

Using Dominators for Solving Constrained Path Problems

Luis Quesada, Peter Van Roy, Yves Deville, and Raphaël Collet

Université catholique de Louvain,
Place Sainte Barbe, 2, B-1348 Louvain-la-Neuve, Belgium
{luque, pvr, yde, raph}@info.ucl.ac.be

Abstract. Constrained path problems have to do with finding paths in graphs subject to constraints. We present a constraint programming approach for solving the Ordered disjoint-paths problem (ODP), i.e., the Disjoint-paths problem where the pairs are associated with ordering constraints. In our approach, we reduce ODP to the Ordered simple path with mandatory nodes problem (OSPMN), i.e., the problem of finding a simple path containing a set of mandatory nodes in a given order. The reduction of the problem is motivated by the fact that we have an appropriate way of dealing with OSPMN based on *DomReachability*, a propagator that implements a generalized reachability constraint on a directed graph based on the concept of graph variables.

The *DomReachability* constraint has three arguments: (1) a flow graph, i.e., a directed graph with a source node; (2) the dominance relation graph on nodes and edges of the flow graph; and (3) the transitive closure of the flow graph.

Our experimental evaluation of *DomReachability* shows that it provides strong pruning, obtaining solutions with very little search. Furthermore, we show that *DomReachability* is also useful for defining a good labeling strategy. These experimental results give evidence that *DomReachability* is a useful primitive for solving constrained path problems over directed 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 between 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 in a given order. We find instances of this problem in Vehicle routing problems [PGPR96, CL97, FLM99] and Bioinformatics [DDD04].

An approach to solve this problem is by using concurrent constraint programming (CCP) [Sch00, Mül01]. In CCP, we solve the problem by interleaving two processes: propagation and labeling. Propagation consists in filtering the domains of a set of finite domain variables, according to the semantics of the constraints that have to be satisfied. Labeling consists in defining the way the search tree is created, i.e., which constraint is used for branching.

In this paper, we present a propagator called *DomReachability*, that implements a generalized reachability constraint on a directed graph. The *DomReachability* constraint

has three arguments: (1) a flow graph, i.e., a directed graph with a source node; (2) the dominance relation graph on nodes and edges of the flow graph; and (3) the transitive closure of the flow graph. The dominance relation graph represents a dominance relation that identifies nodes common to all paths from a source to a destination. By extending the dominator graph we can also identify edges common to all paths from a source to a destination.

Due to the fact that the arguments of *DomReachability* are graph variables that can be partially instantiated, the problem modelled with *DomReachability* can be understood as finding a flow graph that respects the partial instantiations of the flow graph, the dominance relation graph and the transitive closure. For instance, we may be interested in finding a subgraph of a given graph where a node j is reached from a node s and j is dominated by a set of nodes ns with respect to s .

Applicability. The *DomReachability* propagator is suitable for solving the Simple path with mandatory nodes problem [Sel02, CB04]. 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 only once. Certainly, this problem can be trivially solved if the graph has no cycle, since in that case there is only one order in which we can visit the mandatory nodes [Sel02]. However, the presence of cycles makes the problem NP-complete, since we can easily reduce the Hamiltonian path problem [GJ79, CLR90] to this problem.

Note that we can not trivially reduce Simple path with mandatory nodes to Hamiltonian path. One could think that optional nodes (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 paths that are precomputed may share nodes. This may lead to violations of the requirement that a node should be visited at most once.

Figure 1 illustrates this situation. Mandatory nodes are drawn with 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. 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. Therefore the simple path in the second graph is not a valid solution to the original problem since it requires node 3 to be visited twice. Note that the Simple path problem with only one mandatory node, which is equivalent to the 2-Disjoint paths problem [SP78], is still NP-complete.

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, we observe that the suitability of *DomReachability* for dealing with Simple path with mandatory nodes relies on the following aspects:

- The strong pruning that *DomReachability* performs. Due to the computation of dominators, *DomReachability* is able to discover non-viable successors early on.

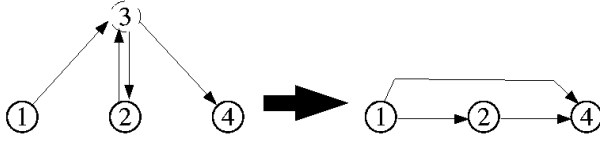


Fig. 1. Relaxing Simple path with mandatory nodes by eliminating the optional nodes

- The information that *DomReachability* provides for implementing smart labeling strategies. *DomReachability* associates each node with the set of nodes that it reaches. This information can be used to guide the search in a smart way. The strategy we used in our experiments tends to minimize the use of optional nodes.

An additional feature of *DomReachability* is its suitability for dealing with a problem that we call the Ordered simple path with mandatory nodes problem (OSPMN) where ordering constraints among mandatory nodes are imposed, which is a common issue in routing problems. Taking into account that a node i reaches a node j if there is a path going from node i to node j , one 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 latter is equivalent to imposing that i is an ancestor of j in the extended dominator tree of the path. Our experiments show that *DomReachability* takes the most advantage of this information to avoid branches in the search tree with no solution.

Related work. The cycle constraint of CHIP [BC94, Bou99] $cycle(N, [S_1, \dots, S_n])$ 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, [Bou99] shows how this constraint can be used to deal with the Euler knight problem (which is an application of Hamiltonian Path). Optional nodes can be modelled by putting each optional node in a separate elementary cycle. However, this constraint is not implemented in terms of dominators.

Sellmann [Sel02] suggests some algorithms for discovering mandatory nodes and non-viable edges in directed acyclic graphs. These algorithms are extended by [CB04] in order to address directed graphs in general with the notion of strongly connected components and condensed graphs. Nevertheless, graphs similar to our third benchmark [SPMc] 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 [DDD04, DDD05]. 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. One of those global constraints is $Path(p, s, d, maxlen)$. This constraint is satisfied if p is a simple path from s to d of length at most $maxlen$. Certainly, Simple path with mandatory nodes can be implemented in terms of $Path$. However, the filtering algorithm of $Path$ does not compute dominators, which makes $Path$ also sensible to cases like SPMN_52a.

Dominators are commonly used in compilers for dataflow analysis [AU77]. Dominance constraints also appear in natural language processing, for building semantic trees from partial information. However, we are not aware of approaches using dominators for implementing filtering algorithms. Even though the information it provides is extremely useful, and can be computed efficiently.

Structure of the paper. The paper is organized as follows. In Section 2, we introduce *DomReachability* by presenting its semantics and pruning rules. In Section 3, we show how we can model Simple path with mandatory nodes in terms of *DomReachability*. Section 4 gives experimental evidence of the performance of *DomReachability* for this type of problem. In Section 5 we show a reduction of the Ordered disjoint-paths problem (ODP) to OSPMN, which can be solved by our approach.

2 The *DomReachability* Propagator

2.1 Extended Dominator Graph

Given a flow graph fg and its corresponding source s , a node i is a dominator of node j if all paths from s to j in fg contain i [LT79, SGL97]:

$$i \in \text{Dominators}(fg, j) \leftrightarrow i \neq j \wedge \forall p \in \text{Paths}(fg, s, j) : i \in \text{Nodes}(p) \quad (1)$$

where

$$p \in \text{Paths}(fg, i, j) \leftrightarrow \begin{cases} p \text{ is a subgraph of } fg \\ \text{Nodes}(p) = \{k_1, \dots, k_n\} \wedge k_1 = i \wedge k_n = j \\ \text{Edges}(p) = \{\langle k_t, k_{t+1} \rangle \mid 1 \leq t < n\} \end{cases} \quad (2)$$

Note that the nodes unreachable from s are dominated by all the other nodes. However, the nodes reachable from s always have an *immediate* dominator, which can be defined as

$$i = \text{ImDominator}(fg, j) \leftrightarrow \begin{cases} i \in \text{Dominators}(fg, j) \\ \neg \exists k \in \text{Nodes}(fg) : i \in \text{Dominators}(fg, k) \wedge k \in \text{Dominators}(fg, j) \end{cases} \quad (3)$$

This property allows to represent the whole dominance relation as a tree, where the parent of a node is its immediate dominator. The dominator tree can be used as an efficient representation of the relation, as there exists incremental algorithms for updating the tree [SGL97]. This paper only presents a non-incremental algorithm to compute the whole relation (see Figure 5).

Let us now consider the extended graph of fg , $\text{Ext}(fg)$, which is obtained by replacing the edges by new nodes, and connecting the new nodes accordingly. This graph can be formally defined as follows:

$$\langle N', E', s' \rangle = \text{Ext}(\langle N, E, s \rangle) \leftrightarrow \begin{cases} s' = s \\ N' = N \cup E \\ e = \langle i, j \rangle \in E \leftrightarrow \langle i, e \rangle \in E' \wedge \langle e, j \rangle \in E' \end{cases} \quad (4)$$

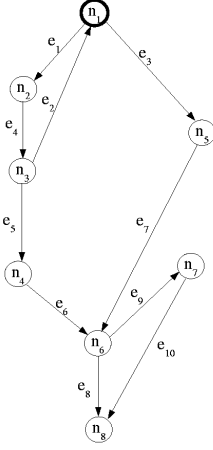


Fig. 2. Flow graph

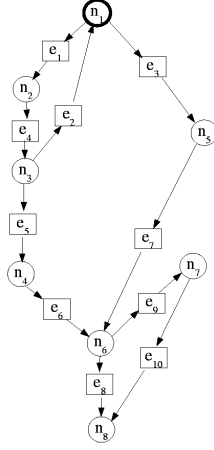


Fig. 3. Extended flow graph

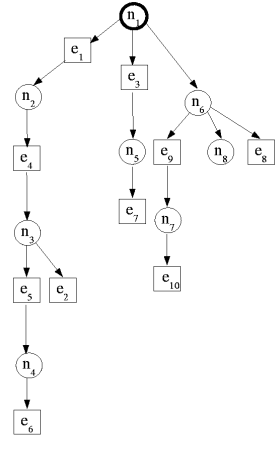


Fig. 4. Extended dominator tree

The extended dominator graph of fg is the dominator graph of its extended graph. Figures 2, 3 and 4 show an example of a flow graph, its extended graph, and its extended dominator tree, respectively. The extended dominator tree has two types of nodes: nodes corresponding to nodes in the original graph (*node dominators*), and nodes corresponding to edges in the original graph (*edge dominators*). The latter nodes are drawn in squares.

The extended dominator tree provides useful information. For instance, consider two node dominators i and j . If $\langle i, j \rangle \in \text{Edges}(\text{DomTree}(\text{Ext}(fg))) \setminus \text{Edges}(fg)$, there are at least two node-disjoint paths from i to j in the flow graph (as it is the case between nodes 1 and 6 in Figure 4). Note also that, if i is an ancestor of j in the extended dominator tree, and the path from i to j does not contain any edge dominator, there are at least two edge-disjoint paths from i to j in the flow graph.

2.2 The *DomReachability* Constraint

The *DomReachability* constraint is a constraint on three graphs:

$$\text{DomReachability}(fg, \text{edg}, tc) \quad (5)$$

where

- fg is a flow graph, i.e., a directed graph with a source node, whose set of nodes is a subset of N ;
- edg is the extended dominator graph of fg ; and
- tc is the transitive closure of fg , i.e.,

$$\begin{aligned} \langle i, j \rangle \in \text{Edges}(tc) &\leftrightarrow \langle i, j \rangle \in \text{Edges}(\text{TransClos}(fg)) \\ \langle i, j \rangle \in \text{Edges}(\text{TransClos}(g)) &\leftrightarrow \exists p : p \in \text{Paths}(g, i, j) \end{aligned} \quad (6)$$

The above definition of *DomReachability* implies the following properties which are important for the pruning that *DomReachability* performs. These properties define relations between the graphs fg , edg and tc . These relations can then be used for pruning, as we show in the next section.

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

$$\forall \langle i, j \rangle \in Edges(fg) : \langle i, j \rangle \in Edges(tc) \quad (7)$$

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

$$\forall i, j, k \in N : \langle i, j \rangle \in Edges(tc) \wedge \langle j, k \rangle \in Edges(tc) \rightarrow \langle i, k \rangle \in Edges(tc) \quad (8)$$

3. If j is reachable from $s = Source(fg)$ and i dominates j in fg , then i is reachable from s and j is reachable from i :

$$\forall i, j \in N : \langle s, j \rangle \in Edges(tc) \wedge \langle i, j \rangle \in Edges(edg) \rightarrow \langle s, i \rangle \in Edges(tc) \wedge \langle i, j \rangle \in Edges(tc) \quad (9)$$

2.3 Pruning Rules

We implement the constraint (5) by the propagator that we note

$$DomReachability(\langle FG, s \rangle, EDG, TC). \quad (10)$$

FG , EDG and TC are graph variables, i.e., variables whose domain is a set of graphs [DDD05]. A graph variable G is represented by a pair of graphs $Min(G) \# Max(G)$. The graph g that G approximates must be a supergraph of $Min(G)$ and a subgraph of $Max(G)$, therefore $Min(G)$ and $Max(G)$ are called the lower and upper bounds of G , respectively. So, $i \in Nodes(G)$ holds if $i \in Nodes(Min(G))$, and $i \notin Nodes(G)$ holds if $i \notin Nodes(Max(G))$ (the same applies for edges). Notice that the source s of the flow graph FG is a known value.

The definition of the *DomReachability* constraint and its derived properties give place to a set of propagation rules. We show here the ones that motivate the implementation of incremental algorithms for keeping the dominance relation and the transitive closure of the flow graph. The others are given in [QVD05b]. A propagation rule is defined as $\frac{C}{A}$ where C is a condition and A is an action. When C is true, the pruning defined by A can be performed.

From property (7) we derive

$$\frac{\langle i, j \rangle \in Edges(Min(FG))}{Edges(Min(TC)) := Edges(Min(TC)) \cup \{\langle i, j \rangle\}} \quad (11)$$

From property (8) we derive

$$\frac{\langle i, j \rangle \in Edges(Min(TC)) \wedge \langle j, k \rangle \in Edges(Min(TC))}{Edges(Min(TC)) := Edges(Min(TC)) \cup \{\langle i, k \rangle\}} \quad (12)$$

From property (9) we derive, for $i \in \text{Nodes}(\text{Min}(FG))$,

$$\frac{\langle s, j \rangle \in \text{Edges}(\text{Min}(TC)) \wedge \langle i, j \rangle \in \text{Edges}(\text{Min}(EDG))}{\text{Edges}(\text{Min}(TC)) := \text{Edges}(\text{Min}(TC)) \cup \{\langle s, i \rangle, \langle i, j \rangle\}} \quad (13)$$

From definition (6) we derive

$$\frac{\langle i, j \rangle \notin \text{Edges}(\text{TransClos}(\text{Max}(FG)))}{\text{Edges}(\text{Max}(TC)) := \text{Edges}(\text{Max}(TC)) \setminus \{\langle i, j \rangle\}} \quad (14)$$

From definition (1) we derive

$$\frac{\langle i, j \rangle \in \text{Edges}(\text{DomGraph}(\text{Ext}(\text{Max}(FG))))}{\text{Edges}(\text{Min}(EDG)) := \text{Edges}(\text{Min}(EDG)) \cup \{\langle i, j \rangle\}} \quad (15)$$

where *DomGraph* is a function that returns the dominator graph of a flow graph, i.e., $\langle i, j \rangle \in \text{Edges}(\text{DomGraph}(fg)) \leftrightarrow i \in \text{Dominators}(fg, j)$.

2.4 Implementation of *DomReachability*

DomReachability has been implemented using a message passing approach [VH04] on top of the multi-paradigm programming language Oz [Moz04]. In [QVD05a], we discuss the implementation of *DomReachability* in detail. In this section we simply refer to the update of the upper bound of *TC* and the lower bound of *EDG*. Both values should be updated when an edge is removed from *Max*(*FG*). However, as explained in [QVD05a], we do not compute these values each time an edge is removed since this certainly leads to a considerably amount of unnecessary computation. This is due to the fact that these two values evolve monotonically. What we actually do is to consider all the removals at once and make one computation per set of edges removed.

```

GetDominators(fg)
  nodes0 := DFS(fg, Source(fg))
  for i ∈ Nodes(fg) do
    doms(i) := if i ∈ nodes0 then ∅ else Nodes(fg) \ {i} end
  end
  for i ∈ nodes0 do
    nodes1 := DFS(RemoveNode(fg, i), Source(fg))
    for j ∈ nodes0 \ (nodes1 ∪ {i}) do
      doms(j) := doms(j) ∪ {i}
    end
  end
  return doms
end

```

Fig. 5. Computation of Dominators

Currently, our way of updating TC 's upper bound is simply by running DFS on each node of TC 's upper bound. So the complexity of this update is $O(N * (N + E))$. Regarding EDG 's lower bound, the set of dominators is computed by using the algorithm in Figure 5 (which is actually equivalent to Aho and Ullman's algorithm for computing dominators [AU77]). $doms(i)$ is the set of dominators of node i in fg . Let us assume that DFS returns the reachable nodes. $doms(i)$ is initialized with $\emptyset$ or $Nodes(fg) \setminus \{i\}$ depending on whether i is reached from $Source(fg)$ (since any node dominates an non-reached node). The basic idea of this algorithm is that, if $Source(fg)$ does not reach j after removing i then i dominates j . So, each node is removed in order to detect the nodes that it dominates. Therefore the computation of dominators is $O(N * (N + E))$ too.

3 Solving Simple Path with Mandatory Nodes with DomReachability

In this section we elaborate on the important role that *DomReachability* 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 src , a destination node dst , and a set of mandatory nodes $mandnodes$, we want to find a path in g from src to dst , going through $mandnodes$ and visiting each node only once.

The contribution of *DomReachability* consists in discovering nodes/edges that are part of the path early on. This information is obtained by computing dominators in each labeling step. Let us consider the following two cases¹:

- Consider the graph variable on the left of Figure 6. 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.
- Consider the graph variable on the left of Figure 7. 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.

Note that the Hamiltonian path problem (finding a simple path between two nodes containing all the nodes of the graph [GJ79, CLR90]) can be reduced to Simple path with mandatory nodes by defining the set of mandatory nodes as $Nodes(g) \setminus \{src, dst\}$.

The above definition of Simple path with mandatory nodes can be formally defined as follows.

$$SPMN(g, src, dst, mandnodes, p) \leftrightarrow \begin{cases} p \in Paths(g, src, dst) \\ NoCycle(p) \\ mandnodes \subset Nodes(p) \end{cases} \quad (16)$$

¹ In Figures 6 and 7, 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 6 is a graph variable whose lower bound is the graph $\langle \{1, 5\}, \emptyset \rangle$, and whose upper bound is the graph $\langle \{1, 2, 3, 4, 5, 6, 7, 8, 9\}, \{\langle 1, 2 \rangle, \langle 1, 3 \rangle, \langle 1, 4 \rangle, \langle 2, 5 \rangle, \langle 3, 5 \rangle, \langle 4, 5 \rangle, \langle 5, 6 \rangle, \langle 5, 7 \rangle, \langle 5, 8 \rangle, \langle 6, 9 \rangle, \langle 7, 9 \rangle, \langle 8, 9 \rangle\} \rangle$.

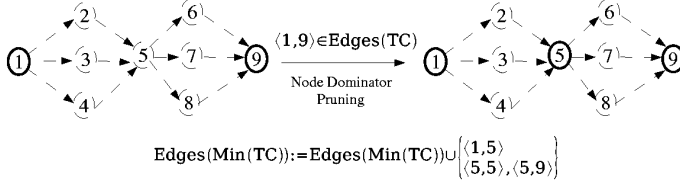


Fig. 6. Discovering node dominators

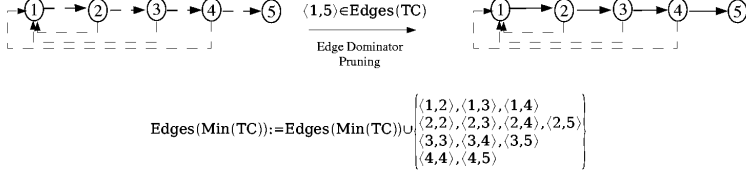


Fig. 7. Discovering edge dominator

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.

$$\text{DomReachability}(p, \text{edg}, \text{tc}) \wedge \langle \text{Source}(p), \text{dst} \rangle \in \text{Edges}(\text{tc}) \wedge \text{mandnodes} \subset \{i \mid \langle \text{Source}(p), i \rangle \in \text{Edges}(\text{tc})\} \quad (17)$$

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 [Ré94] and the *NoCycle* constraint [CL97] 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 *DomReachability*.

Table 1. Two approaches for solving Simple path with mandatory nodes

Approach 1	Approach 2
$\text{SPMN}(g, \text{src}, \text{dst}, \text{mandnodes}, p)$	$\text{SPMN}(g, \text{src}, \text{dst}, \text{mandnodes}, p)$ $\text{DomReachability}(p, \text{edg}, \text{tc})$ $\langle \text{Source}(p), \text{dst} \rangle \in \text{Edges}(\text{tc})$ $\text{mandnodes} \subset \{i \mid \langle \text{Source}(p), i \rangle \in \text{Edges}(\text{tc})\}$

4 Experimental Results

In this section we present a set of experiments that show that *DomReachability* is suitable for Simple path with mandatory nodes. 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

Table 2. Simple path with mandatory nodes instances

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

Table 3. Performance with respect to optional nodes

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. Simple path with mandatory nodes tests

Problem		SPMN		SPMN+R		SPMN+R+ND		SPMN+R+ND+ED	
Instance	Figure	Failures	Time	Failures	Time	Failures	Time	Failures	Time
SPMN_22	[SPMa]	+130000	+1800	91	6.81	40	6.55	13	4.45
SPMN_22full	[SPMb]	213	1.44	19	0.95	0	0.42	0	1.22
SPMN_52b	-	-	-	+900	+1800	+700	+1800	100	402
SPMN_52full	[SPMd]	3012	143	774	765	3	8.51	3	45.03
SPMN_52Order_a	[SPMc]	+12000	+1800	51	46.33	45	81	16	57.07
SPMN_52Order_b	-	+12000	+1800	+1500	+1800	81	157	41	117

uniformly distributed in the graph. We have also observed that the labeling strategy that we implemented with *DomReachability* 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 4². The node 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 measurements are given in seconds. The number of failures means the number of failed alternatives tried before getting the solution.

We have made four types of tests in our experiments: using *SPMN* without *DomReachability* (column “SPMN”), using *SPMN* and *DomReachability* but without considering the dominance graph (column “SPMN+R”), using *SPMN* and *DomReachability* with the dominance graph (column “SPMN+R+ND”), and using *SPMN* and *DomReachability* with the dominance graph of the extended flow graph (node+edge dominators (column “SPMN+R+ND+ED”)).

As it can be observed in Table 4, we were not able to get a solution for SPMN_22 in less than 30 minutes without using *DomReachability*. However, even though the number of failures is still inferior, the use of *DomReachability* does not save too much time when dealing with mandatory nodes only. This is due to the fact that we are basing our

² In order to save space, the figures mentioned in the tables were dropped and made available through references [SPMa], [SPMb], [SPMc] and [SPMd].

implementation of *SPMN* on two things: the *AllDiff* constraint [Rég94] (that lets us efficiently remove branches when there is no possibility of associating different successors to the nodes) and the *NoCycle* constraint [CL97] (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* constraint 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 *DomReachability* on the graph of 52 nodes.

Even though, in *SPMN_22*, the benefit caused by the computation of edge dominators is not that significant, 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 edge dominators. So, the computation of edge dominators pays off in most of the cases, but node dominators should be computed in order to profit from edge dominators.

4.1 Labeling Strategy

DomReachability 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 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 (as it can be observed in [SPMc]).

Nevertheless, in order to reduce the number of failures in finding the solution of [DPC] (which was solved in less than 100 failures), we favored the nodes that were closer to the mandatory nodes, i.e., if the successors of the chosen node are not mandatory the chosen successor is the one closest to the next mandatory node.

4.2 Imposing Order on Nodes

An additional feature of *DomReachability* is its suitability for imposing ordering constraints on nodes (which is a common issue in routing problems). In fact, it might be

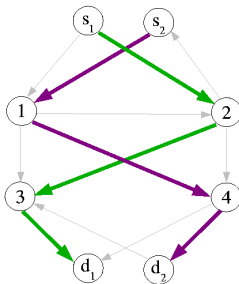


Fig. 8. Finding two disjoint paths

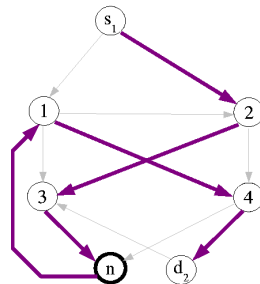


Fig. 9. Finding a simple path passing through n

the case that we have to visit the nodes of the graph in a particular (partial) order. We call this version the “Ordered simple path with mandatory nodes problem” (*OSPMN*).

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 *DomReachability* 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 Reducing the Ordered Disjoint-Paths Problem to the Simple Path with Mandatory Nodes Problem

The k -Disjoint-paths problem consist in finding k pairwise disjoint paths between k pairs of nodes $\langle s_1, d_1 \rangle, \langle s_2, d_2 \rangle, \dots, \langle s_k, d_k \rangle$. Both the node-disjoint version and the edge-disjoint version are NP-complete [SP78]. We will focus on the node-disjoint version.

Let us first look at the problem of reducing the 2-Disjoint-paths problem to *SPMN*. Suppose that we want to find two disjoint paths between the pairs $\langle s_1, d_1 \rangle$ and $\langle s_2, d_2 \rangle$ in g . Let g' and n be defined as follows.

$$\begin{aligned} n &\notin \text{Nodes}(g) \\ g' &= \text{AddEdges}(g_1, E_1 \cup E_2) \\ g_1 &= \text{AddNode}(g_2, n) \\ g_2 &= \text{RemoveNodes}(g, \{d_1, s_2\}) \\ E_1 &= \text{IncEdges}(g, d_1)[d_1/n] \\ E_2 &= \text{OutEdges}(g, s_2)[s_2/n] \end{aligned} \tag{18}$$

Finding the two disjoint paths is equivalent to finding a simple path from s_1 to d_2 passing through n in g' . The correctness of this reduction relies on the fact that the concatenation of the two disjoint paths forms a simple path since each disjoint path is a simple path. Figure 9 shows the the reduction of the two disjoint paths problem of Figure 8. The path found in Figure 9 corresponds to the concatenation of the two disjoint paths of Figure 8.

Let us consider now an extended version of the 2 Node-disjoint path problem that we call *2 Ordered node-disjoint path (2ODP)*. In this version, each pair is associated with a set of mandatory nodes and an order relation on the mandatory nodes. That is, given the directed graph g and the tuples $\langle s_1, d_1, mn_1, order_1 \rangle$ and $\langle s_2, d_2, mn_2, order_2 \rangle$, the goal is to find two paths p_1 and p_2 such that p_1 is a path from s_1 to d_1 visiting mn_1 respecting $order_1$, p_2 is a path from s_2 to d_2 visiting mn_2 respecting $order_2$, and p_1 and p_2 are node-disjoint.

The 2ODP problem $\langle g, \langle \langle s_1, d_1, mn_1, order_1 \rangle, \langle s_2, d_2, mn_2, order_2 \rangle \rangle \rangle$ can be reduced to *OSPMN* $\langle g', s_1, d_2, mn', order' \rangle$ where g' is defined as in the previous reduction, $mn' = mn_1 \cup mn_2 \cup \{n\}$, n is defined as before, and

$$order' = \begin{cases} order_1 \cup \\ order_2 \cup \\ \{ \langle n_1, n_2 \rangle \mid (n_1 \in mn_1 \wedge n_2 = n) \vee (n_1 = n \wedge n_2 \in mn_2) \}. \end{cases} \tag{19}$$

```

ReduceODP( $\langle g, \langle \langle s_1, d_1, mn_1, order_1 \rangle, \dots, \langle s_k, d_k, mn_k, order_k \rangle \rangle \rangle$ )
  ospmn :=  $\langle g, s_1, d_1, mn_1, order_1 \rangle$ 
  for  $i \in \{2, 3, \dots, k\}$  do
     $\langle g', s', d', mn', order' \rangle := ospmn$ 
    ospmn :=  $Reduce\_2\_ODP(\langle g', \langle \langle s', d', mn', order' \rangle, \langle s_i, d_i, mn_i, order_i \rangle \rangle \rangle)$ 
  end
  return ospmn
end

```

Fig. 10. Reducing ODP to OSPMN

The simple path traverses the nodes mn_1 in the order $order_1$, and the nodes mn_2 in the order $order_2$, the nodes mn_1 are visited before n and the nodes in mn_2 after n .

Let $Reduce_2_ODP$ be defined as

$$\begin{aligned}
 Reduce_2_ODP(ODPins) &= OSPMNins \\
 ODPins &= \langle g, \langle \langle s_1, d_1, mn_1, order_1 \rangle, \langle s_2, d_2, mn_2, order_2 \rangle \rangle \rangle \\
 OSPMNins &= \langle g', s_1, d_2, mn', order' \rangle
 \end{aligned} \tag{20}$$

The function $ReduceODP$, which reduces any ordered disjoint path problem (ODP) to OSPMN, can be defined as shown in Figure 10. Certainly, we assume that the pairs $\langle s_1, d_1 \rangle, \langle s_2, d_2 \rangle, \dots, \langle s_k, d_k \rangle$ are pairwise node-disjoint. However, this condition can be easily fulfilled by duplicating the nodes that are used by more than one pair.

Note that the conventional k node-disjoint paths problem can be trivially reduced to ODP. We simply need to map each pair $\langle s_i, d_i \rangle$ to $\langle s_i, d_i, \emptyset, \emptyset \rangle$. We used $ReduceODP$ to solve the case shown in [DPC]. In this case we were interested in finding 14 node-disjoint paths in a directed graph of 165 nodes.

6 Conclusion and Future Work

We presented *DomReachability*, a constrained graph propagator that can be used for solving constrained path problems. *DomReachability* is a propagator that reasons in terms of the three partially defined graphs that it has as arguments. Further definition of one of its graphs may cause the other two graphs to be further defined. After introducing the semantics and pruning rules of *DomReachability*, we showed how its use can speed up a standard approach for dealing with Simple path problem with mandatory nodes. Our experiments show that the gain is increased with the presence of optional nodes. The latter makes the problem harder, and standard approaches perform worse.

It is important to emphasize that both the computation of node dominators, and the computation of edge dominators play an essential role in the performance of *DomReachability*. 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 edge dominators cannot infer

anything since there is no edge dominator. Similarly, Figure 7 represents a context where the computation of edge dominators discovers more information than the computation of node dominators.

As mentioned before, our current approach for maintaining the dominator graph and the transitive closure has complexity $O(N * (N + E))$. However, we are aware of $O(N + E)$ algorithms for updating these structures [SGL97, DI00]. In fact, there is a non-incremental algorithm for computing dominator trees that is more efficient than our current algorithm since it is $O(E\alpha(E, N))$, where $\alpha(E, N)$ is a functional inverse of Ackermann's function [LT79]. Certainly, our next step is to implement these algorithms since we believe that they will remarkably improve the performance of *DomReachability*.

Acknowledgements

We thank Eugene Ressler for pointing out that our former cut nodes [QVD05b] were actually dominators; Christian Schulte, Fred Spiessens and Grégoire Doods for proof-reading earlier versions of this paper; and Martin Oellrich for providing the instance solved in [DPc].

References

- [AU77] A. V. Aho and J. D. Ullman. *Principles of Compiler Design*. Addison-Wesley, 1977.
- [BC94] N. Beldiceanu and E. Contjean. Introducing global constraints in CHIP. *Mathematical and Computer Modelling*, 12:97–123, 1994.
- [Bou99] Eric Bourreau. *Traitement de contraintes sur les graphes en programmation par contraintes*. Doctoral dissertation, Université Paris, Paris, France, 1999.
- [CB04] Hadrien Cambazard and Eric 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.
- [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. Doods, Y. Deville, and P. Dupont. Constrained path finding in biochemical networks. In *5èmes Journées Ouvertes Biologie Informatique Mathématiques*, 2004.
- [DDD05] G. Doods, Y. Deville, and P. Dupont. CP(Graph):introducing a graph computation domain in constraint programming. In *CP2005 Proceedings*, 2005.
- [DI00] Camil Demetrescu and Giuseppe F. Italiano. Fully dynamic transitive closure: Breaking through the $O(n^2)$ barrier. In *IEEE Symposium on Foundations of Computer Science*, pages 381–389, 2000.
- [DPc] A disjoint-paths problem solved with *Reachability*. Available at <http://www.info.ucl.ac.be/~luque/PADL06/DPcase.ps>.
- [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.

- [LT79] T. Lengauer and R. Tarjan. A fast algorithm for finding dominators in a flowgraph. *ACM Transactions on Programming Languages and Systems*, 1(1):121–141, July 1979.
- [Moz04] Mozart Consortium. The Mozart Programming System, version 1.3.0, 2004. Available at <http://www.mozart-oz.org/>.
- [Mül01] Tobias Müller. *Constraint Propagation in Mozart*. Doctoral dissertation, Universität des Saarlandes, Naturwissenschaftlich-Technische Fakultät I, Fachrichtung Informatik, Saarbrücken, Germany, 2001.
- [PGPR96] G. Pesant, M. Gendreau, J. Potvin, and J. Rousseau. An exact constraint logic programming algorithm for the travelling salesman with time windows, 1996.
- [QVD05a] Luis Quesada, Peter Van Roy, and Yves Deville. Reachability: a constrained path propagator implemented as a multi-agent system. In *CLEI2005 Proceedings*, 2005.
- [QVD05b] 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.
- [Rég94] Jean Charles Régin. A filtering algorithm for constraints of difference in cps. 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.
- [SGL97] Vugranam C. Sreedhar, Guang R. Gao, and Yong-Fong Lee. Incremental computation of dominator trees. *ACM Transactions on Programming Languages and Systems*, 19(2):239–252, March 1997.
- [SP78] Y. Shiloach and Y. Perl. Finding two disjoint paths between two pairs of vertices in a graph. *Journal of the ACM*, 1978.
- [SPMa] Spmn_22. Available at http://www.info.ucl.ac.be/luque/PADL06/test_22.ps.
- [SPMb] Spmn_22full. Available at http://www.info.ucl.ac.be/luque/PADL06/test_22full.ps.
- [SPMc] Spmn_52a. Available at http://www.info.ucl.ac.be/luque/PADL06/test_52.ps.
- [SPMd] Spmn_52full. Available at http://www.info.ucl.ac.be/luque/PADL06/test_52full.ps.
- [VH04] P. Van Roy and S. Haridi. *Concepts, Techniques, and Models of Computer Programming*. The MIT Press, 2004.