Solving Constrained Graph Problems using Reachability Constraints based on Transitive Closure and Dominators

Luis Quesada

Thesis submitted in partial fulfillment of the requirements for the Degree of Doctor in Applied Sciences

November 2, 2006

Faculté des Sciences Appliquées
Département d'Ingénierie Informatique
Université catholique de Louvain
Louvain-la-Neuve
Belgium

Solving Constrained Graph Problems
using Reachability Constraints
based on Transitive Closure and Dominators

by Luis Quesada

Computer Science and Engineering Department
Université catholique de Louvain
Place Sainte-Barbe, 2
1348 Louvain-la-Neuve
Belgium

To my wife, my parents and Fray Luis Amigó y Ferrer

Acknowledgments

The fulfillment of this thesis would not have been possible without the contribution of:

- Peter Van Roy, my supervisor, who gave me the opportunity to work as a research assistant at Université Catholique de Louvain (UCL) from July 2001 until April 2005 in the MISURE project. The goal of the project was the development of a system capable of automatically managing the mission of an uninhabited air vehicle, which was expected to perform a mission in hostile territory almost without the support of human operators. The constrained paths problems involved in this project motivated the research presented in this thesis.

- Yves Deville, who kindly accepted to co-supervise my thesis. His pragmatic aptitude and his long experience in constraint programming were essential in narrowing down the scope of the thesis and focusing on the fundamental issues. He also gave me the opportunity to work as a teaching assistant at UCL from October 2005 until August 2006.

- Camilo Rueda, who supervised my B.Sc. thesis and let me join his research group at Pontificia Universidad Javeriana in 1996. Even though Camilo was not directly involved in the work of my Ph.D. thesis, the research I carried out under his supervision initiated me into the field of constraint programming.

- Raphaël Collet, my genius colleague at UCL. The long discussions that we had were fundamental in helping me understand the concepts underlying my research.

- Kevin Glynn, my English colleague at UCL. Apart from helping me with my English, his critical comments let me present my ideas in a more clear way. I must also say that Kevin's advice was of great help when the end of my thesis was not so certain.

- Stefano Gualandi, my colleague in MISURE. The good results that we got in MISURE would not have been possible without Stefano. The fact that we were working on the same project provided us with lots of opportunities to interchange ideas.

- Fred Spiessens, a very open minded colleague. The discussions I had with Fred led us to a nice application of the techniques presented in this thesis to the field of Computer Security.

- Grégoire Dooms and Stephane Zampelli, who helped me in porting my *Oz* code to *Gecode(CP(Graph))*. I appreciate their patience in dealing with my naive questions.

- Eugene Ressler, who pointed out that our former cut nodes [QVD05b] were actually dominators.

- Martin Oellrich, who provided me with a Disjoint Paths instance that shows the advantage of using dominators in that kind of problems.

- Bruno Quoitin, who provided me with the Latex template used in this document.

- Peter Van Roy, Yves Deville, J-Didier Legat, Baudouin Le Charlier, Nicolas Beldiceanu, Christophe Guettier, Xavier Lorca, Jorge Perez, Raphaël Collet, Kevin Glynn, Christian Schulte, Denys Duchier and Boris Mejias, who proofred previous versions of this work.

- Valentin Mesaros and Gustavo Ospina,who helped me to get settled down when I arrived in Louvain la Neuve. Their friendship was very important in adapting to this new culture.

- All my friends in Belgium. Their friendship made my stay more livable and helped me feel less homesick.

- Lucia Garcia, my dear wife. Without her help, patience and support, the achievement of this goal would have been simply impossible.

- Luis Quesada and Rosario Ramirez, my dear parents. In fact, I should have started with them since they built the path to achieve this goal through their support and prayers.

I apologize to all those not mentioned that have helped me in one way or another. Even though I have unconsciously omitted their names, I am very grateful for the help they provided.

Luis Quesada
November 2, 2006

Contents

List of Figures

List of Tables

Concepts

This chapter defines a set of concepts that are used throughout the dissertation:

- A *graph* is a set of nodes connected by a set of edges. We say that a graph is *directed* if an edge from a node i to a node j is not equivalent to an edge from a node j to a node i. If the two edges are equivalent, we say that the graph is *undirected*.

- Given a graph $g = \langle N, E \rangle$, we have that $Nodes(g) = N$ and $Edges(g) = E$.

- Given an edge $e = \langle s, d \rangle$ of a directed graph g, s is the *source* of e, and d its *destination*. We also say that e is an outgoing edge of s and an incoming edge of d in g.

- $AddNodes(g, N)$ denotes the graph obtained from g after adding the nodes in N.

- $AddEdges(g, E)$ denotes the graph obtained from g after adding the edges in E.

- $RemoveNodes(g, N)$ denotes the graph obtained from g after removing the nodes in N.

- $RemoveEdges(g, E)$ denotes the graph obtained from g after removing the edges in E.

- $IncEdges(g, n)$ denotes the set of incoming edges of n in g.

- $OutEdges(g, n)$ denotes the set of outgoing edges of n in g.

- Given two graphs $g_1 = \langle N_1, E_1 \rangle$ and $g_2 = \langle N_2, E_2 \rangle$, g_1 is a (not necessarily proper) *subgraph* of g_2 ($g_1 \subseteq g_2$) if $N_1 \subseteq N_2$ and $E_1 \subseteq E_2$.

- We consider undirected graphs as a special class of directed graphs. We say that the directed graph g is undirected if

$$\forall \langle i, j \rangle \in Edges(g) : \langle j, i \rangle \in Edges(g) \tag{1}$$

Given a directed graph g_d, the *corresponding undirected graph* g_u of g_d is defined as follows:

$$\begin{array}{rcl} Nodes(g_u) & = & Nodes(g_d) \\ Edges(g_u) & = & \{\langle i,j\rangle \mid \langle i,j\rangle \in Edges(g_d) \vee \langle j,i\rangle \in Edges(g_d)\} \end{array} \tag{2}$$

- A *path* from node i to node j in the directed graph g is an element of the set $Paths(g, i, j)$, which is defined as follows:

$$p \in Paths(g,i,j) \leftrightarrow \begin{cases} p \text{ is a subgraph of } g \\ Nodes(p) = \{k_1, \ldots, k_n\} \wedge k_1 = i \wedge k_n = j \\ Edges(p) = \{\langle k_t, k_{t+1}\rangle \mid 1 \leq t < n\} \end{cases} \tag{3}$$

- A *simple path* is a path where each node is visited once.
- A *Hamiltonian path* of a graph g is a simple path that contains all the nodes of g.
- *Weight(p)* is the sum of the weights associated with the edges of the path p.
- Two nodes u and v are *connected* in g_d if the corresponding undirected graph g_u contains a path from u to v.
- A *tree* is a graph in which any two nodes are connected by exactly one path.
- A *spanning tree* of a graph is a selection of edges from the graph that form a tree spanning every node; that is, no node is not connected to the tree.
- The *out-tree* of a node i in a directed graph G is a tree rooted at i whose set of edges is a subset of the edges of G. This tree connects all the nodes in G that are reachable from i.
- The *in-tree* of a node i in a directed graph G is the out-tree of i in G^T (the graph obtained from G after reversing all the edges).
- Given a tree T,
  - $v \xrightarrow{*} w$ in T, if w is a descendant of v in T.
  - $v \xrightarrow{+} w$ in T, if v is different from w and w is a descendant of v in T.
- A *decision problem* is a problem with only two possible solutions: yes or no.

Chapter 1

Context and Contribution

Constrained graph problems have to do with finding graphs respecting a set of given constraints. One way of constraining the graph is by enforcing reachability between nodes. For instance, consider the following problem, which we call *The Bounded Transitive Closure Problem (BTC)*: given the directed graphs g_{min}, g_{max}, tcg_{min} and tcg_{max}, *BTC* is to find a directed graph g such that:

$$\begin{array}{c} g_{min} \subseteq g \subseteq g_{max} \\ \text{and} \\ tcg_{min} \subseteq TC(g) \subseteq tcg_{max} \end{array} \tag{1.1}$$

where $TC(g)$ is the transitive closure of g.

As we will show in section 2.3.1, *BTC* is a generalization of *The Disjoint Paths Problem* [GJ79], which makes *BTC* NP-complete. We find instances of constrained graph problems in Vehicle Routing [QVDC06, PGPR96, CL97, FLM99], Bioinformatics [DDD04] and Computer Security [SQV06].

We use constraint programming for tackling constrained graph problems. In this thesis, in particular, we state that the use of a global constraint defined on top of the notions of transitive closure and dominators play a key role in solving this kind of problems.

1.1 Constraint programming

A Constraint Satisfaction Problem (CSP) is a problem whose solution must satisfy a set of given constraints. A CSP is usually represented by a triple $\langle V, D, C\rangle$, where V is the set of variables involved in the problem, D is a function associating each variable with its domain, and C is the set of constraints that should be respected [Dec03, MS98].

Notice that *BTC* is a constraint satisfaction problem where V is the singleton set containing g, the domain of g is the set of graphs that are subgraph of g_{max} and supergraph of g_{min}, and C is the singleton set containing the constraint $tcg_{min} \subseteq TC(g) \subseteq tcg_{max}$.

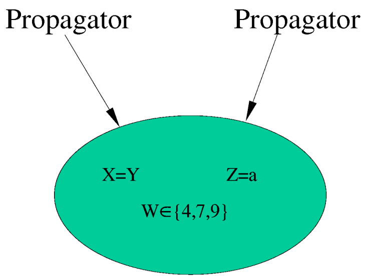


Figure 1.1: CP Model

In Constraint Programming, constraint satisfaction problems are solved by interleaving two processes: propagation and labeling. In Propagation, we are interested in filtering the domains of a set of finite domain variables according to the semantics of the constraints that have to be respected. In Labeling, we are interested in specifying which alternative should be selected when searching for the solution.

Propagators are processes that filter the domains of a set of finite domain variables according to the semantics of the constraint they implement. They share a common store (see Figure 1.1). This store contains the information that is currently known about the variables of the problem. This information can be of the form $X = Y$ (i.e., a variable X is equal to a variable Y), $X \in D$ (i.e., the variable X should be in domain D) or $X = a$ (i.e., the value of the variable X is a). The latter case is actually a subcase of the second case (i.e., $X = a$ is equivalent to $X \in \{a\}$). As soon as a propagator is able to infer new information from the store, it puts this new information in the store.

The search of the solution starts with the work of the propagators. Once the propagators reach a fix point, the labeling process starts. At this point, we may have three possible situations:

- all the variables are bound to a value. In this case the search stops since a solution has been obtained.
- there are some variables that are yet to be determined. In this case, we perform a labeling step. A labeling step consists of:
  - Selecting the variable whose alternatives are to be explored next.
  - Selecting the alternative (of the chosen variable) to explore next.

  Once this selection has been made, we add the corresponding constraint to the store. The addition of this constraint may activate some propagators, so we wait until propagators get stable to perform a new labeling iteration.
- there is at least one variable whose domain is empty. In this case, we go back to the previous labeling step, we impose that the solution to be found should

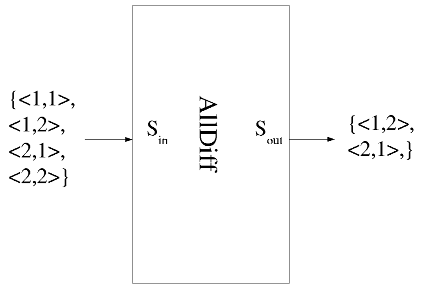


Figure 1.2: AllDiff as a function

be different to the alternative just considered and perform a new labeling iteration after waiting for the stabilization of the propagators.

1.1.1 Propagators as mathematical functions

We see a propagator as a function that maps a set of instances to a set of instances. The set of instances corresponds to the set of possible instances of the constraint the propagator implements. For instance, Figure 1.2 shows *AllDiff*, a propagator implementing an All-different Constraint on a set of finite domain variables [Rég94] applied on two finite domain variables. The initial domain of each variable is the set $\{1, 2\}$, so the corresponding instance set, which corresponds to the Cartesian product of the domains, is the set $\{\langle 1, 1\rangle, \langle 1, 2\rangle, \langle 2, 1\rangle, \langle 2, 2\rangle\}$. As $\langle 1, 1\rangle$ and $\langle 2, 2\rangle$ are not valid instances to the constraint, these values are dropped from the outgoing set.

1.1.2 Properties on propagators

We can also see propagators as theorem provers. Assuming that the propagator f is implementing a constraint c, if $f(S_1) = S_2$, the set of theorems that the propagator proves is $\{t : t \equiv \neg c(v) \land v \in (S_1 - S_2)\}$.

Soundness

A propagator is sound if it does not discard any valid value. In logic terms, this means that all the theorems it proves are logic consequences of the constraint it implements:

$$\forall_{S_1,S_2,v}.f(S_1) = S_2 \land v \in (S_1 - S_2) \rightarrow \neg c(v) \tag{1.2}$$

Completeness

A propagator is complete if it is able to discard all non-valid values. In logic terms, this means that it is able to prove all the theorems that follow from the constraint it implements:

$$\forall S_1, S_2, v : f(S_1) = S_2 \wedge v \in S_1 \wedge \neg c(v) \rightarrow v \notin S_2 \tag{1.3}$$

Monotonicity

A propagator is monotonic if the stronger the knowledge on which it works the stronger the knowledge it is able to infer:

$$\forall_{S_1,S_2}.S_1 \subseteq S_2 \rightarrow f(S_1) \subseteq f(S_2) \tag{1.4}$$

Weak-Completeness

A propagator is weakly-complete if it is always able to map a singleton set to the empty set when the value in the singleton set is not valid:

$$\forall_{S_1,S_2,v}.f(S_1) = S_2 \wedge S_1 = \{v\} \wedge \neg c(v) \rightarrow S_2 = \emptyset \tag{1.5}$$

The propertied just mentioned are a subset of the properties of propagators presented in [Mül01]. Notice that Completeness implies Weak-Completeness. However, Completeness does not imply Soundness. In fact, given a satisfiable constraint c, if we associate c with the propagator $f = \lambda S.\emptyset$, f would be complete because all the non valid values are discarded, but it would not be sound because valid values are discarded too. Usually, propagators are sound, monotonic and weakly-complete. Completeness is not a must because it may not be possible to achieve completeness in polynomial time.

We can now use the properties defined above to say in a precise way what we mean by "implementing a constraint" and "being the semantics of a propagator". We say that f implements c if and only if f is sound, monotonic and weakly-complete under c. We say that the semantics of f is c if and only if c is the strongest constraint under which f is sound and complete taking into account that a constraint c is stronger than a constraint c' if and only if c entails c' ($c \vdash c'$).

1.1.3 Composed propagators

We will see that *DomReachability*, the propagator we are introducing in this thesis, is a composed propagator, i.e., internally, it is composed of several propagators. In this section, we will define the semantics of a composed propagator in terms of the semantics of the propagators that compose it.

Let F be a set of propagators, and let us consider the following order relation among sets:

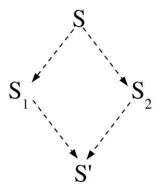


Figure 1.3: Confluence of propagators

$$\forall_{S,S'}.S \leq S' \leftrightarrow S \subseteq S' \tag{1.6}$$

A propagator f composed of a set of propagators F satisfies the following rule:

$$f(S) = max\{S' : S' \leq S \wedge \forall_{f \in F}.S' \in fix(f)\} \tag{1.7}$$

This means that $f(S)$ is the biggest common fix point that is less than S under the order relation defined by Equation 1.6. Certainly, as our order relation is not total, we need to rely on something else to ensure that this function is well defined. In this case, we can always ensure that there is a maximal element because our propagators are monotonic (mathematical) functions. We will elaborate more on this point in the next section.

1.1.4 Confluence of propagators

Theorem 1. *Monotonicity warranties confluence.*

Proof. Let us assume that f is a composed propagator. Suppose also that $f(S)$ is not defined because there are two maximal common fix points S_1 and S_2 such that $S_1 \not\leq S_2 \wedge S_2 \not\leq S_1$, which implies that there exists a value v such that $v \in S_1 \wedge v \notin S_2$.

As a consequence of monotonicity we have that:

$$S \leq S' \rightarrow f_1(f_2(...f_n(S))) \leq f_1(f_2(...f_n(S'))) \tag{1.8}$$

assuming that f_i, $1 \leq i \leq n$, is monotonic.

Then, if v does not belong to $S_2 = f_1(f_2(...f_n(S)))$, where $\forall_{1 \leq i \leq n}.f_i \in F$, then v would not belong to $f_1(f_2(...f_n(S_1)))$ either, which would imply that S_1 is not a fix point. Therefore, it cannot happen that for a given $f(S)$ there are two fix points S_1 and S_2 such that $S_1 \not\leq S_2 \wedge S_2 \not\leq S_1$. □

In fact, if we think of propagators as reduction rules and we consider the sets on which they work as the states of the reduction system, we can say that the reduction system is confluent. I.e., if from a state S we reach two states S_1 and S_2 such that

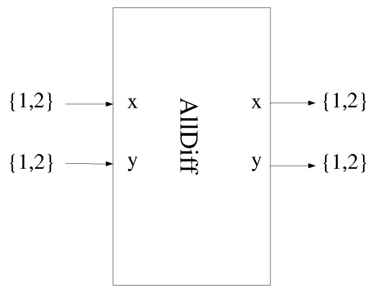


Figure 1.4: Approximation to the AllDiff propagator

$S_1 \not< S_2 \wedge S_2 \not< S_1$, then from S_1 and S_2 one should be able to reach a state S' from which no rule is applicable.

Notice that, due to the fact that we reason in terms of sets, we have the property that equivalent states (i.e., sets) have exactly the same representation. So, proving confluence in our model is simpler than in [AFM99] where two equivalent states may have different syntactic representation.

1.1.5 Approximation to the propagator

We have defined a propagator as a function that goes from a set of instances to a set of instances. However, propagators are defined in terms of a set of finite domain variables in practice. In fact, if x is one of the variables on which the propagator is defined, the propagator is supposed to discard from the domain of x those values that cannot be part of any solution.

Assuming that the propagator is defined in terms of n variables and that the variables share the same domain (D), what we do is to approximate the set $\mathscr{P}(D^n)$ to the set $SolsSet = \{Sols : Sols = \prod_{1 \leq i \leq n} D_i, D_i \subseteq D\}$. Notice that, the first set is indeed a superset of the second set. For instance, assuming that $D = \{1, 2\}$ and $n = 2$, we can observe that the set $\{< 1, 2 >, < 2, 1 >\}$ belongs to $\mathscr{P}(D^n)$ but not to $SolsSet$.

Certainly, with this approximation, we may lose completeness. In Figure 1.4, we present the same example of Figure 1.2, but under the approximation introduced. In fact, if we take into account that the tuple $< \{1, 2\}, \{1, 2\} >$ represents the set $\{1, 2\} \times \{1, 2\}$, we can observe that elements that were discarded before are no longer discarded. However, this is a fact of life. The important thing to notice is that we can be still sound, monotonic and weakly complete.

1.2 CP(Graph)

CP(Graph) introduces a new computation domain focused on graphs including a new type of variable, graph domain variables, as well as constraints over these variables and their propagators [DDD04, DDD05a, Doo06]. 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 also been proposed.

1.3 Levels of consistency

The domain of a finite domain variable X can be represented either by referring explicitly to the elements that compose it or by referring to the bounds of the domain. In the second case we say that $Min(X)$ and $Max(X)$ denote the lower bound and the upper bound of the domain respectively. That is to say that the domain of X is the set of elements greater than or equal to $Min(X)$ and less than or equal to $Max(X)$.

Given a constraint C defined in terms of a set of finite domain variables, we identify two basic levels of consistency:

- *domain consistency*: for every domain, we are interested in removing all the elements that do not participate in any solution.
- *bound consistency*: for every domain, we are interested in updating the bounds so that the lower bound of the domain corresponds to the minimal value accepted by the constraint and the upper bound to the maximal value accepted by the constraint.

While domain consistency is a stronger level of consistency, achieving that level of consistency is unrealistic in many cases. For instance, the size of the domain of a graph variable is exponential with respect to the number of nodes and edges. Aiming at explicitly keeping the domain would lead to use an exponential amount of space.

Even though bound consistency is considerably cheaper than domain consistency, achieving bound consistency may still be prohibited. For graph variables, being bound consistent means being able to discard from the upper bound all nodes and edges that do not participate in any solution, and to include in the lower bound all nodes and edges that participate in every solution. As we will see in the following chapter, computing this information leads to exponential computations in some cases.

1.4 Some applications of dominators

Given a flow graph, i.e., a directed graph g with source node s, node i dominates node j if all paths from s to j contain i [AU77, LT79, Geo05]. In this section we present some applications of this concept in areas different to the one we present in this thesis.

1.4.1 Detecting natural loops

Dominators have been mostly used in code optimization [AU77] where flow graphs are used for representing the execution of programs. One of the important tasks in code optimization is the optimization of loops since programs tend to spend most of their execution time in their inners loops.

In particular, one is interested in finding *natural loops*, i.e., loops having the two following properties:

- The loop must have a single entry point, called the header of the loop. This entry point dominates all the nodes in the loop.
- There must be at least one way to iterate the loop.

In order to detect natural loops, we look at the *back edges* of the flow graph. An edge $\langle j, i \rangle$ is a back edge if i dominates j. So, the natural loop involving $\langle j, i \rangle$ is composed of i plus all the nodes that reach j without passing through i.

The emphasis in natural loops is due to the fact that they offer a useful property that allows us to identify when a loop is included in another one. Given two natural loops l_1 and l_2, if l_1 and l_2 have different headers, they are either disjoint or one is included in the other one. The detection of self-contained loops avoids redundant code optimization.

1.4.2 Detecting domination of species in the ecosystem

In a given ecosystem, a specie depends on another one if the existence of the later warranties the existence of the former. These dependencies are the result of the relation between predators and preys. The relation between predator and prey can be represented with a directed graph. In this graph, an edge from a specie a to a specie b means that b is a predator of a.

Researchers are interested in determining the impact of removing a particular specie from the ecosystem taking into account that a specie a disappear when all the species of which a is predator disappear. In [AB04], the authors show an approach for addressing this question based on the use of dominators. A source node is added to the dependency graph. This source node is connected to all the nodes that do not have incoming nodes. We can see the resulting graph as a flow graph where the added node plays the role of source and species that do not depend on another species are directly connected to the source. Notice that all the species are reachable from the source. A specie will then disappear from the ecosystem if any of its dominators disappears.

1.4.3 Detecting equivalent faults in logic circuits

A fault is said to be detected by an input test vector, if when applying the vector to the circuit, different logic values can be observed, in at least one of the circuit's primary outputs, between the original circuit and the faulty circuit.

Detecting whether two faults are equivalent is important for reducing the number of tests to be performed in order to decide whether a circuit is faulty or not. Computing the complete set of fault equivalence classes in a circuit is a classic problem in digital circuit design. Two faults are functionally equivalent (or indistinguishable) if no input test vector can distinguish them at primary outputs. Functional fault equivalence is a relation that allows faults in a circuit to be collapsed into disjoint sets of equivalent fault classes [VCAS05].

One of the approaches for deciding whether two faults are equivalent consists in looking at the structure of the circuit. The structure of the circuit can be represented as a directed graph where the nodes correspond to the gates of the circuit and the edges are the lines connecting the output of a gate with the input of another one. In this context, dominators are used for focusing the examination on the dominator gates. A dominator gate of a line l is a gate through which all the paths from l to any primary output pass [AFPB01].

1.5 Contribution

1.5.1 Introduction of new NP-complete problems which are generalizations of the Disjoint-Paths problem

In this thesis we introduce two new NP-complete problems:

- Given the directed graphs g_{min}, g_{max}, tcg_{min} and tcg_{max}, The Bounded Transitive Closure Problem (*BTC*) is to find a directed graph g such that:

$$\begin{gathered} g_{min} \subseteq g \subseteq g_{max} \\ \text{and} \\ tcg_{min} \subseteq TC(g) \subseteq tcg_{max} \end{gathered} \tag{1.9}$$

  where TC(g) is the transitive closure of g.

- Given a directed graph g, a source node src, a destination node dst, a set of mandatory nodes $mandnodes$, and set of couples of nodes $order$, the Ordered Simple Path with Mandatory Nodes Problem *OSPMN* is to find a path in g from src to dst, going through $mandnodes$ following $order$ and visiting each node only once.

Both *BTC* and *OSPMN* are generalization of the Disjoint Paths problem. The k-Disjoint-paths problem consist in finding k pairwise disjoint paths between k pairs of nodes $\langle s_1, d_1 \rangle$, $\langle s_2, d_2 \rangle$, …, $\langle s_k, d_k \rangle$. Both the node-disjoint version and the edge-disjoint version are NP-complete even for $k = 2$ [SP78].

1.5.2 Introduction of new global constraints on top of transitive closure and domination

We introduce two global constraints on top of the notions of transitive closure and dominations, which are suitable for tackling the aforementioned problems as well as several other problems presented in chapter 2:

- $Reachability(g, tcg)$, which holds if tcg is the transitive closure of the directed graph g.
- $Domination(fg, edg)$, which holds if edg is the extended dominator graph (i.e., the graph stating the dominance relation among nodes and edge) of the flow graph fg.
- $DomReachability(fg, edg, tcg)$, which holds if edg is the extended dominator graph of the flow graph fg, and tcg is the transitive closure of fg.

1.5.3 Introduction of dominators for solving constrained graph problems

As said before, dominators have been mostly used in code optimization for detecting inner loops. In this thesis, we use dominators for detecting nodes and edges common to a set of paths in a flow graph. This information is important when discarding paths that violate the set of given transitive closure constraints.

Let us consider the case presented in Figure 1.5 where we are interested in finding a simple path (i.e., a path not visiting a node twice) from node 1 to node 22 containing nodes 4, 7, 10, 16, 18 and 21 (which we call mandatory nodes). Notice that choosing the edge $\langle 1, 5 \rangle$ implies that node 5 is visited twice since all the paths from 5 to 22 containing the mandatory nodes include edge $\langle 12, 5 \rangle$, so we can discard that branch and try the one involving edge $\langle 1, 15 \rangle$.

In order to infer that edge $\langle 1, 5 \rangle$ is in all the paths from 5 to 22 containing the mandatory nodes, we need to take into account that:

- 5 reaches all the mandatory nodes, so the nodes and edges reached from the mandatory nodes are also reached from 5.

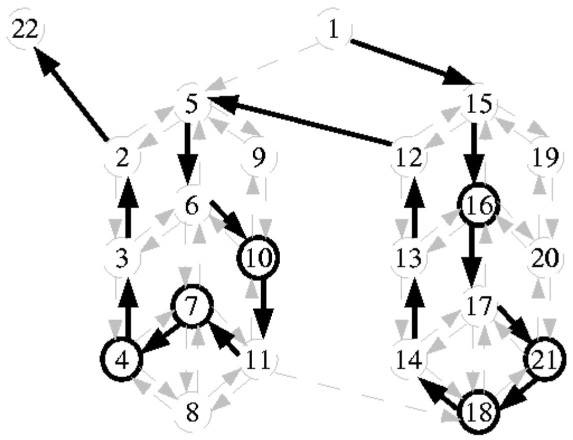


Figure 1.5: A simple path from 1 to 22 containing 4, 7, 10, 16, 18 and 21

- node 22 is dominated by edge $\langle 12, 5\rangle$ with respect to any of the mandatory nodes at the right hand of the graph (i.e., with respect to nodes 16,18 and 21).

This is where the information provided by the dominators becomes fundamental since as soon as we know that a node i reaches a node j we can immediately infer that i reaches all the nodes that dominate j from i. This inference may avoid useless exploration as it is the case in our example.

1.5.4 Pruning algorithms and edge-dominator discovering

The pruning rules of the global constraints introduced are systematically derived from the properties of the constraints and their implementation is done by taking into account state of the art algorithms for computing dominators and transitive closure.

We also introduce an efficient approach for computing edges that participate in all the paths connecting a pair of nodes (edge-dominators) by introducing the notion of extended dominator graph. In the extended dominator graph we map edges to nodes and compute the dominance relation on the resulting graph. Thanks to the fact that the dominance relation can be represented with a dominator tree [AU77], the edge-dominator are computed at the same complexity.

1.5.5 Evaluation of the approaches introduced in realistic scenarios

We have implemented the global constraints introduced in both Gecode [SLT06] and Mozart [Moz04], and tested the performance of the implementation with realistic instances from vehicle routing problems and computer security problems. The Gecode implementation takes advantage of the implementation of CP(Graph) that has been already incorporated into Gecode [DZDD06].

1.6 Structure of the thesis

- **Chapter 2.** In this chapter we first present the different concepts needed for the definition of the global constraints we are introducing. Then we introduce the constraints and describe a set of problems that can me modelled in terms of these constraints. Among the problems described, we have the *The Bounded Transitive Closure Problem* and *The Ordered Simple Path with Mandatory Nodes Problem*: two new NP-complete problems that are being introduced in this thesis. The corresponding NP-complete proofs are also presented in this chapter.

- **Chapter 3.** As *DomReachability* is the conjuntion of *Reachability* and *Domination*, the implementation of *DomReachability* covers the implementation of *Reachability* and *Domination*.

  In this chapter we introduce the algorithms involved in the implementation of *DomReachability*. We start by explaining how the pruning rules are systematically derived from the properties of the constraints. Then we revisit each property and derive the corresponding pruning rules. During this process, we emphasize the pruning gained by the activation of each rule.

  The implementation of *DomReachability* implies maintaining the transitive closure graph and the extended dominator graph of the bounds of the flow graph. So, in this chapter we also study some approaches for maintaining this information.

- **Chapter 4.** *Gecode* [SLT06] is a C++ library that provides an environment for developing constraint-based systems and applications. *Gecode* allows the construction of new variable domains including propagators as implementations of constraints and branchings, and search engines.

  In this chapter we make a summary of the most relevant concepts in *Gecode*. Then, we show how propagators are implemented in Gecode by explaining the implementation of one of the propagators provided by *Gecode*. After explaining how to deal with *CP(Graph)* (a new computation domain that has been added to Gecode), we present the implementation of *DomReachability*.

  In this chapter we also show the implementation of the labeling strategy we have designed to deal with *OSPMN* instances.

- **Chapter 5.** In this chapter we explain how we can implement *DomReachability* using a message passing approach on top of the multi-paradigm programming language Oz [Moz04]. As shown in this chapter, the use of a concurrent language like Oz for implementing global constraints involves the implementation of processes that are non-deterministic in general. This makes Declarative Concurrency not suitable for this need. By using the methodology introduced in [VH04], we show that the definition of the behavior of the agents involved in the implementation of global constraints, and the non-determinism in the communication of these agents are two orthogonal concerns. This separation let us define the behavior of each agent in a declarative way.

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

We also present a cheap way of discovering dominators based on FS pruning, and introduce an approach for implementing Batch propagation using message passing, which plays an important role in the reduction of the time of execution thanks to the minimization of the number of activations of expensive propagators [QVD05a].

- **Chapter 6.** In this chapter we present a set of experiments that show that *DomReachability* is suitable for solving the Simple path with mandatory nodes problem. In the experiments 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.

  – 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.

  In this chapter we also show that *DomReachability* is suitable 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 [QVDC06].

- **Chapter 7.** In software security, the execution of some actions is controlled (allowed or disallowed), in an attempt to restrict their (direct or indirect) effects. Allowed actions are called *permissions*. Different parts of a program (subjects) can have different permissions. The ability of a subject to directly or indirectly induce an effect is called its *authority*.

  The propagation of authority can often be expressed in sufficient detail by reachability in a directed graph. The nodes in the graph each represent a subject and the edges represent permissions. The reflexive and transitive closure of the permission graph then represents an upper bound for reachable authority.

  In this chapter, we show that graph reachability constraints have useful applications in safety analysis and enforcement. We do this by modelling safety analysis and enforcement in terms of The Bounded Transitive Closure Problem, which can be modelled in terms of DomReachability. In order to model a broader set of security problems, we extend The Bounded Transitive Closure Problem with the notion of cardinality [SQV06].

- **Chapter 8** In this chapter we make some concluding remarks and suggest some directions in which the work presented in this thesis can be extended.

Part of the work presented in this thesis appears in previously published conference and workshop proceedings. The list of related publications is shown hereafter:

- F. Spiessens, L. Quesada, and P. Van Roy. Confinement analysis with graph reachabilty constraints. In International Workshop on Constraints in Software Testing, Verification and Analysis (CSTVA06), at the 12th International Conference on Principles and Practice of Constraint Programming (CP2006), 2006.

- Luis Quesada, Peter Van Roy, Yves Deville, and Raphaël Collet. Using dominators for solving constrained path problems. In PADL 2006 Proceedings, Lecture Notes in Computer Science. Springer, 2006.

- Luis Quesada, Peter Van Roy, and Yves Deville. Speeding up constrained path solvers with a reachability propagator. In Colloquium on Implementation of Constraint and Logic Programming Systems(CICLOPS 2005), at the 11th International Conference on Principles and Practice of Constraint Programming (CP2005), 2005.

- Luis Quesada, Peter Van Roy, and Yves Deville. Reachability: a constrained path propagator implemented as a multi-agent system. In CLEI2005 Proceedings, 2005.

- 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.

- L. Quesada and P. Van Roy. A concurrent constraint programming approach for trajectory determination of autonomous vehicles (abstract). In CP 2002 Proceedings, Lecture Notes in Computer Science. Springer, 2002.

Chapter 2

Global Constraints based on Transitive Closure and Domination

In this chapter we present three global constraints whose semantics are defined in terms of the notions of transitive closure and domination in directed graphs:

- The *Reachability* constraint, which has two arguments: a directed graph and its transitive closure.
- The *Domination* constraint, which has two arguments: a flow graph, i.e., a directed graph with a source node, and the dominance relation graph on nodes and edges of the flow graph.
- The *DomReachability* constraint, which has three arguments: (1) a flow graph, (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.

After presenting the semantics of *Reachability*, *Domination* and *DomReachability*, we show a set of problems that can be modeled in terms of them. As all those problems are NP-complete, the modeling of those problems actually represents a proof that achieving bound consistency for any of the constraints is NP-complete.

In this chapter we are also introducing two NP-complete problems:

- The Bounded Transitive Closure Problem, where we are interested in finding a graph respecting respecting some boundaries on itself and its transitive closure.

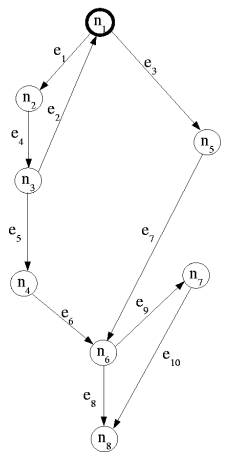


Figure 2.1: An example of a flow graph

- The Ordered Simple Path with Mandatory Nodes Problem, where we are interested in finding a simple path containing a set of nodes in given order.

2.1 Transitive closure and Domination

2.1.1 Reachability

A node is reachable from another one if there is a path from the former to the latter. This relation between nodes of g is represented with a graph which is called *the transitive closure of g* [CLR90].

Definition 1. *$TC(g)$ is the transitive closure of g, i.e.,*

$$\langle i, j\rangle \in Edges(TC(g)) \leftrightarrow \exists p : p \in Paths(g, i, j) \tag{2.1}$$

2.1.2 Flow graph

A flow graph is a directed graph with a source node. Figure 2.1 shows an example of a flow graph. The node that is playing the role of source in this example is node n_1. Flows graphs have been mostly used in Compilers theory to represent the execution of programs where nodes represent basic blocks of the program and edges changes in the control flow [AU77].

We will represent a flow graph fg as a triple $\langle N, E, s\rangle$ where N is the set of nodes, E is the set of edges and s is the source. We will occasionally drop s when no reference to the source occurs.

2.1.3 Dominance relation

Definition 2. *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]:*

$$i \in Dominators(fg, j) \leftrightarrow i \neq j \wedge \forall p \in Paths(fg, s, j) : i \in Nodes(p) \quad (2.2)$$

Let us consider some interesting properties on dominators:

Theorem 2. *The dominance relation is transitive, i.e., if i dominates j and j dominates k, then i dominates k.*

Proof. If k is not reachable from the source, this property is trivially true since every node but k dominates k. Now, let us assume that k is reachable from the source. Every path p from the source to k contains j because j is a dominator of k. The path p also contains i because this is a dominator of j, therefore i is a dominator of k. □

Theorem 2 only applies to reachable nodes though. Notice that if both i and j are unreachable, Theorem 2 would imply that i dominates i, which is by definition not possible.

Theorem 3. *If i_1 and i_2 are both dominators of j, i_1 and i_2 appear in the same order in all the paths from the source to j* [1].

Proof. Suppose that i_1 and i_2 are both dominators of j. Suppose also the that there is a path p_1 where i_1 appears first and a path p_2 where i_2 appears first. Then, a path from the source to j that does not contain i_2 can be obtained as follows:

- go from the source to i_1 using p_1
- go from i_1 to j using p_2

which contradicts the statement that i_2 is a dominator. □

Corollary 1. *All the dominators of j appear in the same order in every path.*

Proof. It follows from the fact that Theorem 3 holds for every pair of dominators of j. □

Note that the nodes unreachable from s are dominated by all the other nodes. However, as a consequence of Theorem 3, the nodes reachable from s always have an *immediate* dominator.

[1]A path in a directed graph denotes a sequence of nodes. We say that i appears first than j in a path if i is first in the corresponding sequence than j

Definition 3. *The immediate dominator of j is the closest dominator of j:*

$$i = ImDominator(fg, j) \leftrightarrow \left\{ \begin{array}{l} i \in Dominators(fg, j) \\ \neg \exists k \in Nodes(fg) : i \in Dominators(fg, k) \wedge k \in Dominators(fg, j) \end{array} \right. \tag{2.3}$$

The notion of *immediate* dominator allows to represent the whole dominance relation as a tree, where the parent of a node is its immediate dominator. Notice, however, that unreachable nodes are not taken into account since $ImDominator(fg, j)$ is not defined if j is not reached from s. In what follows $DomTree(fg)$ will denote the dominator tree of fg, and $DomGraph(fg)$ a graph representing the whole dominance relation of fg, i.e.,$\langle i, j\rangle \in Edges(DomGraph(fg))$ if and only if i dominates j in fg.

Theorem 4. *Given a flow graph $fg = \langle N, E\rangle$ and a node $j \in N$, if all $i \in N \setminus \{j\}$ dominates j, and j is reachable from the source in fg, then fg has a unique Hamiltonian path from the source to j.*

Proof. The fact that there is a Hamiltonian path follows from the definition of domination (Definition 2) since all the nodes but j are dominators and the source reaches j.

Now, suppose there are two different Hamiltonian paths p_1 and p_2 from the source to j. As p_1 and p_2 are different, the orders in which the nodes appear are different, which contradicts Corollary 1. □

Definition 4. *A back edge is an edge whose destination dominates its source [AU77].*

Back edges are used to detect loops in a flow graph. If $\langle i, j\rangle$ is a back edge and both i and j are reachable from the source, then there is a at least a path from j to i (since j dominates i), which forms a loop when concatenated with the edge $\langle i, j\rangle$.

Theorem 5. *Given a flow graph $fg = \langle N, E\rangle$* **with no back edge**, *and a node $j \in N$, if all $i \in N \setminus \{j\}$ dominates j, and j is reachable from the source in fg, then fg is a Hamiltonian path from the source to j.*

Proof. It is enough to prove that, every node in $N \setminus \{j\}$ has at most one outgoing edge. Let us assume the opposite, i.e., that there is a node i with two outgoing edges $e_1 = \langle i, k_1\rangle$ and $e_2 = \langle i, k_2\rangle$. As all the nodes in $N \setminus \{j\}$ dominates j, there is a path where k_1 appears first than k_2 (the one using e_1), and there is another one where k_2 appears first than k_1 (the one using edge e_2) since neither e_1 nor e_2 are back edges. However, this violates Theorem 3. □

The extended dominator graph

We now introduce the notion of *extended dominator graph* which allows us to take advantage of dominator algoritms for computing edges that are common to the set of paths connecting a node with the source of the flow graph.

Definition 5. *The extended graph of fg, $Ext(fg)$, is obtained by replacing the edges by new nodes, and connecting the new nodes accordingly. This graph can be formally defined as follows:*

$$\langle N', E', s' \rangle = Ext(\langle N, E, s \rangle) \leftrightarrow \begin{cases} s' = s \\ N' = N \cup E \\ e = \langle i, j \rangle \in E \leftrightarrow \langle i, e \rangle \in E' \wedge \langle e, j \rangle \in E' \end{cases} \tag{2.4}$$

Definition 6. *The extended dominator graph of fg is the dominator graph of its extended graph.*

Figures 2.2, 2.3 and 2.4 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 that correspond 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.

Theorem 6. *Given two node dominators i and j, if*

$$\langle i, j \rangle \in Edges(DomTree(Ext(fg))) \tag{2.5}$$

then there are at least two different paths from i to j in the flow graph.

Proof. The presence of i and j in $DomTree(Ext(fg))$ ensures that there is at least one path p_1 from i to j. As the immediate dominator of j is a node dominator, there is no edge appearing in all the paths from i to j. This means that if e is an edge of p_1, there is another path p_2 not involving e. □

Set dominators

We will now extend the notion of domination to a set of nodes. In a sense, we can say that this notion of domination is not as strong as the one already introduced since all the nodes of the set must be removed in order to disconnect the dominated node from the source. Formally we have that:

Definition 7. *A set of nodes s is a set dominator of a node j if every path from the source to j has a node in s and the removal of any proper subset of s does not make j unreachable from the source.*

$$\begin{aligned} & domset \in SetDominators(fg, j) \leftrightarrow \\ & \qquad \begin{cases} \forall p \in Paths(fg, source, j). \exists i \in domset : i \in Nodes(p) \\ \forall s \subset domset. \exists p \in Paths(fg, source, j). Nodes(p) \cap s = \emptyset \end{cases} \end{aligned} \tag{2.6}$$

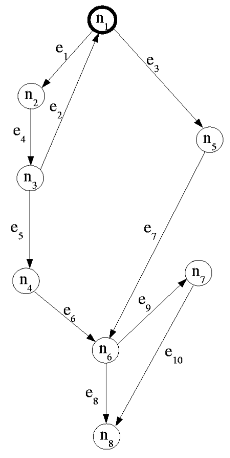


Figure 2.2: Flow graph

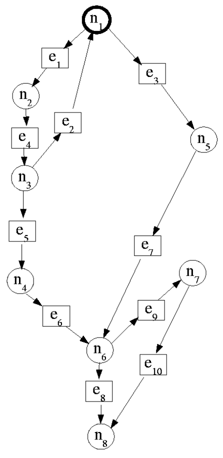


Figure 2.3: Extended flow graph

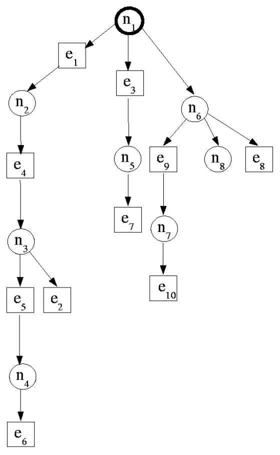


Figure 2.4: Extended dominator tree

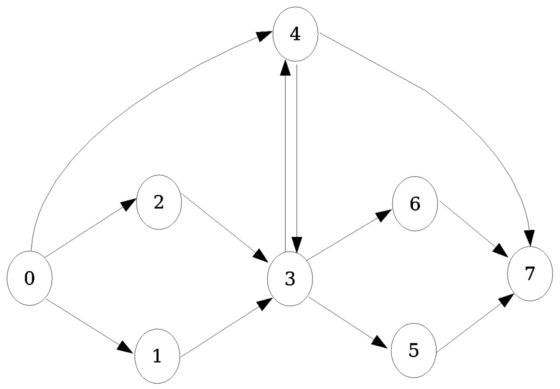


Figure 2.5: Sets $\{1, 2, 4\}$,$\{3, 4\}$ and $\{4, 5, 6\}$ are set dominators of node 7

A set dominator s_1 is maximal in the sense that there is no set $s_2 \supset s_1$ that is also a set dominator. This is why, if j is not reachable from the source, the only set dominator of j is $\emptyset$. If j is reachable from the source and i is a dominator of j, then $\{i\}$ is a set dominator of j.

Let us now look at the relation between two sets s_1 and s_2 if both sets are set dominators of a node i. If either s_1 or s_2 is a singleton set, then s_1 and s_2 are disjoint since a set dominator cannot be a superset of another set dominator. However, if s_1 and s_2 are not singleton, then s_1 and s_2 may share nodes.

Let us consider the case of figure 2.5 where node 0 plays the role of the source. In this case we observe that the sets $\{1, 2, 4\}$,$\{3, 4\}$ and $\{4, 5, 6\}$ are all set dominators of node 7. Indeed, removing nodes 1, 2 and 4 disconnect node 7 from the source, but removing any sub set of $\{1, 2, 4\}$ does not make 7 unreachable from the source. This is also the case for sets $\{3, 4\}$ and $\{4, 5, 6\}$.

2.2 Global constraints on Transitive closure and Domination

2.2.1 The *Reachability* constraint

Definition 8. *The* Reachability *constraint has two arguments: a directed graph and its transitive closure.*

$$Reachability(g, tcg) \tag{2.7}$$

This apparently simple constraint is actually pretty expressive. As we will show in section 2.3.1, we are able to model NP-complete problems in terms of this constraint only.

2.2.2 The *Domination* constraint

Definition 9. *The* Domination *constraint has two arguments: a flow graph, i.e., a directed graph with a source node and the extended dominator graph of the flow graph:*

$$Domination(fg, edg) \tag{2.8}$$

In section 2.3.2 we will see that *Domination* is enough to model The Simple Path with Mandatory Nodes Problem and its applications.

2.2.3 The *DomReachability* constraint

Definition 10. *The* DomReachability *constraint is a constraint on three graphs:*

$$DomReachability(fg, edg, tcg) \tag{2.9}$$

where

- fg *is a flow graph whose set of nodes is a subset of* N*;*
- edg *is the extended dominator graph of* fg*; and*
- tcg *is the transitive closure of* fg*, i.e,*

$$tcg = TC(fg) \tag{2.10}$$

The fact that we can already model NP-complete problems with *Reachability* makes the modelings of NP-complete problems with *DomReachability* not surprising since *DomReachability* is actually an extension of *Reachability*. In section 2.3.2 we will show how we can take advantage of the extended dominator graph in order to express relations on nodes and edges.

2.2.4 Properties of Reachability, Domination and DomReachability

The definitions of *Reachability*, *Domination* and *DomReachability* imply the following properties which are crucial for the pruning that they perform. These properties define relations between the graphs they have as arguments. These relations can then be used for pruning, as we show in section 3.2.

Properties between the graph and its transitive closure

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

$$\forall \langle i, j\rangle \in Edges(g) : \langle i, j\rangle \in Edges(tcg) \tag{2.11}$$

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

$$\forall i, j, k \in N : \langle i, j\rangle \in Edges(tcg) \wedge \langle j, k\rangle \in Edges(tcg) \rightarrow \langle i, k\rangle \in Edges(tcg) \tag{2.12}$$

3. If i does not reach j, $\langle i, j\rangle$ is not an edge edg.

$$\forall i, j \in N : \langle i, j\rangle \notin Edges(TC(g)) \rightarrow \langle i, j\rangle \notin tcg \tag{2.13}$$

Properties between the flow graph and its extended dominator graph

1. If i dominates j, where i/j is either a node or an edge, then , $\langle i, j\rangle$ is an edge of edg.

$$\forall i, j \in N \cup (N \times N) : \langle i, j\rangle \in Edges(DomGraph(Ext(fg))) \rightarrow \langle i, j\rangle \in edg \tag{2.14}$$

2. If j is directly reachable from the source, i.e., $\langle s, j\rangle$ is an edge of fg (where $s = Source(fg)$), then s is the only dominator of j.

$$\forall i, j \in N, i \neq s : \langle s, j\rangle \in Edges(fg) \rightarrow \langle i, j\rangle \notin Edges(edg) \tag{2.15}$$

Property among the flow graph, its extended dominator graph and its transitive closure

If j is reachable from s and i dominates j in fg, then i is reachable from s and j is reachable from i:

$$\begin{aligned}\forall i, j \in N : \langle s, j\rangle \in Edges(tcg) \wedge \langle i, j\rangle \in Edges(edg) \rightarrow \\ \langle s, i\rangle \in Edges(tcg) \wedge \langle i, j\rangle \in Edges(tcg)\end{aligned} \tag{2.16}$$

2.3 Problems modeled with Reachability and DomReachability

2.3.1 The bounded transitive closure problem and its variants

In this section we will present a set of problems that are defined on top of the transitive closure relation. The main contribution of this section is the proof that *The Bounded Transitive Closure Problem (BTC)* is NP complete. We will also present some variants of *BTC* and suggest CP approaches to solve them based on the global constraints we are presenting in this thesis.

The bounded transitive closure problem

Definition 11. *Given the directed graphs g_{min}, g_{max}, tcg_{min} and tcg_{max}, The Bounded Transitive Closure Problem (*BTC*) is to find a directed graph g such that:*

$$\begin{array}{c} g_{min} \subseteq g \subseteq g_{max} \\ \textit{and} \\ tcg_{min} \subseteq TC(g) \subseteq tcg_{max} \end{array} \tag{2.17}$$

Theorem 7. BTC *is NP complete.*

Proof. We will show that *BTC* is NP complete by reducing The Disjoint Path Problem (*DP*) to *BTC*. The k-Disjoint-paths problem consists in finding k pairwise disjoint paths between k pairs of nodes $\langle s_1, d_1\rangle$, $\langle s_2, d_2\rangle$, $\ldots$, $\langle s_k, d_k\rangle$. Both the node-disjoint version and the edge-disjoint version are NP-complete even for $k = 2$ [SP78]. So if we express the problem of (node-disjointedly) connecting $\langle a, b\rangle$ and $\langle c, d\rangle$ in g_{max} in terms of *BTC*, then we prove that *BTC* is NP complete.

Let $p_{\langle a,b\rangle}$ and $p_{\langle c,d\rangle}$ be the paths connecting $\langle a, b\rangle$ and $\langle c, d\rangle$ respectively. The first thing to notice is that, if $p_{\langle a,b\rangle}$ and $p_{\langle c,d\rangle}$ share a node k, the graph composed of $p_{\langle a,b\rangle}$ and $p_{\langle c,d\rangle}$ would be a graph where a reaches d and c reaches b. Notice that in order to reach d from a we just need to go from a to k using $p_{\langle a,b\rangle}$, and then from k to d using $p_{\langle c,d\rangle}$.

If we want to avoid that $p_{\langle a,b\rangle}$ and $p_{\langle c,d\rangle}$ share nodes, we need to impose that a does not reach d and c does not reach b. Then, the problem of finding two disjoint paths connecting $\langle a, b\rangle$ and $\langle c, d\rangle$ in g_{max} can be reduced to the following *BTC* [2]:

$$\begin{array}{rcl} g_{min} & = & \emptyset \\ g_{max} & = & \text{the given graph} \\ tcg_{min} & = & \{\langle a, b\rangle, \langle c, d\rangle\} \\ tcg_{max} & = & TC(g_{max}) - \{\langle a, d\rangle, \langle c, b\rangle\} \end{array} \tag{2.18}$$

If g is a solution of the *BTC* the disjoint paths can be obtained by running *DFS* rooted at a and c respectively. Notice that any path found by *DFS* would be correct since all paths from a to b are pairwise disjoint with all paths from b to d. □

[2]In the following equation, we will represent a graph as a set of edges.

Notice that $Reachability(g, tcg)$ is all what we need to model BTC. g_{min}, g_{max}, tcg_{min} and tcg_{max} correspond to the lower and upper bounds of g and tcg respectively.

The minimum bounded transitive closure

Definition 12. *Given the directed graphs g_{min}, g_{max}, tcg_{min} and tcg_{max}, and an integer k, The Minimum Bounded Transitive Closure Problem (*MinBTC*) is to find a directed graph g, whose number of edges is at most k, such that:*

$$\begin{array}{c} g_{min} \subseteq g \subseteq g_{max} \\ \textit{and} \\ tcg_{min} \subseteq TC(g) \subseteq tcg_{max} \end{array} \tag{2.19}$$

As *MinBTC* is an extension of *BTC*, the fact that *MinBTC* is NP complete is not surprising. Indeed, *DP* can be reduced to *MinBTC* by ignoring k.

It is important to observe that *MinBTC* is also a generalization of *The Minimum Equivalent Digraph Problem (MED)* [GJ79]. In *MED* we are interested in finding a subgraph g_2 of a graph g_1 such that g_1 and g_2 have the same transitive closure, and g_2 has at most k edges. Notice that we can trivially reduce *MED* to *MinBTC* by stating that both tcg_{min} and tcg_{max} are equal to $TC(g_1)$, g_{min} is the empty graph, and $g_{max} = g_1$.

Our way of modeling *MinBTC* is by using $Reachability(g, tcg)$ in conjunction with $Size(g, I)$. The *Size* constraint forces g to have i edges. The model is the following:

$$Reachability(g, tcg) \wedge Size(g, i) \wedge i \leq k \tag{2.20}$$

The maximum bounded transitive closure

Definition 13. *Given the directed graphs g_{min}, g_{max}, tcg_{min} and tcg_{max}, and an integer k, The Maximum Bounded Transitive Closure Problem (*MaxBTC*) is to find a directed graph g, whose number of edges is at least k, such that:*

$$\begin{array}{c} g_{min} \subseteq g \subseteq g_{max} \\ \textit{and} \\ tcg_{min} \subseteq TC(g) \subseteq tcg_{max} \end{array} \tag{2.21}$$

MaxBTC is also a generalization *BTC* so it is also NP complete. The approach for modeling *MaxBTC* is basically the same approach used for *MinBTC*:

$$Reachability(g, tcg) \wedge Size(g, i) \wedge i \geq k \tag{2.22}$$

In section 7 will elaborate on a serie of problems that arrives in Security that can be represented as *MaxBTC* problems.

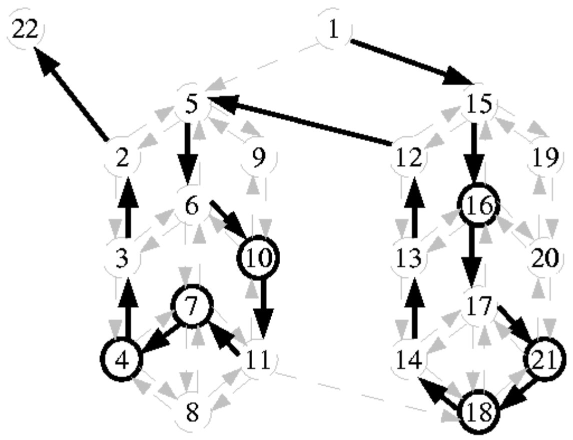


Figure 2.6: A simple path from 1 to 22 containing 4, 7, 10, 16, 18 and 21

2.3.2 The simple path with mandatory nodes problem

Definition 14. *The Simple path with mandatory nodes problem (*SPMN*) is to find a simple path in a directed graph containing a set of mandatory nodes[Sel02, CB04]. A simple path is a path where each node is visited only 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.*

For instance, consider the case shown in Figure 2.6. We want to find a path from from node 1 to nodes 22 containing nodes 4, 7, 10, 16, 18 and 21. Notice that using edge $\langle 1, 5\rangle$ implies that node 5 is visited twice since all paths from 5 to 22, that contain the mandatory nodes, contain edge $\langle 12, 5\rangle$.

Theorem 8. SPMN *is NP complete.*

Proof. Hamiltonian Path (finding a simple path between two nodes containing all the nodes of the graph [GJ79, CLR90]) can be trivially reduced to *SPMN* by defining the set of mandatory nodes as $Nodes(g) \setminus \{src, dst\}$. □

We can model *SPMN* in terms of *DomReachability* by imposing that the source reaches the destination and that all the mandatory nodes dominates the destination.

Formally, if g is the graph where the simple path is to be found, src and dst are the source and the destination respectively, and mn is the set of mandatory nodes, the following constraints are enough to restrict fg to a graph where all the paths from src to dst contain the mandatory nodes:

$$SPMN(g, src, dst, mn, fg) \leftrightarrow \begin{cases} \mathit{Subgraph}(fg, g) \\ \mathit{DomReachability}(fg, edg, tcg) \\ \langle src, dst \rangle \in Edges(tcg) \\ \forall i \in mn : \langle i, dst \rangle \in Edges(edg) \end{cases} \tag{2.23}$$

Once we have found fg, finding the simple path is straightforward. A linear exploration of the graph will suffice to build the path since all the paths from the source to the destination will contain the mandatory nodes.

Notice that as a consequence of Theorem 4, if $Nodes(g) = mn \cup \{src, dst\}$ (i.e., if we are to find a Hamiltonian path), fg contains only one path from src to dst. Notice also that, thanks to Theorem 5, the graph obtained from fg after removing the back edges is a Hamiltonian path.

We can also take advantage of the fact that unreachable nodes are dominated by any node in order to enforce reachability between nodes through the impossion of dominance constraints. For instance, if we want to inforce that a node i is reachable from the source we can do so by stating that a node k does not dominate i.

Assuming that $k \notin Nodes(g)$, the above means that *SPMN* can be expressed in terms of *Domination* as follows:

$$SPMN(g, src, dst, mn, fg) \leftrightarrow \begin{cases} \mathit{Subgraph}(fg, g) \\ \mathit{Domination}(fg, edg) \\ \forall i \in mn : \langle i, dst \rangle \in Edges(edg) \\ \langle k, dst \rangle \notin Edges(edg) \end{cases} \tag{2.24}$$

Another way of modeling this problem is by using *Reachability* in conjunction with the *Path* constraint ($Path(p, s, d)$), which holds if p is a simple path from s to d [DDD05b].

$$SPMN(g, src, dst, mn, fg) \leftrightarrow \begin{cases} \mathit{Subgraph}(fg, g) \\ Path(fg, src, dst) \\ \mathit{Reachability}(fg, tcg) \\ \langle src, dst \rangle \in Edges(tcg) \\ \forall i \in mn : i \in Nodes(fg) \end{cases} \tag{2.25}$$

As the nodes in mn are included in the set of nodes of fg, fg is a simple path containing the mandatory nodes.

In the previous model, the use of *Reachability* is redundant. In fact, the application of *Path* ensures that the graph assigned to fg is a simple path containing the mandatory nodes. However, as we will see in Chapter 6, the information provided by *Reachability* is used to guide the search.

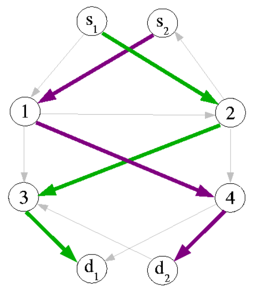


Figure 2.7: Finding two disjoint paths

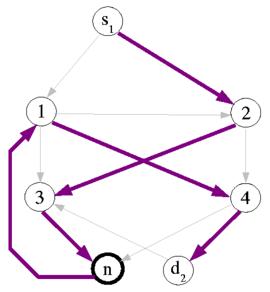


Figure 2.8: Finding a simple path passing through n

2.3.3 The ordered simple path with mandatory nodes problem

Definition 15. *The Ordered Simple Path with Mandatory Nodes Problem* OSPMN *is an extension of* SPMN *where the mandatory nodes are to be visited in a given order, i.e., given a directed graph g, a source node src, a destination node dst, a set of mandatory nodes $mandnodes$, and set of couples of nodes $order$, we want to find a path in g from src to dst, going through $mandnodes$ following $order$ and visiting each node only once.*

As *OSPMN* is an extension of *SPMN*, it is not surprising that *OSPMN* is NP complete. Nevertheless, we will provide a proof of its NP completeness that does not rely on the fact of *SPMN* being NP complete.

Theorem 9. OSPMN *is NP complete.*

Proof. Once again we will take advantage of the NP completeness of The Disjoint Path Problem. 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.

$$\begin{array}{rcl} n & \notin & Nodes(g) \\ g' & = & AddEdges(g_1, E_1 \cup E_2) \\ g_1 & = & AddNodes(g_2, \{n\}) \\ g_2 & = & RemoveNodes(g, \{d_1, s_2\}) \\ E_1 & = & IncEdges(g, d_1)[d_1/n] \\ E_2 & = & OutEdges(g, s_2)[s_2/n] \end{array} \tag{2.26}$$

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 2.8 shows the the reduction of the two disjoint paths problem of Figure 2.7. The path found in Figure 2.8 corresponds to the concatenation of the two disjoint paths of Figure 2.7.

OSPMN can be modeled in term of *DomReachability* as follows:

$$OSPMN(g, src, dst, mn, order, fg) \leftrightarrow \left\{ \begin{array}{l} Subgraph(fg, g) \\ DomReachability(fg, edg, tcg) \\ \langle src, dst\rangle \in Edges(tcg) \\ \forall i \in mn : \langle i, dst\rangle \in Edges(edg) \\ \forall \langle i, j\rangle \in order : \langle j, i\rangle \notin Edges(tcg) \end{array} \right. \tag{2.27}$$

Indeed, given two mandatory nodes i and j, if i should be visited first than j ($\langle i, j\rangle \in order$), it is enough to state that j does not reach i ($\langle j, i\rangle \notin Edges(tcg)$) in order to ensure that the resulting graph fg is a graph where, in all paths from src to dst, i is visited first than j.

2.3.4 The ordered disjoint paths problem

Definition 16. *The Ordered Disjoint Path (*ODP*) is an extension of* DP *where each couple is associated with a set of mandatory nodes and an order relation.*

Let us start with the case of *the 2 Ordered node-disjoint path problem (2ODP)* where, 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* $\langle g, \langle\langle s_1, d_1, mn_1, order_1\rangle, \langle s_2, d_2, mn_2, order_2\rangle\rangle\rangle$ can be reduced to the *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

$$order' = \left\{ \begin{array}{l} order_1 \cup \\ order_2 \cup \\ \{\langle n_1, n_2\rangle \mid (n_1 \in mn_1 \wedge n_2 = n) \vee (n_1 = n \wedge n_2 \in mn_2)\}. \end{array} \right. \tag{2.28}$$

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

$$\begin{array}{rcl} Reduce\_2\_ODP(ODPins) & = & OSPMNins \\ ODPins & = & \langle g, \langle\langle s_1, d_1, mn_1, order_1\rangle, \langle s_2, d_2, mn_2, order_2\rangle\rangle\rangle \\ OSPMNins & = & \langle g', s_1, d_2, mn', order'\rangle \end{array} \tag{2.29}$$

The function $ReduceODP$, which reduces any ordered disjoint path problem (ODP) to OSPMN, can be defined as shown in Figure 2.9. 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.

```
ReduceODP($\langle g, \langle\langle s_1, d_1, mn_1, order_1\rangle, \ldots, \langle s_k, d_k, mn_k, order_k\rangle\rangle\rangle$)
    $ospmn$ := $\langle g, s_1, d_1, mn_1, order_1\rangle$
    for $i \in \{2, 3, \ldots, k\}$ 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
```

Figure 2.9: Reducing ODP to OSPMN

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 $\langle s_i, d_i, \emptyset, \emptyset\rangle$.

2.3.5 The traveling salesman problem

Given a directed graph g, a source node src and a destination node dst, the *Traveling Salesman Problem (TSP)* is to find whether there is a *Hamiltonian Path*, i.e., a path containing all the nodes, whose length is less than or equal to a given value max [GJ79].

We can model this problem as follows:

$$TSP(g, src, dst, max, p) \leftrightarrow \begin{cases} mn = nodes(g) \setminus \{src, dst\} \\ SPMN(g, src, dst, mn, p) \\ Size(p, i) \\ i \leq max \end{cases} \tag{2.30}$$

This definition is using *SPMN* as defined in section 2.3.2. Here, we are basically constraining p to be a simple path of at most max edges from src to dst containing the nodes in mn.

Chapter 3

Algorithms for DomReachability

In chapter 2 we introduced three global constraints on top of the notions of Domination and Transitive Closure: *Reachability*, *Domination* and *DomReachability*. As *DomReachability* is the conjuntion of *Reachability* and *Domination*, the implementation of *DomReachability* covers the implementation of *Reachability* and *Domination*.

In this chapter we introduce the algorithms involved in the implementation of *DomReachability*. We start by explaining how the pruning rules are systematically derived from the properties of the constraint. Then we revisit each property and derive the corresponding pruning rules. During this process, we will emphasize the pruning gained by the activation of each rule.

As we saw in the previous chapter, $DomReachability(FG, EDG, TCG)$ constraints EDG to be the extended dominator graph of FG and TCG to be the transitive closure of FG. In order to discard graphs from the domain of FG that violate this constraint, we need to maintain the transitive closure graph and the extended dominator graph of the bounds of FG. So, in this chapter we also study some approaches for maintaining this information.

3.1 From properties to propagation rules

In this section we present a general approach for transforming properties into propagator rules. In section 3.2, we apply this approach to the properties of *DomReachability* in order to get the pruning rules of *DomReachability*.

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.

The definition of the *DomReachability* constraint and its derived properties give place to a set of propagation rules which are systematically generated as follows:

3.1.1 Generating derived properties

Given a property of the form $P \rightarrow Q$, we generate the corresponding derived properties by applying the following rules:

- If Q is a conjunction of basic formulas $Q_1 \wedge Q_2 \wedge ... \wedge Q_n$, the properties derived from $P \rightarrow Q_1 \wedge Q_2 \wedge ... \wedge Q_n$ are generated by applying the following rule:
$$\frac{P \rightarrow Q_1 \wedge Q_2 \wedge ... \wedge Q_i \wedge ... \wedge Q_n}{P \rightarrow Q_i} \tag{3.1}$$
- If P is a conjunction of basic formulas $P_1 \wedge P_2 \wedge ... \wedge P_n$ and Q is a basic formula, then the properties derived from $P_1 \wedge P_2 \wedge ... \wedge P_n \rightarrow Q$ are generated by applying the following rule:
$$\frac{P_1 \wedge P_2 \wedge ... \wedge P_n \rightarrow Q}{\neg Q \wedge P_1 \wedge ... \wedge P_{i-1} \wedge P_{i+1} ... \wedge P_n \rightarrow \neg P_i} \tag{3.2}$$

3.1.2 Precondition and postcondition rewriting

We approximate a given set by pruning its upper and lower bounds. Given a variable S that approximates a set s, $Max(S)$ refers to the greatest set to which S can be bound, and $Min(S)$ refers to the least set to which S can be bound.

S gets more determined when elements are removed from $Max(S)$ or added to $Min(S)$. The fact that the bounds of S evolve indicates that $Min(S)$ and $Max(S)$ are memory cells. The value that S denotes becomes totally determined when $Min(S)$ and $Max(S)$ become equal.

We check that i is in s by checking that i is in $Min(S)$. Similarly, we check that i is not in s by checking that i is not in $Max(S)$:

$$\frac{i \in s}{i \in Min(S)} \quad (3.3) \qquad \frac{i \notin s}{i \notin Max(S)} \quad (3.4)$$

We ensure that i is in s by adding i to the lower bound of S. Similarly, we ensure that i is not in s by removing i from the upper bound of S:

$$\frac{i \in s}{Min(S) := Min(S) \cup \{i\}} \quad (3.5) \qquad \frac{i \notin s}{Min(S) := Min(S) \setminus \{i\}} \quad (3.6)$$

3.2 Deriving pruning rules

We implement the constraint (2.9) by the propagator that we note

$$DomReachability(\langle FG, s\rangle, EDG, TCG). \tag{3.7}$$

FG, EDG and TCG are graph variables, i.e., variables whose domain is a set of graphs [DDD05b]. A graph variable G is represented by two graphs: $Min(G)$ and $Max(G)$. 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

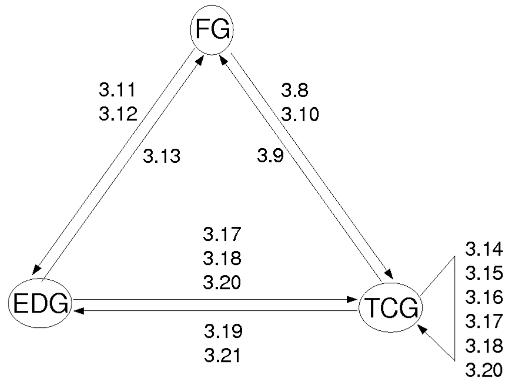


Figure 3.1: Activation dependencies between the graph variables of DomReachability

upper bounds of G, respectively. Notice that the source s of the flow graph FG is a known value.

We will know revisit the properties of *DomReachability* presented in Chapter 2 and define the corresponding pruning rules. Figure 3.1 shows the graph of activation dependencies between the graph variables of DomReachability. An edge labeled with r from G_1 to G_2 means that a change in the domain of G_1 may cause a change in the domain of G_2 through the application of rule r.

3.2.1 Pruning rules of Property 2.11

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

$$\forall\langle i,j\rangle \in Edges(fg) : \langle i,j\rangle \in Edges(tcg)$$

This property represents the basic case of the transitive closure. The pruning rules derived from this property (and from property 2.13) establish the connection between the bounds of FG and the bounds of TCG.

The pruning rules derived from property 2.11 are the following:

- It causes the introduction of edge $\langle i,j\rangle$ in the lower bound of TCG when the edge is in the lower bound of FG:

$$\frac{\langle i,j\rangle \in Edges(Min(FG))}{Edges(Min(TCG)) := Edges(Min(TCG)) \cup \{\langle i,j\rangle\}} \tag{3.8}$$

- It causes the removal of edge $\langle i,j\rangle$ from the upper bound of FG when the edge is not in the upper bound of TCG:

$$\frac{\langle i,j\rangle \notin Edges(Max(TCG))}{Edges(Max(FG)) := Edges(Max(FG)) \setminus \{\langle i,j\rangle\}} \tag{3.9}$$

3.2.2 Pruning rule of Property 2.13

If i does not reach j, $\langle i, j\rangle$ is not an edge of tcg.

$$\forall i, j \in N : \langle i, j\rangle \notin Edges(TC(fg)) \rightarrow \langle i, j\rangle \notin tcg$$

One consequence of this property is that edge $\langle i, j\rangle$ is removed from the upper bound of TCG when j is not reachable from i in the transitive closure of the upper bound of FG.

$$\frac{\langle i, j\rangle \notin Edges(TC(Max(FG)))}{Edges(Max(TCG)) := Edges(Max(TCG)) \setminus \{\langle i, j\rangle\}} \tag{3.10}$$

The implementation of this rules makes it necessary to update the transitive closure of the upper bound of FG after modifying it. As the upper bound of a graph variable evolves monotonically, it is possible to consider decremental approaches for updating the transitive closure of FG. By decremental we mean dynamic algorithms in which the only changes considered are removals of nodes and edges [FMNZ01, RZ02].

3.2.3 Pruning rule of Property 2.14

Let $DomGraph$ be a function that returns the dominator graph of a flow graph, i.e., $\langle i, j\rangle \in Edges(DomGraph(fg)) \leftrightarrow i \in Dominators(fg, j)$.

If i dominates j then , $\langle i, j\rangle$ is an edge of edg.

$$\forall i, j \in N \cup (N \times N) : \langle i, j\rangle \in Edges(DomGraph(Ext(fg))) \rightarrow \langle i, j\rangle \in edg$$

As we are considering the extended dominator graph of fg, i and j may also be edges of fg.

One consequence of property 2.14 is to add the edge $\langle i, j\rangle$ to the lower bound of EDG when i dominates j in the upper bound of FG.

$$\frac{\langle i, j\rangle \in Edges(DomGraph(Ext(Max(FG))))}{Edges(Min(EDG)) := Edges(Min(EGD)) \cup \{\langle i, j\rangle\}} \tag{3.11}$$

Notice that, as the upper bound of FG evolves monotonically, once j is dominated by i in the upper bound of FG it stays dominated by i. This monotonic evolution also implies that decremental algorithms for computing dominators can be considered. However, as the computation of dominators from scratch can be done in linear time [Geo05], we will restrict our attention to this kind of algorithms only.

3.2.4 Pruning rules of Property 2.15

If j is directly reachable from the source, i.e., $\langle s, j\rangle$ is an edge of fg (where $s = Source(fg)$), then s is the only dominator of j.

$$\forall i, j \in N, i \neq s : \langle s, j\rangle \in Edges(fg) \rightarrow \langle i, j\rangle \notin Edges(edg)$$

The consequences of this property are the following:

- Edge $\langle i, j\rangle$ is removed from the upper bound of EDG if edge $\langle s, j\rangle$ is in the lower bound of FG:

$$\frac{\langle s, j\rangle \in Edges(Min(FG))}{Edges(Max(EDG)) := Edges(Max(EDG)) \setminus \{\langle i, j\rangle\}} \tag{3.12}$$

- Edge $\langle s, j\rangle$ is removed from the upper bound of FG if edge $\langle i, j\rangle$ is in the lower bound of EDG:

$$\frac{\langle i, j\rangle \in Edges(Min(EDG))}{Edges(Max(FG)) := Edges(Max(FG)) \setminus \{\langle s, j\rangle\}} \tag{3.13}$$

3.2.5 Pruning rules of Property 2.12

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

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

The pruning rules derived from this property let us propagate the reached nodes of a given node back to their ancestors. In a similar way, unreachable nodes are propagated from a node to its successors.

The pruning rules derived from property 2.12 are the following:

- Edge $\langle i, k\rangle$ is included in the lower bound of TCG when edges $\langle i, j\rangle$ and $\langle j, k\rangle$ are in the lower bound of TCG:

$$\frac{\langle i, j\rangle \in Edges(Min(TCG)) \wedge \langle j, k\rangle \in Edges(Min(TCG))}{Edges(Min(TCG)) := Edges(Min(TCG)) \cup \{\langle i, k\rangle\}} \tag{3.14}$$

- Edge $\langle i, j\rangle$ is removed from the upper bound of TCG when edge $\langle i, k\rangle$ is not in the upper bound of TCG and edge $\langle j, k\rangle$ is in the lower bound of TCG:

$$\frac{\langle i, k\rangle \notin Edges(Max(TCG)) \wedge \langle j, k\rangle \in Edges(Min(TCG))}{Edges(Max(TCG)) := Edges(Max(TCG)) \setminus \{\langle i, j\rangle\}} \tag{3.15}$$

- Edge $\langle j, k\rangle$ is removed from the upper bound of TCG when edge $\langle i, k\rangle$ is not in the upper bound of TCG and edge $\langle i, j\rangle$ is in the lower bound of TCG:

$$\frac{\langle i, k\rangle \notin Edges(Max(TCG)) \wedge \langle i, j\rangle \in Edges(Min(TCG))}{Edges(Max(TCG)) := Edges(Max(TCG)) \setminus \{\langle j, k\rangle\}} \tag{3.16}$$

3.2.6 Pruning rules of Property 2.16

If j is reachable from s and i dominates j in fg, then i is reachable from s and j is reachable from i:

$$\begin{aligned}\forall i, j \in N : \langle s, j\rangle \in Edges(tcg) \wedge \langle i, j\rangle \in Edges(edg) \rightarrow\\ \langle s, i\rangle \in Edges(tcg) \wedge \langle i, j\rangle \in Edges(tcg)\end{aligned}$$

The rules derived from this property are the following:

- Edges $\langle s, i\rangle$ and $\langle i, j\rangle$ are added to lower bound of TCG when edge $\langle s, j\rangle$ is in the lower bound of TCG and edge $\langle i, j\rangle$ is in the lower bound of EDG:

$$\frac{\langle s, j\rangle \in Edges(Min(TCG)) \wedge \langle i, j\rangle \in Edges(Min(EDG))}{Edges(Min(TCG)) := Edges(Min(TCG)) \cup \{\langle s, i\rangle, \langle i, j\rangle\}} \tag{3.17}$$

  Nodes and edges common to all paths from the source to j get included in the lower bound of FG when i is reachable from the source through the activation of this rule.

- Edge $\langle s, j\rangle$ is removed from the upper bound of TCG when edge $\langle s, i\rangle$ is not in the upper bound of TCG and edge $\langle i, j\rangle$ is in the lower bound of EDG:

$$\frac{\langle s, i\rangle \notin Edges(Max(TCG)) \wedge \langle i, j\rangle \in Edges(Min(EDG))}{Edges(Max(TCG)) := Edges(Max(TCG)) \setminus \{\langle s, j\rangle\}} \tag{3.18}$$

- Edge $\langle i, j\rangle$ is removed from the upper bound of EDG when edge $\langle s, i\rangle$ is not in the upper bound of TCG and edge $\langle s, j\rangle$ is in the lower bound of TCG:

$$\frac{\langle s, i\rangle \notin Edges(Max(TCG)) \wedge \langle s, j\rangle \in Edges(Min(TCG))}{Edges(Max(EDG)) := Edges(Max(EDG)) \setminus \{\langle i, j\rangle\}} \tag{3.19}$$

- Edge $\langle s, j\rangle$ is removed from the upper bound of TCG when edge $\langle i, j\rangle$ is not in the upper bound of TCG and edge $\langle i, j\rangle$ is in the lower bound of EDG:

$$\frac{\langle i, j\rangle \notin Edges(Max(TCG)) \wedge \langle i, j\rangle \in Edges(Min(EDG))}{Edges(Max(TCG)) := Edges(Max(TCG)) \setminus \{\langle s, j\rangle\}} \tag{3.20}$$

- Edge $\langle i, j\rangle$ is removed from the upper bound of EDG when edge $\langle i, j\rangle$ is not in the upper bound of TCG and edge $\langle s, j\rangle$ is in the lower bound of TCG:

$$\frac{\langle i, j\rangle \notin Edges(Max(TCG)) \wedge \langle s, j\rangle \in Edges(Min(TCG))}{Edges(Max(EDG)) := Edges(Max(EDG)) \setminus \{\langle i, j\rangle\}} \tag{3.21}$$

3.2.7 Showing activation of pruning rules

Let us see the pruning previously introduced in a concrete example. Figures 3.2, 3.3 and 3.4 show a partially defined flow graph with its corresponding partially defined dominator graph and transitive closure. For the sake of simplicity, in this example, we will assume that the second argument of *DomReachability* is a dominance graph (instead of a extended dominance graph).

Notice, for instance, that we can not say any thing about the domination of node 3 over node 2 since node 3 trivially dominates node 2 if the source (node 1) does not reach node 2. The same applies for node 4.

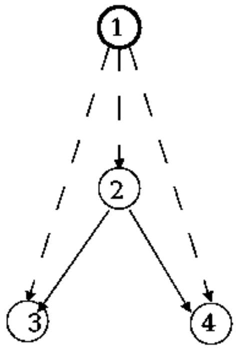


Figure 3.2: Flow graph

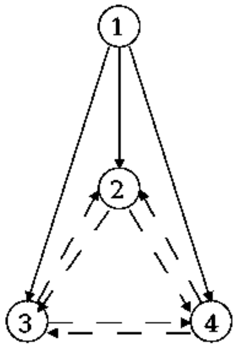


Figure 3.3: Dominance graph

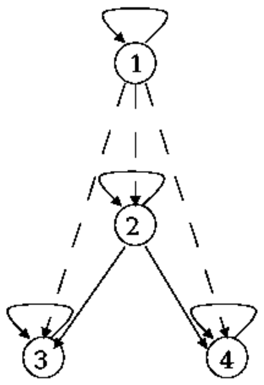


Figure 3.4: Transitive closure

In figures 3.5, 3.6 and 3.7, we show the effect of imposing that edge $< 1, 2 >$ is part of the flow graph. This decreases the upper bound of the dominance graph since neither 3 nor 4 dominate 2. This pruning is caused by rule 3.21 since edge $< 1, 2 >$ is in the lower bound of the transitive closure and edges $< 3, 2 >$ and $< 4, 2 >$ are not in the upper bound of the transitive closure.

The presence of edge $< 1, 2 >$ in the flow graph also implies the determination of the transitive closure as observed in Figure 3.7. This is the result of applying rules 3.8 and 3.14, which basically update the lower bound of the transitive closure after adding a new edge to the lower bound of the flow graph.

Imposing that edges $< 2, 3 >$ and $< 2, 4 >$ are part of the dominance graph decreases the upper bound of the flow graph. All the paths from the source to node 3, and from the source to node 4 should contain node 2. This is why edges $< 1, 3 >$ and $< 1, 4 >$ are removed. This pruning is caused by rule 3.13.

It is important to observe that the bigger the flow graph is the smaller the dominance graph is. Indeed, imposing domination reduces the ways nodes can be connected.

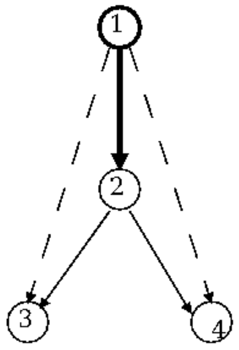


Figure 3.5: Flow graph

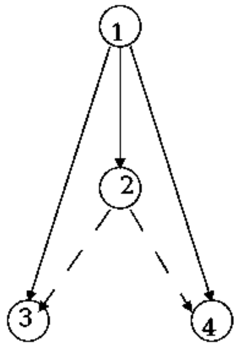


Figure 3.6: Dominance graph

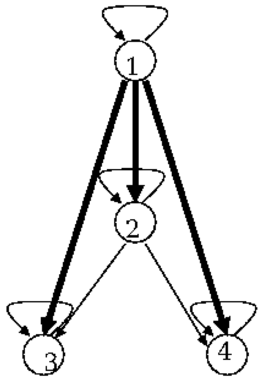


Figure 3.7: Transitive closure

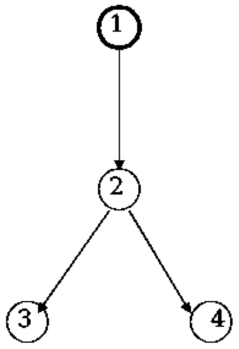


Figure 3.8: Flow graph

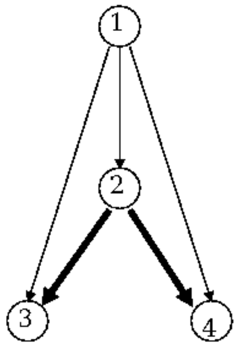


Figure 3.9: Dominance graph

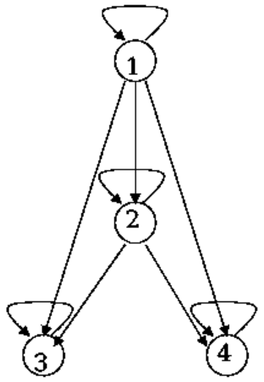


Figure 3.10: Transitive closure

3.2.8 Optimizing the discovery of dominators

Let us revisit the definition of the extended graph of a flow graph $\langle N, E, s\rangle$:

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

The computation of the extended graph of a graph can be done in linear time with respect to the size of the graph since it basically consists in traversing the original graph and performing a constant amount of operation at each step. The resulting graph is a graph whose size complexity is equivalent to the original graph:

$$|N'| + |E'| = (|N| + |E|) + 2 * |E| = |N| + 3 * |E| < 3 * (|N| + |E|) \quad (3.22)$$

Nevertheless, the number of nodes of the resulting graph is, in the worst case, quadratic with respect to the number of nodes of the original graph.

As the computation of the dominance tree depends on the size of the graph, computing the dominance tree of the dominance graph has the same complexity that computing the dominance tree of the extended graph. However, computing the transitive closure of the extended dominance tree is more expensive since the complexity of this operation does depend on the number of nodes.

Pruning rule 3.17 can be optimized by considering the fact that we only need to activate the rule when edge $\langle i, j\rangle$ is in $Edges(DomTree(Max(FG)))$ since the other dominators of j will be added when considering their immediately dominated nodes. This means that rule 3.17 can be re-formulated as follows:

$$\frac{\langle s, j\rangle \in Edges(Min(TCG)) \wedge \langle i, j\rangle \in Edges(DomTree(Ext(Max(FG))))}{Edges(Min(TCG)) := Edges(Min(TCG)) \cup \{\langle s, i\rangle, \langle i, j\rangle\}} \quad (3.23)$$

For instance, suppose that the set of dominators of node j in $Ext(Max(FG))$ is $\{i_1, i_2\}$ and that i_1 is the immediate dominator of i_2, which implies that i_2 is the immediate dominator of j. Suppose also that $\langle s, j\rangle$ is in $Edges(Min(TCG))$. Then, this rule will add edges $\langle s, i_2\rangle$ and $\langle i_2, j\rangle$ to $Edges(Min(TCG))$, which implies the addition of node i_2 to both $Nodes(Min(FG))$ and $Nodes(Min(TCG))$. As edge $\langle s, i_2\rangle$ is now in $Edges(Min(TCG))$ and i_1 is the immediate dominator of i_2, edges $\langle s, i_1\rangle$, $\langle i_1, i_2\rangle$ are added to $Edges(Min(TCG))$.

Indeed, even though we are only considering the extended dominance tree all the dominators of a given node j get added to the lower bound of FG when the sources reaches j. This optimization of the rule avoids unnecessary activations. If we consider the previous example, we were adding i_1 twice (when considering edges $\langle s, j\rangle$ and $\langle s, i_2\rangle$) even though the second addition is not needed since i_1 is already included. With this optimization, we can say that the number of activation with respect to the length of the branch containing j in the extended dominance tree is linear.

3.2.9 DomReachability works on approximations

As we said before, graph variables are represented in terms of bounds. However, there are sets of graphs that cannot be represented in terms of bounds. For instance, consider the case in Figure 3.11. The domain $\{g_1, g_2\}$ cannot be represented in terms of bounds. This domain is approximated as shown in Figure 3.12. The domain of this graph variable is the set of all the graphs with set of nodes $\{1, 2, 3, 4\}$ whose edges are included in the set $\{\langle 1, 2\rangle, \langle 3, 4\rangle, \langle 1, 4\rangle, \langle 3, 2\rangle\}$.

In general, we can say that the domain of a variable is a (not necessarily proper) sub set of its approximation.

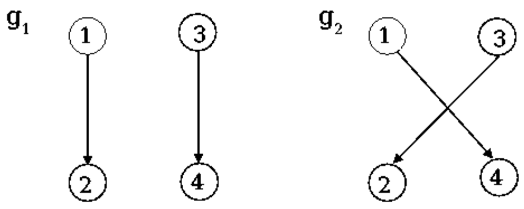


Figure 3.11: The domain $\{g_1, g_2\}$ cannot be represented in terms of bounds

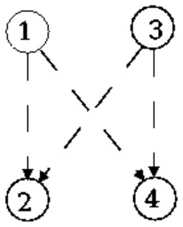


Figure 3.12: Approximation of the domain $\{g_1, g_2\}$

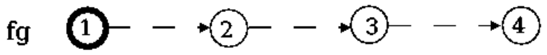


Figure 3.13: There are several ways of preventing 4 to be reachable from 1

One implication of approximating the domains is that the effects of imposing a constraint on one of the arguments of *DomReachability* cannot be directly propagated to the other two arguments. For instance, suppose that the flow graph is instantiated as shown in figure 3.13. The effects of imposing the constraint $\langle 1, 4\rangle \notin tcg$ can not be propagated to the flow graph since the set of possible graphs to which fg can be instantiated, which are shown in Figure 3.14, cannot be represented in terms of bounds.

3.2.10 Level of consistency of DomReachability

As explained in the previous section, the domain of a graph variable can be only pruned by modifying the elements in its bounds. This means that, in the best case,

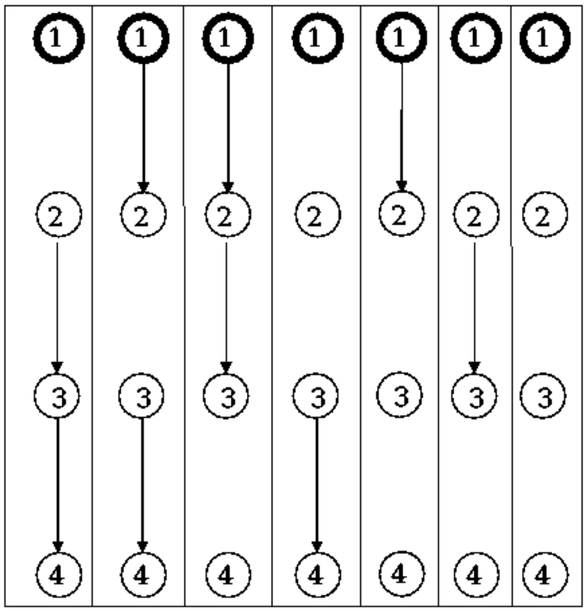


Figure 3.14: Actual domain of fg (in Figure 3.13) after impossing that $\langle 1, 4 \rangle \notin tcg$

bound consistency is the highest level of consistency that can be achieved. Nevertheless, achieving this level of consistency is still a challenge when considering constraints like *DomReachability* since it may lead to NP computations.

The fact that we can model an NP-complete problem like *SPMN* (as shown in section 2.3.2) in terms of *DomReachability* implies that *DomReachability* cannot achieve general consistency in polynomial time. Suppose that we need to find out whether a particular node n in the upper bound of FG needs to be removed. For this, we need to see whether there is at least one simple path containing all the mandatory nodes that contains n. n must be removed from the upper bound if there is not any simple path. Notice that this corresponds to the definition of *SPMN*. As bound consistency relies on the ability to answer this query for every node in the upper bound, this implies that bound consistency can not be achieved in polynomial time.

Notice that, even for *Reachability(g,tcg)* (the constraint introduced in section 2.2.1), checking bound consistency is NP complete. As explained in section 2.3.1, *BTC* is the problem we need to solve in order to determine whether a particular partial instantiation is consistent. As *BTC* is NP complete, checking bound consistency is NP complete too.

3.3 Algorithms for computing dominators

3.3.1 Aho and Ullman's algorithm

Aho and Ullman's algorithm [AU77] relies on the very definition of domination. As we know, i dominates j is all paths from the source to j contain i. An immediate consequence of this is that j should not be reachable from the source after removing i. So, in order to detect the nodes dominated by a node i we look at the set of nodes that are no longer reachable from the source after removing i.

```
Pre : fg is a flow graph
Post : dom(i) is the set of dominators of node $i$ in $fg$

GetDominators(fg)
    $nodes_0$ := $DFS(fg, Source(fg))$
    for $i \in Nodes(fg)$ do
        $doms(i)$ := if $i \in nodes_0$ then $\emptyset$ else $Nodes(fg) \setminus \{i\}$ end
    end
    for $i \in nodes_0$ do
        $nodes_1$ := $DFS(RemoveNode(fg, i), Source(fg))$
        for $j \in nodes_0 \setminus (nodes_1 \cup \{i\})$ do
            $doms(j)$ := $doms(j) \cup \{i\}$
        end
    end
    return $doms$
end
```

Figure 3.15: Aho and Ullman's algorithm

The algorithm is presented in Figure 3.15. The input is a flow graph fg and the output is a map dom that associates each node with its set of dominators, i.e., $doms(i)$ is the set of dominators of node i in fg. Let us assume that *DFS* returns the reachable nodes. $doms(i)$ is initialized with either $\emptyset$ or $Nodes(fg) \setminus \{i\}$ depending on whether i is reachable from $Source(fg)$ (since any node dominates an non-reachable node). Each node is removed in order to detect the nodes that it dominates. Therefore the computation of dominators is $O(N * (N + E))$.

3.3.2 Cooper, Harvey and Kennedy's algorithm

Cooper, Harvey and Kennedy's algorithm [CHK] is an iterative algorithm that relies on the fact that the dominators of a node n is the set composed of n and the intersection of the dominators of its predecessors. In this case the notion of domination presented in Chapter 2 is adapted so that each node is part of its set of dominators:

$$\begin{aligned} doms(n_0) &= \{n_0\} \\ doms(n) &= (\textstyle\bigcap_{p \in preds(n)} doms(p)) \cup \{n\} \end{aligned} \tag{3.24}$$

From this definition of domination we can observe that at most one predecessor of n can be a dominator of n since n can have only one immediate dominator.

```
Pre :
  $fg$ is a flow graph
  $preds(i)$ is the set of incoming nodes of $i$ in $fg$
Post :
  $doms(i)$ is the set of dominators of node $i$ in $fg$

GetDominators(fg)
   for all nodes, $n$ do
      $doms[n] := \{1..|Nodes(fg)|\}$
   end
   $changed := true$
   while ($changed$) do
      $changed := false$
      for all nodes, $n$, in reverse postorder
      $newset := (\bigcap_{p \in preds(n)} doms[p]) \cup \{n\}$
      if ($newset \neq doms[n]$) then
         $doms[n] := newset$
         $changed := true$
      end
      end
   end
   return $doms$
end
```

Figure 3.16: Cooper, Harvey and Kennedy's simplified iterative algorithm for computing dominators

Figure 3.16 shows a simplified version of Cooper et al's algorithm. In each iteration the set of dominators is updated according to Equation 3.24. The loop continues until no further update is performed.

Notice that this algorithm relies on the fact the the source node has no incoming nodes so $(\bigcap_{p \in preds(n)} doms[p]) \cup \{n\}$ is equal to $\{n\}$ if n is the source.

In [CHK], Cooper et al suggest a clever way of implementing the set operations executed at each iteration that allows to compute the dominators in $O(N^2)$.

3.3.3 Lengauer and Tarjan's algorithm

In order to find the dominators of the flow graph G, this algorithm assigns a number to each node. This number corresponds to the order in which the nodes are visited following a *DFS* traversal rooted at r (the source of G). In what follows, these numbers will be used to denote the nodes and D will be the resulting DFS-tree, which will be represented by the array $parent$.

The algorithm is based on the notion of *semidominator* which is defined in terms of the notion of *semidominator path*. These two notions are defined by Lengauer and Tarjan [LT79] as follows:

Definition 17. *A path $P = (u = v_0, v_1, ..., v_{k-1}, v_k = v)$ in G is a* semidominator path *(sdom path) if $v_i > v$ for $1 \leq i \leq k-1$.*

Definition 18. *The* semidominator *of a node v ($s(v)$) is the minimum u such that there is a sdom path from u to v.*

Notice that a node only has one semidominator since the DFS number associated with each node is unique.

In [LT79], Lengauer and Tarjan define a set of properties between dominators and semidominators. In short, these properties imply that, for any $w \neq r$:

- $s(w)$ is a proper ancestor of w in D.
- $d(w)$ is a (not necessarily proper) ancestor of $s(w)$.
- The dominators of nodes in G do not change if the edges that are not in $Edges(D)$ are replaced with the edges in $\{\langle s(w), w\rangle : w \in Nodes(G) \wedge w \neq r\}$.This means that the set of dominators can be computed from D and the set of semidominators.

Figure 3.17, which is taken from [Geo05], shows the skeleton of Lengauer and Tarjan's algorithm (*LT*). *LT* maintains a forest F whose trees are composed of edges of D. In fact, we can say that F is a sub graph of D. The two operations performed on the forest are:

- $link(v)$ that adds the edge $\langle parent(v), v\rangle$ to the forest.
- $eval(v)$ returns $r_F(v)$ if $v = r_F(v)$. Otherwise, it returns $min\{u : r_F(v) \xrightarrow{+} u \xrightarrow{*} v\}$ in D.

In the previous definitions $parent(v)$, is the parent of v in D and $r_F(v)$ is the root of the tree containing v in F.

```
Pre :
   $G = \langle N, E, r \rangle$ is a flow graph with source node $r$
   $D = DFS(G, r)$
   $pred(i)$ is the set of predecessor nodes of $i$ in $D$
   $parent(i)$ is the parent of $i$ in $D$
Post :
   $d(i)$ is the immediate dominator of $i$ in $G$

LT(G)
   for $w \in N - \{r\}$ in reverse preorder of $D$ do
      $s(w) := w$
      for $v \in pred(w)$ do
         $x := eval(v)$
         $s(w) := min\{s(w), s(x)\}$
      end
      add $w$ to $bucket(s(w))$
      link(w)
      z:=parent(w)
      for $v \in bucket[z]$ do
         delete $v$ from $bucket(z)$
         $y := eval(v)$
         if $s(y) < z$ then $d(v) := y$ else $d(v) := z$ end
      end
   end
   for $w \in N - \{r\}$ in preorder of $D$ do
      if $d(w) \neq s(w)$ then $d(w) := d(d(w))$ end
   end
end
```

Figure 3.17: The Lengauer-Tarjan algorithm

In [LT79], Lengauer and Tarjan show that if nodes are processed in reverse preorder, then all the necessary values will be available when needed. This is why the first loop in Figure 3.17 considers the nodes in reverse preorder.

The semidominator of a node u is chosen by considering the current semidominators of the nodes in the path from r to u. *LT* associates each node with the set of nodes that it semidominates. This association is done through the vector $bucket$, i.e., $bucket(i)$ is the set of nodes semidominated by i. Each node v is initially associated with an approximate immediate dominator y. y is an approximation because it might not be the immediate dominator of v, but a node whose immediate

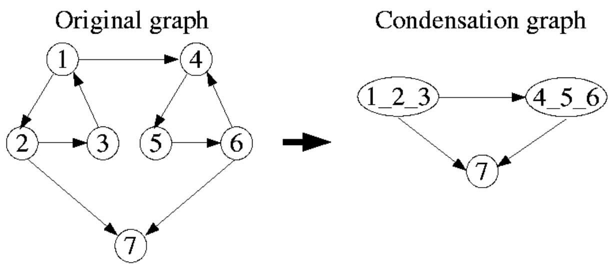


Figure 3.18: An example of a condensation graph

dominator is also the immediate dominator of v. So, the purpose of the last loop is to check whether y is indeed the immediate dominator. If this is not the case, the immediate dominator is updated accordingly.

The complexity of this algorithm depends on how $link$ and $eval$ are implemented. If the complexity of these functions were constant, *LT* would be linear with respect to the size of the graph. In [LT79], Lengauer and Tarjan show one way of implementing these operations that almost achieves this goal. The complexity of the algorithm presented in [LT79] is $O(m\alpha(m,n))$, where n and m are the number of nodes and edges, and α is the inverse of Ackermann's function.

3.4 Algorithms for computing transitive closure

3.4.1 Frigioni et al's decremental algorithm

Frigioni et al's decremental algorithm [FMNZ01] is based on the notion of graph condensation. The condensation graph of G is the graph $G' = \langle N', E' \rangle$ where each node in N' corresponds to a strongly connected component in G and an edge $\langle u, v \rangle$ is in E' if and only if there exists an edge in E connecting any of the nodes in the component of u to any of the nodes in the component of v.

An example of graph condensation is shown in Figure 3.18. Nodes 1, 2 and 3 form a strongly connected component and so do nodes 4, 5 and 6. Node 7 is a strongly connected component on its own. The nodes representing the strongly connected component are connected as mentioned before. For instance, As node 1 and node 4 are in different connected components, edge $\langle 1\_2\_3, 4\_5\_6 \rangle$ is in the set of edges of the condensation graph.

Notice that a condensation graph is always acyclic. Indeed, if the nodes representing the strongly connected components i and j participate in a cycle in the condensation graph, the union of i and j would form a strongly connected component too since the nodes of i would be reachable from the nodes of j and vice versa.

This is why Frigioni et al decompose the problem of maintaining the transitive closure of a directed graph $G = \langle N, E\rangle$ into two sub problems:

- Maintaining the condensation graph of G.
- Maintaining the association of nodes with strongly connected components.

In the description of the algorithm, n will refer to $|N|$.

Decremental maintenance of transitive closure for DAGs.

The algorithm presented here is an optimization of Italiano's algorithm [Ita88] done by Frigioni et al [FMNZ01]. Italiano associates every node $u \in N$ with a set $DESC[u]$ containing all the descendants of u in G. Each one of those sets is organized as a out-tree rooted at u. In addition, an $n \times n$ matrix of pointers, called $INDEX$, is maintained which allows fast access to nodes in these trees. If $j \in DESC[i]$, $INDEX[i, j]$ points to node j in $DESC[i]$; otherwise, it is *Null*. $INDEX$ allows to check weather a node is reachable from another in constant time.

Let $\langle i, j\rangle$ be the edge to be deleted. If $\langle i, j\rangle$ does not belong to any $DESC$ tree, the data structure does not need to be updated. If $\langle i, j\rangle$ belongs to $DESC[u]$, $DESC[u]$ has to reconstructed since the deletion of the edge splits the tree. In order to do this, one has to check that there is still a path from u to j. This is done by checking whether there is an edge $\langle v, j\rangle$ in $DESC[u]$ such that there is a path from u to v that avoids $\langle i, j\rangle$. If such edge exists, edge $\langle i, j\rangle$ is replaced with $\langle v, j\rangle$ in $DESC[u]$. In this case, v is considered a *hook* for j. If such edge does not exist, then j must be removed from $DESC[u]$ and the outgoing edges of j in $DESC[u]$ should be deleted by applying the same procedure.

In order to find a hook for j, each node y is associated with the set of tails of its incoming edges $IN[y]$. Assuming that each set is correctly updated after each edge deletion, checking whether there is a hook for j in $DESC[u]$, after the deletion of $\langle i, j\rangle$, is done by checking if $IN[j] \cap DESC[u] \neq \emptyset$. Indeed, if $IN[j] \cap DESC[u] \neq \emptyset$ it is because there is at least one node reachable from u that is an incoming node of j.

Notice that nodes that are not hook for a node j remains in that condition after edge removals. This means that if it has been already detected that k is not a hook for j, k does not need to be reconsidered after removing an edge. This is why an $n \times n$ matrix $HOOK$ is maintained. $HOOK[u, j]$ stores the pointer to the first yet to be considered node in $IN[j]$. If there is no more nodes to be considered, $HOOK[u, j]$ is $Null$.

Frigioni et al optimize the representation of the information by using a single $n \times n$ matrix $PARENT$ containing the information in $DESC$ and $INDEX$. $PARENT[i, j]$ stores a pointer to the edge that connects j to its parent in $DESC[i]$ if $j \in DESC[i]$. If $j \notin DESC[i]$, $PARENT[i, j]$ is $Null$.

Figure 3.19 shows the algorithm for deleting an edge $\langle i_0, j_0 \rangle$. The first thing done is to update IN since i_0 is no longer an incoming node of j_0. Then, all the nodes u whose out-tree contains edge $\langle i_0, j_0 \rangle$ are considered. In each iteration a queue Q that keeps the edges to be removed from the out-tree of u is created. As $\langle i_0, j_0 \rangle$ must be removed, it is put in the queue Q. Then, each edge of Q is considered. In each iteration, the first edge $\langle i, j \rangle$ of Q is dequeued and a valid replacement is searched. As said before, this replacement exists if $HOOK[u, j]$ is not $Null$. If this is the case, $\langle i, j \rangle$ is replaced with $\langle HOOK[u, j], j \rangle$ and $HOOK$ is updated. If not, $Null$ is assigned to $PARENT[u, j]$ indicating that j is no longer reachable from u and all the outgoing edges of j that are in the out-tree of u are added to Q since j cannot be used to reach any node k from u.

Figure 3.20 shows how $HOOK$ is updated. In this algorithm it is assumed that the elements of $IN[j]$ are indexed with respect to the initial graph. So, $IN[j][k]$ is the k_{th} incoming node of j in the initial graph. As the index of the hook for j before removing the edge is $GetIndex(IN[j][HOOK[u, j]])$, ind (the index of the hook for j after removing the edge) is initialized with the next index. Notice that, in this initialization, one relies on the fact that hook candidates are not reconsidered since nodes that are not hooks do not become hooks after removing edges. Once ind is initialized, one loops until either one finds an incoming node that is reachable from i or runs out of incoming nodes. In the former case, ind is assigned to $HOOK[u, j]$. In the later, $Null$ is assigned to $HOOK[u, j]$ indicating that there are no more valid replacements.

Figure 3.21 shows the implementation of $Reach(i, j)$. In fact the implementation of this function is straightforward since it only consists in looking up $PARENT$ in order to see whether j is reachable from i or not.

Deleting m edges takes $O(nm_0)$, where n is the number of nodes and m_0 the number of initial edges. This means that the complexity of updating the transitive closure after removing one edge is linear with respect to the size of the graph in average. This linear cost is due to the fact that each edge is only considered once in a given out-tree. Once an edge becomes unsuitable for being added to the descendant tree of a node it remains unsuitable.

The complexity of $Reach$ is constant since it is a matrix look up.

Maintaining the association of nodes with strongly connected components.

As the previous decremental algorithm for maintaining transitive closure only considers acyclic graph, the original graph G is transformed into a graph G' by condensing its strongly connected components. The deletion of an edge from G does not necessarily change G'. Indeed, if the edge that is being deleted is inside a strongly connected component and the connectivity of the component is not affected by the removal, G' remains unchanged.

Even though the computation of strongly connected components takes linear time, the update of the data structures used in the decremental algorithm for the transitive closure should be done carefully in order to keep the complexity accept-

```
Pre :
    $PARENT[u, j]$ is the edge that connects $j$ to its parent in the out
    tree rooted at $u$. It is $Null$ if j is not reachable from $u$.
    $HOOK[u, j]$ is the first yet to be considered node $k$ such that $\langle k, j\rangle$
    is a valid replacement for $PARENT[u, j]$. It is null is such replacement
    does not exist.
    $IN[j]$ is the set of incoming nodes of $j$

Post :
    $PARENT$,$HOOK$ and $IN$ are updated considering the removal of $\langle i_0, j_0\rangle$


Delete_Edge($\langle i_0, j_0\rangle$)
  $IN[j_0] := IN[j_0] \setminus \{i_0\}$
  for $u$ : $PARENT[u, j_0]$==$\langle i_0, j_0\rangle$ then
    Q=NewQueue
    Enqueue($Q$,$\langle i_0, j_0\rangle$)
    while NotEmpty($Q$) do
      $\langle i, j\rangle$:=Dequeue($Q$)
      if $HOOK[u, j] \neq Null$ then
        $PARENT[u, j]$:=$\langle HOOK[u, j], j\rangle$
        $Update\_Hook(u, j)$
      else
        $PARENT[u, j]$:=$Null$
        for $k$ : $PARENT[u, k] == \langle j, k\rangle$ do
          Enqueue($Q$,$\langle j, k\rangle$)
        end
      end
    end
  end
end
```

Figure 3.19: Delete_Edge

able. In particular, one has to deal with the fact that a strongly connected component may split into several components after removing an edge. In order to cope with this, the set of data structures considered are the following:

- A Boolean matrix $INDEX$ whose entry $INDEX[i, j]$ is true or false de-

```
Pre :
    $PARENT[u,j]$ is the edge that connects $j$ to its parent in the out
    tree rooted at $u$. It is $Null$ if j is not reachable from $u$.
    $HOOK[u,j]$ is the first yet to be considered node $k$ such that $\langle k,j \rangle$
    is a valid replacement for $PARENT[u,j]$. It is null is such replacement
    does not exist.
    $IN[j][k]$ is the $k_{th}$ incoming node of $j$ in the initial graph

Post :
    $HOOK[u,j]$ is updated considering the information in $IN$

Update_Hook($u$,$j$)
  $ind$:=GetIndex($IN$,$HOOK[u,j]$)+1
  while $PARENT[u, IN[j][ind]] == Null \wedge ind \leq MaxIndex(IN[j])$ do
    $ind := ind + 1$
  end
  if $ind \leq MaxIndex(IN[j])$ then
    $HOOK[u,j] := ind$
  else
    $HOOK[u,j] := Null$
  end
end
```

Figure 3.20: Update_Hook

pending on whether there is a path from i to j.

- An array Scc such that, for all v in N, $Scc[v]$ is the strongly connected component containing v.
- For each strongly connected component C one has:
  - An array In such that $C.In$ refers to the incoming edges of C.
  - An array Out such that $C.Out$ refers to the outgoing edges of C.
  - An array $PARENT$ such that, for all v in N, $C.PARENT[v]$ is the edge in $DESC[v]$ that connects C to v, in case C is reachable from v and v is not inside C. Otherwise, $C.PARENT[v]$ is $Null$.
  - An array $HOOK$ such that $C.HOOK[v]$, is the first yet to be considered edge in $C.In$ that is a valid replacement for $C.PARENT[v]$.

```
Pre :
    $PARENT[u, j]$ is the edge that connects $j$ to its parent in the out
    tree rooted at $u$. It is $Null$ if j is not reachable from $u$.
Post :
    Reach($u$,$j$) returns true or false depending on whether $j$ is reachable from $u$

Reach($u$,$j$)
  if $PARENT[u, j]$==$Null$ then
    return false
  else
    return true
  end
end
```

Figure 3.21: Reach

- A sparse certificate, which is a sub-graph of C with the same nodes that conserves the connectivity between each pair of nodes of C. In other words, there is a path from i to j in C if and only if there is a path in its certificate.

A sparse certificate of a strongly connected component C is computed as follows:

1. Take any node r of C and perform DFS rooted at r in order to obtain the out-tree t of i in C.

2. Inverse the sense of the edges in C and perform DFS rooted at r again to obtain the inverse out-tree t' of i.

3. Restore the original direction of the edges in t' and assign $t \cup t'$ to the sparse certificate.

In order to delete edge $\langle i, j\rangle$ the following is done: if $\langle i, j\rangle$ belongs G' then the strongly connected components are not affected, so the decremental algorithm for computing transitive closure in acyclic graphs is used to update the data structures. If $\langle i, j\rangle$ does not belong to G' it means that the edge is in a strongly connected component C. In this case, if $\langle i, j\rangle$ does not belong to the sparse certificate of C, the edge is simply removed from C. If $\langle i, j\rangle$ belongs to sparse certificate and the connectivity of C is not affected by the removal of $\langle i, j\rangle$, the certificate is rebuilt.

If the connectivity is affected, then C is replaced by the components into which it splits and the data structure are updated accordingly.

The total complexity of this decremental algorithm for computing transitive closure in general graph is $O(n^2+nm_0)$ (as shown in [FMNZ01]), which is slightly higher than the complexity of the algorithm not involving cycles ($O(nm_0)$) when the graph is not dense, i.e., when the number of edges is proportional to the number of nodes. This small penalization is due to the update of the data structures when a component splits into several sub components.

3.4.2 Roditty's algorithm

Roditty's algorithm [Rod03] also maintains a forest of trees, but it does not keep a tree per node. This algorithm maintains two trees per insertion operation. The edges inserted in each insertion operation are centered on a given node, i.e., all the edges are incident edges of the same node.

The two trees associated with an insertion operation whose center node is u correspond to the in-tree and the out-tree of u in the resulting graph. Nodes are grouped into blocks that correspond to the strongly connected components of the graph that is obtained just after the insertion. In what follows, G_i will be the graph composed of the two trees that are obtained after the $i-th$ insertion operation, and B_i will be the set of blocks of G_i.

Center nodes are not repeated. A center node is associated with only one graph. When inserting a set of edges whose center node has been already used, the graph corresponding to the previous insertion operation is removed, the indexes of the graphs and blocks updated, and the trees that are associated with the new insertion operation are built taking into account both set of edges: the edges of the graph that has been removed and the edges that are being inserted.

The trees of the forest evolve in a decremental way. When deleting a set of edges from the current graph, blocks that are no longer subset of a strongly connected component in the current graph are replaced by the sub blocks that constitute them. When a set of edges is inserted, the new trees are created without modifying the previous trees.

As only edges are added/removed, the set of nodes remains constant. Let N be the set of nodes. G_0 is the graph with no edges whose set of nodes is N, and B_0 is $\{\{x\} : x \in N\}$.

If the set of edges of the initial graph is not empty, one can assume that the initial forest is set up by inserting edges in groups of incident edges of a given node. In order to minimize the number of trees, bigger sets of incident edges are inserted first as done in Figure 3.22.

Due to the fact that G_{i-1} is a sub graph of G_i, the blocks of G_i are composed of blocks of G_{i-1}. So for any index k, it is always the case that there are at most $2|N|-1$ different blocks in $\bigcup_{i=1}^{k} B_i$.

Figure 3.23 shows Roditty's Algorithm. This algorithm uses the following data:

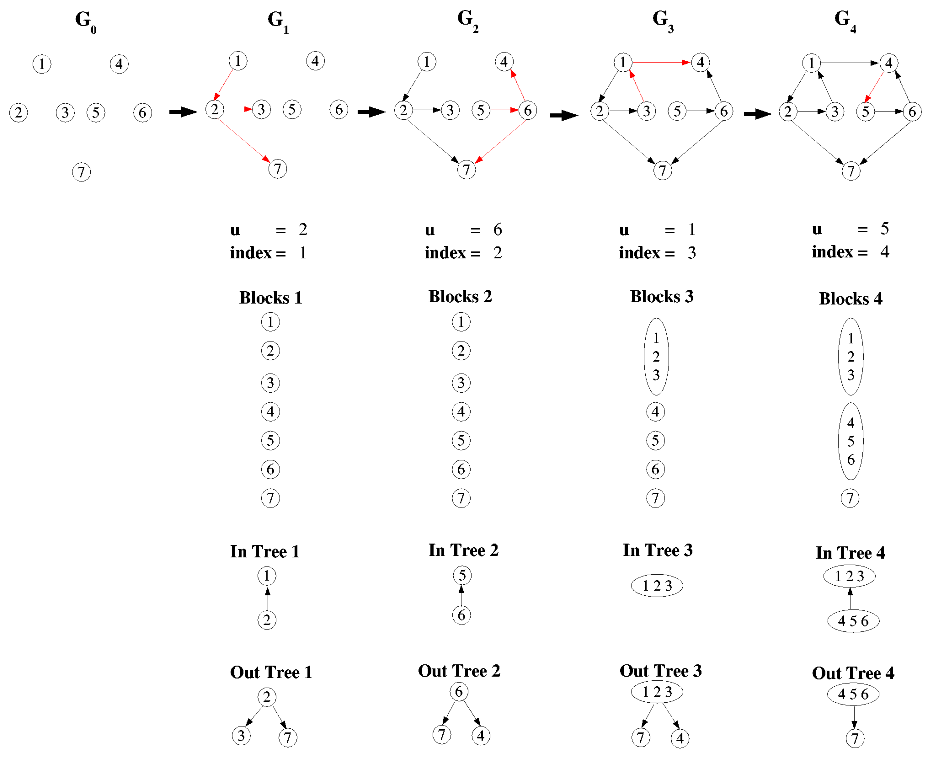


Figure 3.22: Setting up the forest in order to use Roditty's algorithm

- n is $|N|$ (the number of nodes of the current graph).
- $Index[k]$ is the center node of G_k.
- $Forest[i]$ represents G_i. Each forest contains the following data structures:
  - $A[i,j]$ is the index of edge $\langle i,j\rangle$. $A[i,j]$ is ∞ if $\langle i,j\rangle$ has not been inserted.
  - $M[i,j]$ is the index of edge $\langle i,b\rangle$, where b is the sub block of j that is connected to i through the edge with smallest index.
  - $col[i]$ is the column of block i in M.
  - $row[i]$ is the row of block i in M.
  - $elem[i]$ are the nodes of block i.
  - $subblock[i]$ are the sub blocks of block i.

Let us elaborate on the role of each functions of the algorithm:

- *InitForest* just initializes $Index$.
- *BuildMatrix* builds the forest dynamically. $Forest[i]$ is created in terms of $Forest[i-1]$. If a block b in $Forest[i]$ already existed in $Forest[i-1]$ ($|subblock[j]| = 1$), then its adjacency information is copied from $Forest[i-1]$ to $Forest[i]$. If b has been formed in $Forest[i]$ ($|subblock[j]| > 1$), then it is connected to every node v for which there is a sub block b' of b such that b' is connected to v ($M[v,m] := min_{b \in subblock[j]} M[v, Forest[i-1].col[b]]$).
- *InsertF(E_u,u)* adds G_k, where u is $Index[k]$. If there is a graph in the forest associated with the same center node, that graph is removed and its edges are added to G_k.
- *DeleteF(E')* re-builds the forest dynamically. Blocks that cease to exist are marked as unreachable. The new blocks that are formed after the removal are marked as "pending" ("*"). *ReconnectTree* will decide whether the blocks associated with "*" are reachable or not.

Even though the algorithm does not keep an out-tree per node, the information in the kept trees is enough to compute the current condensation graph (and therefore the current transitive closure). This is due to the following facts:

Lemma 1. *The current set of strongly connected components Cs is always a subset of $\bigcup_{i=1}^{k} B_i$.*

Proof. This is trivially true for strongly connected components of size one. For strongly connected components containing more than one node, we have to take into account that they are formed after inserting edges, so, for every strongly connected component, there must be one insertion operation that gave place to the component. □

Lemma 2. *If $\langle x, y \rangle$ is an inter component edge in the current graph, then there is a G_i having $\langle Block(x), Block(y) \rangle$ as one of its edges, where $Block(v)$ is the block containing node v in G_i.*

Proof. The only edges that are condensed are the intra-component ones. So, if a inter-component edge is added, it remains in the corresponding tree until it is removed. □

The total running time of the algorithm presented in Figure 3.23, after performing (ins+del) operations, is $O((ins + del)n^2)$. This means each update operation is computed in n^2.

In order to answer queries regarding the existence of a path between two given nodes, a matrix $count$ is maintained following the approach suggested in [Kin99]. $count[i, j]$ is the number of insertion centers that lie on a path between i and j. Certainly, if $count[i, j]$ is greater than zero, it means that there is at least one G_i with insertion center u such that u is reached from i and j is reached from u.

When adding a new G_i to the forest, $count$ is updated by considering the Cartesian product $InN \times OutN$, where InN and $OutN$ are the nodes of the out-tree and the in-tree composing G_i respectively. Each $count[i, j]$, $\langle i, j \rangle \in InN \times OutN$, is incremented in one because there is at least one more path connecting i and j.

If a set of edges is removed, the algorithm considers the set of couples that no longer belong to $InN \times OutN$, for each G_i. For each of those couples, the corresponding cell in $count$ is decremented in one.

Answering each reachability query takes constant time because it consists in looking up the corresponding cell in the matrix. Updating $count$ after an insertion operation is $O(n^2)$ because we only need to consider the graph that is being inserted. Updating $count$ after a delete operation is $O(n^2)$ in average. Even though it is true that updating $count$ with respect to a particular G_i is $O(n^2)$, it is also true that G_i evolves decrementally. This means that updating $count$ with respect to G_i gets cheaper each time edges are removed. In the extreme case, it will take constant time to update $count$, so in average, we may say that the complexity, with respect to a given G_i is linear.

3.5 Algorithms used in the current implementation of DomReachability

As explained in sections 3.2.2 and 3.2.3, in the implementation of *DomReachability*, we need to maintain the upper bound of the transitive closure graph and the lower bound of the extended dominator graph.

We use Lengauer and Tarjan's algorithm for updating the lower bound of the extended dominator graph. This choice is based on the fact that this is the algorithm that works best in practice even though it does not have the smallest complexity [Geo05].

To update the upper bound of the transitive closure graph, we use the non decremental algorithm provided by Boost [LLS01]. This algorithm is shown in Figure 3.24. The basic idea is to update the set of reachable nodes of every node taking into account that a node reaches all the reachable nodes of its successors. As two nodes belonging to the same strongly connected component reach the same set of nodes, the algorithm works on the condensed graph of the graph given and updates the information accordingly after computing all the reachable sets. The complexity of this algorithm is O(N*E) in the worst case (i.e., the case where there is no component containing more than one node).

```
**InitForest:**
  $k := 0$, allocate an array $Index$ of size $n$

**BuildMatrix($Forest$):**
  $M[1:n, 1:n] := A, m := n$
  for $i := 1$ to $k$
    using $Forest[i]$.
      for $j := 1$ to $2n$
        if $|subblock[j]| = 1$ then $col[j] := Forest[i-1].col[subblock[j][1]]$
          if $|subblock[j]| > 1$ then
            $m := m+1$,$col[j] := m$
            for $v := 1$ to $n$
              $M[v,m] := min_{b \in subblock[j]} M[v, Forest[i-1].col[b]]$

**InsertF($E_u$,$u$):**
  if $\exists i$ such that $Index[i] = u$ then
    for $j := i$ to $k-1$
      $Index[j] := Index[j+1], Forest[j] := Forest[j+1]$
    else
      $k := k+1$ and add $Forest[k]$
  update arrays $subblock$ and $elem$ of $Forest[k]$ using $GetSCC(G)$
  $Index[k] := u$
  BuildMatrix($Forest$)
  ReconnectTree($Forest[k]$,$Index[k]$,$k$)

**DeleteF($E'$):**
  Update $A$ with $E'$ and $S := GetSCC(G)$
  for $i := 1$ to $k$
    $subblock' := UpdateBlocks(Forest[i].subblock, S)$
    for $j := 1$ to $2n$
      if $Forest[i].subblock[j] \neq NULL$ and $subblock'[j] = NULL$ then
        $Forest[i].reach[j] := 0$
      if $Forest[i].subblock[j] = NULL$ and $subblock'[j] \neq NULL$ then
        $Forest[i].reach[j] := *$
    $Forest[i].subblock := subblock'$
  BuildMatrix($Forest$)
  for $i := 1$ to $k$ do ReconnectTree($Forest[i]$,$Index[i]$,$i$)
```

Figure 3.23: Roditty’s Algorithm

```
**Pre :**
    $G = \langle N, E \rangle$ is a directed graph
    $Succ(i) = \{j \mid \langle i, j \rangle \in Edges(G)\}$
    $Rep(i)$ is the node that represents the strongly connected
        component containing $i$.
    $CondenseGraph(G)$ is the condensed graph of $G$
**Post :**
    $Reach(i)$ is the set of nodes reachable from $i$ in $G$

TC(G)
    $G' = CondenseGraph(G)$
    for each node $u$ in $Nodes(G')$ in reverse topological order
        for each node $v$ in $Succ(i)$
            if ($v$ not in $Reach(u)$)
                $Reach(u) := Reach(u) \cup \{v\} \cup Succ(v)$
    for each $u$ in $Nodes(G)$
        if $u \neq Rep(u)$
            $u := Reach(Rep(u))$
end
```

Figure 3.24: Boost's Transitive Closure Algorithm

Chapter 4

Implementing DomReachability in Gecode using CP(Graph)

Gecode [SLT06] is a C++ library that provides an environment for developing constraint-based systems and applications. *Gecode* allows the construction of new variable domains including propagators as implementations of constraints and branchings, and search engines.

Search in *Gecode* is based on recomputation and copying, which allows the implementation of advanced search engines like adaptive search engines and search engines on top of batch recomputation. In fact, the use of batch recomputation drastically reduces the propagation time during recomputation.

Gecode offers finite domain constraints and finite set variables implemented on top of its generic kernel. Thanks to the way *Gecode* has been designed, it is simple to add new computation domains. *CP(Graph)* [DZDD06] is a new computation domain that has been added to Gecode.

In this chapter we will make a summary of the most relevant concepts in *Gecode*. Then, we will show how propagators are implemented in Gecode by explaining the implementation of *Distinc*: one of the propagators provided by *Gecode*. After explaining how to deal with *CP(Graph)* in *Gecode*, we will present the implementation of the ad-hoc propagator of *DomReachability* sketched in Figure 5.4. In section 4.6, we will show the implementation of the labeling strategy we have designed to deal with *OSPMN* instances.

Taking into account that there is not a tutorial in *Gecode* that explains how to implement a propagator, the explanation of the implementation of the *Distinct* constraint would be appreciated by those trying to implement a propagator for the first time.

4.1 Basic concepts in Gecode

In this section we will present a set of definitions (which are given in [SLT06]) that are fundamental for the presentation of the next sections.

4.1.1 Actor

An actor is either a branching (see section 4.1.2) or a propagator (see section 4.1.7). Actors provide the functionality that is common for propagators and branchings such as member functions for copying during cloning, memory allocation, and so on.

4.1.2 Branching

A branching defines the shape of the search tree. Branchings are also known as labeling or distributors, and a branching creates a series of choice points.

4.1.3 Branching description

A branching description speeds up recomputation by providing batch recomputation. It is created by a branching (see section 4.1.2) and allows to replay the effect of that branching without the need to first perform constraint propagation.

4.1.4 Computation space

A computation space (space for short) comprises all entities for a constraint problem to be solved, including all actors (see section 4.1.1) and variables (see section 4.1.9). A space can be seen as corresponding to a node in the search tree. It organizes constraint propagation, the branching process, exploration, and memory management.

4.1.5 Modification event

A modification event describes how a view (see section 4.1.11) or variable implementation (see section 4.1.10) is changed by an update operation performed on the view or variable. Each variable domain defines its own modification events. However modification events that describe generic events such as failure, no modification, or assignment to a single value are predefined (see Generic modification events and propagation conditions).

4.1.6 Propagation condition

A propagation condition defines when a propagator requires to be re-executed. Re-execution is controlled by the modification events that occur on the variables the propagator depends on (see section 4.1.7). Propagation conditions and the relation between propagation conditions and modification events depends on the variable domain.

4.1.7 Propagator

A propagator implements a constraint. Execution by a propagator is defined by its dependencies: the views (referring to some variables) together with their propagation conditions.

4.1.8 Propagator modification event

A propagator maintains for each variable domain a modification event. This event is called a *propagation modification event*. These modification events describe which modification events occurred on all views (variables) the propagator depends on. A propagator modification event is available through a view or variable implementation.

4.1.9 Variable

A variable is used for modeling problems, be it for direct modeling or for modeling through some interface. A variable provides only those operations useful for modeling and excludes in particular operations that can modify the variable domain directly. A variable is implemented by a variable implementation (see below).

4.1.10 Variable implementation

A variable implementation implements the variable domain and provides operations to access and modify the domain.

4.1.11 View

A view offers essentially the same interface as a variable implementation and allows both domain access and modification. Typically, several views exist for the same variable implementation to obtain several constraints from the same propagator.

4.2 CP(Graph) in Gecode

As explained in [DZDD06, Doo06], CP(Graph) defines a new computation domain in constraint programming: graph domain variables and constraints over these variables. The implementation of graph variables use the "view" concept of Gecode [ST06]. One view implements a graph as a set of nodes and a set of edges, the other view uses a set of nodes and N sets of adjacent nodes. Some constraints, such as $Complement(G1, G2)$, $Path(G, n1, n2)$ and $Path(G, n1, n2, I, w)$, are also provided.

In Figure 4.2, we show the creation of a graph variable (line 1) and the elimination of the edge $\langle 0, 1\rangle$ from its upper bound (line 2). In this case, we are using

```
template <class View0, class View1>
class Distinct :
  public InhomBinaryPropagator<...> {
protected:
  using InhomBinaryPropagator<...>::x0;
  using InhomBinaryPropagator<...>::x1;
  /// Constructor for cloning p
  Distinct(Space* home, bool share,Distinct& p);
  /// Constructor for posting
  Distinct(Space* home,View0,View1);
public:
  /// Copy propagator during cloning
  virtual Actor*      copy(Space* home,bool);
  /// Perform propagation
  virtual ExecStatus  propagate(Space* home);
  /// Post propagator  x \neq y
  static  ExecStatus  post(Space* home,View0,View1);
};

template <class View0, class View1>
class DistinctDoit :
  public UnaryPropagator<View0,PC_SET_ANY> {
protected:
  using UnaryPropagator<View0,PC_SET_ANY>::x0;
  /// The view that is already assigned
  View1 y;
  /// Constructor for cloning \a p
  DistinctDoit(Space* home, bool share,DistinctDoit&);
  /// Constructor for posting
  DistinctDoit(Space* home, View0, View1);
public:
  /// Copy propagator during cloning
  virtual Actor*      copy(Space* home, bool);
  /// Perform propagation
  virtual ExecStatus propagate(Space* home);
  /// Post propagator  x \neq y
  static ExecStatus post(Space* home, View0, View1);
};
```

Figure 4.1: Partial Definition of Distinct and DistinctDoit. The complete code can be obtained from Gecode's web site (*http://www.gecode.org*)

```
fg=OutAdjSetsGraphView(this,fg_ub);
GECODE_ME_FAIL(this, fg._arcOut(this,0,1));
```

Figure 4.2: Creating a graph variable and removing edge $\langle 0, 1\rangle$ from its upper bound

```
1  // J in tcg.outN(I) => I not in tcg.outN(J)
2  for (int i=0;i<fg_ub_numNodes;i++){
3    for (int j=0;j<fg_ub_numNodes;j++){
4      if (i!=j){
5        BoolVar a(Space,0,1); //a=1 means J in tcg.outN(I)
6        BoolVar b(Space,0,1); //b=1 means I in tcg.outN(J)
7
8        dom(Space, tcg.outN[i], SRT_SUP, j, a);
9        dom(Space, tcg.outN[j], SRT_SUP, i, b);
10       post(Space, ff(a && b));
11
12     }
13   }
14 }
```

Figure 4.3: Accessing the adjacency set of a node in the imposition of an antisymmetric relation

the view OutAdjSetsGraphView which associates each node with its set of outgoing nodes. fg.outN[i] refers to the FS variable representing the outgoing nodes of node i in fg.

In Figure 4.3, we show an example where we are accessing the FS variable associated with each node of the graph variable tcg. In this particular case, we are imposing an antisymmetric relation among the nodes of tcg, i.e., if edge $\langle i, j\rangle$ is in tcg, edge $\langle j, i\rangle$ is not. In order to impose this relation, we are reifying the presence of a node in the set of outgoing nodes of another one. For instance, in line 8 we are reifying the present of node j in the set of outgoing nodes of i in the Boolean variable a. This means that a is equal to 1 if and only if j is in the set of outgoing nodes of i. Once we have reified the presence of the corresponding edges, we impose the antisymmetric relation through the instruction post(Space, ff(a && b)), which means that it can not be true that both edges are part of the graph.

```
template <class View0, class View1>
ExecStatus
Distinct<...>::post(Space* home, View0 x, View1 y) {
  if (x.assigned())
    GECODE_ES_CHECK((DistinctDoit<...>::post(home,y,x)));
  if (y.assigned())
    GECODE_ES_CHECK((DistinctDoit<...>::post(home,x,y)));
  (void) new (home) Distinct<...>(home,x,y);
  return ES_OK;
}
```

Figure 4.4: Implementation of method post of class Distinct
.

```
transitive_closure(fg_ub,fg_ub_tc);
list< pair<int,int> > tcg_ub_delta;

for(;tcg_ub();++tcg_ub){
  int s=tcg_ub.val().first;
  int d=tcg_ub.val().second;
  if (!(tcg_vs[s] && tcg_vs[d] &&
        edge(s,d,fg_ub_tc).second)){
      tcg_ub_delta.push_back(tcg_ub.val());
  }
}

ItValEdges
tcg_ub_delta_it(tcg_ub_delta.begin(),tcg_ub_delta.end());
GECODE_ME_CHECK(this->g3._arcsOut(home,tcg_ub_delta_it));
```

Figure 4.5: Pruning the upper bound of the transitive closure of *DomReachability*

4.3 Implementing user-defined propagators in Gecode

In order to introduce the framework, in this section we present the implementation of the Distinct propagator for set variables provided by *Gecode*[1]. Given two set variables S_1 and S_2, $Distinct(S_1, S_2)$ holds if and only if the set approximated by S_1 is different to the one approximated by S_2.

Figure 4.1 shows the definition of class Distinct that implements *Distinct*.

[1]The source code presented in this section has been taken from the source code of *Gecode* [SLT06]. However, the code has been commented by the author of this thesis.

Distinct is implemented in terms of *DistinctDoit*. Once one of the set variables is instantiated, *Distinct* launches *DistinctDoit* which is the one that actually performs the pruning on the non-instantiated variable [2] .

A class implementing a propagator in *Gecode* always has at least the methods composing the classes in Figure 4.1. Notice that each class has two constructors: one for creating the propagator and another one for cloning the propagator when the space to which the propagator belongs is being cloned. When cloning the space, the search engine invokes method copy, and copy clones the propagator by invoking the constructor for cloning.

The propagator is created when the method post is invoked. However, in some cases like the one of method post of class Distinct (see Figure 4.4), the propagator may reduce to another one depending on the status of its argument at the moment of posting the propagator. Notice that, if set x is already determined, instead of creating a *Distinct* propagator, a *DistinctDoit* propagator is created. The same situation occurs if set y is the one that is already instantiated.

The most important method in a class defining a propagator is the method *propagate*. This method may return the following values:

- ES_FAILED, if the current domains of the variables violates the constraint implemented by the propagator.
- ES_SUBSUMED, if the current domains of the variables entails the constraint implemented by the propagaror.
- ES_FIX, if the the current instantiation of the variables neither violates nor entail the constraint implemented by the propagator [3].

In Figure 4.10, we are showing the propagate method of class DistinctDoit . As *DistinctDoit* is launched when one of the set is already instantiated, the first thing that *propagate* does is to check whether the non-instantiated set variable (x0) has been instantiated (line 6). If this is the case, *propagate* returns either ES_FAILED or ES_SUBSUMED depending on whether the set variables have been instantiated to the same set or not. If x0 has not been instantiated yet, we check whether the cardinality of the sets is already known to be different (lines 14 and 15). If this is so, ES_SUBSUMED is returned. If the cardinalities might be equal, we check whether one of the set is already known to be not contained in the other one (lines 21 and 25). Notice that, if set x0 is not contained in set y, there is at least one element in x0 that is not in y, which means that the two sets are different and therefore that ES_SUBSUMED should be returned. If the previous checks

[2]In Figure 4.1, the template arguments in the Definition of Distinct (<View0,PC_SET_VAL,View1,PC_SET_VAL>) have been omitted in order to respect the margins).

[3]When the methods returns ES_FIX, it usually means that the fix point has been reached,i.e., that no further pruning on the domains can be performed by the propagator. However, propagators in *Gecode* are not required to be idempotent, i.e., it is not mandatory to reach the fix point when the method is invoked.

have returned neither ES_FAILED nor ES_SUBSUMED, the upper bound of x0 contains the instantiated set (y) and y contains the lower bound of x0. So, if the cardinality of the upper bound x0 is the same as the cardinality of y (line 30), the propagator states that the cardinality of x0 is less than the cardinality of y since, otherwise, the variables would be assigned to the same set. If the cardinality of the lower bound of x0 is the same as the cardinality of y (line 37), the propagator states that cardinality of x0 is greater than the cardinality of y in order to avoid that the variables are assigned to the same value. Notice that in the two last cases *propagate* does not need to be called again. Indeed, we can say that, after stating that the cardinality of x0 is less/greater than the cardinality of y, the propagator is entailed since the rest of the job will be done by the cardinality propagator.

4.4 Implementing DomReachability

We split the implementation of *DomReachability* into two part: the one implemented in terms of the FS propagators provided by *Gecode*, and the one that requires the implementation of an ad-hoc propator in *Gecode*.

4.4.1 Using Gecode's FS propagators

We use pseudo-code for the presentation of the implementation of the rules of DomReachability that are implemented on top of the propagators already provided by *Gecode*. The translation of this pseudo-code into the actual *Gecode* code is straightforward. For instance, the code in Figure 4.6 is the actual *Gecode* code associated with $|tcg.outN(i)| > 0 \rightarrow i \in fg.nodes$.

```
IntVar card_tc_i(Ex,0,n);
IntVar int_i(Ex,i,i);
BoolVar bool_i_a(Ex,0,1);
BoolVar bool_i_b(Ex,0,1);

cardinality(Ex,tcg.outN[i],card_tc_i);
rel(Ex,card_tc_i,IRT_GR,cero,bool_i_a);
rel(Ex,int_i,SRT_SUB,fg.nodes,bool_i_b);
bool_imp(Ex,bool_i_a,bool_i_b,boolOne);
```

Figure 4.6: Actual *Gecode* code associated with $|tcg.outN(i)| > 0 \rightarrow i \in fg.nodes$

Transitive closure of *DomReachability* (Rules 3.8 and 3.14)

$$|tcg.outN(i)| > 0 \rightarrow i \in fg.nodes \tag{4.1}$$

$$i \in fg.nodes \rightarrow i \in tcg.outN(i) \tag{4.2}$$

Statement 4.1 imposes an implication between the cardinality of $tcg.outN(i)$ being greater than 0 and the presence of node i in fg, i.e., a node should be part of the flow graph in order to reach another one.

Statement 4.2 imposes an implication between the presence of i in fg and i reaching itself. This is because every node of fg reaches itself.

$$j \in tcg.outN(i) \rightarrow tcg.outN(j) \subseteq tcg.outN(i) \tag{4.3}$$

Statement 4.3 imposes an implication between i reaching j, and $tcg.outN(j)$ being a subset of $tcg.outN(i)$.

Pruning the upper bound of $RN(i)$ (Rule 3.10)

We first have to ensure that, for every node i that is already known to belong to fg, $tcg.outN(i)$ gets determined when i has no successors:

$$|fg.outN(i)| = 0 \leftrightarrow |tcg.outN(i)| \leq 1 \tag{4.4}$$

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

$$tcg.outN(i) = \{i\} \cup \bigcup_{j \in fg.outN(i)} tcg.outN(j) \tag{4.5}$$

This is all what is needed for pruning a flow graph without cycles since the sets of reached nodes of the leaves get bound because of Statement 4.4, and this information is propagated to the corresponding predecessor because of Statement 4.5.

However, if fg has cycles, tcg do not get determined even if fg is already determined. For instance, suppose that the lower and upper bound of fg is the graph $\langle\{1,2,3\},\{\langle 1,2\rangle,\langle 2,1\rangle\}\rangle$. The propagators above mentioned will basically constrain $tcg.outN(1)$ to be equal to $tcg.outN(2)$ (and $tcg.outN(3)$ to be $\{3\}$). Additionally, due to Statements 4.1 and 4.2, nodes 1 and 2 get into the lower bound of $tcg.outN(1)$ and $tcg.outN(2)$. However, no propagator removes 3 from the upper bound of neither $tcg.outN(1)$ and $tcg.outN(2)$. Updating the upper bound of tcg and the lower bound of edg is the task of the ad hoc propagator presented in the following section.

4.4.2 The ad hoc propagator of DomReachability

Figure 4.7 shows the skeleton of the *Gecode* implementation of the propagate method of DomReachAdHocPropag.

The first thing to notice is that this propapagator is a propagator on two graph variables: `fg` and `tcg`. DomReachAdHocPropag is awaked when the domain of one of its arguments have changed. As the operations performed in the propagate method are expensive, the highest cost to DomReachAdHocPropag is assigned so that it is scheduled only when all the other propagators have reached their fix point.

```
template <class GD_FG, class GD_TCG>
ExecStatus
DomReachAdHocPropag<...>::propagate(Space* home) {

  // *******************
  // fg<->tcg interaction
  // *******************

  // 1. Computing the transitive closure of
  //    fg's upper bound (TC(fg_max))
  // 2. Pruning tcg's upper bound based on TC(fg_max)



  //*******************
  //fg<->edg interaction
  //*******************

  // Computing the extended dominator tree
  // of fg's upper bound (EDT(fg_max))


  //*******************
  //edg<->tcg interaction
  //*******************

  // Pruning tcg's lower bound based on EDT(fg_max)


  // The propagator is entailed when the flow graph
  // is assigned
  if (this->fg.assigned())
    return ES_SUBSUMED;

  return ES_FIX;
}
```

Figure 4.7: Skeleton of the Ad Hoc propagator of *DomReachability*

In the implementation of the propagate method we basically have three parts:

- The interaction between the flow graph which consists in updating the upper bound of the transitive with respect to the transitive closure of the upper bound of the flow graph.
- The computation of the extended dominator tree of the upper bound of the flow graph. In the computation of this tree we took advantage of the already existing implementations of algorithms for computing dominators which are publicly available [WG04, GWT$^+$04].
- The pruning of the lower bound of the transitive closure taking into account the information in the extended dominator tree of the upper bound of the flow graph.

Figure 4.5 shows a fragment of the implementation of the propagate method of DomReachAdHocPropag. In the implementation of propagate we have taken advantage of the algorithms provided by *The Boost Graph Library (BGL)* [LLS01] and *The C++ Standard Library* [Str97, Jos99, Eck00, Eck03]. For instance, in line 1, the transitive closure of the upper bound (fg_ub_tc) is computed by using the transitive_closure function provided by *BGL*.

In order to prune the upper bound of the transitive closure (tcg_ub), we traverse the list of edges in tcg_ub) and see whether there are edges that should be removed. An edge is removed from tcg_ub if it is not in fg_ub_tc. The update of the upper bound is optimized by collecting the edges in the list tcg_ub_delta and removing them in one single operation (line 15).

4.5 Pseudo-optimizing rules by using ad hoc propagators

Let us consider again the imposition of the antisymmetric relation presented in Figure 4.3. Notice that, if n is the number of nodes of the upper bound of the transitive closure graph, we launch $O(n^2)$ sub set propagators. Let us compare this option with the option of implementing an ad hoc propagator that achieve the same level of pruning. We show this option in Figure 4.9.

Even though in the second option seems more efficient (with respect to the computation time) since there is only one propagator taking care of the filtering, it performs worse. The reason is that we still need to traverse the whole lower bound even in cases where the size of the delta (i.e., the edges added) is pretty small when the propagator is activated. Notice that the number of edges in the lower bound is $O(n^2)$.

The propagators launched in the first option are constant. Indeed, after one of those propagator is awaken it takes constant time to execute the corresponding propagate method since it only consist in checking whether an element is in a set [4].

[4]We are assuming that there are not holes in the set. In general, the complexity of checking whether an element is in a set in Gecode is $O(h)$, where h is the number of holes in the set

```
template <class SPACE, class GV1, class GV2>
class OSPMNBranching : public Branching {
protected:
  GV1 fg;
  GV2 tcg;
  GBD *bd;
  map<pair<int,int>,int> dists;
  list<int> pendMandNodes;

  OSPMNBranching(Space* home, bool share, OSPMNBranching& b);
public:
  // Constructor for creation
  OSPMNBranching(Space* home, GV1 &fg, GV2 &tcg,
                     map<pair<int,int>,int>&
                     dists, list<int>& pendMandNodes);
  std::pair<bool,GBD*> getOption(Branching *b);
  // Perform branching (selects view)
  virtual unsigned int branch(void);
  // Return branching description
  virtual BranchingDesc* description(void);
  // Perform commit for alternative a and branching
  // description d
  virtual ExecStatus
  commit(Space* home, unsigned int a, BranchingDesc* d);
  // Perform cloning
  virtual Actor* copy(Space* home, bool share);
};
```

Figure 4.8: Definition of the branching implementing the A* labeling strategy used for solving *OSPMN* instances

So, when the outgoing set of nodes of a given node is modified $O(n)$ propagators are awaken, which leads to an overall complexity of $O(n)$.

4.6 Implementing user-defined labelling strategies in Gecode

In this section we will explain the labelling strategy used to solve the instance of the Disjoint Paths Problem reported in [QVDC06]. As explained in section 2.3.3, we translated the Disjoint Paths instance into a OSPMN instance. So, as our aim is to find a path where the mandatory nodes are visited in the given order, our labelling

```
list < pair<int,int> > tcg_ub_delta;

for(;tcg_lb();++tcg_lb){
   int s=tcg_ub.val().first;
   int d=tcg_ub.val().second;
   tcg_ub_delta.push_back(make_pair(d,s));
}

ItValEdges
tcg_ub_delta_it(tcg_ub_delta.begin(),tcg_ub_delta.end());
GECODE_ME_CHECK(this->g3._arcsOut(home,tcg_ub_delta_it));
```

Figure 4.9: Pseudo-optimizing the imposition of the antisymmetric relation of figure 4.3

strategy builds the path incrementally starting from the source. At each labeling step, we choose the node that is closer to the next mandatory node to be reached. In fact we can think of our heuristic as a kind of A* heuristic [RN03].

In Figure 4.8, we show the definition of class OSPMNBranching: the class implementing the labeling strategy we just described. A class implementing a labeling strategy in *Gecode* must be a subclass of the class Branching.

OSPMNBranching has the following attributes:

- fg: the view associated with the flow graph.
- tcg:the view associated with the transitive closure graph.
- bd: a pointer to the descriptor to be considered in the next commit operation.
- dists : the matrix of distances between nodes.
- pendMandNodes: the list of pending mandatory nodes. The mandatory nodes in $pendMandNodes$ appear in the order they should be visited, so the first node in pendMandNodes is the next mandatory node to be reached.

A branching also has two constructors: one for launching the branching and another that it is used when the space has to be copied. Apart from these two constructors, a branching has the following methods:

- branch computes the information on which the creation of the descriptor to be used in the commit operation is based. The descriptor define the options associated with the choice point created when the commit operation is performed. The number options can be zero meaning that no choice point should be created.

- description creates the descriptor taking into account the information computed by branchbranch

- commit creates the choice point based on the descriptor created by the description method.

- copy creates a copy of the branching by calling the constructor for cloning.

Descriptors are needed for implementing *batch recomputation* in *Gecode* since this allows to consider a set of decision at once by applying the conjunction of the descriptors associated with each decision. Before explaining the concept of *batch recompuptation*, let us first refer to *recomputation*.

As explained in [Sch00], search demands that nodes of the search tree must possibly be available at a later stage of exploration. A search engine must take precaution by either memorizing nodes or by means to reconstruct them. States are memorized by cloning. Techniques for reconstruction are trailing and recomputation. While recomputation computes everything from scratch, trailing records for each state-changing operation the information necessary to undo its effect. The basic idea of recomputation is to compute a node in the search tree from the root node of the search tree and a description of the node's path p. One option to compute such a node is to do it in n steps, where n is the length of p. This basically means to perform a commit operation per edge in p. Another option is to do it all at once by considering the conjunction of the descriptors along the path. The second option corresponds to the notion of *batch recompuptation* in *Gecode*.

Apart from the methods already introduced, in OSPMNBranching, we have the extra method getOption, which choose the edge e on which the choice point will be defined. The first option of the choice point is to include e to the lower bound. The second one is to exclude e from the upper bound.

The implementation of getOption is shown in Figure 4.11. We first initialize the map of nodes to their out-going degrees in the lower bound of the transitive closure graph. This map is kept in tcg_lb_od, i.e., tcg_lb_od[i] is the number of out-going edges of node i in the lower bound of the transitive closure graph. Then, we traverse the list of un-known edges of the flow graph (i.e., edges that are in the upper bound but not in the lower bound) in order to pick the edge whose source is the node that reaches the most nodes and destination is the node that is closer to next mandatory node.

Notice that by choosing the edge whose source is the node that reaches the most nodes,i.e., the node with the higher out-degree in the lower bound of the transitive closure, we incrementally build the path from the source to the destination. At the beginning of the search, the source of the edge chosen is the source of the path ($source(path)$) since $source(path)$ reaches itself, the mandatory nodes and the destination node. After choosing the first edge of the path,the source of the next edge chosen is the successor of source(path) ($suc(source(path))$) since $suc(source(path))$ is the node that reaches most nodes. Notice that $suc(source(path))$

reaches all the nodes that $source(path)$ reaches excepting $source(path)$. The situation will be the same after choosing the i_{th} edge of the path, which warranties that the path is incrementally built from the source to the destination.

4.7 Contribution to Gecode

We are extending the list of graph propagators already available in *Gecode(CP(Graph))* [DZDD06] by integrating the implementation of *DomReachability*. Documentation on how to use *DomReachability* and examples reproducing the results published in this thesis will be available through *CP(Graph)*'s web site:

http://cpgraph.info.ucl.ac.be

```
template <class View0, class View1>
ExecStatus
DistinctDoit<View0,View1>::propagate(Space* home) {
  // Testing whether the two sets have been assigned to
  // the same set
  if (x0.assigned()) {
    GlbRanges<View0> xi(x0);
    GlbRanges<View1> yi(y);
    if (Iter::Ranges::equal(xi,yi)) {return ES_FAILED;}
    else { return ES_SUBSUMED; }
  }
  // Testing whether the cardinality of the two sets is
  // already known to be different.
  if (x0.cardMin()>y.cardMax()) { return ES_SUBSUMED; }
  if (x0.cardMax()<y.cardMin()) { return ES_SUBSUMED; }
  // Testing whether ~(y \subseteq lub(x)) or
  // ~(glb(x) \subseteq y).
  // In both cases the propagator is entailed.
  GlbRanges<View0> xi1(x0);
  LubRanges<View1> yi1(y);
  if (!Iter::Ranges::subset(xi1,yi1))
    {return ES_SUBSUMED;}
  LubRanges<View0> xi2(x0);
  GlbRanges<View1> yi2(y);
  if (!Iter::Ranges::subset(yi2,xi2))
    {return ES_SUBSUMED;}
  // At least one element from X0's upper bound should
  // be removed in order to ensure that the two sets are
  // different.
  if (x0.lubSize() == y.cardMin() && x0.lubSize() > 0) {
    GECODE_ME_CHECK(x0.cardMax(home, x0.lubSize() - 1));
    return ES_SUBSUMED;
  }
  // At least one element from X0's upper bound should
  // be added in order to ensure that the two sets are
  // different.
  if (x0.glbSize() == y.cardMin()) {
    GECODE_ME_CHECK(x0.cardMin(home, x0.glbSize() + 1));
    return ES_SUBSUMED;
  }
  return ES_FIX;
}
```

Figure 4.10: Implementation of method propagate of class DistinctDoit

```
template <class SPACE, class GV1, class GV2>
std::pair<bool,GBD*>
OSPMNBranching<SPACE,GV1,GV2>::getOption(Branching *b){

  // Initializing out-going degree for the nodes of the
  // lower bound of the transitive closure (tcg_lb_od[i])
  // -  tcg_lb_od[i] is the out-degre of i in


  //Traversing the list of  edges in order to pick best one
  BGL_FORALL_EDGES(e, b_fg.UB, GG){
    int sid = b_fg.UB[source(e,b_fg.UB)].id;
    int tid = b_fg.UB[target(e,b_fg.UB)].id;

    // If edge <sid,tid> is an unknown edge:
    if (!(b_fg.LB_v[sid] && b_fg.LB_v[tid] &&
          edge(b_fg.LB_v[sid],b_fg.LB_v[tid],b_fg.LB).second)){
      if (tcg_lb_od[sid]>maxDegree){
        bestedge = make_pair(sid,tid);
        maxDegree=tcg_lb_od[sid];
        minDist=dists[make_pair(tid,pendMandNodes.front())];
      }
      else
        if (tcg_lb_od[sid]==maxDegree){
          int newMinDist=
              dists[make_pair(tid,pendMandNodes.front())];
          if (minDist>newMinDist){
            bestedge = make_pair(sid,tid);
            minDist=newMinDist;
          }
        }
    }
  }

  if (bestedge == make_pair(-1,-1))
    return make_pair(false,(GBD*)NULL);

  return make_pair(true,new GBD(b,bestedge,true));
}
```

Figure 4.11: Implementation of method getOption of class OSPMNBranching

Chapter 5

Implementing DomReachability Using Message Passing in Oz

Chapter 5 of [VH04] presents message passing as a programming style that allows building highly reliable systems. This style of programming is based on the asynchronous communication of independent entities (agents). After extending the kernel language of Oz [Moz04] and giving the formal semantics of the new concepts introduced, the authors show how the behavior of each independent entity can be defined by using declarative functions.

In this chapter we will explain how we can implement *DomReachability* using a message passing approach on top of the multi-paradigm programming language Oz [Moz04] [1]. As we will show in this chapter, the use of a concurrent language like Oz for implementing global constraints involves the implementation of processes that are non-deterministic in general. This makes Declarative Concurrency not suitable for this need. By using the methodology introduced in [VH04], we will show that the definition of the behavior of the agents involved in the implementation of global constraints, and the non-determinism in the communication of these agents are two orthogonal concerns. This separation allows the behavior of each agent to be defined in a declarative way.

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

We will also present a cheap way of discovering bridges based on FS pruning, and introduce an approach for implementing Batch propagation using message passing, which plays an important role in the reduction of the time of execution thanks to the minimization of the number of activations of expensive propagators.

[1]The results presented in this chapter have been published in [QVD05a].

5.1 The DomReachability constraint

As we implement the propagator using FS variables, in this chapter we reformulate the definition of the constraint given in section 2.2.3. Here, we define the DomReachability constraint as follows:

$$DomReachability(g, source, rn, cn, be) \equiv \ \forall_{i \in N}. \begin{array}{l} rn(i) = Reach(g,i) \wedge \\ cn(i) = CutNodes(g, source, i) \wedge \\ be(i) = Bridges(g, source, i) \end{array} \tag{5.1}$$

Where:

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

$$j \in Reach(g,i) \leftrightarrow \exists_p . p \in Paths(g,i,j) \tag{5.2}$$

$$p \in Paths(g,i,j) \leftrightarrow \begin{array}{l} p = \langle k_1, ..., k_h \rangle \in nodes(g)^h \wedge k_1 = i \wedge k_h = j \wedge \\ \forall_{1 \le f < h} . \langle k_f, k_{f+1} \rangle \in edges(g) \end{array} \tag{5.3}$$

$$k \in CutNodes(g,i,j) \leftrightarrow \forall_{p \in Paths(g,i,j)} . k \in nodes(p) \tag{5.4}$$

$$e \in Bridges(g,i,j) \leftrightarrow \forall_{p \in Paths(g,i,j)} . e \in edges(p) \tag{5.5}$$

In this chapter cut nodes and bridges will refer to node dominators and edge dominators respectively. Notice that the two definitions of *DomReachability* are equivalent. The $source$ of Equation 5.1 is implicit in the flow graph fg of Equation 2.9. The tcg in equation 2.9 is represented by rn in Equation 5.1 since $rn(i)$ corresponds to the outgoing nodes of i in tcg. $i \in cn(j)$ means $\langle i,j \rangle \in edg$. Similarly, $e \in be(j)$ means $\langle e,j \rangle \in edg$.

The reader may think that Equation 2.9 is stronger than Equation 5.1 because the former associates an edge with its dominators. However, this information is implicit in Equation 5.1. Notice that, if we want to impose that node i dominates edge

$\langle j,k\rangle$ ($\langle i,\langle j,k\rangle\rangle \in edg$), it is enough to to force i to dominate j ($i \in cn(j)$) since any dominator of j is also a dominator $\langle j,k\rangle$. We can drop the same conclusion with respect to edge dominators.

5.2 The DomReachability propagator

We implement the constraint in Equation 5.1 with the propagator

$$DomReachability(G, Source, RN, CN, BE) \tag{5.6}$$

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 $\langle N_2, E_2\rangle$. 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) [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 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.

5.3 Using Message passing in Oz

We will define the DomReachability propagator using a concurrent functional language, namely the declarative subset of Oz. This language is a concurrent constraint language in the sense of Saraswat [Sar93]. 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

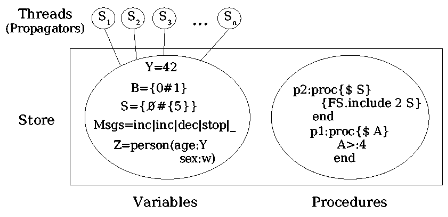


Figure 5.1: The Oz Execution Model (Declarative subset)

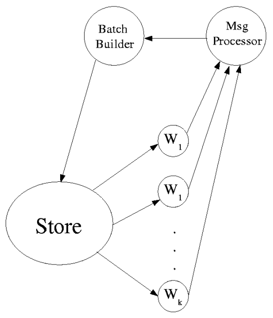


Figure 5.2: Architecture of a Graph variable propagator

example, in Figure 5.1 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.

```
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 5.3: A thread reading messages from a stream

For instance, in Figure 5.3, `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 [VH04]). An agent is supposed to receive messages from different threads, so the order in 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.

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** {X $Y_1 \ldots Y_n$} S **end** $\mid$ {X $Y_1 \ldots Y_n$}	Procedure
$\mid$ **if** X **then** S **else** S **end**	Conditional
$\mid$ **thread** S **end**	Thread

Table 5.1: The Oz declarative kernel language.

```
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 5.2. Notice that in this figure there is an additional component that we are going to introduce in section 5.4.3.

Each of the pruning rules of Chapter 3.2 can be implemented 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 [VH04]). 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 [VH04].

All Oz execution can be defined in terms of a kernel language whose semantics are given in chapter 13 of [VH04]. We will just refer to the declarative part of it.

Table 5.1 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.

5.4 Implementation of *DomReachability*

5.4.1 Implementing CP(Graph) using message passing

In [QVD05b], we re-implemented part of CP(Graph) using a Message Passing approach, for implementing *DomReachability* 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)|_
```

5.4.2 Pruning of *DomReachability*

```
proc {DomReachability 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 5.4: Skeleton of DomReachability

The skeleton of the implementation of *DomReachability* is shown in Figure 5.4. In the implementation of *DomReachability* 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 *DomReachability* 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`.

Transitive closure of *DomReachability* (Rules 3.8 and 3.14)

```
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`.

Pruning the upper bound of $RN(i)$ (Rule 3.10)

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] 3:nil). The propagators above mentioned will basically constrain RN.1 to be equal to RN.2 (and RN.3 to be $\{3\}$). 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 *DomReachability*:

```
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`.

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 3.11 [2].

```
/*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}
```

In order to perform the pruning of rules derived from 2.16. 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 when we take into account that nodes dominatators are nodes of the flow graph. An example illustrating the pruning performed by these statements is shown in Figure

[2]We present the algorithms that we use for computing cut nodes and bridges in [QVD05b]. These algorithms are based on DFS [QVD05b].

5.5. 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.

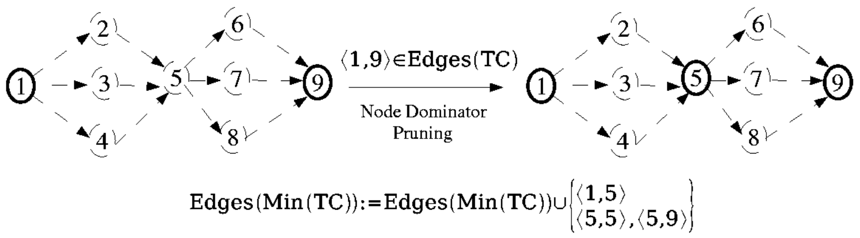


Figure 5.5: Discovering cut nodes

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 takes into account the fact the edge dominators are edges of the flow graph. An example illustrating the pruning performed by this statement is shown in Figure 5.6. 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.

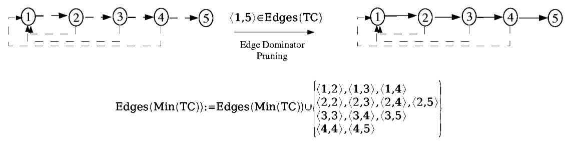

⟨1,5⟩∈Edges(TC)
Edge Dominator Pruning
Edges(Min(TC)):=Edges(Min(TC))∪{⟨1,2⟩,⟨1,3⟩,⟨1,4⟩ ⟨2,2⟩,⟨2,3⟩,⟨2,4⟩,⟨2,5⟩ ⟨3,3⟩,⟨3,4⟩,⟨3,5⟩ ⟨4,4⟩,⟨4,5⟩}

Figure 5.6: Discovering bridges

5.4.3 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

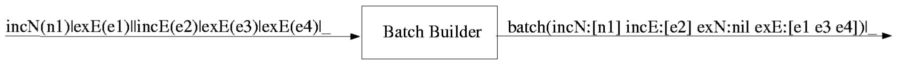

incN(n1)|exE(e1)||incE(e2)|exE(e3)|exE(e4)|_
Batch Builder
batch(incN:[n1] incE:[e2] exN:nil exE:[e1 e3 e4])|_

Figure 5.7: Building batches

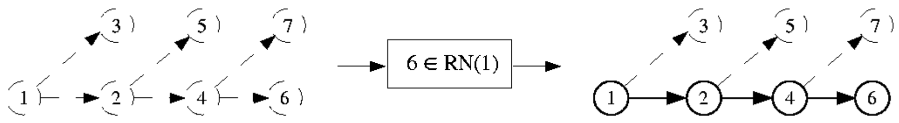


Figure 5.8: Simple Bridge Discovering

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 5.7). In this way, the transition function of the global propagator of *DomReachability* 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
      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 5.8, 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

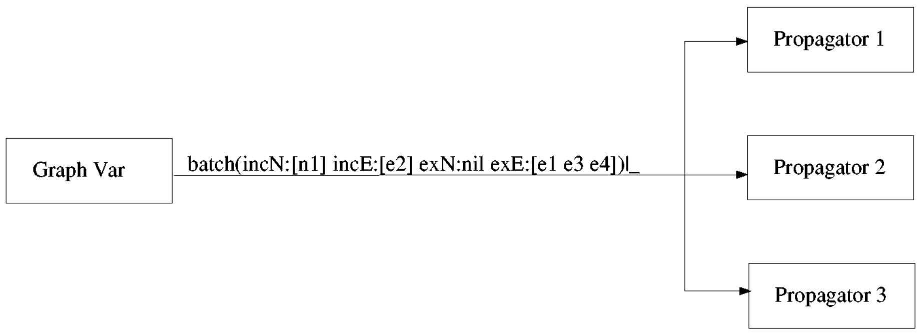


Figure 5.9: Three Propagators sharing the events stream of the graph variable on which they have been imposed

and the expensive one is not activated.

5.5 Using DomReachability with MaxReachability

The approach we have presented here lets us connect several propagators imposed on the same variable by letting the propagators share the events stream associated with the graph variable.

In Figure 5.9 we show three propagators sharing the stream of a graph variable. As soon as a batch of events is available in the stream, each propagator reads this batch independently and reacts accordingly.

In this section, we will consider $MaxReachability(fg, max)$: another propagator on graph variables that is basically a reformulation (on top of CP(Graph)) of the propagator presented in [QGV03]. $MaxReachability(g, source, max)$ states that each node of g should be reachable from $source$ through a path of at most max cost. The constraint implemented by this propagator can be formally defined as follows:

$$\begin{aligned} &MaxReachability(g, source, max) \leftrightarrow \\ &\quad \forall i \in nodes(g), \exists p \in Path(g, source, i) : Weight(p) \leq max \end{aligned} \tag{5.7}$$

Where:

- g is a directed graph whose edges are associated with positive integer costs.
- max is the upper bound of the weight of the lightest path from node i to $dest$, for every node i of g.
- $Weight(p)$ is the weight of path p.

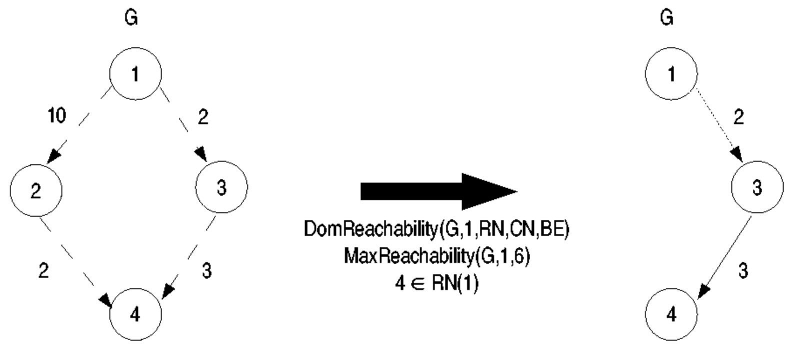


Figure 5.10: Pruning caused by the interaction between *DomReachability* and *MaxReachability*

The propagator $MaxReachability(G, Source, Max)$ implements *MaxReachability*. In the implementation we assume that $Source$ and Max are fixed values. Notice that achieving bound consistency is polynomial since the information in the lower bound of G does not actually matter. Indeed, the presence of an incoming edge e of a node i in the lower bound of G does not mean that e is in the shortest path from $Source$ to i. So, in order to check the consistency of the constraint, we only need to care about the information in upper bound, i.e., all we need to do is to find,for every node, the shortest path in the upper bound and check whether the cost of that shortest path is not greater than Max.

In Figure 5.10 we show an example of the pruning that can be obtained by the interaction between *DomReachability* and *MaxReachability*. As explained before, dashed edges represent edges that are in the upper bound of the graph variables but not in the lower bound, i.e., edges for which we do not know whether they are part of the graph denoted by the graph variable. By imposing the constraint that all nodes should be reachable from 1 with a cost of at most 6 ($MaxReachability(G, 1, 6)$), we discard node 2 from the set of nodes since the cheapest way of getting to 2 has cost 10. As node 4 should be reachable from 1, the removal of node 2 from the upper bound of G causes the determination of the only path to 4.

In Figure 5.11 we show the stream of batches of events that results from the interaction between *DomReachability* and *MaxReachability*. The first batch:

```
batch(incN:nil incE:nil exN:[2] exE:[<1,2>,<2,4>])
```

is the result of the pruning of *MaxReachability*. When *DomReachability* reads this batch it performs the pruning summarized in the second batch:

```
batch(incN:nil incE:[<1,3>,<3,4>] exN:nil exE:nil)
```

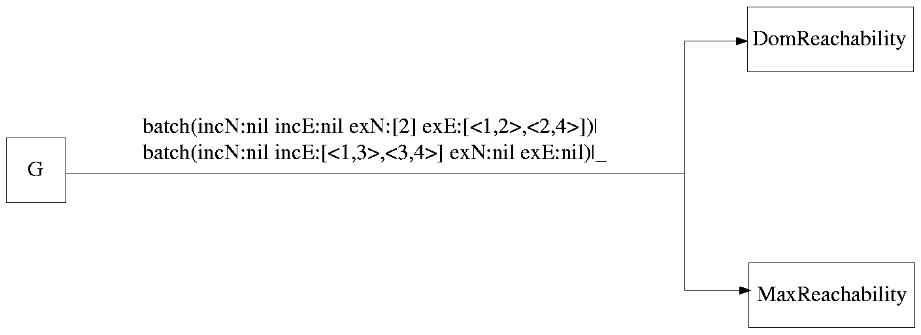

G
batch(incN:nil incE:nil exN:[2] exE:[<1,2>,<2,4>])|
batch(incN:nil incE:[<1,3>,<3,4>] exN:nil exE:nil)|_
DomReachability
MaxReachability

Figure 5.11: Stream summarizing the pruning of *DomReachability* and *MaxReachability*

Chapter 6

Solving the Simple Path with Mandatory Nodes Problem with DomReachability

In this chapter we present a set of experiments that show that *DomReachability* 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 6.1 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

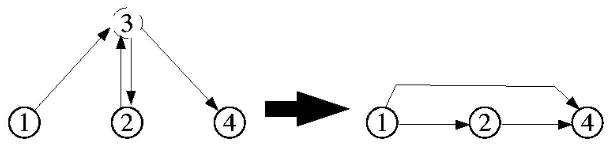


Figure 6.1: Relaxing Simple path with mandatory nodes by eliminating the optional nodes

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.
- 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

6.1 Related work

The cycle constraint of CHIP [BC94, Bou99] $cycle(N, [S_1, \ldots, S_n])$ 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 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 (see Figure 6.6) 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, DDD05b, Doo06]. 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 $Path(p, s, d, maxlength)$. This constraint is satisfied if p is a simple path from s to d of length at most $maxlength$. Certainly, Simple path with mandatory nodes can be implemented in terms of *Path*. However, the filtering algorithm of *Path* does not compute dominators, which makes *Path* also sensible to cases like SPMN_52a.

In [BFL06], the authors introduce *Tree*: a global constrain for dealing with directed graph partitioning. This constraints allows to model precedence constraints, incomparability constraints and degree constraints. *OSPMN* can be certainly modeled in terms of *Tree*, as pointed out in [BFL06]. We will elaborate on this particular approach in section 6.6.

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, there are not approaches using dominators for implementing filtering algorithms. Even though the information it provides is extremely useful, and can be computed efficiently.

6.2 Solving *Simple path with mandatory nodes* with *DomReachability*

As explained before, 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 cases[1]:

[1]In Figures 6.2 and 6.3, nodes and edges that belong to the lower bound of the graph variable are in solid line. For instance, the graph variable on the left side of Figure 6.2 is a graph variable whose lower bound is the graph $\langle\{1,5\},\emptyset\rangle$, and whose upper bound is the graph $\langle\{1,2,3,4,5,6,7,8,9\},\{\langle 1,2\rangle,\langle 1,3\rangle,\langle 1,4\rangle,\langle 2,5\rangle,\langle 3,5\rangle,\langle 4,5\rangle,\langle 5,6\rangle,\langle 5,7\rangle,\langle 5,8\rangle,\langle 6,9\rangle,$ $\langle 7,9\rangle,\langle 8,9\rangle\}\rangle$.

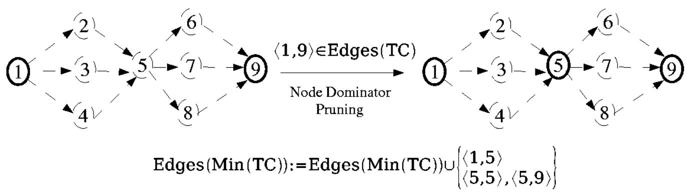


Figure 6.2: Discovering node dominators

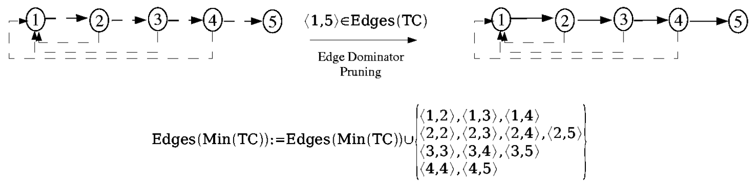

⟨1,5⟩∈Edges(TC)
Edge Dominator Pruning
Edges(Min(TC)):=Edges(Min(TC))∪{⟨1,2⟩,⟨1,3⟩,⟨1,4⟩ ⟨2,2⟩,⟨2,3⟩,⟨2,4⟩,⟨2,5⟩ ⟨3,3⟩,⟨3,4⟩,⟨3,5⟩ ⟨4,4⟩,⟨4,5⟩}

Figure 6.3: Discovering edge dominator

- Consider the graph variable on the left of Figure 6.2. Assume that node 1 reaches node 9. This information is enough to infer that node 5 belongs to the graph, node 1 reaches node 5, and node 5 reaches node 9.
- Consider the graph variable on the left of Figure 6.3. Assume that node 1 reaches node 5. This information is enough to infer that edges $\langle 1, 2\rangle$, $\langle 2, 3\rangle$,$\langle 3, 4\rangle$ and $\langle 4, 5\rangle$ are in the graph, which implies that node 1 reaches nodes 1,2,3,4,5, node 2 at least reaches nodes 2,3,4,5, node 3 at least reaches nodes 3,4,5 and node 4 at least reaches nodes 4,5.

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.

$$SPMN(g, src, dst, mandnodes, p) \leftrightarrow \begin{cases} p \in Paths(g, src, dst) \\ NoCycle(p) \\ mandnodes \subset Nodes(p) \end{cases} \tag{6.1}$$

$SPMN$ 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.

$$\begin{aligned} &DomReachability(p, edg, tc) \wedge \langle Source(p), dst\rangle \in Edges(tc) \wedge \\ &\quad mandnodes \subset \{i \mid \langle Source(p), i\rangle \in Edges(tc)\} \end{aligned} \tag{6.2}$$

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 [Rég94] and the *NoCycle* constraint [CL97] suggest the two approaches for Simple path with mandatory nodes summarized in Table 6.1 (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*.

Approach 1
$SPMN(g, src, dst, mandnodes, p)$

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

Table 6.1: Two approaches for solving Simple path with mandatory nodes

6.3 Experimental results with the implementation of DomReachability in Oz

In this section we present a set of experiments that show that *DomReachability* is suitable for Simple path with mandatory nodes. The experiments have been carried out with the implementation of DomReachability done in Oz. We will repeat these experiments with the *Gecode(CP(Graph))* implementation in section 6.7.

In our experiments *Approach 2* (in Table 6.1) outperforms *Approach 1*. These experiments also show that Simple path with mandatory nodes tends to be harder when the number of optional nodes increases if they are uniformly distributed in the graph. We have also observed that the labeling strategy that we implemented with *DomReachability* tends to minimize the use of optional nodes (which is a common need when the resources are limited).

In Table 6.2, we define the instances on which we made the tests of Tables 6.4 and 6.5. 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

Name	Figure	Source	Destination	Mand. Nodes	Order
SPMN_22	6.4	1	22	4 7 10 16 18 21	false
SPMN_22full	6.5	1	22	all	false
SPMN_52a	6.6	1	52	11 13 24 39 45	false
SPMN_52b	6.6	1	52	4 5 7 13 16 19 22 24 29 33 36 39 44 45 49	false
SPMN_52full	6.7	1	52	all	false
SPMN_52Order_a	6.6	1	52	45 39 24 13 11	true
SPMN_52Order_b	6.6	1	52	11 13 24 39 45	true

Table 6.2: Simple path with mandatory nodes instances

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

Table 6.3: Performance with respect to optional nodes

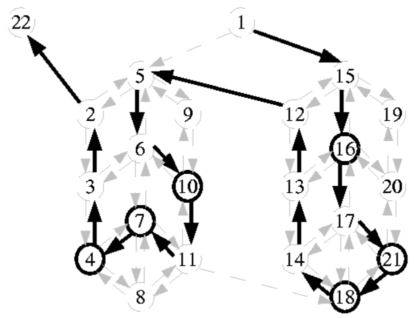


Figure 6.4: SPMN_22:A path from 1 to 22 visiting 4 7 10 16 18 21

SPMN and *DomReachability* with the dominance graph of the extended flow graph

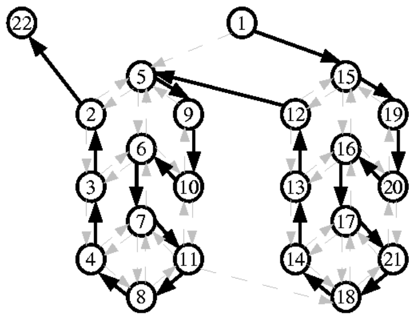


Figure 6.5: SPMN_22full:A path from 1 to 22 visiting all the nodes

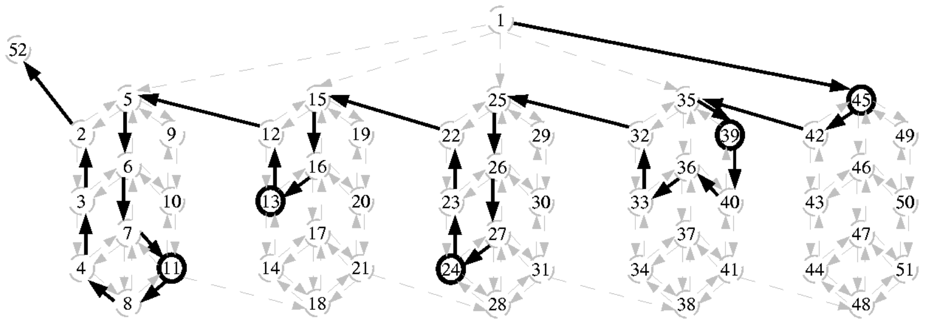


Figure 6.6: SPMN_52a:A path from 1 to 52 visiting 11 13 24 39 45

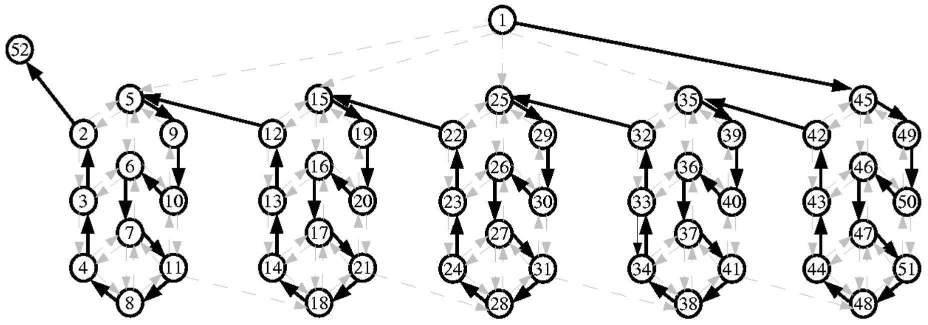


Figure 6.7: SPMN_52full:A path from 1 to 52 visiting all the nodes

(node+edge dominators (column “SPMN+R+ND+ED”)).

Problem		**SPMN**		**SPMN+R**	
Instance	**Figure**	**Failures**	**Time**	**Failures**	**Time**
SPMN_22	6.4	+130000	+1800	91	6.81
SPMN_22full	6.5	213	1.44	19	0.95
SPMN_52b	_	_	_	+900	+1800
SPMN_52full	6.7	3012	143	774	765
SPMN_52Order_a	6.6	+12000	+1800	51	46.33
SPMN_52Order_b	_	+12000	+1800	+1500	+1800

Table 6.4: Simple path with mandatory nodes tests (not using dominators)

Problem		**SPMN+R+ND**		**SPMN+R+ND+ED**	
Instance	**Figure**	**Failures**	**Time**	**Failures**	**Time**
SPMN_22	6.4	40	6.55	13	4.45
SPMN_22full	6.5	0	0.42	0	1.22
SPMN_52b	_	+700	+1800	100	402
SPMN_52full	6.7	3	8.51	3	45.03
SPMN_52Order_a	6.6	45	81	16	57.07
SPMN_52Order_b	_	81	157	41	117

Table 6.5: Simple path with mandatory nodes tests (using dominators)

As it can be observed in Table 6.4, 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 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 6.3, 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.

6.4 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 Figure 6.6).

Nevertheless, in order to reduce the number of failures in finding the solution of Figure 6.9 (which was solved in around two hours with 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. In fact we can think of our heuristic as a kind of A* heuristic [RN03].

6.5 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 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.

Notice that an alternative implementation for ordering constraints is to do it in terms of the extended dominator graph. As the resulting graph is a path, we have that i dominates j if and only if i is reached before j. Nevertheless, this is only true if j is reached from the source since, by definition, unreachable nodes are dominated any node.

6.6 Modeling OSPMN with the Tree constraint

In [BFL06], the authors introduce *Tree*: a global constrain for dealing with directed graph partitioning. This constraints allows to model precedence constraints, incomparability constraints and degree constraints.

6.6.1 Definition of the Tree constraint

$Tree(ntree, nprop, ver)$ holds if g, the graph represented by ver, is a forest (a disjoint union of trees) having $ntree$ trees, of which $nprop$ are proper trees, i.e., trees with at least two nodes.

ver is a collection of nodes. Each node i is associated with the following attributes:

- L is the label of i, i.e., $ver[i].L$ is i.
- F is the label of the father of i in the tree containing i, i.e., if $ver[i].F = j$ then $\langle j, i\rangle \in Edges(g)$ and there is no $k \neq j$ such that $\langle k, i\rangle \in Edges(g)$ [2].
- P is the set of mandatory predecessors of i, i.e., for every node $j \in ver[i].P$, there is a unique simple path from j to i in g.
- I is the set of incomparable nodes of i, i.e., for every node $j \in ver[i].I$ there is a path in g neither from i to j nor from j to i.
- D is the number of outgoing edges of i in g.

Due to the fact that g is a forest, precedence means domination. If i precedes j, i is a dominator of j with respect to r, the root of the tree containing i and j. Indeed, as there is only one path from r to j, i trivially fulfills the condition of being in all the paths from r to j.

Lorca et al make special emphasis in avoiding redundant information in their data structures. For the case of the precedence relation, they internally keep a graph which represents the precedence relations among nodes. When a new precedence edge is inferred during propagation, they first check whether the edge can not be computed from the edges that are already in the precedence graph before adding it to the graph. The invariant they keep is that the whole set of precedence relations is the transitive closure of the graph kept.

6.6.2 Modeling OSPMN

OSPMN can be easily modeled in terms of $Tree$ by stating that the mandatory nodes precede the destination, and are preceded by the source. The order among nodes is directly modeled by imposing the corresponding precedence constraints. Formally speaking, we can model *OSPMN* as follows:

$$OSPMN(gmax, src, dst, mn, order, ver) \leftrightarrow \begin{cases} Subgraph(ver, gmax) \\ Tree(1, 1, ver) \\ \forall i \in mn : src \in ver[i].P \wedge i \in ver[dst].P \\ \forall \langle i, j\rangle \in order : i \in ver[j].P \end{cases} \tag{6.3}$$

[2] In order to make the definition of $Tree$ more intuitive, we have modified the definition of ver[i].F. In [BFL06], $ver[i].F = j$ if $\langle i, j\rangle \in Edges(g)$

By $Subgraph(ver, g)$ we mean that the graph represented by the collection of nodes ver is a subgraph of $gmax$ (the graph in which the simple path should be found).

Any solution to the above CSP binds ver to a tree containing a unique path from src to dst containing the mandatory nodes. So, once a solution has been found, finding the simple path is straightforward since it is a matter of running *DFS* rooted at src.

Thanks to the fact that, in the *Tree* constraint, each node has an out degree attribute, the CSP can be further refined so that the solution found corresponds to a simple path containing the mandatory nodes.

6.6.3 Dealing with precedence constraints

In chapter 2, we showed one way of Modeling *OSPMN* in terms of *DomReachability* only. When comparing this model with the one based on *Tree*, we observe that the *Tree* model is more constrained since the solution is a tree. This additional restriction allows to discard potential edges as soon as it is known that its destination already has an incoming edge.

In the implementation of *Tree*, the authors take advantage of the notion of strong articulation point to infer precedence relations among nodes. Given a strongly connected component c, we say the node n is a strong articulation point if c is split up into several strongly connected components after removing c.

Once the strong articulation points have been computed, the authors check whether the removal of them violates the precedence constraints. The removal of a strong articulation point p violates a precedence constraint $\langle i, j\rangle$ if there is no path from i to j after removing p. If the removal of p violates the precedence constraint $\langle i, j\rangle$, the precedence constraints $\langle i, p\rangle$ and $\langle p, j\rangle$ are added to the precedence graph in case they are not redundant.

Let us consider the *OSPMN* instance shown in Figure 6.4 where we are interested in finding a simple path from 1 to 22 containing 4 7 10 16 18 21. The graph we obtain after removing node 22 is a strongly connected component. In this strongly connected component, we observe that node 12 and 5 are strong articulation points. In the *Tree* model corresponding to this instance, we have that precedence $\langle 16, 22\rangle$ is included in the set of precedence constraints. As the removal of node 12 makes node 22 unreachable from node 16, the precedence constraints $\langle 16, 12\rangle$ and $\langle 12, 22\rangle$ are added. As the removal of node 5 makes node 22 unreachable from node 12 , the precedence constraints $\langle 12, 5\rangle$ and $\langle 5, 22\rangle$ are added. As edge, $\langle 12, 5\rangle$ is the only outgoing edge of node 12, $\langle 12, 5\rangle$ belongs to the solution. Notice that stating that edge $\langle 12, 5\rangle$ belongs to the solution implies that edge $\langle 1, 5\rangle$ does not belong to the solution since node 5 can only have one incoming edge.

In the *DomReachability* model, edge $\langle 1, 5\rangle$ can not be removed at the initial propagation phase because in this model node 5 is not forced to have one incoming edge. Even if the edge were forced to have one incoming edge, we would be still unable to remove edge $\langle 1, 5\rangle$ because the source of the dominator tree that we keep

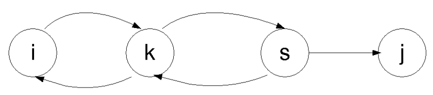


Figure 6.8: In this graph, the set of dominators of j with respect to i is a proper superset of the set of strong articulation points of the graph

is node 1 and edge $\langle 12, 5\rangle$ is not an edge dominator of node 22 with respect to node 1. In our approach, the edge $\langle 12, 5\rangle$ is only discovered after choosing edge $\langle 1, 15\rangle$.

The approach of discovering precedence constraints by using strong articulation points applies even in some cases where the nodes are in different components. For instance, consider the graph in Figure 6.8 and suppose that there is a precedence constraint $\langle i, j\rangle$. Even though i and j are in different components, it is still possible to apply the strong articulation points rule since the removal of k (which is a strong articulation point) makes j unreachable from i.

However, even though s is not a strong articulation point, it should be inferred that i precedes s ,s precedes j and k precedes s when i precedes j. In this particular case, using dominators instead of strong articulation points leads to better pruning. However, keeping only one dominator tree is not enough to maintain this difference in pruning because the dominance relation depends on the source (as shown in the previous example).

6.7 Experimental results with the implementation of DomReachability in Gecode(CP(Graph))

In this section we repeat some of the tests presented in section 6.3. In the implementation of the approach, we have used the *Path* constraint to constraint the solution to be a simple path [DDD04, DDD05b, Doo06]. $Path(p, s, d)$ is satisfied if p is a simple path from s to d.

The approach can be formally specified as follows:

$$OSPMN(g, src, dst, mn, order) \leftrightarrow \begin{cases} Subgraph(fg, g) \\ Path(fg, src, dst) \\ DomReachability(fg, edg, tcg) \\ \langle src, dst\rangle \in Edges(tcg) \\ \forall i \in mn : \langle i, dst\rangle \in Edges(edg) \\ \forall \langle i, j\rangle \in order : \langle i, j\rangle \in Edges(edg) \end{cases} \tag{6.4}$$

In table 6.6, we compare the results that we got with the *Gecode(CP(Graph))* implementation with those obtained by Lorca et al [BFL06]. We have carried out

OSPMN instances	Lorca et al [BFL06]		DomReachability+Path	
	Failures	Time	Failures	Time
SPMN_22	0	0.071	5	0.110
SPMN_22full	0	0.036	0	0.070
SPMN_52b	0	1.685	6	0.920
SPMN_52full	0	0.692	0	0.580
SPMN_52Order_a	0	0.892	0	0.500
SPMN_52Order_b	0	0.020	4	0.280

Table 6.6: Tree Vs DomReachability+Path

the experiments in the same machine where the Oz tests were done. This machine is a 3060 MHz Linux Red Hat machine with 3805136 KB of RAM.

Our first observation is that the *Gecode(CP(Graph))* implementation remarkably outperforms the Oz implementation. This is basicly due to the following reasons:

- *Gecode(CP(Graph))* is a C++ library whereas *Oz* is compiled into a byte code, which is emulated.
- The *Gecode(CP(Graph))* implementation is using state-of-the-art algorithms for computing dominators and transitive closure [Geo05, LLS01].
- *Path* prunes more than the path propagator used in the Oz experiments which is basically a conjunction of a *NoCycle* constraint and an *AllDiff* constraint.

We also notice that, with this implementation, we are competitive with respect to Lorca et al's approach. This implementation also solves the real world case presented in Figure 6.9 in 58 seconds without failing, which shows the scalability of our approach.

6.8 Conclusion

We showed how *DomReachability* can speed up a standard approach for dealing with *SPMN*. Our experiments show that the gain is increased with the presence of optional nodes.

We presented another approach for solving *SPMN* built on top of the *Tree* constraint [BFL06]. We elaborated on the difference in pruning that this approach has with respect to our approach.

We compared the two implementations of *DomReachability* and showed that the *Gecode(CP(Graph))* implementation remarkably outperforms the *Oz* implementation. We also showed that the *Gecode(CP(Graph))* implementation allows us to be competitive with the approach presented in [BFL06].

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 6.2 is a context where the computation of edge dominators cannot infer anything since there is no edge dominator. Similarly, Figure 6.3 represents a context where the computation of edge dominators discovers more information than the computation of node dominators.

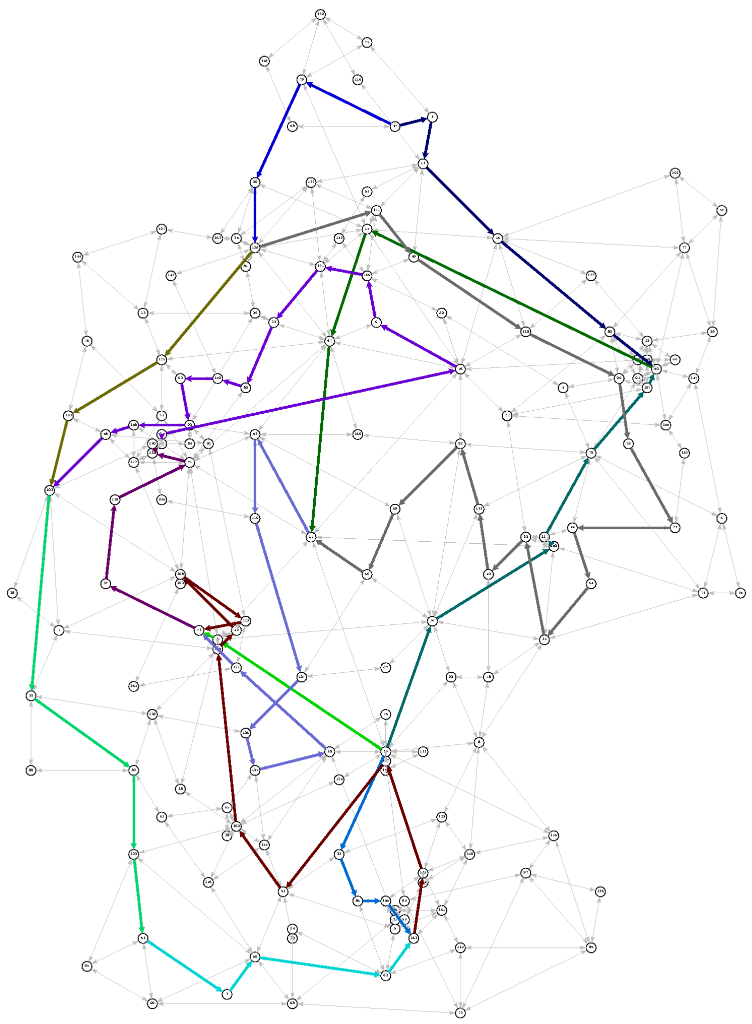


Figure 6.9: A Disjoint Paths instance solved with *DomReachability* involving 14 paths in a graph of 165 nodes. This instance was kindly provided by Martin Oellrich from the Combinatorial Optimization & Graph Algorithm Center at TU Berlin. This instance will be available in the web site of DomReachability, which will be reachable through the web site of CP(Graph) (*http://cpgraph.info.ucl.ac.be*)

Chapter 7

Using DomReachability for Confinement Analysis

In software security, the execution of some actions is controlled (allowed or disallowed), in an attempt to restrict their (direct or indirect) effects. Allowed actions are called *permissions*. Different parts of a program (subjects) can have different permissions. The ability of a subject to directly or indirectly induce an effect is called its *authority*.

The difference between permission and authority is the difference between action and effect. Illegal authority are called “safety properties”. A program breaks a safety property if the illegal effect is reachable. When analyzing if a program can break a safety property, the following are important:

1. What permissions are initially available to (the different subjects of) the program.

2. How the subjects use their permissions to generate effects.

It was shown in [HRU76] that these kinds of problems are not computable in general. Therefore, security analysis has to approximate the problem from the safe side, by looking for *proof* that the safety property remains unbroken. If no such proof can be found, the problem is *assumed* to be unsafe. We can safely approximate a program by considering only the authority *enhancing* parts of the actions. This is a *monotonic* safe approximation, which can provide a reasonably accurate estimate of the original program’s safety, if the preconditions for the actions are sufficiently detailed.

The propagation of authority can often be expressed in sufficient detail by reachability in a directed graph. The nodes in the graph each represent a subject and the edges represent permissions. The reflexive and transitive closure of the permission graph then represents an upper bound for reachable authority.

For example, consider a set of subjects that can have *read* and/or *write* permissions to each other. Set up a graph and depict the write-permissions as edges from the writer to the subject written, and depict the read-permissions as edges that point *towards* the reader. The edges now all point in the direction of the information flow, and the effects of exerting the permissions (the actual flow of information in the graph) propagate by transitive reachability in the graph.

In this chapter, we show that graph reachability constraints have useful applications in safety analysis and enforcement. We do not claim that this approach is appropriate, useful, or feasible in *all* circumstances.

Most graph-based formal security models have labelled edges to differentiate between types of permissions, and labelled nodes to differentiate between types of subjects. To model types of permissions, we can use a dedicated reachability graph for every permission type. Permissions types that result in the same type of authority can be represented in a single graph (e.g. *read* and *write* permissions provide authority of the type: "pass information" and can be represented in the same data-flow graph).

We will show how, in certain conditions and to a certain extent, node types can be expressed as subgraphs that represent the node's "inner workings".

We expect that graph reachability constraints, when used in combination with purpose built tools for constraint based security analysis [SJV05], can boost the latter's expressive power and scalability.

This chapter is structured as follows. In section 7.1 we express a security problem in terms of the Bounded Transitive Closure problem (*BTC*). The rest of the chapter describes several ways of using *DomReachability* for safety analysis. Section 7.2 demonstrates how to calculate strategic positions for interposition of controllable subjects in a network of interacting entities. Section 7.3 explains how entities with restricted behavior can be expressed by subgraphs and by adding additional constraints to the subgraph.

Section 7.4 presents two extensions of *BTC*, and discusses their additional expressive power. We extend the safety analysis to networks of interconnected systems in Section 7.5, and compare the scalability of the extended *BTC* approach with an existing approach based on the "Scollar" tool [SJV05]. We then present future work, that will combine the strength of both approaches.

Remark: This chapter is joint work with Fred Spiessens and was published in [SQV06]. This work will be also part of Fred Spiessens's PhD thesis [Spi06].

7.1 Expressing security constraints with DomReachability

Our security problems have two concerns:

1. some authority should not be reachable for safety (safety properties)

2. some other authority should be reachable for functionality (liveness properties)

Both concerns can be expressed in terms of *The Bounded Transitive Closure Problem (BTC)*: given the directed graphs g_{min}, g_{max}, tcg_{min} and tcg_{max}, find a directed graph g such that:

$$\begin{array}{c} g_{min} \subseteq g \subseteq g_{max} \\ \text{and} \\ tcg_{min} \subseteq TC(g) \subseteq tcg_{max} \end{array} \tag{7.1}$$

The set of liveness properties will be tcg_{min}, tcg_{max} will be the complement of the set of safety properties, and g_{min} and g_{max} will just be suitable bounds for the safe configuration of permissions we are looking for. In section 2.3.1 we proved that *BTC* is NP-complete. This implies that any problem that reduces to it is also NP-complete.

7.2 Confinement by interposition

Suppose we have a set of previously unconnected, uncontrollable components, and we want to find out how we can connect them, using controllable components, to allow them to perform their collaborative tasks, but also prevent them from breaking a given security policy. The tools we have to solve this problem are:

- a set of controllable components (subjects) to be strategically inter-positioned between the uncontrolled components.
- a set of permissions to be granted to the controllable components.

The assignment is: find a configuration (graph) with a *minimal number of controllable nodes* (not exceeding a fixed practical upper limit), that guarantees the requirements for liveness (the uncontrolled components get enough authority) as well as the requirements for safety (the uncontrolled components do not get too much authority).

7.2.1 Practical example

We take a well known example, expressing a simple *Multi-Level Security Problem (MLS)* [BL74]. Two external subjects *Bond* and *Q*, with respective clearances *Top Secret* and *Confidential*, have to be given access to two external storage devices, one for *Top Secret* content, and one for *Confidential* content.

We have to construct the content of a black box in (e.g. Figure 7.1), with a minimal number of components. Since the uncontrollable components cannot be restricted, their connection to the box is bi-directional. Even the devices are not trusted to be passive containers, they are unknown components and could be of any type.

The security policy we want to enforce between these four entities is simply to make sure that no Top Secret information leaks (down) to the Confidential level.

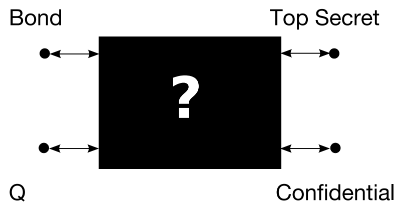


Figure 7.1: The $*$-property black box

Therefore we will enforce the $*$-*property* (star-property) that states: agents should be able write to all levels above (and including) their own level of confidentiality, and read from all levels below (and including) their own level of confidentiality, but no agent should be able to write strictly below his confidentiality level, or read strictly above his confidentiality level. This is a policy that specifies both liveness requirements and safety requirements, so we will express it as suggested in section 7.1.

7.2.2 Expressing the problem in terms of DomReachability

The *BTC* for the instance of the problem presented above is:

$$\begin{aligned} g_{min} &= \emptyset \\ g_{max} &= \{\langle x,y\rangle | x,y \in \{b,q,t,c\} \cup \{o_1,o_2,...,o_{max}\}\} \\ tcg_{min} &= \{\langle b,t\rangle, \langle t,b\rangle, \langle q,c\rangle, \langle c,q\rangle, \langle c,b\rangle, \langle q,t\rangle\} \\ tcg_{max} &= g_{max} - \{\langle b,q\rangle\langle b,c\rangle\langle t,q\rangle\langle t,c\rangle\} \end{aligned} \tag{7.2}$$

In the problem, b stands for Bond, q for Q, t for the top-secret device, and c for the confidential device. The controllable nodes are o_1, o_2,...,o_{max}.

Apart from the *BTC* constraints, we have to express the fact that b, q, t, and c are uncontrolled, by making sure that all their connections are bi-directional. We therefore added the necessary implication constraints to the problem:

$$\forall 0 \leq x \leq max, i \in \{b,q,t,c\} : \langle i,o_x\rangle \in g \Leftrightarrow \langle o_x,i\rangle \in g \tag{7.3}$$

To minimize the number of controlled components, we can start with zero controlled nodes, and iteratively add one more, until we find a solution.

We also experimented with a labeling strategy that tends to find the solution with the least nodes first. By first trying to remove all possible edges from the controlled node that reaches the most nodes, the strategy tends to minimize both the number of edges and the number of controlled nodes, although there are cases where the number of controlled nodes used is not the smallest one possible.

The pruning performed by *DomReachability*, and the aforementioned labeling strategy provided the solution in Figure 7.2, in 40ms.

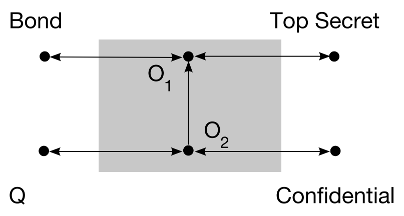


Figure 7.2: A solution with the minimal number of controllable components

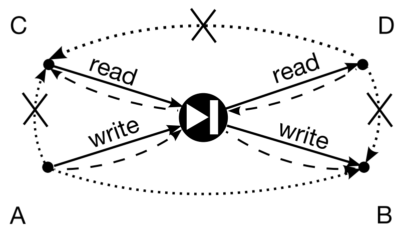


Figure 7.3: Data Forwarder (dataflow diode)

7.3 Confinement by restricted behavior

In the previous section we relied on the ability of the system to enforce the permissions. There could for instance have been a reference monitor that checked the permissions before they were exerted. Alternatively, the internal subjects could be trusted parts of the system: trusted to behave exactly as allowed by their permissions. Capability systems [DH65] rely on such subjects (called capabilities).

We could as well rely in our home-brewn trusted subjects to behave in “smarter” ways than simply using or not using certain permissions. We can program them to use their permissions in a way that would allow the desired *effects* and prevent the other ones. This allows for much more accurate analysis of the reachable effects in a system. An account of the different ways in which the boundaries of authority can be calculated is given in chapter 8 of [Mil06].

Suppose we want to express the behavior of a subject that only passes information if:

- other subjects wrote that information to it (it did not read the information itself from other subjects), and

- it writes that information itself to other subjects (it does not reveal that information to its own readers)

Such a subject acts as a forward diode for data flow, depicted in figure 7.3. The full edges denote the access rights and the dashed edges represent the corresponding flow of data. The data-flow is only transitive in one direction: from A to B, as

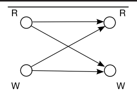
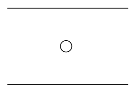
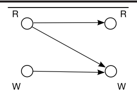
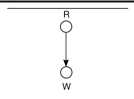
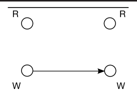
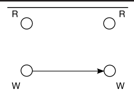
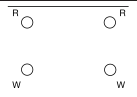
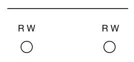
behavior graph	**simplified graph**	**behavior**
		unrestricted behaviour
		hides its writers data from its readers
		data forwarder of figure 7.3
		non-tranparent subject

Figure 7.4: Subgraphs for behaviour-based internal dataflow

indicated by the dotted edges. The behaviour of the diode in the middle prevents data to pass in the three other directions.

We can express similar restricted behavior in a subgraph with four nodes: two $in - ports$ and two $out - ports$, one of each kind for reading, and the other one for writing. All external edges will be connected to one of the four ports: the incoming flow to the in-ports, the outgoing flow to the out-ports, the flow via read permissions to the read-ports, and the flow via write permissions to the write-ports. These restrictions can directly be expressed in *BTC*, by removing the illegal external connections from g_{max}.

Figure 7.4 shows some behaviour subgraphs with four *internal* ports (not to be confused with the graph in figure 7.3). The internal flow (edges) always goes from an in-ports (left) to an out-ports (right). These subgraphs are to replace the monolithic subject nodes in the *data-flow graph*. The edges here correspond to the *dotted* edges (flow-through) in the example of figure 7.3. Depending on which of the four possible edges are present, the behaviour-graph can be simplified (second column of figure 7.4).

7.4 Extending BTC for additional expressive power

In the previous sections we had to use additional constraints to express the security problems. We now propose two extensions to the BTC problem that incorporate the implication constraints that allow us to express interesting security problems.

7.4.1 The conditional *BTC* problem (CondBTC)

In section 7.2.2 we had to use extra constraints for all four uncontrolled components, to express that they should take only bidirectional connections. This means,

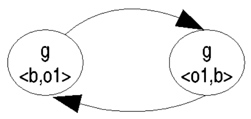


Figure 7.5: A fraction of the *condg* for the problem in section 7.2.2

if an edge $\langle A, B\rangle$ is in the graph, then so should $\langle B, A\rangle$. We can express this condition as an edge from $\langle A, B\rangle$ to $\langle B, A\rangle$ in a graph whose nodes are edges in other graphs and whose edges represent implications. This allows us to express inter-graph conditions on edges. If we consider also the complements of the graphs, (the complement of graph g is denoted as $(g)\prime$), we can express negations as well as implications.

Therefore we add a directed graph $condg$ such that:

$$\begin{gathered} G_1, G_2 \in \{g, (g)', TC(g), (TC(g))'\} \\ \langle e_1^{G_1}, e_2^{G_2}\rangle \in condg \Leftrightarrow (e_1 \in G_1 \Rightarrow e_2 \in G_2) \end{gathered} \tag{7.4}$$

The security problems in sections 7.2 and 7.3 are direct applications of *CondBTC*. The implications involving edges of the solution graph and its transitive closure can be directly represented in terms of $condg$.

Figure 7.5 shows a bi-directional connection constraint as 2 edges in $condg$.

7.4.2 The cardinal *BTC* problem (CardBTC)

Instead of representing edges in another graph, let the nodes in $condg$ now represent *mixed sets of edges from any of the DomReachability graphs*. An edge $\langle A, B\rangle$ in $condg$ now represents a composite condition: if *all* edges in the set A are present, then so should *at least one* edge in the set B.

The extended definition of $condg$ allows us to simplify the definition of the problem. The *BTC* graphs g_{min}, g_{max}, tcg_{min} and tcg_{max} can be defined in $condg$ as follows:

$$\begin{array}{rcl} \forall e \in g_{min} & : & \langle \emptyset, \{e^g\}\rangle \in condg \\ \forall e \notin g_{max} & : & \langle \emptyset, \{e^{(g)'}\}\rangle \in condg \\ \forall e \in tcg_{min} & : & \langle \emptyset, \{e^{TC(g)}\}\rangle \in condg \\ \forall e \notin tcg_{max} & : & \langle \emptyset, \{e^{(TC(g))'}\}\rangle \in condg \end{array} \tag{7.5}$$

The expressivity of *CardBTC* can be further extended by labelling edges with constraints on the cardinality of the target set. For instance, figure 7.6 graphically shows the following constraint in extended Higraph notation [Har95] :

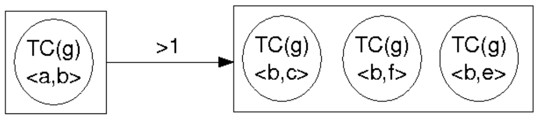


Figure 7.6: A graphical presentation of a CardBTC constraint

$$\begin{aligned}
&b_1, b_2, b_3 \in \{0, 1\} \\
&b_1 = 1 \Leftrightarrow \langle b, c\rangle \in TC(g) \\
&b_2 = 1 \Leftrightarrow \langle b, f\rangle \in TC(g) \\
&b_3 = 1 \Leftrightarrow \langle b, e\rangle \in TC(g) \\
&\langle a, b\rangle \in TC(g) \Rightarrow b_1 + b_2 + b_3 > 1
\end{aligned} \tag{7.6}$$

We can say that, when the label of the edge is ommitted, the implicit constraint is “> 0”, i.e., at least one of the constraints in the set must be true.

The cardinality constraints involved in *CardBTC* can be managed by using standard approaches based on cardinality propagators [VD91]. However, we can reason at a higher level of abstraction by looking at the Boolean Satisfiability instance that results from associating each basic graph constraint with literals. This level of abstraction would let us take advantage of BDD propagators to narrow down the literals composing a given disjunction [HLS05]. We could also consider hybrid approaches, like the one suggested in [HS06], in order to inherit the advantages offered by SAT solvers.

7.4.3 Applying CardBTC for practical security problems

CardBTC allows us to express complex conditions on the propagation of authority in several ways we did not yet explore:

- It can be used to express more complex ways of authority propagation than transitive closure.
- It can be used to represent fine-grained conditional behavior of trusted subjects, without the need to represent every subject as a complex subgraph.

7.5 Secure interoperation

In this section we present a security problem for which the scalability of the approach using DomReachability considerably exceeds that of *Scollar* [SJV05], a more general constraint-based tool for security analysis.

A system of interacting subjects can be secure, but when two or more secure systems become interconnected, the result may again introduce safety breaches.

A secure reconfiguration removes a set of permissions from the subjects in the system, to make sure that no authority that was unreachable in any single system (and may have been forbidden by that systems policy), becomes reachable by the interconnection.

The use of constraint programming to find secure reconfigurations of interoperating systems, is proposed in [BFO05], for systems that are interconnected via shared subjects. By default, the approach makes no effort to prevent authority between two subjects that are not both included in a single system. The rationale behind this decisions is that, while authority between such subjects is always due to the interconnection, no single system can be held responsible for managing these effects. Additional safety and liveness requirements can be added to the interconnected system.

Following this approach, we show how to use DomReachability, to find a minimal secure reconfiguration for a set of interconnected systems. A secure reconfiguration is a set of permissions such that, when each of these permissions are revoked in all systems that granted the permission before the interoperation, the interconnection will make no additional effects reachable between two subjects of the same system. This approach can then easily be extended to include additional constraints on the reachability of effects in the interconnected system.

7.5.1 Calculating secure reconfigurations with Scollar

The constraint based tool “Scollar”, written in Mozart-Oz [VH04, Sch00], analyzes safety in configurations of permission-restricted and behavior-restricted subjects, and calculates the minimal (additional) restrictions that are necessary to guarantee the safety requirements.

Scollar was recently extended to compute safe reconfigurations for interoperating systems as well. To analyze secure interoperation, the tool first computes the transitive closures of every individual system, derives from these transitive closures the safety requirements for the global, interconnected system, and then calculates a minimal reconfiguration, that removes some initial permissions (and/or behavior of the trusted subjects, when specified)

Since Scollar’s primarily aim is to analyze small patterns of interacting subjects with relatively complex behavior, its scalability in terms of number of subjects was not an initial concern. For secure interoperation, the problems tend to be larger in number of subjects, and marginally lower in complexity of the subject behavior. Recent experiments with the current implementation revealed that the practical limit allows no more than approximately 100 subjects in the interconnected system, even when the behavior complexity is reduced to simple on-off (active/passive), no propagation of permissions is modeled, and the mechanisms for effect propagation are reduced to simple transitive closure.

7.5.2 Comparing scalability

We conducted a very preliminary set of experiments to get a rough idea of the relative scalability of DomReachability in comparison to Scollar.

We can expect that the DomReachability approach will perform best when the rules that model the propagation of authority are similar to reachability by transitive closure. At the same time we wanted to test the practical feasibility of modelling simple restrictive behaviour as subgraphs.

We decided to run a very preliminary and small set of experiments with the following setup:

- N systems are inter-connected in a network that has a small-world topology, generated following the Watts-Strogaz approach [WS98] from a structured undirected graph in which every system (node) has 4 neighbours. A small world graph is a graph with a high clustering coefficient (of every system, most neighbouring systems are connected) and a low characteristic path length (mean distance in the network between any two systems).
- Systems S_1 and S_2 are connected $\Leftrightarrow$ they share exactly two subjects.
- Unconnected systems have no common subjects
- Subjects are shared by at most two systems.
- Every system has exactly as many subjects as are required for its connections to its neighbours in the network.
- Half of the subjects in every system have unrestricted behaviour, the other half are non-transparent (See Figure 7.4).

The same instances of the generated problems where fed to the Scollar based solver and to the DomReachability based solver. All safety properties were precalculated in Scollar, during the generation of the examples, and were not recalculated in the experiments.

The rules that govern data-flow in all systems were kept simple, and are illustrated by the following Horn Clauses, used in Scollar:

$$readPermission(Y,X) \Rightarrow flow(X,Y)$$

$$flow(X,Y) \wedge flow(Y,Z) \wedge transparent(Y) \Rightarrow flow(X,Z)$$

We made the experiments as simple as was reasonably possible, by considering only one permission (read), with an effect (data transfer towards the reader) that propagates transitively when the behavior of the subject transferring the data does not prevent it. We arranged for 50% of all subjects to be unrestricted (allowing data to flow through them in all directions), and the other 50% to be non-transparent (See Figure 7.4). The transparency of a subject was considered to be a fixed and was not optimized. Only the readPermissions were optimised in the experiments.

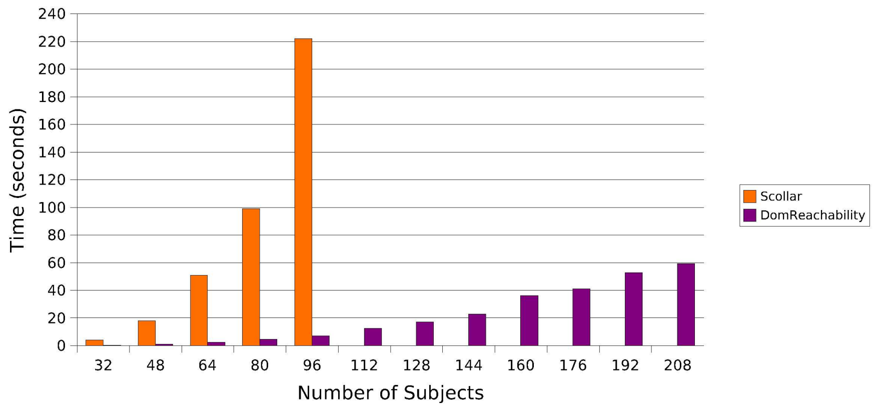


Figure 7.7: *Scollar* and *DomReachability* calculating Secure Reconfigurations

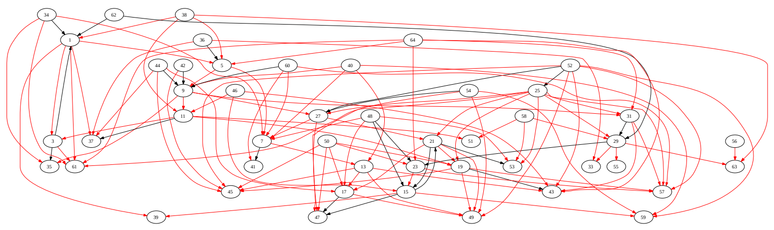


Figure 7.8: A secure inter-operation problem instance involving 48 subjects. Edges dropped are in black.

All these restrictions where set up to restrict the influence of random choices on our measurements, and improve the accuracy with which our results reflect the influence of the size of the problem (number of systems).

All experiments where conducted on a dedicated Linux machine with 2GB of memory and 4 processors at 3.06 GHz. Figure 7.7 shows the time it took to find a first secure reconfiguration (in seconds), for networks with 8 to 52 systems (32 to 208 subjects). We performed only one calculation for every size of the problem. No results could be calculated in Scollar for problems of more than 24 systems (96 subjects), due to virtual memory exhaustion.

Even if only one problem instance was solved for every size, the results leave no doubt about the winner in this scalability contest. DomReachability is much more suited to solve problems of big size.

The labeling strategy implemented on top of the information computed by DomReachability indeed tends to minimize the number of edges dropped. Figure 7.8 shows an instance of the inter-operation problem. Notice that the number of edges dropped in order to satisfy the constraints in Figure 7.9 is small with respect

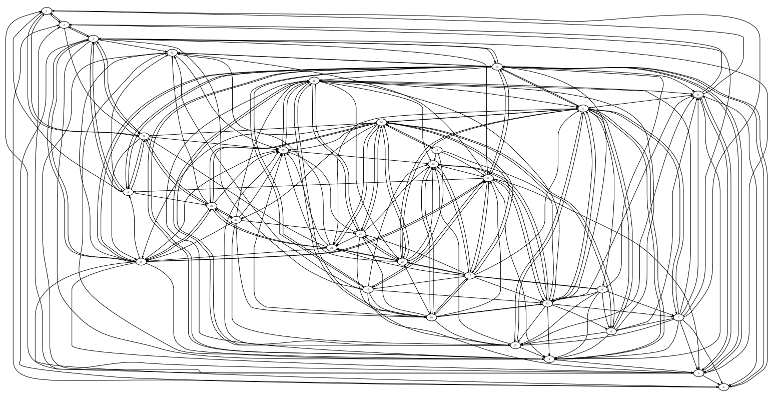


Figure 7.9: No-reachability constraints of the BTC problem of Figure 7.8

to the total number of edges.

We expect that Scollar is still more suitable for solving complex problems of small size, to model complex rules and subject behaviour that express a refined approximation to how authority propagates. It will be interesting future work to find out exactly where the borderline for choosing between the two approaches lies, and even more interesting to see how the approaches can be combined to get the best of both.

7.6 Conclusion

We have shown how the monotonic propagation of effects can be modelled with reachability constraints in a directed graph by associating nodes in the graph with subjects, and edges with permissions between the corresponding subjects. We elaborated on the relation of the resulting constrained graph problem with the Bounded Transitive Closure problem (BTC) and suggested extensions of BTC that let us express more security requirements.

Some of the problems that we have presented can be solved in polynomial time. For instance, if there is no constraint on neither the lower bound of the interconnected graph of the interoperability problem nor on its transitive closure, the *BTC* instance can be solved in polynomial time. Indeed, the empty graph would be a valid solution to the problem. Even finding a maximal graph, i.e., a graph which is not included in another one respecting the safety properties, is still polynomial since it is always possible to find a graph not containing a particular edge that respects the safety properties.

The adoption of *DomReachability*, which is normally used in combinatorial problems, is justified because: (a) it offers an incremental approach for computing transitive closure, and (b) it discards invalid edges early on, since the addition of an edge may imply that some other edges are not part of the graph.

In section 7.2.2 we constrained the size of the graph by using a size constraint that takes the edges in the lower bound into account, but not the structure of the graph. A smarter global constraint would take into account the current boundaries of the graph and its transitive closure, to anticipate violations of the limit. For instance, suppose that i reaches j and the shortest path p from i to j contains x edges. Suppose also that the size of the graph is less than max. Then if the graph contains at least the edges in g_{min}, $max - |g_{min}| < |p - g_{min}|$ implies that there is no solution since reaching j from i would imply that the number of edges in the graph is greater than the limit(max). To detect this kind of information, we can use an approach like the one suggested in [Sel02] for incrementally keeping the shortest paths between each pair of nodes.

Towards a synergy of both approaches

We expect DomReachability to be most useful in collaboration with our existing Scollar tool. Scollar is most suitable to express a system's rules that govern the propagation of permissions and authority, and a subject's behaviour. System rules can express realistic models for propagation, that can take the restrictions of the behaviour of the trusted subjects into account. Subject behaviour can be expressed in a way that depends on the information that a subject has from initial conditions, and has required during the collaboration with other subjects. Its expressive power makes Scollar a tool that can (also) be used to study the propagation of authority in capability systems and patterns of collaborating entities.

The restriction to monotonic approximations (that are safe but may possibly be too crude) prevents us to directly express the revocation of authority. This is relevant for capability systems too because, even if access permissions cannot be revoked, it is very well possible (and easy) for a subject to revoke the authority it used to provide to its clients, for instance by refusing to collaborate any further, and no longer pass on any data or capabilities to them.

This is where the dominator part of DomReachability can be of direct use: to add expressive power to the safety requirements. Instead of simply stating that some effect (authority) should be prevented, we could instead require that all authority of a certain kind should only ever be available via a trusted subject that is able to revoke the authority. In the "authority-flow" graph (to be derived from the access-graph) a trusted subject Alice can revoke all Bob's authority over a third subject Carol, if Alice dominates Bob in the authority-flow graph that originates with Carol.

Chapter 8

Conclusions and Future Work

We have introduced two new NP-complete problems which are generalization of the Disjoint Paths Problem. We have defined three new constraints: *Reachability*, *Domination* and *DomReachability* for tackling those problems. We have defined the operational semantics of the propagators implementing these constraints by providing the corresponding pruning rules. We have implemented those pruning rules on top of state-of-the-art algorithms for computing dominators and transitive closure.

We have tested our approach in two real-case scenarios:

- **Solving constrained path problems.** We presented the Ordered Simple Path with Mandatory Nodes Problem (*OSPMN*) as an example of constrained path problems. *OSPMN* is to find a simple path in a directed graph that passes through a set of mandatory nodes respecting a given order on the mandatory nodes. We showed how *DomReachability* can speed up a standard approach for dealing with *OSPMN*. Our experiments show that the gain is increased when not all the nodes are mandatory.

  We compared the two implementations of *DomReachability* and showed that the *Gecode(CP(Graph))* implementation remarkably outperforms the *Oz* implementation. We also showed that the *Gecode(CP(Graph))* implementation allows us to be competitive with the approach presented in [BFL06].

  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.

- **Solving computer security problems.** We showed how the monotonic propagation of effects can be modeled with reachability constraints in a directed graph by associating nodes in the graph with subjects, and edges with permissions between the corresponding subjects. We elaborated on the relation of the resulting constrained graph problem with the Bounded Transitive

Closure problem (*BTC*): the problem of finding a directed graph that respects a set of reachability constraints (see section 2.3.1).

Even though the techniques presented in this thesis are mostly for dealing with combinatorial problems, we find, in computer security, polynomial problems that are better addressed with our approach. The main reason is that the information in *DomReachability* is updated incrementally.

We now suggest some ways of extending the work presented in this thesis:

8.1 Implementing more sophisticated algorithms for computing transitive closure

As explained in section , the current implementation of *DomReachability* is computing from scratch the transitive closure of the upper bound of the transitive closure graph each time a set of edges is removed. We believe that the algorithms presented in section 3.4 should outperform the current algorithm. Notice that each time we remove a set of edges, we pay $O(N * E)$ for updating the upper bound with the current algorithm. As shown in section 3.4, Roditty suggests a decremental approach for updating this information that is linear with respect to the size of the graph considering a set of edges removed.

8.2 Using dominators for detecting precedences

In section 6.6, we saw that dominators are more powerful than strong articulation points in conjunction with articulation points for detecting precedence constraints. In fact, we can see that the advantage is still kept even if *unique winners* are considered to enhance the approach of articulation points.

From the definition of winners given in [BFL05], we say that, given a strongly connected component (containing at least two nodes), a *unique winner* is a node i such that any tree (subgraph of the component) connecting the nodes of the components has i as a root. Unique winners are used for discovering precedences since a unique winner is preceeded by all the nodes in the components. This means that, if i is a unique winner of a component, and j is another node of the component, the edge $\langle i, j \rangle$ is invalid.

Nevertheless, there are situations where there are dominators but there are neither articulation points, nor strongly articulation points, nor unique winners, as shown in Figure 8.1. In this Figure, rounded rectangles represent strongly connected components. Assuming that s precedes t, using dominators in this cases will let us infer that s precedes d and that d precedes t.

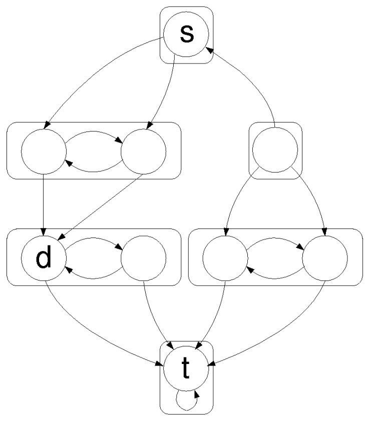


Figure 8.1: A directed graph, containing a dominator, that has neither articulation points, nor strong articulation points, nor unique winners

8.3 Towards a global constraint for CardBTC

In chapter 7, we introduced *CardBTC*: a very simple problem that allows us to express interesting graph problems on top of the notion of transitive closure and cardinality constraints. Our current approach for implementing *CardBTC* is to use *DomReachability* in conjuntion with Cardinality Constraints [VD91]. However, we can achieve a stronger level of pruning by taking into account the structure of the disjunctions and the bounds of the graph variables.

For instance, let us consider the CardBTC instance in Figure 8.3, whose semantics is the following:

$$\langle j,d\rangle \in TC(g) \wedge \langle k,d\rangle \in TC(g) \wedge (\langle i,j\rangle \in TC(g) \vee \langle i,k\rangle \in TC(g)) \quad (8.1)$$

From the previous constraint we can infer that $\langle i,d\rangle \in TC(g)$. Notice that this inference cannot be made by only looking at the bounds of the graph and its transitive closure. We also need to take into consideration that either i reaches j or i reaches k. In order to take this into account, we need to extend the pruning rules presented in chapter 3 so that the conditions include the information in the disjuction.

Let us re-write the semantics of the BTC instance as follows:

$$\begin{array}{c} \langle j,d\rangle \in TC(g) \wedge \langle k,d\rangle \in TC(g) \wedge \langle i,j\rangle \in TC(g) \\ \vee \\ \langle j,d\rangle \in TC(g) \wedge \langle k,d\rangle \in TC(g) \wedge \langle i,k\rangle \in TC(g) \end{array} \quad (8.2)$$

We can see that what we are suggesting is actually an application of Constructive Disjunction [VSD95] since $\langle i,d\rangle \in TC(g)$ is inferred from every conjunction

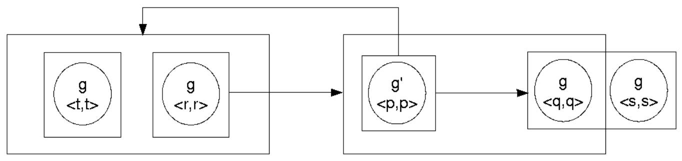


Figure 8.2: *CardBTC* instance associated with *SAT3* instance of Equation 8.3

composing the disjunction.

8.4 The underlying SAT problem of CardBTC

The price of allowing disjunctions in *CardBTC* is reflected in its complexity. Notice that *SAT3* [GJ79] can be trivially reduced to *CardBTC* by associating each proposition with an edge in g, which means that *CardBTC* is NP-complete even if no constraint on the transitive closure is imposed. For instance, the following *SAT3* instance is represented in Figure 8.2:

$$(p \vee q \vee s) \wedge (\neg p \vee q \vee \neg r) \wedge (p \vee r \vee t) \tag{8.3}$$

As explained in chapter 7, the cardinality constraints involved in *CardBTC* can be managed by using standard approaches based on cardinality propagators [VD91]. However, we can reason at a higher level of abstraction by looking at the Boolean Satisfiability instance that results from associating each basic graph constraint with literals. For instance, let us consider the following *CardBTC* instance:

$$(\langle i,j\rangle \in g \vee \langle j,k\rangle \in TC(g)) \wedge (\langle i,j\rangle \in g \vee \langle j,k\rangle \in TC(g)') \tag{8.4}$$

Notice that, the only possible solutions to the above *CardBTC* are graphs containing the edge $\langle i,j\rangle$. This inference can be made by only looking at the corresponding SAT instance:

$$(p \vee q) \wedge (p \vee \neg q) \tag{8.5}$$

which is equivalent to p.

Notice tha this is an application of Constructive Disjuction too since we are basically imposing that the intersection of the solutions satisfying every clause must be true.

Reasoning at the symbolic level would let us take advantage of BDD propagators to narrow down the literals composing a given disjunction [HLS05]. We could also consider hybrid approaches, like the one suggested in [HS06], in order to inherit the advantages offered by SAT solvers.

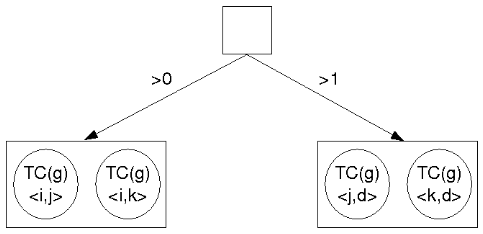


Figure 8.3: A *CardBTC* instance that implies that i reaches d

Bibliography

[AB04] Stefano Allesina and Antonio Bodini. Who dominate whom in the ecosystem? energy flow bottlenecks and cascading extinctions. *Journal of Theoretical Biology*, 230(3):351–358, 2004.

[AFM99] Slim Abdennadher, Thom W. Fruhwirth, and Holger Meuss. Confluence and semantics of constraint simplification rules. *Constraints*, 4(2):133–165, 1999.

[AFPB01] M. Amyeen, W. Fuchs, I. Pomeranz, and V. Boppana. Fault equivalence identification using redundancy information and static and dynamic extraction. In *Proc. IEEE VLSI Test Symp.*, 2001.

[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.

[BFL05] N. Beldiceanu, P. Flener, and X. Lorca. The tree constraint. In Roman Barták and Michela Milano, editors, *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems: Second International Conference, CPAIOR 2005, Proceedings*, volume 3524, pages 64–78, 2005.

[BFL06] N. Beldiceanu, P. Flener, and X. Lorca. Combining tree partitioning, precedence, incomparability, and degree constraints, with an application to phylogenetic and ordered-path problems. Technical Report 2006-020, Department of Information Technology, Uppsala University, 2006.

[BFO05] Stefano Bistarelli, Simon N. Foley, and Barry O'Sullivan. Reasoning about secure interoperation using soft constraints. In *Proc. IFIP TC1 WG1.7 Workshop on Formal Aspects in Security and Trust (FAST)*, volume 173 of *FIP International Federation for Information Processing*. Kluwer, August 2005.

[BL74] D.E. Bell and L. LaPadula. Secure computer systems. In *ESD-TR*, pages 83–278. Mitre Corporation, 1974. Electronically available at: *http://www.albany.edu/acc/courses/ia/classics/belllapadula1.pdf*.

[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.

[CHK] Keith D. Cooper, Timothy J. Harvey, and Ken Kennedy. A simple, fast dominance algorithm.

[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.

[DDD05a] 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.

[DDD05b] G. Dooms, Y. Deville, and P. Dupont. CP(Graph):introducing a graph computation domain in constraint programming. In *CP2005 Proceedings*, 2005.

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

[DH65] J. B. Dennis and E. C. Van Horn. Programming semantics for multiprogrammed computations. Technical Report MIT/LCS/TR-23, M.I.T. Laboratory for Computer Science, 1965.

[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/*.

[Doo06] Grégoire Dooms. *The CP(Graph) Computation Domain in Constraint Programming*. Doctoral dissertation, Université catholique de Louvain, Louvain-la-Neuve, Belgium, 2006.

[DZDD06] Grégoire Dooms, Stéphane Zampelli, Yves Deville, and Pierre Dupont. CP(Graph), 2006. Available at *http://cpgraph.info.ucl.ac.be*.

[Eck00] Bruce Eckel. *Thinking in C++: Introduction to Standard C++*, volume 1. Prentice Hall, 2000.

[Eck03] Bruce Eckel. *Thinking in C++: Practical Programming*, volume 2. Prentice Hall, 2003.

[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.

[FMNZ01] D. Frigioni, T. Miller, U. Nanni, and C. Zaroliagis. An experimental study of dynamic algorithms for transitive closure, 2001.

[Geo05] Loukas Georgiadis. *Linear-Time Algorithms for Dominators and Related Problems*. Doctoral dissertation, Princeton University, Princeton,USA, 2005.

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

[GWT$^+$04] L. Georgiadis, R. Werneck, R. Tarjan, S. Triantafyllis, and D. August. Finding dominators in practice. In *12th Annual European Symposium on Algorithms (ESA 2004)*, volume 3221 of *Lecture Notes in Artificial Intelligence*, pages 677–688. Springer-Verlag, 2004.

[Har95] David Harel. On visual formalisms. In Janice Glasgow, N. Hari Narayanan, and B. Chandrasekaran, editors, *Diagrammatic Reasoning*, pages 235–271. The MIT Press, Cambridge, Massachusetts, 1995.

[HLS05] P.J. Hawkins, V. Lagoon, and P.J. Stuckey. Solving set constraint satisfaction problems using robdds. *Journal of Artificial Intelligence Research*, 24:109–156, 2005.

[HRU76] Michael A. Harrison, Walter L. Ruzzo, and Jeffrey D. Ullman. Protection in operating systems. *Commun. ACM*, 19(8):461–471, 1976.

[HS06] Peter Hawkins and Peter Stuckey. A hybrid bdd and sat finite domain constraint solver. In *PADL 2006 Proceedings*, volume 3819 of *Lecture Notes in Computer Science*. Springer, 2006.

[Ita88] Giuseppe F. Italiano. Finding paths and deleting edges in directed acyclic graphs. *Inf. Process. Lett.*, 28(1):5–11, 1988.

[Jos99] Nicolai Josuttis. *The C++ Standard Library - A Tutorial and Reference*. Addison Wesley Professional, 1999.

[Kin99] V. King. Fully dynamic algorithms for maintaining all-pairs shortest paths and transitive closure in digraphs. In *40th Annual Symposium on Foundations of Computer Science*, pages 81–89, 1999.

[LLS01] Lie Quan Lee, Andrew Lumsdaine, and Jeremy Siek. *The Boost Graph Library: User Guide and Reference Manual*. Addison Wesley Professional, 2001.

[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.

[Mil06] Mark Samuel Miller. *Robust Composition: Towards a Unified Approach to Access Control and Concurrency Control*. PhD thesis, Johns Hopkins University, Baltimore, Maryland, USA, May 2006.

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

[MS98] Kim Marriott and Peter Stuckey. *Programming with Constraints*. The MIT Press, 1998.

[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.

[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.

[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.

[QVDC06] Luis Quesada, Peter Van Roy, Yves Deville, and Raphaël Collet. Using dominators for solving constrained path problems. In *PADL 2006 Proceedings*, volume 3819 of *Lecture Notes in Computer Science*. Springer, 2006.

[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.

[RN03] S. Russell and P. Norvig. *Artifical Intelligence: A Modern Approach*. Prentice Hall, 2003.

[Rod03] Liam Roditty. A faster and simpler fully dynamic transitive closure. In *SODA '03: Proceedings of the fourteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 404–412, Philadelphia, PA, USA, 2003. Society for Industrial and Applied Mathematics.

[RZ02] L. Roditty and U. Zwick. Improved dynamic reachability algorithms for directed graphs, 2002.

[Sar93] Vijay Saraswat. *Concurrent Constraint Programming*. The MIT Press, 1993.

[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.

[SJV05] Fred Spiessens, Yves Jaradin, and Peter Van Roy. Using constraints to analyze and generate safe capability patterns. Research Report INFO-2005-11, Département d'Ingénierie Informatique, Université catholique de Louvain, Louvain-la-Neuve Belgium, 2005. Presented at CPSec'05. Available at
http://www.info.ucl.ac.be/people/fsp/rr2005-11.pdf.

[SLT06] Christian Schulte, Mikael Lagerkvist, and Guido Tack. Gecode, 2006. Available at *http://www.gecode.org*.

[SP78] Y. Shiloach and Y. Perl. Finding two disjoint paths between two pairs of vertices in a graph. *Journal of the ACM*, 1978.

[Spi06] Alfred Spiessens. *Patterns of Safe Collaboration*. Doctoral dissertation, Université catholique de Louvain, Louvain-la-Neuve, Belgium, 2006.

[SQV06] F. Spiessens, L. Quesada, and P. Van Roy. Confinement analysis with graph reachabilty constraints. In *International Workshop on Constraints in Software Testing, Verification and Analysis (CSTVA06), at the 12th International Conference on Principles and Practice of Constraint Programming (CP2006)*, 2006.

[ST06] Christian Schulte and Guido Tack. Views and iterators for generic constraint implementations. In *Recent Advances in Constraints (2005)*, volume 3978 of *Lecture Notes in Artificial Intelligence*, pages 118–132. Springer-Verlag, 2006.

[Str97] Bjarne Stroustrup. *The C++ Programming Language*. Addison Wesley Professional, 1997.

[VCAS05] Andreas Veneris, Robert Chang, Magdy S. Abadir, and Sep Seyedi. Functional fault equivalence and diagnostic test generation in combinational logic circuits using conventional atpg. *J. Electron. Test.*, 21(5):495–502, 2005.

[VD91] Pascal Van Hentenryck and Yves Deville. The cardinality operator: A new logical connective for constraint logic programming. In *Proceedings of the Eighth International Conference on Logic Programming*, pages 745–759. MIT Press, 1991.

[VH04] P. Van Roy and S. Haridi. *Concepts, Techniques, and Models of Computer Programming*. The MIT Press, 2004.

[VSD95] Pascal Van Hentenryck, Vijay Saraswat, and Yves Deville. Design, implementation, and evaluation of the constraint language cc(FD). In Andreas Podelski, editor, *Constraint Programming: Basics and Trends*. Springer, 1995.

[WG04] R. Werneck and L. Georgiadis. Implementation of the most relevant algorithms for computing dominators, 2004. Available at *http://www.cs.princeton.edu/ rwerneck/dominators/*.

[WS98] D.J. Watts and S.H. Strogatz. Collective dynamics of 'small-world' networks. *Nature*, 393:440–442, June 1998.