Reachability: a constrained path propagator implemented as a multi-agent system

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

Reachability is a propagator 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.

The effectiveness of our *Reachability* propagator has been shown by applying it to the Hamiltonian Path problem. The experimental evaluations that we have done in [10] show that it provides strong pruning, obtaining solutions with very little search. Furthermore, the experiments 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.

In this paper we elaborate on the implementation of *Reachability*. *Reachability* has been implemented using a message passing approach on top of the multi-paradigm programming language Oz [7]. I.e., *Reachability* is a multi-agent system where agents send each other synchronous and asynchronous messages and their transition state functions rely on data flow and constraint programming primitives [14]. In the implementation, we profited from the Finite Integer Set module provided by the Mozart system, which implements Oz [5].

Keywords: constraint programming, filtering algorithms, message passing, concurrent programming.

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 [9, 1, 6] and Bioinformatics [3].

An approach to solve this problem is by using Concurrent Constraint Programming (CCP) [13, 8]. 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.

In [10], we present *Reachability* as a propagator that is suitable for solving Hamiltonian Path with optional nodes. Given a directed graph g, a source node $source$ and a destination node $dest$, the Hamiltonian Path problem [2] consists in finding a path going from $source$ to $dest$ visiting every node of g once. In Hamiltonian Path with optional nodes, we are forced to visit only a specific subset of the nodes (instead of visiting all the nodes). We also show how a standard approach for dealing with this kind of problem, which is based on the use of *AllDiff* [11] and *NoCycle* [1], can be radically enhanced by using *Reachability*.

This paper is organized as follows: In section 2 we present the Reachability Constraint and a subset of its pruning rules. In section 3, we show how we can use a concurrent constraint language for defining propagators. In section 4, after presenting the CP(Graph) framework and its role in the implementation of *Reachability*, we show the implementation of the pruning rules in Oz. In this section, we also show that the composition of propagation, which we call *Batch Propagation*, can be easily achieved when using a message passing approach. The implementation of *Batch*

Propagation is an important result since it plays an important role in the reduction of the computation time. This is because it minimizes the number of activations of expensive propagators.

2 The reachability propagator

2.1 Reachability constraint

The Reachability constraint is defined as follows:

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

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:

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

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

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

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

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.

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

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

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

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

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

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 [3] 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 $\langle N_2, E_2\rangle$. 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) [5] 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 [10]. 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)$\forall_{i \in N}.Reach(g, i) = rn(i)$ 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:

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

One effect of these propagation rules is that, if $Source$ reaches j, and there is only one path p from $Source$ to j, then p is in G. For instance, consider the example in Figure 1. Let us assume that the source node is 1 and that 1 reaches 4 (this is why 4 is in the lower bound of node 1's FS variable). Then, let us add the constraint that edge $\langle 1, 3 \rangle$ is not in G. This constraint and the information we already had imply that the path from 1 to 4 passing through 2 is in G since there is not other way of reaching 4 from 1.

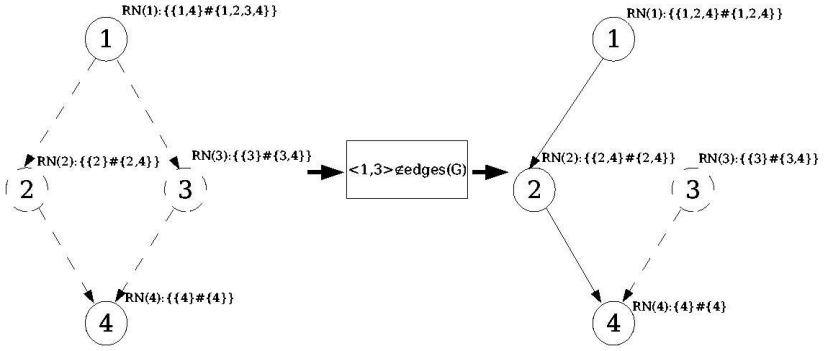

Figure 1: Pruning done by *Reachability* when the constraint $\langle 1, 3 \rangle \notin G$ is added. Dashed nodes/edges are nodes/edges that may not be part of G (i.e., nodes/edges that are in the upper bound of G but not in the lower bound of G.

3 Defining propagators using a concurrent constraint language

We will define the Reachability propagator using a concurrent functional language, namely the declarative subset of Oz. This language is a concurrent constraint language in the sense of Saraswat [12]. For our purposes, it can be considered as a functional language that executes concurrently over a constraint store. The constraint store consists of a conjunction of primitive constraints. For example, in Figure 2 we observe that `Y` is the integer 42, `B` is a Finite Domain(FD) variable whose domain is $\{0, 1\}$, `S` is a FS variable whose lower and upper bounds are $\emptyset$ and $\{5\}$, `Msgs` is a list that is partially determined, and `Z` is a record with label `person` that has two fields: `age` whose value is the value of the variable `Y`, and `sex` whose value is `w`.

Information can only be added to the constraint store, by a "tell" operation, and never removed. Threads synchronize on information becoming available in the store, by an "ask" operation.

In our framework we distinguish three types of propagators:

- **Level 1**. These propagators are optimizations of propagators belonging to the two other levels that are provided by Mozart and implemented in C++. A propagator in this level can be considered as a thread that waits for information to become available, and then adds new information. For example, the propagator implementing the constraint `X=<:Y` reduces the upper bound of X to 10 when the constraint store knows that Y has upper bound 10.

- **Level 2**. A propagator in this level can be considered as a set of threads, each of which executes a recursive function that continuously waits for information to be added to the store, in order to add other information to the store. For instance, in Figure 4, `CreateCounter` creates a thread that reads its messages from the stream `S` and updates its state accordingly. This thread ceases to exist when reading the message `stop`. Notice that this thread computes a list containing the state values.

- **Level 3**. Propagators in this level can be seen as agents: active entities with which one can exchange messages (see chapter 5 of [14]). An agent is supposed to receive messages from different threads, so the order in

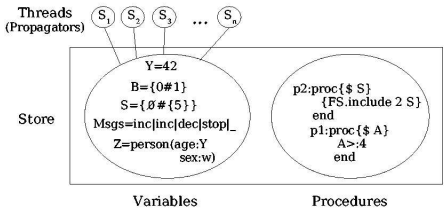

Figure 2: The Oz Execution Model (Declarative subset)

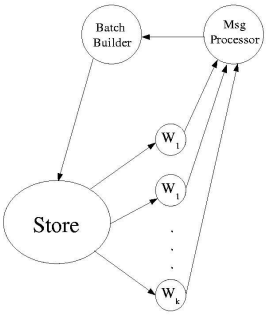

Figure 3: Architecture of a Graph variable propagator

```
proc {CreateCounter InitState S}
   fun {NextState state(val:Val output:Output) Msg}
      NewOutput
   in
      Output=Val|NewOutput
      case Msg of
         inc then state(val:Val+1 output:NewOutput)
      [] dec then state(val:Val-1 output:NewOutput)
      end
   end
   proc {ProcessMsgs State=state(val:Val output:Output) S}
      case S of stop|_ then Output=nil
      [] Msg|RestS then
         {ProcessMsgs {NextState State Msg} RestS}
      end
   end
in
   thread {ProcessMsgs InitState S} end
end
{CreateCounter state(val:0 output:Output) Msgs}
Msgs=inc|inc|dec|stop|_
```

Figure 4: A thread reading messages from a stream

which the agent receives the messages is completely indeterministic. This is why the agent is equipped with a communication channel (port) through which the messages are sent.

The global propagator of the graph variable that we are going to introduce in the next section is a level 3 propagator. The need of the communication channel comes from the fact that the order in which nodes/edges are introduced/excluded is not known a priori. Our solution is to have a thread per node/edge watching the insertion/exclusion of the node/edge. Once the node/edge is include/exclude the thread (which we call watcher) sends the corresponding message to the port. For instance, the following is the implementation of a node watcher. `Graph.N1.isIn` is 1/0 if `N1` is/is not in the graph. Once it is known that `N1` is/is not in the graph the watcher sends the message `includeNode(N1)`/`excludeNode(N1)` to the message processor.

```
thread
   if Graph.N1.isIn==1 then {Send MsgProcessor includeNode(N1)}
   else {Send MsgProcessor excludeNode(N1)} end
end
```

The interaction between the watchers and the message processor of the graph variable is shown in Figure 3. Notice that in this figure there is an additional component that we are going to introduce in section 4.4.

$S ::= S\ S$	Sequence
$\mid X = f(l_1 : Y_1 \ldots l_n : Y_n) \mid$	Value
$\mid X =$<number> $\mid X =$<atom>	
$\mid$ **local** $X_1 \ldots X_n$ **in** S **end** $\mid X = Y$	Variable
$\mid$ **proc** $\{\mathrm{X}\ Y_1 \ldots Y_n\}\ S$ **end** $\mid \{\mathrm{X}\ Y_1 \ldots Y_n\}$	Procedure
$\mid$ **if** X **then** S **else** S **end**	Conditional
$\mid$ **thread** S **end**	Thread

Table 1: The Oz declarative kernel language.

Each of the pruning rules of section 2.2 can be implemented in a straightforward way as a propagator using this computation model.

The declarative language we introduce here is based on procedures; semantically a procedure is similar to a process in a process calculus. This is because procedures can create threads and a thread can exist indefinitely as a running entity if it is implementing a propagator. We can still consider the language to be declarative, however, because it is confluent (see chapter 13 of [14]). Because of the monotonicity of the store, the concurrency executes in a restricted form that is deterministic and has no race conditions. This is clearly explained in chapter 4 of [14].

All Oz execution can be defined in terms of a kernel language whose semantics are given in chapter 13 of [14]. We will just refer to the declarative part of it.

Table 1 defines the abstract syntax of a statement S in the declarative subset of the Oz kernel language. Statement sequences are reduced sequentially inside a thread. All variables are logic variables, declared in an explicit scope defined by the local statement. Values (records, numbers, etc.) are introduced explicitly and can be equated to variables. Procedures are defined at run-time with the **`proc`** statement and referred to by a variable. Procedure applications block until the first argument references a procedure name. The **`if`** statement defines a conditional that blocks until its condition is **`true`** or **`false`** in the variable store. Threads are created explicitly with the **`thread`** statement. Each thread has a unique identifier that is used for thread-related operations.

In the following section, we are going to be using a bit of syntactic sugar to make programs easier to read. We will do so by considering that:

- **`proc`** `{...}` `...` **`in`** `...` **`end`** is equivalent to **`proc`** `{...}` **`local`** `...` **`in`** `...` **`end end`**.
- **`fun`** `{F V1 V2 ... Vn}` `<Stm>` `<Exp>` **`end`** is equivalent to **`proc`** `{F V1 V2 ... Vn R}` `...` `<Stm>` `R=<Exp>` **`end`**, where `<Exp>` is an expression representing a value and `<Stm>` is any statement.
- **`fun`** `{...}` `...` **`in`** `...` **`end`** is equivalent to **`fun`** `{...}` **`local`** `...` **`in`** `...` **`end end`**.

Procedures are values in Oz. This means that a variable may be bound to a procedure. In particular, we have that **`proc`** `{X V1...Vn}...` **`end`** is equivalent to `X=`**`proc`** `{$ V1...Vn}...` **`end`**, where the RHS is a procedure value.

4 Implementation of *Reachability*

4.1 CP(Graph)

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 [3, 4]. CP(Graph) also introduces node variables and edge variables, and is integrated with the finite domain and finite set computation domain.

The kernel constraints of CP(Graph) are:

- $Nodes(G, SN)$: SN is the set of nodes of G.
- $Edges(G, SE)$: SE is the set of edges of G.
- $EdgeNode(E, N_1, N_2)$: the edge variable E is an edge from node N_1 to node N_2.

Consistency techniques have been developed, graph constraints have been built over the kernel constraints and global constraints have been proposed. $CP(Graph)$ has also been implemented in Oz [4].

4.2 Implementing CP(Graph) using message passing

In [10], we re-implemented part of CP(Graph) using a Message Passing approach, for implementing our *Reachability* propagator. We focussed on graph variables and provided the following implementation of the two first kernel constraints:

- `{G incN(N)}` results in $Nodes(G, SN) \wedge N \in SN$
- `{G exN(N)}` results in $Nodes(G, SN) \wedge N \notin SN$
- `{G incE(E)}` results in $Edges(G, SE) \wedge E \in SE$
- `{G exE(E)}` results in $Edges(G, SE) \wedge E \notin SE$
- `{G isN(N B)}` results in $Nodes(G, SN) \wedge (B = true \vee B = false) \wedge (N \in SN \leftrightarrow B = true)$
- `{G isE(E B)}` results in $Edges(G, SE) \wedge (B = true \vee B = false) \wedge (E \in SE \leftrightarrow B = true)$

Additionally, in our implementation, `{G stream($)}` is the stream that contains the messages associated with the constraints that have been imposed on `G`. So, if we have imposed the constraints:

```
{G incN(1)} {G incN(2)} {G exE(1#2)} {G incE(2#1)} {G exN(3)}
```

the partial value of `S` would be:

```
incN(1)|incN(2)|exE(1#2)|incE(2#1)|exN(3)|_
```

4.3 Pruning of *Reachability*

The skeleton of the implementation of *Reachability* is shown in Figure 5. In the implementation of *Reachability* there are two basic components: a set of already provided FS/FD propagators and a global (user defined) propagator. In this section, we will elaborate on the different propagators that constitute *Reachability* by referring to the pruning rules that they implement.

Notice that `CreateGlobalPropagator` creates an agent whose behavior is defined by the function `NextState`. The agent ceases to exist when encountering the message `determined` in the stream. `determined` signals the determination of the graph variable. G is determined when its lower bound is equal to its upper bound (i.e.,$min(G) = max(G)$). The determination of G implies that no message comes after `determined`.

4.3.1 Transitive closure of Reachability *(Rules 13 and 14)*

```
%% For every potential node I of G
/*1*/{FD.impl ({FS.card RN.I} >: 0) {G isN(I $)} 1}
/*2*/{FD.impl {G isN(I $)} {FS.reified.isIn I RN.I} 1}
```

Statement 1 imposes an implication between the cardinality of `RN.I` being greater than 0 and the presence of `I` in `G`. I.e., a node should be part of the graph in order to reach another one.

Statement 2 imposes an implication between the presence of `I` in `G` and `I` reaching itself. This is because every node of `G` reaches itself.

```
/*3*/Ss={G sucs($)}

%% For every potential pair of nodes <I,J> of G
/*4*/{FD.impl {FS.reified.isIn J Ss.I} {ReifiedSubSet RN.J RN.I} 1}
```

`Ss.I` is the set of successors of `I`. As these variables are already present in the implementation of graph variables, we simply make the corresponding associations between those variables and `Ss`(Statement 3).

Statement 4 imposes an implication between `J` being in `Ss.I` and `RN.J` being a subset of `RN.I`.

```
proc {Reachability G Source RN CN BE}
   ...
   proc {CreateGlobalPropagator G Source RN CN BE}
      fun {NextState state(graph:G) Msg}
         ...
      end
      proc {ProcessMsgs state(graph:G) Stream}
         case Stream of
            determined|_ then
            %% End of message processing
         [] Msg|RestStream then
            {ProcessMsgs {NextState state(graph:G) Msg} RestStream}
         end
      end
   in
      thread
         {ProcessMsgs state(graph:{MakeCompleteGraph NumNodes}) {G stream($)}}
      end
   end
in
   for I in 1..NumNodes do
      %% Unary propagators
      ...
      for J in 1..NumNodes do
         %% Binary propagators
         ...
      end
   end
   {CreateGlobalPropagator G Source RN CN BE}
end
```

Figure 5: Skeleton of Reachability

4.3.2 Pruning the upper bound of $RN(i)$ (Rule 17)

We first have to ensure that, for every `I` that is already known to belong to `G`, `RN.I` gets determined when `I` has no successors:

```
%% For every potential node I of G
/*5*/{FD.impl
      ({FS.card RN.I} >: 0)
      {FD.impl ({FS.card Ss.I} =: 0) ({FS.card RN.I} =: 1)}
      1}
```

We also have to ensure that `I` only reaches itself and the nodes that its successors reach. The following statement does that:

```
/*6*/local
        fun {Accumulate Sets J}
           if I\=J then S={FS.var.decl} in
              /*8*/{Select {G isInEdge(I#J $)} RN.J FS.value.empty S}
              S|Sets
           else Sets end
        end
        /*7*/SucSets={FoldL NodesIds Accumulate nil}
        /*9*/ReachedNodes={FS.unionN {FS.value.singl I}|SucSets}
     in
        /*10*/{Select ({FS.card RN.I} >: 0) ReachedNodes FS.value.empty RN.I}
```

```
   end
```

`SucSets`, defined in Statement 7, is bound to the sets of nodes reached by the successor. As we may not know a priori whether `J` is going to be successor of `I`, the corresponding set `S` is a set that is either the empty set (in case `J` is not a successor) or `RN.J`. This relation is imposed by the application of `Select`:

```
proc {Select Cond S1 S2 S3}
   {FS.subset S3 {FS.union S1 S2}}
   {FS.subset {FS.intersect S1 S2} S3}
   thread
      or Cond=1 S3=S1 [] Cond=0 S3=S2 end
   end
end
```

Depending on `Cond`, `Select` binds `S3` to `S1` or `S2`. Moreover, as `S3` is either `S1` or `S2`, `Select` constrains `S3` to have only the elements that `S1` and `S2` have and to include the elements that `S1` and `S2` have in common.

Statement 10 is the one that actually constrains `RN.I` to be the set containing `I` and the nodes reached by the successors of `I`. However, this is done on the condition that `I` is a node of `G` (i.e., `({FS.card RN.I} >: 0)`).

This is all what is needed for pruning a graph without cycles since the sets of reached nodes of the leaves get bound because of Statement 5, and this information is propagated to the corresponding predecessor because of Statement 10.

However, if `G` has cycles, the reached nodes sets do not get determined even if `G` is already determined. For instance, suppose that the lower and upper bound of `G` is `graph(1:[2] 2:[1])` and that the potential set of nodes is $\{1, 2, 3\}$. The propagators above mentioned will basically constrain `RN.1` to be equal to `RN.2` (and `RN.3` to be the empty set). Additionally, due to Statement 1 and 2, nodes 1 and 2 get into the lower bound of `RN.1` and `RN.2`. However, no propagator removes 3 from the upper bound of neither `RN.1` nor `RN.2`.

The upper bound of each reached nodes set is updated in the transition function of the global propagator of *Reachability*:

```
fun {NextState state(graph:G) Msg}
   case Msg of exE(N1#N2) then
      /*11*/NewG={RemoveEdge G N1#N2}
   in
      /*12*/{FS.subset RN.N1 {FS.value.make {DFS.reach N1 NewG}}}
      /*13*/{UpdateCutNodes CN Source NewG}
      /*14*/{UpdateBridges BE Source NewG}
      state(graph:NewG)
   else
      state(graph:G)
   end
end
```

The internal state of the global propagator is the upper bound of G. Each time an edge is removed, this upper bound is updated (Statement 11) and so are the upper bounds of the reached nodes sets affected (Statement 12). Notice that it is enough to update the reached nodes set of the origin of the edge removed (`N1`) since the rest will be done by Statement 10. Notice that `RN.N1` is updated by imposing that `RN.N1` is a subset of the nodes reached by `N1` in the upper bound `G`.

4.3.3 Discovering cut nodes

We have to start by keeping track of the cut nodes between the source and each other node (`CN.I`). As the set of cut nodes may change when an edge is removed, we update `CN.I` each time an edge removal takes place by invoking `UpdateCutNodes` (Statement 13). Notice that, in this statement, we are taking care of Rule 18 [1].

```
/*15*/{FD.impl {FS.reified.isIn I RN.Source} {ReifiedSubSet CN.I RN.Source} 1}

/*16*/{FD.impl {FS.reified.isIn J RN.I} {G isN(J $)} 1}
```

[1] We present the algorithms that we use for computing cut nodes and bridges in [10]. These algorithms are based on DFS [10].

In order to perform the pruning of rules 15 and 16. We impose an implication between `I` belonging to `RN.Source` and `CN.I` being a subset of `RN.Source` (Statement 15), and between `J` belonging to `RN.I` and `J` belonging to the nodes of `G` (Statement 16). In fact, this last statement also takes the pruning performed by rules 20 and 21 into account. An example illustrating the pruning performed by these statements is shown in Figure 6. In this example we impose the constraint that node 1 should reach node 9. As 5 is a cutnode between 1 and 9, 5 is included in `G` and forced to reach 9. Additionally, 1 is constrained to reach 5.

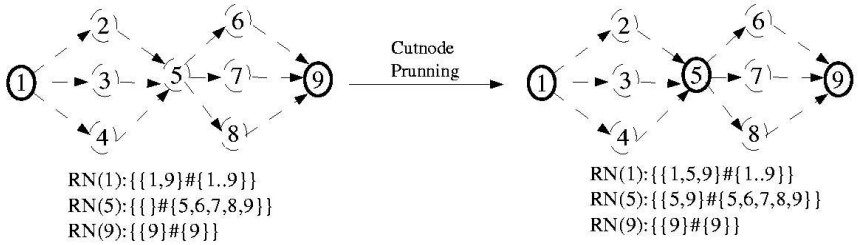

Figure 6: Discovering cut nodes

4.3.4 Discovering bridges

As in the previous case, `BE.I` is updated each time an edge removal takes place by invoking `UpdateBridges` (Statement 14).

```
/*17*/{FD.impl {FS.reified.isIn I RN.Source} {ReifiedEdgesInGraph BE.I G} 1}
```

We impose an implication between `I` belonging to `RN.Source` and the bridges between `Source` and `I` belonging to the edges of `G` (Statement 17). This statement covers the pruning of Rule 22 into account. An example illustrating the pruning performed by this statement is shown in Figure 7. In this example we impose the constraint that node 1 should reach node 5. This constraint is enough to determine the only path between 1 and 5.

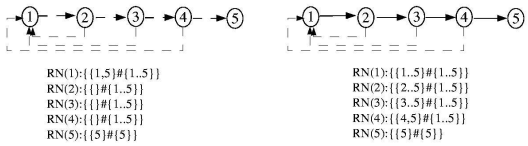

Figure 7: Discovering bridges

4.4 Batch propagation

In the previous implementation, we compute cut nodes and bridges each time an edge is removed. This certainly leads to a considerably amount of unnecessary computation since the set of cut nodes/bridges evolves monotonically. Another approach is to consider all the removals at once and make one computation of cut nodes and bridges per set of edges removed. This optimization can be implemented by adding a concurrent process to the implementation of graph variables. The task of this process is to batch together the messages according to their types (as shown in Figure 8). In this way, the transition function of the global propagator of *Reachability* will consider all the edges that have been removed at once:

```
fun {NextState state(graph:G)  batch(exE:Es ...)}
   if Es==nil then state(graph:G)
   else
```

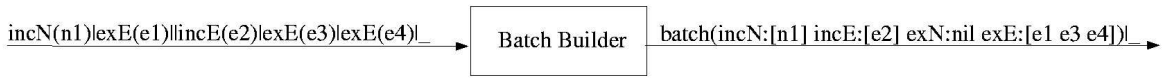

Figure 8: Building batches

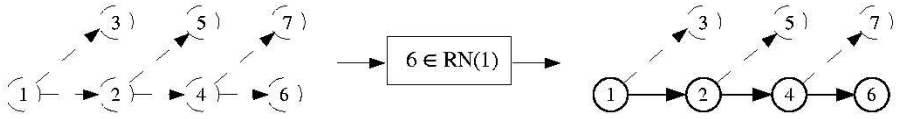

Figure 9: Simple Bridge Discovering

```
      NewG={RemoveEdges Es G}
   in
      {UpdateRNs Es NewG}
      {UpdateCutNodes CN Source NewG}
      {UpdateBridges BE Source NewG}
      state(graph:NewG)
   end
end
```

In fact, this transition function is very similar to the previous one. The only different thing is that `NewG` is considering all the nodes that have been removed.

Statement 6 is a cheap way of computing bridges when there is no cycle. Notice that, in the situation of Figure 9, the pruning performed by Statement 6 is enough for discovering the bridges between node 1 and node 6. However, the global propagator also discovers this information. The point in having this redundancy in propagation is that, thanks to the fact that the expensive propagator works on batches, there are cases where the expensive computation of bridges is not activated. Suppose, for instance, that discovering the bridge $\langle 2, 4\rangle$ raises a failure because 4 is not reached by 2. This failure is discovered by the cheap propagator and the expensive one is not activated.

5 Conclusion and future work

We presented the implementation of *Reachability*, which has been implemented using a message passing approach on top of the multi-paradigm programming language Oz [7]. We showed how the use of FS variables simplified the implementation of most of the rules.

In the implementation of *Reachability* we distinguished two basic components: a set of already provided FS/FD propagators and a global (user defined) propagator. We showed the global propagator as an agent that reads messages from a stream generated by the graph variable on which *Reachability* is applied.

We presented a cheap way of discovering bridges based on FS pruning. After introducing our implementation of Bath propagation using message passing, we explained why this can play an important role in the reduction of the computation time.

From our observations in [10], 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.

Our experiments in [10] also show that the appropriateness of *Reachability* is increased with the presence of optional nodes. This is basically because we are no longer able to apply the global *AllDiff* propagator on the successors of the nodes since we do not know a priori which nodes participate in the path. However, the complexity of the problem tends to increase with the number of optional nodes if they are uniformly distributed.

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 6 is a context where the computation of bridges cannot infer anything since there is no bridge. Similarly, Figure 7 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?.

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, what we want is to use the pruning previously performed in order to repair the pruning.

References

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

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

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

[4] G. Dooms, Y. Deville, and P. Dupont. CP(Graph):introducing a graph computation domain in constraint programming. Research Report INFO-2005-06, Université catholique de Louvain, Louvain-la-Neuve, Belgium, 2005.

[5] 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/*.

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

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

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

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

[10] Luis Quesada, Peter Van Roy, and Yves Deville. The reachability propagator. Research Report INFO-2005-07, Université catholique de Louvain, Louvain-la-Neuve, Belgium, 2005. Available at *http://www.info.ucl.ac.be/~luque/SPMN/paper.pdf*.

[11] 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.

[12] Vijay Saraswat. *Concurrent Constraint Programming*. The MIT Press, 1993.

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

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