Discrimination Nets: Improvement and Extension to Bang Graphs

Fabio Massimo Zennaro
Balliol College
University of Oxford

A thesis submitted for the degree of
MSc in Mathematics and Foundations of Computer Science
September 2012

Abstract

String graphs constitute a graphical language to represent many types of processes and phenomena, including quantum phenomena. Because of their discrete nature, string graphs can be easily processed by a computer. A well-known family of filtering techniques used to speed up the process of graph matching are discrimination nets. Discrimination nets were applied to string graphs by Douglas in 2010. In this dissertation we first propose an improved algorithm for discrimination nets when working with string graphs; then we define an extension of the algorithm for discrimination nets when dealing with !-graphs. We give a proof of the correctness of our algorithms and we offer an implementation of these algorithms within the context of Quantomatic. Finally we present a set of results obtained by running simulations of our algorithms with randomly generated string graphs and !-graphs.

Acknowledgements

First of all, I would like to thank Aleks Kissinger. Thanks for your constant help during the months I spent working on my thesis, for your patience and your being always available to meet with me, for your suggestions and your explanations during our discussions.

An equally sincere thank is for Bob Coecke. Thanks for your overwhelming passion and trust which persuaded me to undertake this project.

I would also like to express my gratitude to all the people that made this course and my study here possible: Dan Olteanu, Michael Collins, Wendy Adams, Margaret Sloper, Torimitsu Keiichi, Matteo Matteucci, Rossella Blatt, Gianluigi Furioli, Roberto Sassi, Roberto Negrini and Franca Chiusi. A list of names will never do justice to the unique contribution that each of you offered me.

The last line is for my parents and my friends. You do not have an explicit thanks here on this page. It would be diminutive. You have my daily, most heartfelt thanks.

Contents

Chapter 1

Introduction

1.1 Motivation and Aim

This dissertation stands at the convergence of two main fields of study: categorical quantum mechanics and term rewriting theory.

Categorical quantum mechanics (CQM) is a recent field of research devoted to the study of quantum mechanics using the tools offered by *category theory*.

One of the main advantages of CQM is the possibility of representing and reasoning about quantum phenomena in a graphical way. *String diagrams* provide an intuitive diagrammatic language to represent quantum phenomena in a visual, yet rigorous, way. The diagrammatic language of string diagrams defines an abstraction layer which allows a researcher in the field of quantum physics or quantum information to reason at a higher level than the level of physical quantum phenomena.

In order to work with string diagrams, it is crucial to develop our ability to process these diagrams efficiently. We can discretize and convert string diagrams into *string graphs*. In this way, a string graph can be fed into a computer and can be quickly processed and mechanically manipulated.

Term rewriting theory is a branch of mathematics and computer science dealing with abstract term systems and studying ways in which terms can be rewritten into other terms. The original problem that term rewriting tries to solve is the so-called word problem: given two terms and a set of identities, we would like to know if it is possible to transform the first term in the second one using the identities as a set of rewrite rules. Term rewriting defines then a set of mechanical procedures by which we can manipulate a term in order to generate new, potentially interesting, terms.

By implementing term rewriting systems on a computer we can create basic automated reasoning software tools or basic proof assistants. These software tools can work with terms or graphs and mechanically process them in order to find out ways in which terms or graphs can be rewritten. As an exhaustive search of all the possible ways in which a term or a graph can be rewritten is, in general, uncomputable, different techniques have been developed and implemented to make this process more efficient than a simple exhaustive research.

In this dissertation we will consider the application of term rewriting techniques to CQM string graphs.

As we have said before, after encoding quantum phenomena with the language of string diagrams and after feeding string graphs to a computer, we need a fast and efficient way to process these diagrams. Term rewriting theory provides us with the theory and with the techniques to manipulate mechanically string graphs. Indeed, if we start from string graphs encoding a set of theorems from the field of quantum mechanics, we can use term rewriting techniques to manipulate the set of graphs and potentially discover new, unforeseen theorems.

Being able to reason automatically and efficiently about string graphs is then a very important task which allows us to discover new theorems and new facts about quantum physics and quantum information in a quick and systematic way. Different algorithms can be implemented to make this task computable and efficient.

Quantomatic, for example, is a software tool built to offer to the researcher in the field of CQM a tool to help him working and reasoning with string graphs.

An extension to the language of string graphs which has been proposed recently is the definition of the language of *!-graphs* ("*bang graphs*"). !-graphs provide a way to express abstract string graphs which can then be instantiated into several different string graphs. While a string graph rewrite rule can represent a single quantum theorem, a !-graph rewrite rule can represent a family of quantum theorems. From a single !-graph we can derive an infinite number of different string graphs.

Several algorithms for term rewriting which could be straightforwardly implemented in the case of string graphs require to be reviewed and modified in order to be applied to !-graphs.

Among these techniques, *discrimination nets* can be used to improve and speed up the process of rewriting through a reduction of the search space of the rewrite rules. While the implementation of discrimination nets for string graphs is known, the application to !-graphs requires a redefinition and a generalization of the algorithm.

The main aim of this dissertation is then to analyze how discrimination nets work for string graphs, examine how they can be improved and extended in the case of !-graphs, define a new algorithm and prove its correctness.

Once a new algorithm has been defined and proved to be correct, the next goal of the dissertation is to provide a working implementation of the algorithm as an extension of Quantomatic.

After this introduction, in the second chapter we will give a brief outline of the work done in the fields of categorical quantum mechanics and graph rewrite theory; we will review all the background notions required to deal with the problem of working with !-graphs. In the third chapter we will introduce an existing technique used to improve the performance of graph matching for string graphs; we will then present an improved version of this algorithm working with string graphs and then an extended version of the algorithm working with !-graphs. In the fourth chapter we will discuss how our algorithms have been implemented within the framework of Quantomatic and we will comment on their performance. In the fifth chapter we will summarize the results we obtained and we will suggest possible future developments of this project.

In a final appendix we will give a more detailed and exhaustive explanation of the algorithm we implemented in Poly/ML.

Chapter 2

Background

The research in the field of CQM started in 2004 with the proposal by Abramsky and Coecke [1] to adopt a categorical approach to the study of quantum mechanics. In their article, they showed that only abstract, categorical structure was necessary to explain many quantum phenomena; quantum information phenomena, such as teleportation or entanglement-swapping, could be easily formalized using the language of strongly compact closed categories with biproducts. In 2005 Selinger [26] formalized the graphical language for dagger compact closed categories (which he called strongly compact closed categories) and showed their completeness for equational reasoning. The graphical language used by Abramsky, Coecke and Selinger to deal with dagger compact categories is rooted in the work on string diagrams started by Penrose [25] and in the work on tensor categories done by Joyal and Street [14].

Term rewriting is born out from the study of logic, universal algebra, automatic theorem proving and functional programming [3]. The study of term rewriting systems dates back to the results on λ-calculus by Church and Rosser in the beginning of the 20th century, to the development of combinatory logic by Curry in the 1960s and to the research on denotational semantics of programming languages by Scott and Plotkin in the 1970s [19]. Tackling the problem of termination and convergence, Knuth and Bendix proposed in 1970 the *Knuth-Bendix completion algorithm* to convert a set of terms into a confluent term rewriting system [20, 6].

The effort to make implemented rewrite systems more efficient led to the development of several algorithms. In the 1970s researchers in the field of expert systems developed *discrimination net* algorithms to speed up the solution of matching problems; at the beginning of 1990s these algorithms were first used to improve the performance of rewriting and theorem proving systems [4, 22]. More recently, other algorithms and techniques have been introduced to simplify and improve rewriting; in 2010 Johansson, Dixon and Bundy proposed the

conjecture synthesis technique to prevent the generation of reducible formulas during the process of rewriting [13]; in the same year Montano-Rivas, McCasland, Dixon and Bundy described the *scheme-based synthesis* technique used to direct the rewriting process through the definition of schemes.

The study of string graph theory within the field of CQM is based on the results proved in the recent years by Dixon, Duncan and Kissinger; in 2010 they described string graphs using the formalism of open graphs and they showed how string graph rewriting can be performed using the double-pushout technique [9]; in 2012 Kissinger showed how the conjecture synthesis technique can be applied to the synthesis of graphical theories [16].

The idea of !-graphs was introduced for the first time in 2008 by Dixon and Duncan [7]; later, in 2012, Kissinger, Merry and Soloviev formalized the concept and the theory of !-graphs [18].

Tools to help researchers working with string graphs and !-graphs have been implemented within the Quantomatic Project [17]; the Quantomatic Project was developed using the Isabelle theorem prover libraries [23] and provides the user with a graphical user interface to work with string graphs and !-graphs.

In this chapter we will give an introduction to the background topics in which this dissertation is grounded; we will deal with topics from categorical quantum mechanics (section 2.1), theory of string graphs (section 2.2 and 2.3), rewrite theory (section 2.4), string graph rewrite theory (section 2.5) and regular expressions (2.6).

2.1 Categorical Quantum Mechanics

Quantum computing is an area of research devoted to the exploitation of *quantum mechanics* to perform computations; in particular, quantum computing tries to exploit particular quantum phenomena, such as superposition or entanglement, to develop algorithms which can perform significantly better than their classical counterparts.

Traditionally, (pure state) quantum mechanics is mathematically represented using the Von Neumann formalism in which quantum states are vectors in a Hilbert space, state evolution is represented by linear unitary maps and measurements are represented by projections [8].

An alternative approach to the study of quantum mechanics using category theory was proposed by Abramsky and Coecke in 2004 [1]. This new field of research, called *categorical quantum mechanics (CQM),* defines a new way to model quantum phenomena using the

formalism of category theory and relies on the use of a diagrammatic notation to perform computations [11, 5]. Instead of working in the context of Hilbert spaces, CQM focuses on more abstract structures, such as *monoidal categories*, in order to study the categorical and compositional aspects of quantum phenomena [15].

Monoidal categories provide a versatile setting to describe quantum phenomena and offers an intuitive and elegant way to represent these phenomena using diagrams [12]. A *(planar) monoidal category* is a category $\mathbf{C}$ equipped with:

- A *tensor product* $\otimes : \mathbf{C} \times \mathbf{C} \to \mathbf{C}$;
- A *unit object* $I \in \mathbf{C}$;
- *Associators*, that is a family of natural isomorphisms $\alpha_{A,B,C} : (A \otimes B) \otimes C \to A \otimes (B \otimes C)$ for all objects $A, B, C \in \mathbf{C}$;
- *Left unitors*, that is a family of natural isomorphisms $\lambda_A : I \otimes A \to A$ for all objects $A \in \mathbf{C}$;
- *Right unitors*, that is a family of natural isomorphisms $\rho_A : A \otimes I \to A$ for all objects $A \in \mathbf{C}$;
- A *coherence property* for α, λ and ρ, guaranteeing that every well-formed equation built from $\circ, \otimes, id, \alpha, \alpha^{-1}, \lambda, \lambda^{-1}, \rho, \rho^{-1}$ is satisfied.
  The coherence property can be stated as the condition that, given the natural isomorphisms α, λ and ρ, then for any $A, B, C \in \mathbf{C}$ the triangle diagram (see figure 2.1) commutes and for any $A, B, C, D \in \mathbf{C}$ the pentagon diagram (see figure 2.2) commutes. It was proved by Mac Lane [21] that for any monoidal category $\mathbf{C}$, the natural isomorphisms α, λ and ρ are coherent if and only if the triangle equation and the pentagon equation hold.

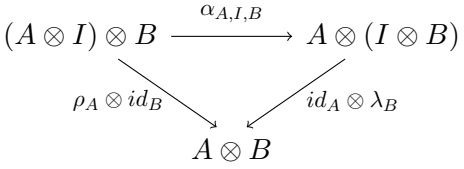


Figure 2.1: Triangle diagram.

Monoidal categories define a framework to work with generalized processes: given a monoidal category $\mathbf{C}$, its morphisms can be interpreted as processes, its objects define how processes can be combined, the categorical composition ($\circ$) allows the composition of processes in time and the tensor product ($\otimes$) allows composition of processes in space [16].

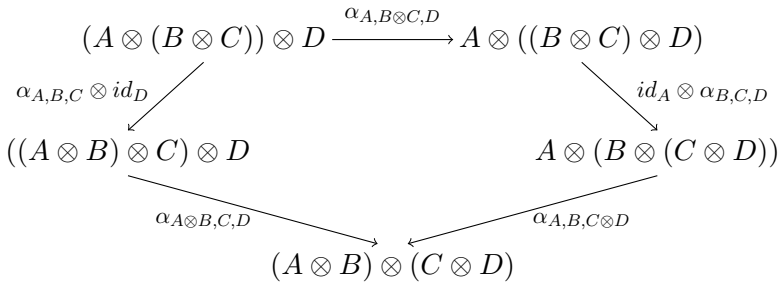


Figure 2.2: Pentagon diagram.

FdHilb, the category of finite-dimensional Hilbert spaces, is an example of a monoidal category.

In the context of CQM, it is often convenient to work with categories having more structure than simple monoidal categories. One specific kind of monoidal category of interest is a *symmetric traced category* [27]. A symmetric traced category is a monoidal category $\mathbf{C}$ equipped with:

- A family of natural isomorphisms $\sigma_{A,B} : A \otimes B \to B \otimes A$ for all objects $A, B \in \mathbf{C}$;
- A *symmetry property* $\sigma_{B,A} \circ \sigma_{A,B} = id_{A\otimes B}$ for all objects $A, B \in \mathbf{C}$;
- A *right trace*, that is a family of operations between hom-sets $tr_X : hom(A \otimes X, B \otimes X) \to hom(A, B)$ satisfying the following axioms:

  Tightening (naturality in A,B): $tr_X((g \otimes id_X) \circ f \circ (h \otimes id_X)) = g \circ (tr_X(f)) \circ h$;

  Sliding (dinaturality in X): $tr_Y(f \circ (id_A \otimes g)) = tr_X((id_B \otimes g) \circ f)$, where $f : A \otimes X \to B \otimes Y$ and $g : Y \to X$;

  Vanishing: $tr_I(f) = f$ and $tr_{X\otimes Y}(f) = tr_X(tr_Y(f))$;

  Strength: $tr_X(g \otimes f) = g \otimes tr_X(f)$

Symmetric traced categories extend simple monoidal categories allowing us new operations on the processes: the symmetry property allows us to perform swaps in space, the trace allows us to perform feedbacks in time [16].

Diagrammatic notation is an elegant way to express complex term-based expressions using a graphical, intuitive diagram relying on the idea of processes and composition of processes. When representing a planar monoidal category in the diagrammatic notation, morphisms or processes are represented as *boxes*, the composition of processes is represented by *wires* connecting boxes and the tensor product is represented as the simple juxtaposition

of boxes (see figure 2.3); when representing a symmetric category, the symmetry property allows us the *crossing* of wires (see figure 2.4); when representing a traced symmetric category, the trace allows us to draw *feedbacks* to the boxes (see figure 2.5).

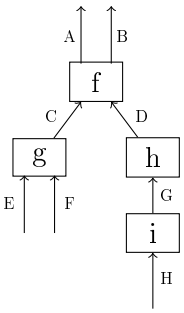


Figure 2.3: Example of diagrammatic notation for planar monoidal categories.

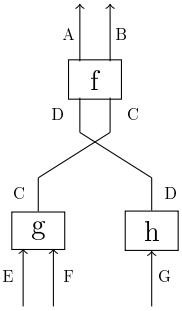


Figure 2.4: Example of diagrammatic notation for symmetric categories.

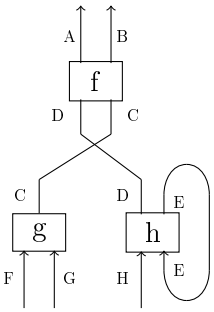


Figure 2.5: Example of diagrammatic notation for symmetric traced categories.

Diagrams are then a versatile and powerful way to represent information graphically and to reason about it. The specific family of diagrams used within the context of monoidal categories is called *string diagrams*.

2.2 String Graphs

As explained in the previous section, string diagrams arise from the diagrammatic notation used to represent monoidal categories. String diagrams are widely used to model several

types of compositional structures such as physical processes, logic circuits or tensor networks [18].

Generally speaking, a string diagram G is composed by *nodes* connected by directed edges, called *wires* (see figure 2.6). Nodes are used to represent processes with a fixed number of inputs and outputs, while wires are used to represent transitions from one process to another.

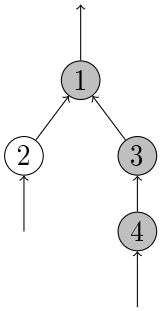


Figure 2.6: Example of a string diagram.

Topological graphs such as string diagrams are difficult to represent and manipulate on a computer; for this reason, it is convenient to discretize string diagrams and convert them to *string graphs* [9]. A string graph is a discrete version of a string diagram, in which every node has been substituted with an equivalent *node-vertex* and every wire has been substituted with a chain of *wire-vertices*.
So, a string graph G is a graph made up by (see figure 2.7):

Node-vertices are vertices representing a process; every node-vertex n has a given *kind,* $kind_{NV}(n)$, a fixed input arity, $input(n)$, and a fixed output arity, $output(n)$; the set of all the node-vertices of a string graph G is denoted as $NV(G)$; the set of all the kinds of node-vertices is denoted as $NVK(G)$;

Wire-vertices are vertices representing connections between processes; wire-vertices are connected to other wire-vertices or to node-vertices by directed edges; every wire-vertex w has a given kind, $kind_{WV}(w)$, and it has at most one input and one output; wire-vertices can not split nor merge; the set of all the wire-vertices of a string graph G is denoted $WV(G)$; the set of all the kinds of the wire-vertices in G is denoted as $WVK(G)$.

When we convert a wire into a chain of wire-vertices, the specific number of wire-vertices instantiated is not semantically relevant (see figure 2.8) [16]. Indeed, we can define an equivalence relationship called *wire-homeomorphism* between a wire chain with a single wire-vertex and a wire chain with an arbitrary number of wire-vertices. Given two graphs differing only for the number of wire-vertices in their wire-chains, we say that the two graph

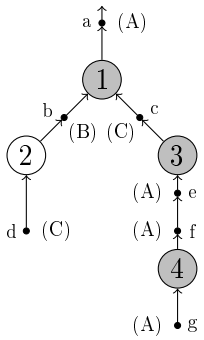


Figure 2.7: Example of a string graph where $NV = \{1, 2, 3, 4\}$, $NVK = \{"white", "gray"\}$, $WV = \{a, b, c, d, e, f, g\}$, $WVK = \{A, B, C\}$, $Input = \{d, g\}$, $Output = \{a\}$ and $Bound = \{a, d, g\}$.

are *wire-homeomorphic* and then, for the task of representing string diagrams, equivalent. Wire-homeomorphism can be formalized as a convergent graph rewrite system (see section 2.4). We define a string graph a *reduced string graph* if all the wire chains are composed by a single wire-vertex [15]. In the following chapter we assume that we will always work with reduced string graphs.

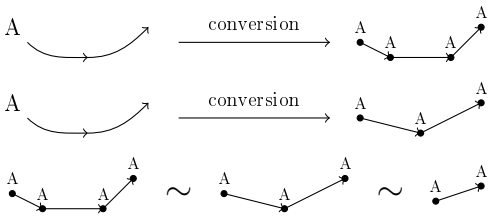


Figure 2.8: Conversion of a wire into a wire chain.

Given a string graph G and a directed edge e, we define $source(e)$ the node-vertex $n \in NV(G)$ or the wire-vertex $w \in WV(G)$ from which the edge e originates; we define $target(e)$ the node-vertex $n \in NV(G)$ or the wire-vertex $w \in WV(G)$ in which the edge e ends.

Given a string graph G, a wire-vertex $w \in WV(G)$ with an output, but no input, is defined *input*; we denote $Input(G)$ as the set of all the input wire-vertices of the graph G, that is all the $w \in WV(G)$ such that there is no edge e such that $target(e) = w$. Conversely, given a graph G, a wire-vertex w with an input, but no output, is defined *output*; we denote $Output(G)$ as the set of all the output wire-vertices of the graph G, that is all the $w \in WV(G)$ such that there is no edge e such that $source(e) = w$.

Together, $Input(G)$ and $Output(G)$ constitute the boundary of a string graph G. We define $Bound(G) = Input(G) \cup Output(G)$.

An important notion to deal with string graphs is *neighbourhood*. Given a string graph G and a node-vertex $n \in NV(G)$, we define *neighbourhood* of n, $nhd(n)$, the set of all the edges adjacent to n, that is all the edges e such that $target(e) = n$ or $source(e) = n$.

String graphs can be interpreted as objects in the category $\mathbf{SGraph}_T$ [16]. Let T be a (small, strict) monoidal signature composed of:

- a set $\mathcal{O}$ of objects;
- a set $\mathcal{M}$ of morphisms;
- two functions dom, $cod : \mathcal{M} \to w(\mathcal{O})$, where $w(\mathcal{O})$ is the set of lists over the set $\mathcal{O}$, assigning input and output kinds to each morphism.

Then $\mathbf{SGraph}_T$ is the category of string graphs parametrized by the monoidal signature T, where the objects of $\mathbf{SGraph}_T$ are string graphs and the morphisms of $\mathbf{SGraph}_T$ are string graph homomorphisms, that is graph homomorphisms respecting the types of node-vertices and wire-vertices.

In the next pages, we will consider string graphs parametrized by a signature T in which the set of objects $\mathcal{O}$ is composed by a single element; this means that the domain and the codomain of all the morphisms is the same. We will also assume that all the morphisms are commutative and cocommutative. Practically, instead of working with specific morphisms $(f, g, h...)$ we will work with families of morphisms having a given input arity and a given output arity $(f_0^0, f_0^1, f_1^0, f_1^1, f_0^2...)$.

Given a category $\mathbf{C}$ and an object C belonging to $\mathbf{C}$ the slice category $\mathbf{C}/C$ is a category such that [2]:

- the objects of $\mathbf{C}/C$ are the arrows f belonging to $\mathbf{C}$ such that $cod(f) = C$
- the morphisms of $\mathbf{C}/C$ are morphisms g from $f : X \to C$ to $f' : X' \to C$ such that $g : X \to X'$ is a morphism in $\mathbf{C}$ such that $f' \circ g = f$ (see figure 2.9)

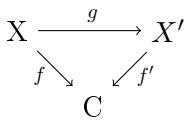


Figure 2.9: Morphisms for $\mathbf{C}/C$.

Using this definition, $\mathbf{SGraph}_T$ can also be considered as a full subcategory of the slice category $\mathbf{Graph}/G_T$ [9], where:

- **Graph** is the category of graphs, defined as a functor category $[\mathbb{G}, \mathbf{Set}]$, $\mathbb{G}$ being:

$$E \underset{t}{\overset{s}{\rightrightarrows}} V$$

where E are the edges of the graph, V the vertices (node-vertices and wire-vertices) of the graph, s the function taking each edge to its source and t the function taking each edge to its target.

- G_T is the derived typegraph, defining the type and the kind of the vertices and the way in which they are connected; given a graph G we say that a graph is typed if there is a typing morphism $\tau : G \to G_T$; for a string graph composed by node-vertices having a single kind and wire-vertices having a single kind we will use the typegraph 2_G (see figure 2.10) defining node-vertices (V) and wire-vertices (W); for a string graph composed by node-vertices having two kinds and wire-vertices having a single kind we will use the typegraph represented in figure 2.11.

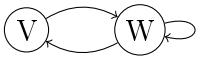


Figure 2.10: Typegraph 2_G.

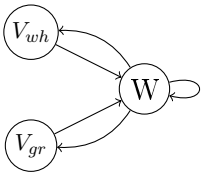


Figure 2.11: Typegraph defining "white" node-vertices (V_{wh}), "gray" node-vertices (V_{gr}) and wire-vertices (W).

2.3 Bang Graphs

A useful extension of string graphs are *!-graphs* (*"bang" graphs*) [15]. !-graphs are created from string graphs by introducing *!-vertices* or *!-boxes*. Given a string graph G a !-vertex is a special type of vertex connected to a portion of the graph (including its incident edges) which can be instantiated an arbitrary number of times. For simplicity a !-graph can be represented drawing !-boxes instead of !-vertices: all the portion of the graph reached by a !-vertex can be drawn inside a !-box (see figure 2.12 and 2.13)

So a !-graph is a graph made up of:

Node-vertices as in the case of string graphs;

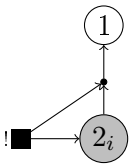


Figure 2.12: Example of a !-graph with !-vertices.

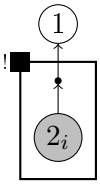


Figure 2.13: Example of a !-graph with !-boxes.

Wire-vertices as in the case of string graphs, but with the exception that a wire-vertex w can now have multiple inputs, a single input coming from a node-vertex or a wire-vertex and multiple optional inputs coming from !-vertices;

!-vertices are abstract vertices representing the possibility of duplicating a set of processes an arbitrary number of times; every !-vertex b can have out-edges going to node-vertices, wire-vertices and other !-vertices, but it can have in-edges coming only from other !-vertices; the set of all the !-vertices of the !-graph is denoted as $!V(G)$.

For consistency, we require the content of a !-box to be an open graph. Given a string graph G we say that a subgraph O contained in G is an *open subgraph* if $Input(G\backslash O) \subseteq Input(G)$ and $Output(G\backslash O) \subseteq Output(G)$. The notion of open subgraph guarantees that by subtracting the subgraph O from G no new boundaries are created and so it implies that the content of a !-box contains complete wires [18]. This means that if a !-vertex b has an out-edge to a wire-vertex w_1 in a wire chain, then all the wire-vertices w_i in the same wire chain must have an in-edge coming from b; this is equivalent to say that if a wire-vertex w_1 belonging to a wire chain is in the !-box b, then all the wire-vertices w_i in the same wire chain must be inside the !-box b. In this way, graphs as the one shown in figure 2.14 are ruled out as incorrect. Indeed, considering the graph in figure 2.14, if the !-box would be instantiated more than one time, so it would be the wire-vertex w_2; but this would force the input arity of the wire-vertex w_1 to be greater than one, which is not acceptable.

Notice that, because of the way in which the inputs of a wire-vertex has been redefined, also the notion of input of a !-graph must be redefined. Given a !-graph G, we define *input* of G a wire-vertex $w \in WV(G)$ without any incoming edge or a wire-vertex $w \in WV(G)$ such that all its incoming edges come from !-vertices; the set of these input wire-vertices is

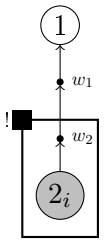


Figure 2.14: Example of an incorrect !-graph.

denoted $Input(G)$.

The definition of outputs $Output(G)$ does not change.

Because of the presence of !-boxes (or !-vertices) a !-graph can not be considered a concrete string graph anymore; indeed a !-graph can be interpreted as a rule to generate concrete string graphs which can be instantiated using operations inspired to the bang operations of the linear logic [18] (see figure 2.15). Given a !-graph G, the *instantiation* $G \preceq_! G^*$ is a string graph G^* generated from G applying one or more of the following four operations:

- *Copy*: duplicates a !-box and its content;
- *Drop*: removes a !-box;
- *Kill*: removes a !-box and its content;
- *Merge*: joins together two !-boxes and their content.

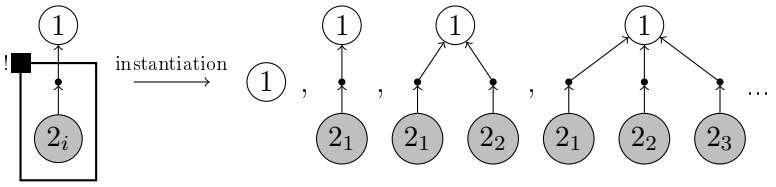


Figure 2.15: Instantiation of a !-graph.

!-graphs can be interpreted as objects in the category **!Graph**, whose objects are !-graphs and whose morphisms are !-graph homomorphisms, that is graph homomorphisms respecting the types of node-vertices, wire-vertices and !-vertices.

!Graph can also be considered as a full subcategory of the slice category $\mathbf{Graph}/G_T$, where the typegraph G_T defines the type and the kind of vertices in the !-graph; for a !-graph having a single kind of node-vertices and a single kind of wire-vertices the typegraph

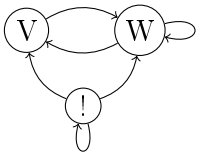


Figure 2.16: Typegraph 3_G.

3_G (see image 2.16) defines how node-vertices (V), wire-vertices (W) and !-vertices (!) can be connected [18].

A graph G is then a !-graph if it is a typed graph and if it respects the following constraints:

1. the full subgraph with vertices $NV(G) \cup WV(G)$ is a string graph;
2. the full subgraph with vertices $!V(G)$ is posetal, that is !-vertices have at most one wire between them and they form a partial order;
3. for all $b \in !V(B)$, the graph B in the !-box associated with b is an open subgraph of G;
4. for all $b, b' \in !V(B)$, if b' is contained in the !-box associated with b, then all the elements in the !-box associated with b' must also be in the !-box associated with b.

2.4 Rewriting Systems

We now introduce the definition of a rewriting system, which will be applied to string graphs and !-graphs.

An abstract *rewriting (or reduction) system* $\mathcal{R}$ is a pair $(A, \rightarrow)$ where:

- A is a set;
- $\rightarrow \subseteq A \times A$ is a binary relation on the set A.

Interpreting A as a set of formulas, $\mathcal{R}$ defines a set of rewrite rules $f \rightarrow f'$ specifying how a formula f can be rewritten into an equivalent formula f' [3].

Given a rewrite rule $f \rightarrow f'$ and a generic formula g, *matching* is the process in which we look for a matching between the formula g (or a subpart of the formula g) and the formula f; *replacement* is the process in which the formula g (or the subpart of the formula g) which matched f is substituted with f'.

Given a rewrite system $\mathcal{R}$ and a formula f, we say that:

- f is *reducible* if $\mathcal{R}$ contains a rewrite rule matching f (or a subpart of f);

- f is *irreducible* if $\mathcal{R}$ does not contain a rewrite rule matching f (or a subpart of f); in this case we say that f is in *normal form*.

We define $\stackrel{*}{\rightarrow}$ as the reflexive and transitive closure of the binary relation $\rightarrow$. $f \stackrel{*}{\rightarrow} f'$ is equivalent to $f \rightarrow f_1 \rightarrow f_2 \rightarrow \ldots \rightarrow f'$, which means that in the rewrite system $\mathcal{R}$ there is a sequence of rules which allow us to rewrite f as f'.

When working with a rewrite system $\mathcal{R}$ we are interested in the following properties :

- *Termination*: a rewrite system $\mathcal{R}$ is terminating if, given any formula $f \in A$, then any chain of reductions, $f \stackrel{*}{\rightarrow} \hat{f}$ leading from f to the normal form $\hat{f}$, contains a finite number of reductions; termination guarantees that the reduction process will not end up in infinite loops.

- *Confluence*: a rewrite system $\mathcal{R}$ is confluent if, given any formula $f \in A$, then for any two chains of reductions $f \stackrel{*}{\rightarrow} f_1$ and $f \stackrel{*}{\rightarrow} f_2$, leading from f to f_1 and from f to f_2, there is a formula f_3 such that there exist the reflexive and transitive closures $f_1 \stackrel{*}{\rightarrow} f_3$ and $f_2 \stackrel{*}{\rightarrow} f_3$; if f has a normal form $\hat{f}$, then confluence guarantees the uniqueness of the normal form $\hat{f}$.

- *Convergence*: a rewrite system $\mathcal{R}$ is convergent if it is terminating and confluent; convergence guarantees the well-behaviour of a rewriting system: for any formula f it is always possible to get to its normal form $\hat{f}$ in a finite number of reductions [3].

It is worth noticing that if it is possible to find an ordering in the rewrite rules of a rewrite system $\mathcal{R}$, then $\mathcal{R}$ is terminating. More precisely, given a well-founded poset $(P, \leq)$, that is a partially ordered set with a smallest element and no infinite sequence of strictly decreasing elements, if the rewrite system $\mathcal{R}$ admits a function $\omega : A \rightarrow P$ such that for any $f_1, f_2 \in A$, $f_1 \rightarrow f_2 \Rightarrow \omega(f_1) > \omega(f_2)$, then $\mathcal{R}$ is terminating [15].

An example of very well-known and well-studied concrete rewrite systems are *term rewriting systems*. In a term rewrite system the set A is a set of terms, where a term is a formula built out from a set of variables V and from a set of function symbols F, and the binary relation $\rightarrow$ is the relation specifying how a term is reduced to another term.

Another example of concrete rewrite systems closer to our interest are *graph rewrite systems*. In a general graph rewrite system $(A, \rightarrow)$ the set A is a set of graphs and the binary relation $\rightarrow$ is the relation determining how a graph can be redrawn as another graph.

A specific type of graph rewrite systems are *string graph rewrite systems*.

2.5 String Graph Rewriting Systems

A string graph rewriting system $\mathcal{R} = (A, \rightarrow)$ is a pair composed by a set A of string graphs and by a binary relation $\rightarrow$ determining how to convert a string graph into another string graph. The set of rules composing $\mathcal{R}$ is a set of rules of type $L \rightarrow R$, defining how the string graph L on the left hand-side can be converted into the string graph R on the right hand-side.

Let's recall that given a category $\mathbf{C}$ and two objects X and Y belonging to $\mathbf{C}$, then a span is a diagram of the form shown in figure 2.17, where S is an object of $\mathbf{C}$.

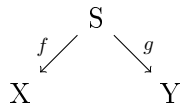


Figure 2.17: Span of X and Y in $\mathbf{C}$.

A string graph rewrite rule $L \rightarrow R$ can be defined as [16]:

Definition (String graph rewrite rule) A string graph rewrite rule $L \rightarrow R$ is a span of monomorphisms $L \overset{b_1}{\leftarrow} B \overset{b_2}{\rightarrow} R$ such that:

- $b_1(B) = Bound(L)$
- $b_2(B) = Bound(R)$
- $\forall b \in B,\ b_1(B) \in Input(L) \Leftrightarrow b_2(B) \in Input(R)$

Therefore, the monomorphisms b_1 and b_2 map the objects in B to the boundary of L and to the boundary of R in such a way that if $b \in B$ is mapped by b_1 to the input (resp. output) of L then it is mapped by b_2 to the input (resp. output) of R and vice versa.

Now, given a string graph rewrite rule and a string graph G we are interested in finding a possible matching between G (or a subpart of G) and L. Before giving a definition of a matching, it is useful to introduce the concept of local isomorphism [16]:

Definition (Local isomorphism for string graphs) Given two string graphs G and H and a string graph homomorphism between them $f : G \rightarrow H$, f is a *local isomorphism* if for every node-vertex $n \in NV(G)$ then the restriction of f on the $nhd(n)$ restricts to a bijection of the edges $f\,|_{nhd(n)}$: $nhd(n) \tilde{\rightarrow} nhd(f(n))$.

In other words, $f : G \to H$ is a *local isomorphism* if f is a monomorphism and for every node-vertex $n \in NV(G)$ the image of the neighbourhood of n is equal to the neighbourhood of the image of n, $f(nhd(n)) = nhd(f(n))$.

This definition can be easily extended to the case in which we have a string graph and a !-graph:

Definition (Local isomorphism for bang graphs) Given a !-graph G, a string graph H, a map $f^* : G^* \to H$ is a *local isomorphism* if there exists an instantiation $G \preceq_! G^*$ such that f^* is a string graph homomorphism and for every node-vertex $n \in NV(G^*)$ then the restriction of f to $nhd(n)$ restricts to an bijection of the edges $f\mid_{nhd(n)}$: $nhd(n) \tilde{\to} nhd(f(n))$.

Similarly to the previous case, given a !-graph G and a string graph H, a map $f^* : G^* \to H$ is a *local isomorphism* if we can find an instantiation G^* of G such that f^* is a string graph homomorphism and for every node-vertex $n \in NV(G^*)$ the image of the neighbourhood of n is equal to the neighbourhood of the image of n, $f(nhd(n)) = nhd(f(n))$.

Now we can define a string graph matching as:

Definition (String graph matching) Given a string graph rewrite rule $L \to R$ and a string graph G, we define *matching* a monic local isomorphism $m : L \to G$.

After finding finding a matching m between the string graphs L and G, we can replace the occurrence of L within G with R. The replacement operation is made up of the following steps:

- Disconnection and removal of L from G;
- Insertion and re-connection of R to G.

During these operations it is important to keep constantly track of how L (or R) are connected to G.

Let's define *interface* (or boundary) B of L the set of node-vertices belonging to L which are directly connected to G; let's also define *interior* of L as $L - B$, that is the set of node-vertices belonging to L which are not directly connected to G. Notice that by the definition of string graph rewrite rule we have given above, the boundary B of L and R are the same. The replacement operation can then be performed more precisely in the following way:

- Identify the boundary B of L;

- Disconnect and remove the interior of L, that is $L - B$;
- Identify the interior of R, that is $R - B$;
- Insert the interior of R and connect it back to the boundary B.

This procedure can be formalized using the notion of pushouts and it takes the name of *double pushout technique* (*DPO*) [15, 9]. Let's define the graph G' as the graph G with the interior of L removed, that is the graph whose node-vertices and edges are:

$$\begin{aligned} V_{G'} &= V_G - (V_L - V_B) \\ E_{G'} &= E_G - (E_L - E_B) \end{aligned}$$

and where the source s and the target t of the edges are defined as the restrictions $s_{G'} = s_G|_{G'}$ and $t_{G'} = t_G|_{G'}$. Notice that these restrictions are well-defined as the matching m of L on G is a local isomorphism; this guarantees that in G' there are no edges without a target or a source. We can now interpret G' as the pushout complement of $B \overset{b_1}{\hookrightarrow} L \overset{m}{\to} G$ and G as the pushout of $B \overset{m'}{\to} G'$ and $B \overset{b_1}{\hookrightarrow} L$ (see figure 2.18).

$$\begin{array}{ccc} B & \overset{b_1}{\hookrightarrow} & L \\ {\scriptstyle m'}\downarrow & & \downarrow{\scriptstyle m} \\ G' & \hookrightarrow & G \end{array}$$

Figure 2.18: Pushout complement.

Similarly, if we consider R and we define H as the final string graph obtained by replacing L with R in G, we can interpret G' as the pushout complement of $B \overset{b_2}{\hookrightarrow} R \overset{m}{\to} H$ and H as the pushout of $B \overset{m'}{\to} G'$ and $B \overset{b_2}{\hookrightarrow} R$ (see figure 2.19).

$$\begin{array}{ccc} B & \overset{b_2}{\hookrightarrow} & R \\ {\scriptstyle m'}\downarrow & & \downarrow \\ G' & \hookrightarrow & H \end{array}$$

Figure 2.19: Pushout complement.

The technique of graph rewriting based on the computation of these two pushouts is called double pushout and it is usually expressed in a single diagram (see figure 2.20).

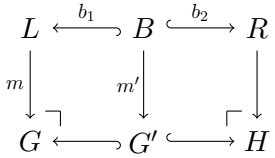


Figure 2.20: Double Pushout.

This means that G can be interpreted as the gluing of L and G' along the boundary B and that the final string graph H can be computed by the gluing of R (instead of L) and G' along the boundary B. The picture in figure 2.21 exemplifies how the double pushout technique is used in a concrete case.

Even if in a category with all pushouts, pushout complements does not need to exist or to be unique, it is possible to prove the uniqueness of rewriting. The following theorem, proved in [9], guarantees for the uniqueness of rewriting:

Theorem (Existence and uniqueness of pushout complement) Given a boundary span $L \overset{b_1}{\leftarrow} B \overset{b_2}{\rightarrow} R$ and a matching $m : L \rightarrow G$, then $B \overset{b_1}{\rightarrow} L \overset{m}{\rightarrow} G$ has a unique pushout complement G' and both the pushout squares in the DPO diagram (see figure 2.20) are preserved by the embedding of $\mathbf{SGraph_T}$ into $\mathbf{Graph}/G_T$.

2.6 Regular Expressions

Finally, we briefly recall the formalism of regular expressions, which will be useful when dealing with the representation of discrimination nets for !-graphs (see chapter 3).

A *regular expression* is a rule used to generate strings. More formally, given a finite alphabet $\Sigma = \{a_1, a_2, a_3 \dots a_n\}$, a regular expression r is a rule to produce strings s from the alphabet Σ.

Given the alphabet $\Sigma = \{a_1, a_2, a_3 \dots a_n\}$ a string s over the alphabet Σ can be:

- $s = \epsilon$: the empty string;
- $s = a_i$: a literal element belonging to alphabet Σ;
- $s = s_i.s_j$ (or simply $s = s_i s_j$): the concatenation of two strings s_i and s_j over the alphabet Σ;

A regular expression r over a finite alphabet $\Sigma = \{a_1, a_2, a_3 \dots a_n\}$ can be the rule:

- $r = s_i$ generating s_i (constant);
- $r = r_i.r_j$ (or simply $r = r_i r_j$) generating the concatenation of the string produced by r_i and the string produced by r_j (concatenation).
- $r = (r_i)|(r_j)$ generating r_i or r_j (alternative);
- $r = (r_i)^*$ generating ϵ or r_i or $r_i r_i$ or $r_i r_i r_i ...$ (repetition zero or more times);
- $r = (r_i)^+$ generating r_i or $r_i r_i$ or $r_i r_i r_i ...$ (repetition one or more times);
- $r = (r_i)^?$ generating ϵ or r_i (occurrence zero or one time);

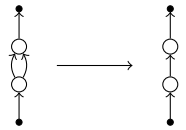
Notice that the repetition one or more time $(r_i)^+$ and the occurrence zero or more time $(r_i)^?$ are derived expressions as they could be written using other expressions:

- $(r_i)^+ = r_i(r_i)^*$
- $(r_i)^? = (\epsilon|r_i)$

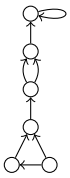
Notice also that when we define r we use parentheses to group those literals comprising r_i which are the object of the operation of alternative, repetition or occurrence; if r_i is a single literal, parentheses can be omitted.

Given a regular expression r and a string s, we say that r *matches* s if s can be generated from the regular expression r.

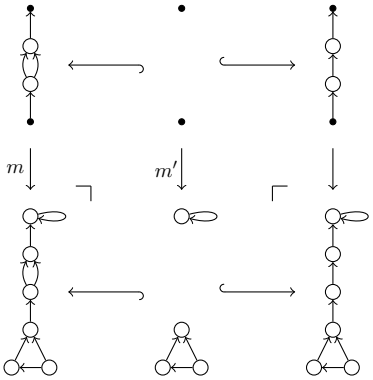
Rewrite rule $L \to R$:



String Graph G :



Double pushout:



Final String Graph H :

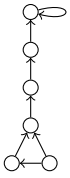


Figure 2.21: Example of use of the double pushout.

Chapter 3

Algorithms for Rewriting

In this chapter we focus on the main problem we tackle in this dissertation; we will first introduce a family of algorithms devised to improve the performance of graph matching (section 3.1); then we will described a standard implementation of these algorithms for string graphs (section 3.2); next we will explain how we improved this algorithm in the case of string graphs (section 3.3) and finally how we extended it to !-graphs (section 3.4).

3.1 Discrimination Nets

In this section we introduce the basic concepts common to all the algorithms belonging to the family of discrimination nets.

3.1.1 Definition of Discrimination Nets

Discrimination nets are a family of algorithms introduced in the 1990s to speed up the search for matchings in term rewriting systems [22]. A discrimination net is a *filtering algorithm*, not a matching algorithm; it can implemented before a matching algorithm, but it does not replace it. Given a rewrite system $\mathcal{R}$ made up of a set of rewrite rules r_i and a generic formula f, a discrimination net is not designed to evaluate exact matchings between f and r_i, but to exclude in a fast and efficient way all those rewrite rules r_i that certainly do not match f; a discrimination net returns a smaller set of rewrite rules $\mathcal{R}' \subseteq \mathcal{R}$, containing only those rewrite rules r_i which are likely to match f. Therefore, a discrimination net is implemented in order to improve the overall performance of a matching algorithm by reducing the search space of the matching algorithm.

Discrimination nets are based on the idea of building a data structure, called *index*, which can be used to efficiently query many rewrite rules at the same time. Given a rewrite

system $\mathcal{R}$ made up of a set of rewrite rules r_i, a discrimination net generates a data structure containing information about all the rewrite rules r_i. Given a new formula f, the discrimination net can efficiently compare the formula f against all the rewrite rules stored in its data structure and return only the formulas r_i which are likely to match f.

When using discrimination nets with string graphs, it is convenient to consider the input and output wire-vertices of a string graph connected to a special type of node-vertex called *boundary node-vertex*; a boundary node-vertex n is a node-vertex with $kind_{NV}(n) =$ "*boundary*" and with an undefined input arity and an undefined output arity. Conceptually, a boundary node-vertex n is just a placeholder which can be substituted with any other node-vertex or with an arbitrary string graph; through a boundary node-vertex a string graph G can be connected to another string graph H. For example, the string graph in figure 2.7 will be represented as the string graph in figure 3.1.

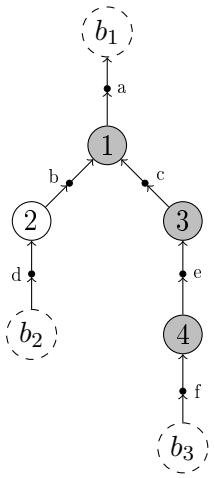


Figure 3.1: Example of a string graph with boundary node-vertices.

We will use the term *concrete node-vertex* to refer to a node-vertex having a kind different from "*boundary*".

3.1.2 Correctness of Discrimination Nets

In returning the reduced set of rewrite rules $\mathcal{R}' \subseteq \mathcal{R}$, the discrimination net is allowed to commit false positives (keeping in $\mathcal{R}'$ rules which do not match f), but it is not allowed to commit any false negative (not keeping in $\mathcal{R}'$ rules which match f). The discriminatory power of the discrimination net can be computed as inversely proportional to the number of false positives.

In order to enforce these constraints, we say that a discrimination net is *correct* if:

(i) $\mathcal{R}' \subseteq \mathcal{R}$: the final reduced rewrite system $\mathcal{R}'$ produced by the discrimination net is a subset of the original rewrite system $\mathcal{R}$.

(ii) $\forall r_i \in \mathcal{R},\ r_i\ matches\ f \Rightarrow r_i \in \mathcal{R}'$: if a rewrite rule r_i matches the formula f, then r_i must belong to the reduced rewrite system $\mathcal{R}'$ produced by the discrimination net.

In the specific case of graph matching we will make the following assumptions:

1. The generic object f is a concrete string graph; we will call this graph *target graph* T. We assume that target graphs are always concrete string graphs, containing any kind of node-vertex, excluding boundary node-vertices; this choice is justified by the idea that our aim is to find matchings between specific instances of a graph (target graph) and a set of generic rules for generating graphs, which can be non-concrete string graphs or !-graphs. As boundary node-vertices and !-vertices are used to convert concrete graphs in abstract graphs which can be instantiated in several ways, we forbid the use of boundary node-vertices and !-vertices in target graphs.

2. The rewrite system $\mathcal{R}$ is a collection of graph rewrite rules r_i of the type $l \rightarrow r$; as we are interested only in the matching between the LHS l of a rewrite rule r_i and a generic graph f, we will work only with the set of LHS graphs l; we will call the set of all the LHS graphs l *set of pattern graphs* $\mathcal{P}$ and we will call an element of this set *pattern graph* P_i.

Two families of discrimination nets which have been studied and applied to the problem of graph matching are [10]:

1. Hashing-based discrimination nets: relying on a index containing hashed values describing the pattern graphs (e.g. histogram-based discrimination nets);

2. Topography-based discrimination nets: relying on a index built as the tree containing information about the pattern graphs.

Given the better performance on a theoretical and a practical level [10], we will focus on topography-based discrimination nets.

3.2 Standard Topography-based Discrimination Net for String Graphs

In this section we describe a topography-based discrimination net algorithm proposed by Douglas in [10] to speed up graph matching.

3.2.1 Algorithm

In a topography-based discrimination net the index data structure is built as a tree, generated by the merging of the contour lists of all the pattern graphs.

Definition (Contour) Given a pattern graph P_i from the pattern set $\mathcal{P}$ we define m^{th} contour of P_i, the set p_m of all the node-vertices n belonging to P_i such that:

(i) n belongs to the set of node-vertices connected to the neighbourhood of the $(m-1)^{th}$ contour, p_{m-1};

(ii) for all $j < m$, n does not belong to the j^{th} contour, p_j.

Definition (Contour List) Given a pattern graph P_i, the *contour list* (or *topography*) of P_i is the list of all the contours starting from an initial node-vertex n_0 assumed as the 0^{th} contour.

So, in order to build the contour list of a pattern graph P_i we need a starting point; to choose the initial node-vertex we define a function, called *targeting function*, which receives as input a pattern graph P_i and returns a concrete node-vertex $n_0 \in P_i$.

By definition, the node-vertex n_0 returned by the targeting function will be considered the 0^{th} contour of the pattern graph P_i. Starting from this contour, all the following contours can be computed one after the other (see figure 3.2).

The algorithm of the targeting function may vary from implementation to implementation: it may return a random node-vertex n_0 or it can perform a search in order to return a specific node-vertex; as it will become clear in the next sections, a sensible choice could be to return a node-vertex n_0 which maximizes the number of contours between the 0^{th} contour and the first contour containing a boundary node-vertex.

In order to be efficient, every contour should store only the information useful to the discrimination task. For each node-vertex, we need to know its kind and its input and output arity.

When dealing with string graphs, we consider each contour as a multiset of node-vertices. We can represent the node-vertices graphically or, for the sake of writability, as tuples of the form (k_i, in_i, out_i) where k_i is the kind, in_i is the input arity and out_i is the output arity of the i^{th} node-vertex (see figure 3.3). Notice that since boundary node-vertices have an undefined input arity and an undefined output arity, then we denote their arities as $in_i =$ "$-$" and $out_i =$ "$-$".

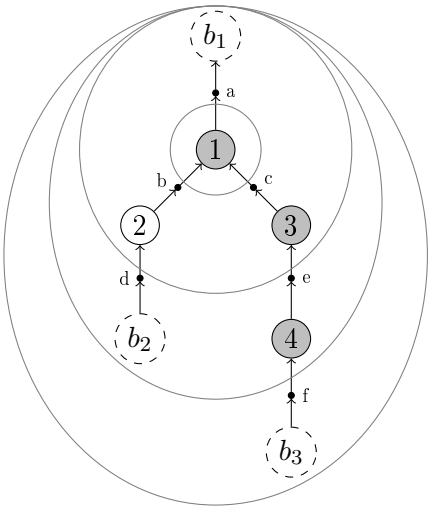
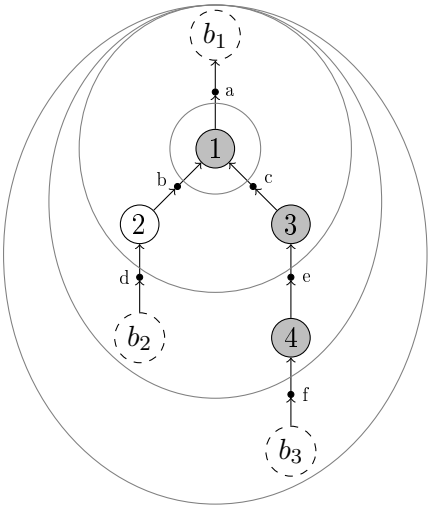


Figure 3.2: Example of a topography where the 0^{th} *contour* $= \{1\}$, 1^{st} *contour* $= \{2, 3, b_1\}$, 2^{nd} *contour* $= \{4, b_2\}$, 3^{rd} *contour* $= \{b_3\}$.



0^{th} contour:

[("grey",2,1)]

1^{st} contour:

[("white",1,1), ("gray",1,1), ("bound",-,-)]

2^{nd} contour:

[("gray",1,1), ("bound",-,-)]

3^{rd} contour:

[("bound",-,-)]

Figure 3.3: Example of a topography for a string graph.

Once the contour lists for all the pattern graphs P_i have been built, all the contour lists l_i can be merged into a *topography tree (or discrimination tree)* $\mathcal{T}$. The topography tree $\mathcal{T}$ is built starting from a tree made up by a single empty root node r. Given a contour list l_i, we insert it in $\mathcal{T}$ by inserting a contour at each level. Starting from the root node r, we check if the 0^{th} contour of l_i is present among the children of r; if the contour is present we move to the corresponding tree node n_1; if the contour is not present we create a tree node n_1 containing the 0^{th} contour of l_i, we append it to r and we move to the tree node n_1. Then, one by one, for all the contours in l_i we follow the same procedure: if the contour is present among the children of the current tree node n_i we move on to the corresponding tree node n_{i+1}; if the contour is not present we create the tree node n_{i+1}, we append it and

we move to it. At the last stage, when all the contours of l_i have been added to the tree, we create and append a last tree node containing a reference to the original pattern graph P_i we have processed. In this way we can build the complete topography tree $\mathcal{T}$ containing the contour lists of all the pattern graphs (see figure 3.4).

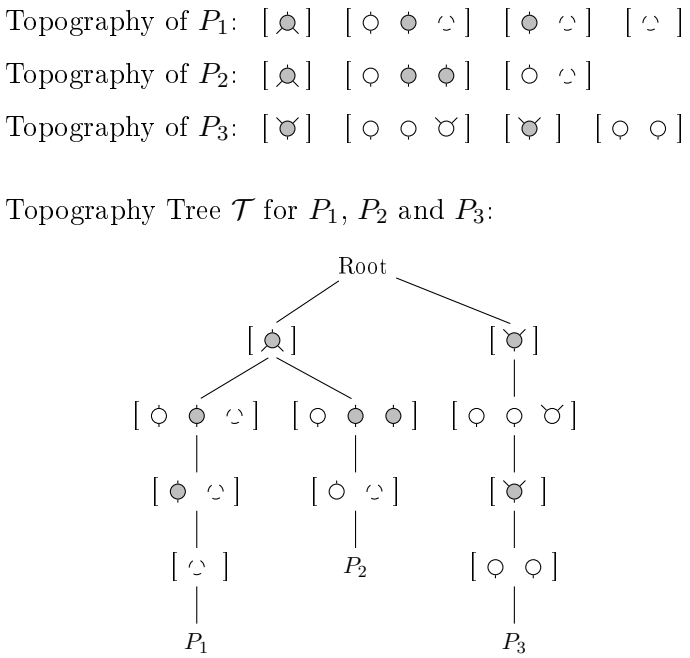


Figure 3.4: Example of a topography tree.

After building the topography tree $\mathcal{T}$, the discrimination net algorithm receives and processes a target graph T. For each node-vertex $n \in T$, the discrimination net algorithm builds a contour list c_n starting at n. Each contour list c_n is then checked against the topography tree $\mathcal{T}$. The contour list c_n is used to prune from $\mathcal{T}$ all those branches which are not *compatible* with the target graph T.

The operation of pruning is based on the notion of *compatibility*. When we work with string graphs, contours are represented as multisets and the definition of compatibility is:

Definition (Compatibility) Given two contours c_1 and c_2, we say that c_1 is compatible with c_2, that is $c_1 \rhd c_2$, if we can define an injective function $f : c_2 \to c_1$ preserving the kind and the arities of the node-vertices in c_2.

Practically, if we are working with string graphs and we are given a contour t belonging to a target graph T and a contour p belonging to a pattern graph P_i, we say that t is

compatible with p, $t \rhd p$, if we can find an injective function $f : p \to t$ such that for every node-vertex $p_j \in p$, $t_l \in t$, if $f(p_j) = t_j$ then:

(i) $kind(p_j) = kind(t_l)$ and $input(p_j) = input(t_l)$ and $output(p_j) = output(t_l)$

OR

(ii) $kind(p_j) =$ "$boundary$"

that is: if p_j is a node-vertex with kind different from "boundary", then p_j and t_l must have the same kind, the same input arity and the same output arity; if p_j is a node-vertex with kind "boundary", then t_l can have any kind, any input arity and any output arity. See figure 3.5.

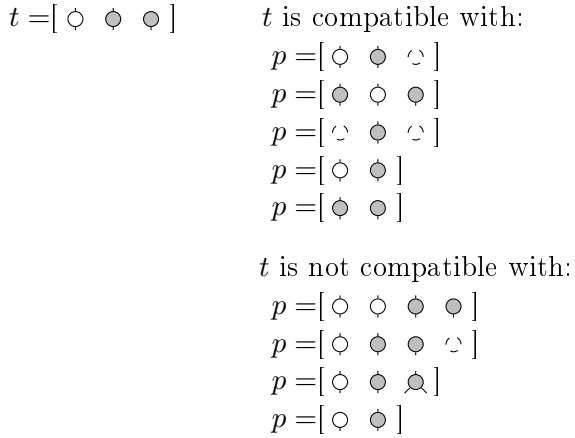


Figure 3.5: Examples of compatibility for string graphs.

Notice that, because of the injectivity of p, the relation of compatibility is not symmetric; $c_1 \rhd c_2$ ("c_1 is compatible with c_2") does not imply $c_2 \rhd c_1$ ("c_2 is compatible with c_1"). The relation of compatibility is indeed antisymmetric as $c_1 \rhd c_2$ and $c_2 \rhd c_1$ implies $c_1 = c_2$. Because of the injectivity of p, it is also easy to see that the relation of compatibility is reflexive ($c_1 \rhd c_1$) and transitve ($c_1 \rhd c_2$ and $c_2 \rhd c_3$ implies $c_1 \rhd c_3$). The relation of compatibility defines then a partial order.

In general, we will say that, given two string graphs T and P_i, T is compatible with P_i, $T \rhd P_i$, if, for all the contours j in P_i, then $t_j \rhd p_j$.

Using the notion of compatibility, for each contour list c_n, the discrimination net algorithm prunes $\mathcal{T}$ and returns a set $\mathcal{R}'_n \subseteq \mathcal{P}$ of pattern graphs P_i which could match T. This set $\mathcal{R}'_n$ is then sent along with T to the matching algorithm in order to check for exact

matchings between T and the pattern graphs $P_i \subseteq \mathcal{R}'_n$. See figure 3.6.

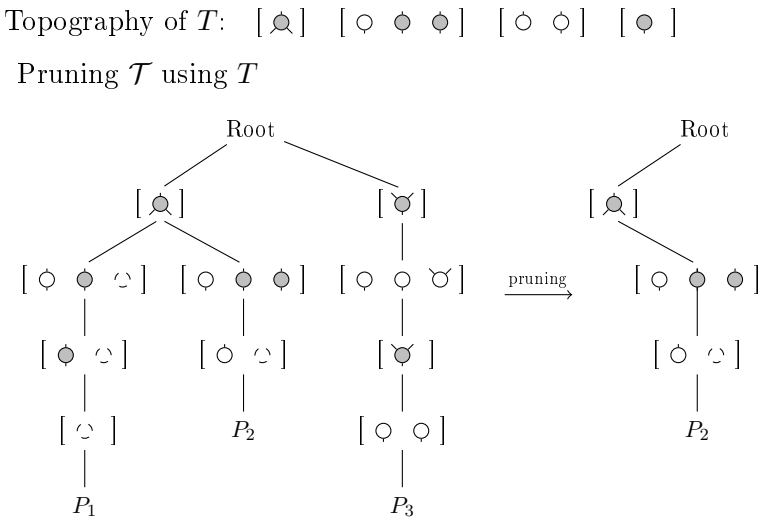


Figure 3.6: Example of pruning of a topography tree.

The overall discrimination net algorithm is summarized in the algorithm table 1.

3.2.2 Correctness

The proof of the correctness of the algorithm for the standard topography-based discrimination net is given by Douglas in [10].

3.2.3 Complexity

The asymptotic complexity of each step of the standard topography-based discrimination net is computed in the algorithm table 2.

If we analyze the algorithm we could divide it in two parts: the first part (steps 1 to 3) which is a setup part, executed only once in order to build the topography tree $\mathcal{T}$; and a second part (steps 4 and 5) which is executed every time we want to check a target graph T for matching.

In the worst case the asymptotic complexity of the first part is $O(1) + O(|\mathcal{P}|\,|P_i|^2 + |\mathcal{P}|\,|P_i|) + O(|\mathcal{P}|\,|P_i|) \sim O(|\mathcal{P}|\,|P_i|^2)$; so the complexity grows linearly with the number of pattern graphs in $\mathcal{P}$ and quadratically with the number of node-vertices in the pattern graphs P_i.

Algorithm 1 Standard Algorithm for the Discrimination Net

(i) Receive the set $\mathcal{P}$ of pattern graphs P_i

(ii) For each pattern graph P_i in $\mathcal{P}$:

- Send P_i to the targeting function and receive back a concrete node-vertex $n \in P_i$
- Generate the contour list c_i starting at n

(iii) For each contour lists c_i:

- Add c_i to the topography tree $\mathcal{T}$

(iv) Receive the target graph T

(v) For all node-vertices n in T:

- Generate the contour list c_n starting at n
- Prune $\mathcal{T}$ using c_n according to the notion of compatibility
- Obtain a set $\mathcal{R}'_n \in \mathcal{P}$ of pattern graphs P_i potentially matching T
- Send $\mathcal{R}'_n$ and T to a matching algorithm

For the second part, the asymptotic complexity in the worst case is $O(1) + O(|T|^2 + |T|^2\,|\mathcal{P}| + |T|) \sim O(|T|^2\,|\mathcal{P}|)$; in this case, the complexity grows linearly with the number of pattern graphs in $\mathcal{P}$ and quadratically with the number of node-vertices in the target graph T. The algorithm could be improved, without changing its overall worst-case complexity though, by merging together the first two sub-steps of step 5; instead of generating a contour list for T in advance, we can generate it just-in-time when we need to compare one contour with the topography tree; this would avoid the generation of unused contour in case the topography tree is completely pruned before processing all the contours of the target graph.

Notice that by implementing the discrimination net algorithm we reduce the complexity of the following matching algorithm from $O(|\mathcal{P}|\,|T|\,M)$, where M is the complexity of the matching operation, to $O(|\mathcal{R}|\,|T|\,M)$, where $|\mathcal{R}|$ is the number of pattern graphs P_i contained in the reduced set of pattern graphs [10]. A good discrimination net algorithm is one with a high discriminative power reducing the size of the set of pattern graphs $\mathcal{P}$ as much as possible.

Algorithm 2 Complexity of the Discrimination Net Algorithm

(i) $O(1)$

(ii) $O(|\mathcal{P}|)$ where $|\mathcal{P}|$ is the number of pattern graphs P_i in $\mathcal{P}$;

- $O(targeting\,function)$ depending on the implementation of the targeting function; it can range from $O(1)$ if the targeting function returns a random node-vertex in P_i to $O(|P_i|^2\,k)$, where $|P_i|$ is the number of node-vertices n in P_i, if it has to build the contour list of every node-vertex and evaluate it in constant time k;
- $O(|P_i|\,k)$ where k is the cost of generating a contour; the number of repetitions depends on the number of contours which is bounded by the number of node-vertices in P_i.

(iii) $O(|\mathcal{P}|)$;

- $O(|P_i|\,k)$ where k is the cost of appending a node and moving inside the topography tree; the number of repetitions depends on the number of contours which is bounded by the number of node-vertices in P_i.

(iv) $O(1)$

(v) $O(|T|)$ where $|T|$ is the number of node-vertices n in T:

- $O(|T|\,k)$ where k is the cost of generating a contour; the number of repetitions depends on the number of contours which is bounded by the number of node-vertices in T.
- $O(|T|\,|\mathcal{P}|\,k)$ where k is the cost of making a comparison and executing a pruning; the number of repetitions depends on the number of levels to check, bounded by the number of node-vertices in T, and on the number of nodes on each level of the tree $\mathcal{T}$, bounded by the number of pattern graphs in $\mathcal{P}$.
- $O(1)$
- $O(1)$

3.3 Improved Topography-based Discrimination Net for String Graphs

In this section we show how to redefine the standard topography-based discrimination net algorithm by modifying the notion of compatibility. In this way we will have a more strict and rigorous notion of compatibility which guarantees the same or a higher discriminative power. Moreover this new notion will make it easier to extend a topography-based discrimination net to !-graphs.

3.3.1 Algorithm

In the improved algorithm we substitute the original and basic idea of compatibility with two different notions of compatibility between contours: *strong compatibility* and *weak compatibility*.

Definition (Strong Compatibility) Given two contours c_1 and c_2, we say that c_1 is strongly compatible with c_2 if we can define an bijective function $f : c_2 \to c_1$ preserving the kind and the arities of the node-vertices in c_2.

Definition (Weak Compatibility) Given two contours c_1 and c_2, we say that c_1 is weakly compatible with c_2 if we can define an injective function $f : c_2 \to c_1$ preserving the kind and the arities of the node-vertices in c_2.

Practically, if we are given a contour t belonging to a target graph T and a contour p belonging to a pattern graph P_i, we look for a function $f : p \to t$, such that, given a node-vertex $p_j \in p$ and a node-vertex $t_l \in t$, if $f(p_j) = t_l$ then the kind-preservation and arity-preservation constraints given above are satisfied:

(i) $kind(p_j) = kind(t_l)$ and $input(p_j) = input(t_l)$ and $output(p_j) = output(t_l)$

OR

(ii) $kind(p_j) =$ "$boundary$"

In conclusion, t is strongly compatible with p if all the node-vertices in p are matched to t and vice versa; if all the node-vertices in p are matched to t but not vice versa then t is weakly compatible with p. See figure 3.7.

Notice that, because of bijectivity, the relation of strong compatibility with no boundaries is symmetric ($c_1 \rhd c_2$ implies $c_2 \rhd c_1$) while the relation of weak compatibility remains

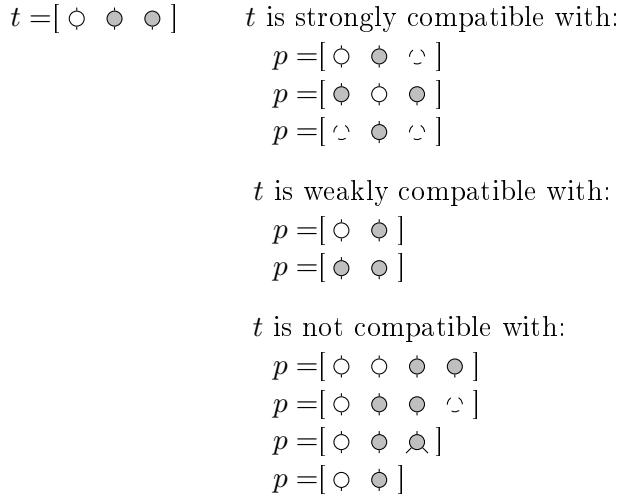


Figure 3.7: Examples of strong compatibility and weak compatibility for string graphs.

antisymmetric ($c_1 \rhd c_2$ and $c_2 \rhd c_1$ implies $c_1 = c_2$).

Notice also that the previous notion of compatibility and the new notion of weak compatibility coincide; this means that by using strong compatibility we enforce stronger conditions for compatibility, while the weak compatibility works as in the case of the standard algorithm.

We will keep saying that two contours c_1 and c_2 are compatible, $c_1 \rhd c_2$, if c_1 and c_2 are strongly compatible or if c_1 and c_2 are weakly compatible.

We can now modify the algorithm for the standard topography-based discrimination net adding the notions of strong compatibility and weak compatibility (see algorithm table 3).

After building the topography tree $\mathcal{T}$ as explained in the previous section, we start pruning using the notion of strong compatibility to compare contours; when, during the processing of the contours, we find a boundary node-vertex, then we switch from using the notion of strong compatibility to using the notion of weak compatibility.

The intuition behind this algorithm is the following: before finding a boundary node-vertex, we require a (kind-wise and arity-wise) bijective matching between the contour of the pattern graph P_i and the contour of the target graph T; after finding a boundary node, part of the target graph T can develop unconstrained and, therefore, we can require only a (kind-wise and arity-wise) injective matching between the contour of the pattern graph P_i and the contour of the target graph T. The correctness of this new version of the algorithm is formally proved in the next section.

Algorithm 3 Improved Algorithm for the Discrimination Net

(i) Generate topography tree $\mathcal{T}$ and receive the target graph T as before.

(ii) For all node-vertices n in T:

- Generate the contour list c_i starting at n
- Prune $\mathcal{T}$ using c_i according to the new notions of compatibility
  - If no boundary node-vertex has occurred in a previous contour, prune according to the notion of strong compatibility
  - If a boundary node-vertex has been found, prune according to the notion of weak compatibility
- Obtain a set $\mathcal{R}_n \in \mathcal{P}$ of pattern graphs P_i potentially matching T
- Send $\mathcal{R}_n$ and T to a matching algorithm

3.3.2 Correctness

In order to prove the correctness of the discrimination net algorithm using strong compatibility and weak compatibility, it will be useful to introduce two lemmas about compatibility between contours.

Lemma (Strong compatibility of contours without boundary node-vertices) Let A and B be two string graphs, such that A matches B, that is there is a matching $m : A \to B$.

Assume n_0 to be the starting node-vertex of the topography of A.

We know that there are no boundary node-vertices in B and we assume also that the first boundary node-vertex has been found at radius k from n_0 in A.

Then for every contour j within or on radius k, b_j is strongly compatible with a_j.

Proof. To prove this lemma we can show that for every contour j within or on radius k, $a_j \subseteq b_j$ and $b_j \subseteq a_j$.

First of all, $a_j \subseteq b_j$ is easily proved, because the monomorphism $m : A \to B$ guarantees that every element in A (and therefore every element in any contour of A) has an image in B.

As a second step, let's focus on proving $b_j \subseteq a_j$. We will prove this by induction:

(i) Base case: let's prove $b_1 \subseteq a_1$.

This is trivially true, since the first contour of every graph is just a single node-vertex; and, since by assumption we have a matching between A and B, then it must be

$b_1 = a_1 = \{n_0\}$, which implies $b_1 \subseteq a_1$.

(ii) Inductive step: let's prove $b_k \subseteq a_k$ assuming that $b_{k-1} \subseteq a_{k-1}$.
Consider a node-vertex $n_{b,k} \in b_k$. There must exist a node-vertex $n_{b,k-1} \in b_{k-1}$ connected to $n_{b,k}$; since $b_{k-1} \subseteq a_{k-1}$, there must also exist a node-vertex $n_{a,k-1} \in a_{k-1}$ such that $m(n_{a,k-1}) = n_{b,k-1}$. Now, since by assumption $n_{a,k-1}$ is not a boundary node-vertex, to respect the property of local isomorphism of the matching, the image of the neighbourhood of $n_{a,k-1}$ must be in bijection with the neighbourhood of the image $m(n_{a,k-1}) = n_{b,k-1}$. Therefore there must exist a node-vertex $n_{a,k} \in p_k$ such that $m(n_{a,k}) = n_{b,k}$. And so it holds that $b_k \subseteq a_k$.

So, having proved that for every contour j within or on radius k, $a_j \subseteq b_j$ and $b_j \subseteq a_j$, we can conclude that for every contour j within or on radius k, b_j is strongly compatible with a_j. □

Lemma (Weak compatibility of contours with boundary node-vertices) Let A and B be two string graphs, such that A matches B, that is there is a matching $m : A \to B$.

Assume n_0 to be the starting node-vertex of the topography of A.

We know that there are no boundary node-vertices in B and we assume also that the first boundary node-vertex has been found at radius k from n_0 in A.

Then for every contour j beyond radius k, b_j is weakly compatible with a_j.

Proof. To prove this lemma we have to show that for every contour j beyond radius k, $a_j \subseteq b_j$. As before, this is easily proved, because the monomorphism $m : A \to B$ guarantees that every element in A (and therefore every element in any contour of A) has an image in B.

Therefore, we can state that for every contour j beyond radius k, b_j is weakly compatible with a_j. □

We now present a proof of the theorem of the correctness of the improved algorithm for the topography-based discrimination net following a similar proof given in [10].

Theorem (Correctness of the algorithm for the improved topography-based discrimination net) The algorithm for the improved topography-based discrimination net for string graphs is correct, i.e. it satisfies the two conditions for the correctness of a discrimination net we stated in 3.1.2.

Proof. Let $\mathcal{P}$ be a set of pattern graphs $\{P_1, P_2 \dots P_n\}$ and let T be a target graph. Let $\mathcal{T}$ be the topography tree built from the set of pattern graphs $\mathcal{P}$.

Suppose that P_i matches T, that is there a matching $m : P_i \to T$.

Assume n_0 to be the starting node-vertex of the topography of P_i.

In order to prove that the discrimination procedure of the improved topography-based algorithm is correct we have to prove the two conditions for the correctness of a discrimination net we stated in 3.1.2.

First of all we have to prove $\mathcal{P}' \subseteq \mathcal{P}$, that is the set of pattern graphs $\mathcal{P}'$ returned by the topography-based algorithm is contained in the original set of pattern graphs $\mathcal{P}$.
This holds because of the way in which the algorithm is constructed: starting from the topography tree $\mathcal{T}$ containing all the pattern graphs in $\mathcal{P}$, the algorithm proceeds exploring and pruning the tree; the data structure returned by the algorithm is a pruned tree $\mathcal{T}'$ containing branches leading only to a finite subset of the original pattern graphs. Therefore $\mathcal{P}' \subseteq \mathcal{P}$.

Next, we have to prove $P_i \in \mathcal{P}'$, that is the pattern graph P_i matching the target graph T belongs to the set of pattern graphs $\mathcal{P}'$ returned by the improved topography-based algorithm.
In order for this statement to hold, the branch of the topography tree $\mathcal{T}$ leading to the pattern graph P_i must not be pruned; this branch will not be pruned if and only if $T \rhd P_i$.
As the algorithm iterates through all the possible lists of contours of T, assume that the list of contours of T we consider is the one starting at $m(n_0)$. For T to be compatible with P_i it must hold that for every contour j then $t_j \rhd p_j$; more precisely, in the improved topography-based algorithm, if p_k is the contour where we found the first occurrence of a boundary node-vertex, it must hold that for every $j \leq k$, t_j is strongly compatible with p_j and, for every $j > k$, t_j is weakly compatible with p_j.
Let's prove the two parts of this assertion:

(i) Assume that the first boundary node-vertex has been found at radius k from n_0.
Then, by the lemma on the strong compatibility of contours it follows that for every contour j within or on radius k, t_j is strongly compatible with p_j.

(ii) Again, assume that the first boundary node-vertex has been found at radius k from n_0.
Now, by the lemma on weak compatibility of contours, it follows that for every contour j beyond radius k, t_j is weakly compatible with p_j.

In this way we have proved that $P_i \subseteq \mathcal{P}'$.

Hence, in conclusion, the algorithm for the improved discrimination net has been proved to be correct. □

3.3.3 Complexity

The main difference between the standard algorithm and the improved algorithm is in the way in which compatibility is evaluated between contours; the standard algorithm uses weak compatibility during the whole execution, while the improved algorithm starts using strong compatibility and switches to weak compatibility only when it finds a boundary node-vertex.

As the complexity of evaluating strong and weak compatibility is asymptotically equivalent, the complexity of the two algorithms is the same (see algorithm table 2 and the discussion in section 3.2.3).

3.4 Extended Topography-based Discrimination Net for Bang Graphs

In this section we explain our idea to extend our improved algorithm for topography-based discrimination net from string graphs to !-graphs.

3.4.1 Algorithm

Analogously to the previous algorithms, also the algorithm for the extended topography-based discrimination net for !-graph is based on the construction of an index, a topography tree $\mathcal{T}$ containing the contour lists of all the pattern graphs P_i in the set of pattern graphs $\mathcal{P}$.

When dealing with !-graphs, we are working with a set of pattern graphs $\mathcal{P}$ containing string graphs and !-graphs and a target graph T. As specified above (see section 3.1.2) the target graph is required to be a concrete string graph, that is a string graph without boundary node-vertices.

Given a pattern graph $P_i \in \mathcal{P}$ we use the targeting function to select a node-vertex $n_0 \in P_i$ to be the 0^{th} contour. As P_i could be a !-graph we impose a new constraint on the targeting function: we require the targeting function to return a concrete node-vertex n_0, a node-vertex that is not a boundary node-vertex nor is contained in a !-box. The reason for this constraint is that we want the contour list for the pattern graph P_i to be built starting from a node-vertex which will be present and which will have a given kind and arity in all

the possible instances of the pattern graph P_i.

Now, in order to represent contours, using multisets is not convenient anymore. The notation using multisets does not indeed provide a natural way to represent node-vertices that can be instantiated an arbitrary number of times. To solve this problem, we use the formalism of regular expressions. Contours can now be represented using regular expressions r built from the alphabet Σ containing the node-vertices making up the pattern graphs P_i and the target graph T. As before, the elements of the alphabet Σ can be represented graphically or, for the sake of writability, they can be represented as tuple of the form $(k_i, in_i^{\square}, out_i^{\square})$ where k_i, in_i and out_i are the kind, the input arity and the output arity of the i^{th} node-vertex and $^{\square}$ is a placeholder that, as it will be explained soon, can be dropped or replaced with the * operator (see figure 3.9).

The language of regular expression provides us with the two operators that allow us to deal with !-boxes:

- the repetition operator (*) allows us to specify that a node-vertex is contained in a !-box; we will use the repetition operator:

  1. if a node-vertex n_i is inside a !-box b_i, then we will mark the node-vertex n_i as n_i^* (as the node-vertex n_i inside the !-box b_i can be instantiated an arbitrary number of times);
  2. if a node-vertex n_i has an input edge coming from a node-vertex n_j^* inside a !-box b_j, then we will mark the input arity of the node-vertex n_i as $in_{n_i}^*$, where in_{n_i} is the number of input edges from concrete node-vertices; the input arity $in_{n_i}^*$ represents now a minimum number of input edges for the node-vertex n_i: in_{n_i} is the fixed number of input edges coming from concrete node-vertices, while * denotes that more input edges can be added if b_j is instantiated multiple times;
  3. if a node-vertex n_i has an output edge going to a node-vertex n_j^* inside a !-box b_j, then we will mark the output arity of the node-vertex n_i as $out_{n_i}^*$, where out_{n_i} is the number of output edges to concrete node-vertices; the output arity $out_{n_i}^*$ represents now a minimum number of output edges for the node-vertex n_i: out_{n_i} is the fixed number of output edges going to concrete node-vertices, while * denotes that more output edges can be added if b_j is instantiated multiple times;

- the optional operator ($^?$) allows us to specify that a node-vertex could end up being disconnected by the graph in case a !-box is killed or in the case a boundary node-vertex is merged with another boundary node-vertex; we will use the optional operator:

1. if a node-vertex n_i, which is not in a !-box and belongs to the m^{th} contour, has connections only to node-vertices n_j^* inside !-boxes and belonging to the $(m-1)^{th}$ contour, then we will mark the node-vertex n_i as $n_i^?$ (as the node-vertex n_i may end up being disconnected from the graph in case the !-boxes in the $(m-1)^{th}$ contour are killed);
2. if a node-vertex n_i, which is not in a !-box and belongs to the m^{th} contour, has connections only to node-vertices n_j^* belonging to the $(m-1)^{th}$ contour or to node-vertices $n_j^?$ belonging to the $(m-1)^{th}$ contour, then we will mark the node-vertex n_i as $n_i^?$ (as the node-vertex n_i may end up being disconnected if n_j too is disconnected);
3. if a contour contains more than one boundary node-vertex n_i, then all the boundary node-vertices are marked as $n_i^?$ since multiple boundary node-vertices may be joined in a single concrete node-vertex (see figure 3.8).

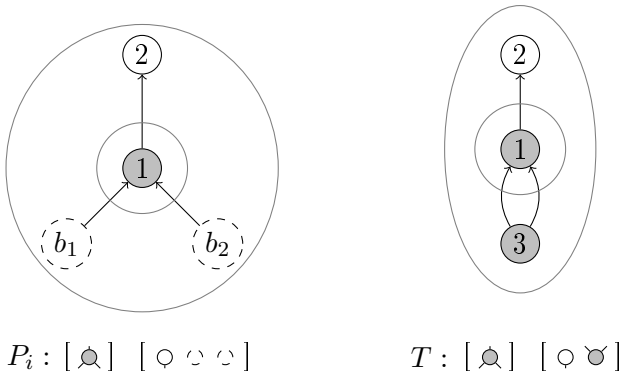


Figure 3.8: Instantiation of two boundary node-vertices in a single concrete node.

When we represent a contour using regular expressions, we want the order of the single elements to be irrelevant; therefore we define a particular type of regular expressions where the order of the elements does not matter:

Definition (Permutative regular expression) Given a regular expression r, we define *permutative regular expression* a regular expression r' given by the alternative of all the permutations of the elements in r.

Recall that when we build a contour for a !-graph we use regular expressions r built from the alphabet Σ containing the node-vertices making up the pattern graphs P_i and the target graph T. The elements comprising r are therefore tuples of the form $a_i = (k_i, in_i^{\square}, out_i^{\square})^{\square}$ where $^{\square}$ is again a placeholder that can be dropped or replaced with * or with $^?$; we will

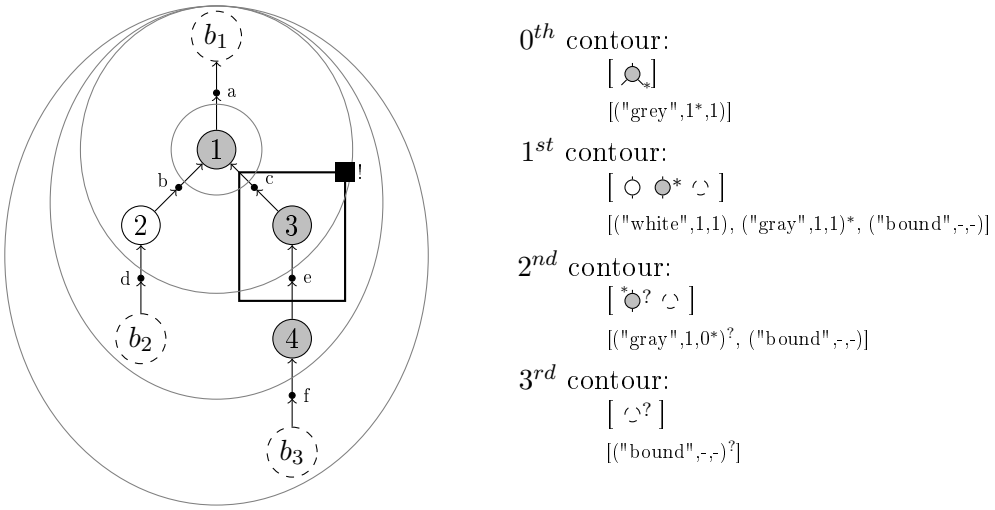


Figure 3.9: Example of a topography for a !-graph.

call these elements *atomic expressions*. The permutative regular expression r' derived from r is then given from the alternative of all the permutations of the atomic expressions; for example, if $r = a_1a_2a_3$ then $r' = a_1a_2a_3|a_1a_3a_2|a_2a_1a_3|a_2a_3a_1|a_3a_1a_2|a_3a_2a_1$. Given a string s and a permutative regular expression r' generated from r, we say that r' matches s if s can be generated from a regular expression obtained as a permutation of the atomic expressions comprising r.

Notice that, even if the expansion and the computation of all the permutations of a regular expression r is computationally very expensive, this will never be carried out in practice; in the implementation, we will just ignore the ordering of the atomic expressions.

Once the contour lists of all the pattern graphs have been generated, they can be merged in a tree in the same way we did for the standard topography-based discrimination net (see figure 3.10).

Now, in order for the algorithm to prune the topography tree $\mathcal{T}$ and produce a reduced set of pattern graphs $\mathcal{R}' \subseteq \mathcal{P}$, the notions of strong compatibility and weak compatibility must be redefined to take into account the fact that we are working with regular expressions.

Definition (Strong Compatibility) Given two contours c_1 and c_2, if c_1 is a string and c_2' is a permutative regular expression, then c_1 is *strongly compatible* with c_2 if c_2' matches c_1.

In other words, c_1 is strongly compatible with c_2 if the string c_1 can be generated from one of the permutations of the atomic expressions of c_2.

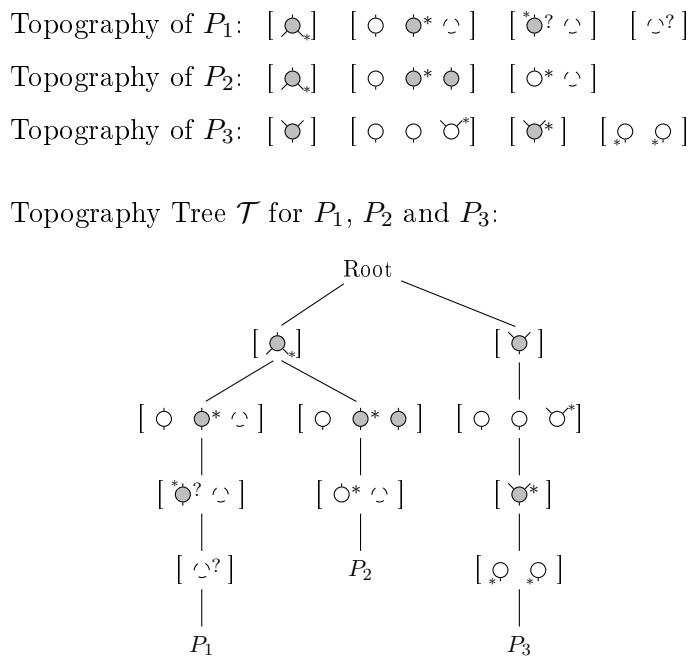


Figure 3.10: Example of a topography tree with !-graphs.

Definition (Weak Compatibility) Given two contours c_1 and c_2, if c_1 is a string and c_2' is a permutative regular expression, then c_1 is *weakly compatible* with c_2 if there exists a substring $\bar{c_1} \subseteq c_1$ such that c_2' matches $\bar{c_1}$.

In other words, c_1 is weakly compatible with c_2 if a substring of c_1 can be generated from one of the permutations of the atomic expressions of c_2.

So, c_1 and c_2 are strongly compatible if the entire string c_1 can be generated by c_2'; if only a substring of c_1 can be generated by c_2' then c_1 and c_2 are weakly compatible. See figure 3.11.

The definitions we have given of strong and weak compatibility fit the task of verifying the compatibility between the contour t of a target graph T and the contour p of a pattern graph P_i. According to the requirements given above, a target graph T never contains boundary node-vertices or !-boxes; all the atomic expressions comprising the contour t have the form $a_i = (k_i, in_i, out_i)$; t can then be treated as a string. Vice versa, as the pattern graph P_i may contain boundary node-vertices and !-boxes, then the atomic expressions comprising p have the form $a_i = (k_i, in_i^{\square}, out_i^{\square})^{\square}$; therefore p must be treated as a permutative regular expression.

Given an atomic expression of the form $t_l = (k_l, in_l, out_l)$ from the contour t of the target graph T, we say that an atomic expression p_j from the contour p of a pattern graph P_i can

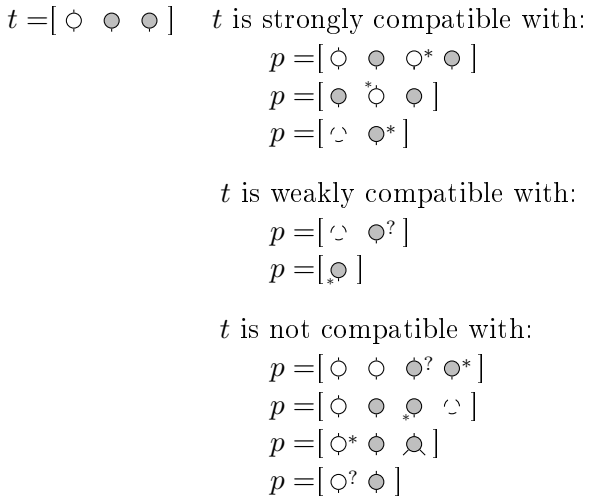


Figure 3.11: Examples of strong compatibility and weak compatibility for !-graphs.

generate a_i if one of the following condition is satisfied:

- $p_j = (\text{“}boundary\text{”}, -, -)^{\square}$
- $p_j = (k_j, in_j, out_j)^{\square}$, $k_j \neq \text{“}boundary\text{”}$, and $k_l = k_j$, $in_l = in_j$, $out_l = out_j$
- $p_j = (k_j, in_j^*, out_j)^{\square}$, $k_j \neq \text{“}boundary\text{”}$, and $k_l = k_j$, $in_l \geq in_j$, $out_l = out_j$
- $p_j = (k_j, in_j, out_j^*)^{\square}$, $k_j \neq \text{“}boundary\text{”}$, and $k_l = k_j$, $in_l = in_j$, $out_l \geq out_j$
- $p_j = (k_j, in_j^*, out_j^*)^{\square}$, $k_j \neq \text{“}boundary\text{”}$, and $k_l = k_j$, $in_l \geq in_j$, $out_l \geq out_j$

So, if p_j represents a boundary node-vertex, it can generate any t_l. If p_j represents a concrete node-vertex, then the kinds of t_l and p_j must be the same; moreover it must be possible to make the input and the output arities of p_j equal to the input and the output arities of t_l. This means that if the arities of p_j do not have any * operator, arities in p_j and in t_l must be exactly the same; if an arity in p_j has a * operator then the arity in p_j must be less or equal to the corresponding arity in t_l (this is due to the fact that if an arity in p_j is marked by * operator, then the arity can increase as a node-vertex inside a !-box connected to the current atomic expression is instantiated more times).

Notice that, while verifying the compatibility between the contour t of a target graph T and the contour p of a pattern graph P_i, the concept of weak compatibility could be reduced to the concept of strong compatibility if we artificially insert a boundary node-vertex of the form $(\text{“}boundary\text{”}, -, -)^*$ in the contour p.

In general, we will say that two contours c_1 and c_2 are compatible, $c_1 \rhd c_2$, if c_1 and c_2 are strongly compatible or if c_1 and c_2 are weakly compatible.

Compatibility could equivalently be stated using the notion of multisets by considering each atomic expression of the regular expression as an element of a multiset. In this case, if we are given two contours c_1 and c_2, we define $c_i^* \subseteq c_i$ and $c_i^? \subseteq c_i$ as the multisets containing all the elements of c_i having a * operator or a $^?$ operator. We say that c_1 is strongly compatible with c_2 if we can define a function $f : c_1 \to c_2$ which:

- preserves the kind and the arities of the node-vertices in c_1;
- is injective on the elements of c_1 whose image belongs to $c_2 - c_2^*$;
- is surjective on $c_2 - (c_2^* \cup c_2^?)$.

This definition requires that all the elements of c_1 mapped to concrete or optional elements of c_2 must be mapped in a bijective fashion; moreover, the definition requires that all the elements of c_2 which do not have a * operator or a $^?$ operator must be mapped in a surjective fashion by f. The first condition guarantees that all the elements of c_1 are correctly mapped to elements of c_2; the second condition guarantees that all the concrete elements of c_2 have a correct mapping from elements of c_1.

We say that c_1 is weakly compatible with c_2 if we can define a function $f : c_2 - (c_2^* \cup c_2^?) \to c_1$ which:

- preserves the kind and the arities of the node-vertices in $c_2 - (c_2^* \cup c_2^?)$;
- is injective in $c_2 - (c_2^* \cup c_2^?)$.

This definition simply requires that all the concrete elements in c_2 are injectively mapped to elements in c_1; the mapping of elements of c_2 which have a $^?$ operator or which have a * operator is considered irrelevant in evaluating weak compatibility; all the elements of c_2 which have a $^?$ operator or which have a * operator can indeed be dropped.

Notice that the definition of f could be extended to take into consideration the mapping of elements of c_2 which have a $^?$ operator (by representing the operation of dropping them as a mapping to an undefined element $\perp$), but it cannot be extended to take into consideration the mapping of elements of c_2 which have a * operator as these elements could be mapped to many elements in c_1.

The definition of compatibility using multisets is equivalent to the definition of compatibility using regular expression and all the results stated using the formalism of regular expressions could be restated using the formalism of multisets.

3.4.2 Correctness

In order to prove the correctness of the algorithm for the extended topography-based discrimination net for !-graphs, we will start by proving a couple of lemmas on the compatibility between contours.

Lemma (Strong compatibility of contours without boundary node-vertices) Let A be a !-graph and B a string graph, such that A matches B, that is there is an instantiation $A \preceq_! A^*$ such that there is a matching $m : A^* \to B$.

Assume n_0 to be the starting node-vertex of the topography of A^*.

We know that there are no boundary node-vertices in B and we assume also that the first boundary node-vertex has been found at radius k from n_0 in A.

Then for every contour j within or on radius k, b_j is strongly compatible with a_j.

Proof. To prove this lemma we have to show that for every contour j within or on radius k, b_j is generated by a_j. First, we will prove that every atomic expression in a_j has a corresponding generated atomic expression in b_j; then, we will prove that every atomic expression in b_j has a corresponding generating atomic expression in a_j.

First, let's focus on proving that every atomic expression $(k, in, out)_a$ in a_j has a corresponding generated atomic expressions $(k, in, out)_b$ in b_j.
Since A matches B, there is an instantiation $A \preceq_! A^*$ such that we have a monic local isomorphism $m : A^* \to B$; as every contour in A^* is an instantiation of a contour in A, then:

(i) every concrete node-vertex in A must be instantiated to an equivalent concrete node-vertex in A^*; this node-vertex is represented by the atomic expression $(k_l, in_l, out_l)_A$ in a_j and by $(k_l, in_l, out_l)_{A^*}$ in a_j^*; $(k_l, in_l, out_l)_{A^*}$ is mapped through the matching $m : A^* \to B$ to the corresponding generated equivalent atomic expression $(k_l, in_l, out_l)_B$ in b_j;

(ii) every concrete node-vertex in A which has $^?$ operator is instantiated to an equivalent concrete node-vertex in A^* which is connected to the graph, to a concrete node-vertex in A^* which is disconnected from the graph (because one or more !-boxes were killed) or is not instantiated in A^*. The node-vertex is represented by the atomic expression $(k_l, in_l, out_l)_A{}^?$ in a_j and by $(k_l, in_l, out_l)_{A^*}$ or by $\emptyset$ in a_j^*. If $(k_l, in_l, out_l)_A{}^?$ is instantiated to $(k_l, in_l, out_l)_{A^*}$ and it is connected to the graph, then $(k_l, in_l, out_l)_{A^*}$ in a_j is mapped through the matching $m : A^* \to B$ to the corresponding generated

equivalent atomic expression $(k_l, in_l, out_l)_B$ in b_j; if $(k_l, in_l, out_l)_A{}^?$ is instantiated to $(k_l, in_l, out_l)_{A^*}$ but it is disconnected from the graph, $(k_l, in_l, out_l)_{A^*}$ does not belong to any contour of A^* and therefore there is no atomic expression in any contour a_j to map; if $(k_l, in_l, out_l)_A{}^?$ is instantiated to $\emptyset$, then there is no atomic expression to map.

(iii) every node-vertex in a !-box in A is instantiated to an arbitrary number of equivalent concrete node-vertices in A^*; this node-vertex is represented by the atomic expression $(k_l, in_l, out_l)_A{}^*$ in a_j and by zero or more $(k_l, in_l, out_l)_{A^*}$ in a_j^*; now, for every $(k_l, in_l, out_l)_{A^*}$ in a_j, $(k_l, in_l, out_l)_{A^*}$ is mapped through the matching $m : A^* \to B$ to a corresponding generated equivalent atomic expression $(k_l, in_l, out_l)_B$ in b_j.

Let's now focus on proving that every atomic expression $(k, in, out)_b$ in b_j has a corresponding generating atomic expression $(k, in, out)_a$ in a_j. We will prove this by induction:

(i) Base case: let's prove b_1 has a generating atomic expression in a_1.
This is trivially true, since the first contour of every graph is always a single concrete node-vertex; let n_0 be the concrete node-vertex returned by the targeting function applied to A or to A^*; since by assumption we have a matching between A and B, then it must be $b_1 = a_1 = \{n_0\}$; we then have the same atomic expression $(k_{n_0}, in_{n_0}, out_{n_0})$ in a_1 (generating) and in b_1 (generated).

(ii) Inductive step: let's prove that for every atomic expression in b_k there is a generating atomic expression in a_k, assuming that b_{k-1} is strongly compatible with a_{k-1}.
We want to prove that all the atomic expressions in b_k are generated by atomic expressions in a_k or, equivalently, by the instantiation a_k^*. Suppose there is an atomic expression $(k_l, in_l, out_l)_B$ in b_k corresponding to a node-vertex $n_{b,k} \in b_k$; we want to prove that there must be an atomic expression $(k_l, in_l, out_l)_{A^*}$ in a_k^* generating it and corresponding to to the node-vertex $n_{a^*,k} \in a_k^*$ such that $m(n_{a^*,k}) = n_{b,k}$. Of course there must be an atomic expression $(k_l, in_l, out_l)_B$ in b_{k-1} corresponding to the node-vertex $n_{b,k-1} \in b_{k-1}$ connected to $n_{b,k}$; since b_{k-1} is strongly compatible with a_{k-1}, there must also exist an atomic expressions $(k_l, in_l, out_l)_{A^*}$ in a_{k-1}^* generating $(k_l, in_l, out_l)_B$ in b_{k-1} and corresponding to $n_{a^*,k-1} \in a_{k-1}^*$ such that $m(n_{a^*,k-1}) = n_{b,k-1}$. Now, by assumption $n_{a^*,k-1}$ is not a boundary node-vertex; moreover, according to the property of local isomorphism of the matching there must be a bijection between image of the neighbourhood of $n_{a^*,k-1}$ and the neighbourhood of the image of $n_{a^*,k-1}$. This means that there must be a node-vertex $n_{a,k} \in a_k$ such that its instantiation $n_{a^*,k} \in a_k$ guarantees $m(n_{a^*,k}) = n_{b,k}$. Consequently, there must be an atomic expression $(k_l, in_l, out_l)_A$ in a_k and an instantiation $(k_l, in_l, out_l)_{A^*}$ in a_k^* that generates $(k_l, in_l, out_l)_B$.

So, having proved that for every contour j within or on radius k, every atomic expression in a_j has a corresponding generated atomic expression in b_j and every atomic expression in b_j has a corresponding generating atomic expression in a_j, we can conclude that for every contour j within or on radius k, b_j is strongly compatible with a_j. □

Lemma (Weak compatibility of contours with boundary node-vertices) Let A be a !-graph and B a string graph, such that A matches B, that is there is an instantiation $A \preceq_! A^*$ such that there is a matching $m : A^* \to B$.

Assume n_0 to be the starting node-vertex of the topography of A^*.

We know that there are no boundary node-vertices in B and we assume also that the first boundary node-vertex has been found at radius k from n_0 in A.

Then for every contour j beyond radius k, b_j is weakly compatible with a_j.

Proof. To prove this lemma we have to show that for every contour j beyond radius k, there is a substring $b'_j \subseteq b_j$ generated by a_j. To do this, notice that we can ignore the atomic expressions in a_j which have an operator $^?$ or * as they can always be instantiated to no node-vertex in a^*_j; we then focus on finding a substring $b'_j \subseteq b_j$ generated by the concrete node-vertices of a_j which can not be dropped. Now, it is easy to find a substring $b'_j \subseteq b_j$ such that b'_j is generated by a_j; indeed because of the matching $m : A^*_i \to B$, it is guaranteed that every element in A_i not belonging to a !-box (marked by * operator) or being droppable (marked by $^?$ operator) has an image in B, which means that for every atomic expression which has no operator $^?$ or * in a_j there is a generated atomic expression in b_j.

Therefore, we can state that every contour j beyond radius k, b_j is weakly compatible with a_j. □

We can now state our theorem on the correctness of the algorithm for the extended topography-based discrimination net for !-graphs.

Theorem (Correctness of the algorithm for the extended topography-based discrimination net) The algorithm for the the extended topography-based discrimination net for !-graphs is correct, i.e. it satisfies the two conditions for the correctness of a discrimination net we stated in 3.1.2.

Proof. Let $\mathcal{P}$ be a set of pattern graphs $\{P_1, P_2 \dots P_n\}$ and let T be a target graph. Let $\mathcal{T}$ be the topography tree built from the set of pattern graphs $\mathcal{P}$.

Suppose that the !-graph P_i matches the string graph T; this means that there exists an instantiation $P_i \preceq_! P_i^*$ such that there is a matching $m : P_i^* \to T$.

Assume n_0 to be the starting node-vertex of the topography of P_i.

In order to prove that the discrimination procedure of the extended topography-based algorithm is correct, we have to prove two conditions for the correctness of a discrimination net (see section 3.1.2).

First, we have to prove $\mathcal{P}' \subseteq \mathcal{P}$; this condition can be proved as in the proof for the correctness of the improved discrimination net (see section 3.3.2).

Next, we have to prove $P_i \in \mathcal{P}'$, that is the pattern graph P_i matching the target graph T belongs to the set of pattern graphs $\mathcal{P}'$ returned by the extended topography-based algorithm.
In order for this statement to hold, the branch of the topography tree $\mathcal{T}$ leading to the pattern graph P_i must not be pruned; this branch will not be pruned if $T \rhd P_i$.
As the algorithm iterates through all the possible lists of contours of T, assume that the list of contours of T we consider is the one starting at $m(n_0)$. For T to be compatible with P_i it must hold that for every contour j then $t_j \rhd p_j$; more precisely, in the extended topography-based algorithm, if p_k is the contour where we found the first occurrence of a boundary node-vertex, it must hold that for every $j \leq k$, t_j is strongly compatible with p_j and, for every $j > k$, t_j is weakly compatible with p_j.
Let's prove the two parts of this assertion:

(i) Assume that the first boundary node-vertex has been found at radius k from n_0.
Then, by the lemma on the strong compatibility of contours it follows that for every contour j within or on radius k, t_j is strongly compatible with p_j.

(ii) Again, assume that the first boundary node-vertex has been found at radius k from n_0.
Now, by the lemma on weak compatibility of contours, it follows that for every contour j beyond radius k, t_j is weakly compatible with p_j.

In this way we have proved that $P_i \subseteq \mathcal{P}'$.

Hence, in conclusion the extended algorithm has been proved to be correct. □

3.4.3 Complexity

The main difference between the extended algorithm and the improved algorithm is in the type of operation required to check compatibility. The improved algorithm requires to

compare two multisets and checks if the elements of one multiset (the contour of a pattern graph) are contained in the other multiset (the contour of the target graph); the extended algorithm requires to compare if a permutative regular expression (the contour of a pattern graph) can generate a string (the contour of the target graph). However, we have already shown that compatibility in terms of regular expressions can be restated as compatibility in terms of multisets. Practically, the problem of checking if a permutative regular expression can generate a given string (or a substring of it) can be reduced to the equivalent problem of the evaluation of the function defined in section 3.4.1 between multisets; the elements composing the permutative regular expression and the elements composing the string can be converted into elements of a multiset and then they can be checked for compatibility by evaluating the function mapping one multiset to the other.

Therefore, as the complexity of evaluating compatibility with multisets and permutative regular expressions is asymptotically equivalent, the complexity of the extended algorithm is the same as the complexity of the standard algorithm (see algorithm table 2 and the discussion in section 3.2.3).

Chapter 4

Implementation

In this chapter we discuss how the algorithms for the topography-based discrimination net discussed in the previous chapter have been implemented within the Quantomatic framework. We will first give a short presentation of the framework in which the algorithms were developed (section 4.1); we then give details of the implemented algorithm in a high-level functional-like pseudo-code (section 4.2); and finally we discuss the performance of the algorithms (section 4.3).

4.1 Quantomatic

Quantomatic [17] is a set of software tools developed to support reasoning and manipulation of string graphs and !-graphs. Quantomatic is a modular software composed by three main components [15].

The core of the system is a module called *QuantoCore*, a library written in Poly/ML, a dialect of Standard ML [24]. This module is responsible for representing, manipulating and rewriting graphs.

In order to interact with QuantoCore, *QuantoGUI* offers a high-level graphical user interface. Through QuantoGUI the user can draw, edit and process (manually or automatically) graphs.

Over QuantoCore, a third component, *QuantoCoSy*, has been built. QuantoCoSy is the module responsible for performing automatic reasoning using the conjecture synthesis technique [13]; QuantoCoSy synthesizes conjectures (graphical identities) and checks them in a concrete model [16].

4.2 Implementation of the Algorithms

The algorithms we described in the previous chapter have been implemented in Poly/ML inside the QuantoCore module of Quantomatic.

The implementation can be examined analyzing separately the two main functionalities offered by the code: generating a topography tree and pruning the topography tree.

Before considering how these functionalities are implemented, we recall the definition of few general-purpose standard functions:

$$\begin{aligned} :: &: \alpha \to \alpha\,\mathsf{list} \to \alpha\,\mathsf{list} \\ \mathsf{rev} &: \alpha\,\mathsf{list} \to \alpha\,\mathsf{list} \\ \mathsf{map} &: (\alpha \to \beta) \to \alpha\,\mathsf{list} \to \beta\,\mathsf{list} \end{aligned}$$

:: is the infix notation of the cons operator, which add an element of type α to the head of a list of elements of the same type. rev receives a list of elements of type α and returns the same list in reverted order. map receives a function f from type α to type β and a list of elements of type α; it then applies f to each element in the list and it returns a list of elements of type β.

The first functionality implemented is the generation of the topography tree. The process by which the topography tree is built can be seen as a top-down process that add one by one each pattern graph P_i in the set of pattern graphs $\mathcal{P}$ to the topography tree.

First of all, let's represent in ML the set $\mathcal{P}$ of pattern graphs as a list P of pattern graphs. For each pattern graph in P we need to compute its contour list:

$$\mathsf{cl} = \mathsf{map\ generate\_contour\_list\ P}$$

cl is then the list of all the contour lists built from P.

For each pattern graph $\mathsf{p_i}$, its contour list $\mathsf{cl_i}$ is built by calling a targeting function to select the node-vertex comprising the 0^{th} contour, $\mathsf{contour_0}$, and then recursively building all the following contours, $\mathsf{contour_n}$:

$$
\begin{aligned}
&\mathsf{contour_0} = \mathsf{targeting\_function\ p_i} \\
&\mathsf{contours} = \mathsf{recursively\_build\_contours\ p_i\ contour_0} \\
&\mathsf{contours} = \mathsf{rev\ contours} \\
&\mathsf{cl_i} = \mathsf{contour_0} :: \mathsf{contours}
\end{aligned}
$$

where recursively_build_contours, which receives as inputs a graph g and a contour c and which returns as output a contour list, is defined as:

$$
\begin{aligned}
\mathsf{fun}\quad & \mathsf{recursively\_build\_contours\ g\ c} = \\
& \mathsf{new} = \mathsf{compute\_contour\ g\ c} \\
& \textbf{if}\ (\mathsf{new\ is\ empty}) \\
& \textbf{then return}\ \mathsf{new} \\
& \textbf{else return}\ \mathsf{new} :: (\mathsf{recursively\_build\_contours\ g\ new})
\end{aligned}
$$

$\mathsf{cl_i}$ is then the contour list of the pattern graph $\mathsf{p_i}$ built from the concatenation of all the contours $\mathsf{contour_n}$.

For each contour list $\mathsf{cl_i}$, a contour, $\mathsf{contour_n}$, is generated by finding all the node-vertices which are adjacent to the last contour $\mathsf{contour_{n-1}}$ and by filtering the atomic expressions of the node-vertices which appear in previous contours:

$$
\mathsf{contour_n} = (\mathsf{get\_adjacent\_atomic\_expr\ p_i\ contour_{n-1}}) - \mathsf{contour_{i<n}}
$$

where the function get_adjacent_atomic_expr collects all the node-vertices adjacent to the contour $\mathsf{contour_{n-1}}$, and transforms each of them in an atomic expression.

Once all the contour lists $\mathsf{cl_i}$ have been built they can be added to the topography tree by calling for each contour list $\mathsf{cl_i}$ the following instruction:

$$
\mathsf{tree} = \mathsf{add\_cl\_to\_tree\ cl\ tree}
$$

add_cl_to_tree then recursively calls the function add_contour_to_tree to add each contour in $\mathsf{cl_i}$ to the tree:

```
fun  add_contour_to_tree tree cl c =
     tree = update_tree tree c
     new = next_contour cl c
     if (new is empty)
     then return tree
     else return add_contour_to_tree tree cl new
```

At each iteration add_contour_to_tree updates tree by checking if the contour c already exists in tree at the current level; if not, a new tree node is instantiated in the tree. At the end, the function return the new tree in which all the contours belonging to cl_i have been added.
Once this operation has been performed for all the contour lists cl_i, then the topography tree is ready.

The second functionality implemented is the pruning of the tree. Given a topography tree, tree, we can now use a string graph, target, to prune the tree and to find the graphs contained in the topography tree that are likely to match target.

The first step of the pruning is the generation of the list as containing all the atomic expressions composing target:

```
as = get_atomic_expressions target
```

For each atomic expression a in as we then generate the contour list of target starting at a:

```
cls = generate_contour_list target as
```

where cls is the list of all the possible contour lists generated from target.

Now we can prune tree calling the following function:

```
trees = strong_pruning tree cls
```

where strong_pruning invokes the function strong_prune for each contour list cl in cls:

```
tree = strong_prune tree cl c
```

The function strong_prune receives the tree, tree, to traverse, a contour list cl and the current contour c and executes the following code:

```
fun  strong_prune tree cl c =
     if (tree traversal is over)
     then return tree
     else if (c has no boundaries)
         then tree = strong_update tree
              new = next_contour cl c
              strong_prune tree cl new
         else tree = strong_update tree
              new = next_contour cl c
              weak_prune tree cl new
```

At each recursive call, strong_prune checks if the exploration of the tree has terminated, in which case it returns the pruned tree. If the exploration is still active, then the function checks if the current contour contains any boundary node-vertex. If there are no boundary node-vertices then the function uses strong compatibility to prune nodes at the current level of the tree and then it recursively calls itself. If it finds a boundary node, then the function uses strong compatibility to prune nodes at the current level of the tree and then it switches to weak compatibility by calling the function weak_prune:

```
fun  weak_prune tree cl c =
     if (tree traversal is over)
     then return tree
     else tree = weak_update tree
          new = next_contour cl c
          weak_prune tree cl new
```

weak_prune works exactly like strong_prune except for the fact that it skips the check on the boundaries as now there is no need to switch from one type of compatibility to the other.

In this way, strong_pruning implements the algorithm for extended discrimination nets

	String graphs	!-graphs
Standard pruning (weak pruning)	*Algorithm for standard discrimination net* (sec 3.2)	*Algorithm for weaker extended discrimination net*
Extended pruning (strong pruning)	*Algorithm for improved discrimination net* (sec 3.3)	*Algorithm for extended discrimination net* (sec 3.4)

Table 4.1: Algorithms implemented in the code.

we have described in section 3.4.

Beside strong_pruning we have also implemented another variant of the pruning function called weak_pruning which works by calling since the beginning weak_prune instead of strong_prune.

So, practically, the code we have written implements two different pruning functions:

(i) *Extended pruning (*or *strong pruning)* is the algorithm which starts the pruning process using the notion of strong compatibility and switches to weak compatibility after finding a boundary;

(ii) *Standard pruning (*or *weak pruning)* is the algorithm which uses only weak compatibility during the entire pruning process.

Considering that we have two different types of inputs (string graph or !-graph), at the end we practically have four different algorithms (see table 4.1). We have implemented all the three algorithms described in chapter 3 and moreover we got for free a fourth algorithm (*algorithm for weaker extended discrimination net*) which works as a weaker version of the extended algorithm without using the notion of strong compatibility.

4.3 Results

Using the implementation of the algorithms, we were able to run simulations to compare the performance through the evaluation of the size of the final search space produced by each algorithm.

4.3.1 Simulations

Simulations work by generating a set of random pattern graphs and a random target graph, by creating the topography tree of the pattern graphs and, finally, by pruning the topography tree using the target graph and one of the algorithms implemented. The generation of the pattern graphs and the target graph follows the same procedure and it is regulated by a set of parameters defined by the user (*number of node-vertices, set of kinds, number of edges, number of !-boxes, maximum number of node-vertices per !-box, number of boundary node-vertices*). The random generation of a graph is executed step-by-step in the following way:

(1) Generate a number of node-vertices as specified by the parameter *number of node-vertices*; determine the kind of each one selecting a random value from the parameter *set of kinds* (this parameter is supposed to contain only concrete kinds and not to contain the "boundary" kind);

(2) Instantiate a number of edges as specified by the parameter *number of edges*; for each edge, select randomly a source and a target among the node-vertices instantiated during the previous step;

(3) Instantiate a number of !-boxes as specified by the parameter *number of !-boxes*; for each !-box, select randomly a number of adjacent node-vertices between one and the parameter *maximum number of node-vertices per !-box* and add them to the !-box;

(4) Instantiate a number of boundary node-vertices as specified by the parameter *number of boundary node-vertices*; for each boundary node-vertex, select randomly a node-vertex; then connect the boundary node-vertex to the selected node by randomly choosing the direction of the connecting edge.

Whenever generating a target graph, the parameters *number of !-boxes, maximum number of node-vertices per !-box, number of boundary node-vertices* are forced to zero.

A final user-defined parameter, *number of graphs*, defines how many graphs must be instantiated. When generating pattern graphs this parameter is equal to the size of the pattern graphs set; when generating a target graph this parameter is forced to one.

Notice that the outcome of the random graph generation has a certain probability (given by the number of node-vertices and the number of edges) of being disconnected. This means that we could end up working with a connected component made up by a number of node-vertices smaller than the one defined by the parameter *number of node-vertices*.

In all the simulations we run, our aim was to evaluate the efficiency of our algorithm by computing the size of the final search space, that is counting the number of pattern graphs contained in the branches of the topography tree which were not pruned. Indeed, the number of pattern graphs which are present in the topography tree at the end of the pruning procedure defines the number of times the full matching algorithm must be run. The efficiency of the discrimination procedure is therefore inversely proportional to the number of times the full matching algorithm must be run.

Every time we computed the efficiency of our algorithms, we estimated the figure of the efficiency over one hundred iterations; we repeated the process of random graph generation, topography tree building and pruning one hundred times and then we computed the sample average and the sample variance of the efficiency.

We executed two main sets of simulations:

- *Abstract simulations*: simulations to evaluate the performance of the algorithms in general cases;
- *Concrete simulations*: simulations to evaluate the performance of the algorithms in cases similar to ones that researchers in the field of CQM have to face.

4.3.2 Results of Abstract Simulations

First of all, we implemented a simulation to evaluate how the performance of the standard algorithm (section 3.2) and the performance of the improved algorithm (section 3.3) differ. In order to work only with concrete graphs, the parameters *number of !-boxes, maximum number of node-vertices per !-box, number of boundary node-vertices* were set to zero for all the graphs. Keeping the parameters of the target graph and the number of pattern graphs constant we evaluated the performance as a function of the number of node-vertices (and the number of edges) in the pattern graphs (see figure 4.1).

Both the algorithms provide a significant improvement to the overall matching algorithm by reducing the size of the search space of more than one order of magnitude. The standard algorithm and the improved algorithm have a very similar performance (the scale of the figure 4.1 does not allow to see this minimal difference). Nonetheless, it is interesting to notice that their performance, even if extremely close, is not exactly the same. The improved algorithm guarantees a stricter filtering in that, using strong compatibility, it is able to rule out some instances of target graphs which do not match a pattern graph, but still satisfy weak compatibility (see figure 4.2).

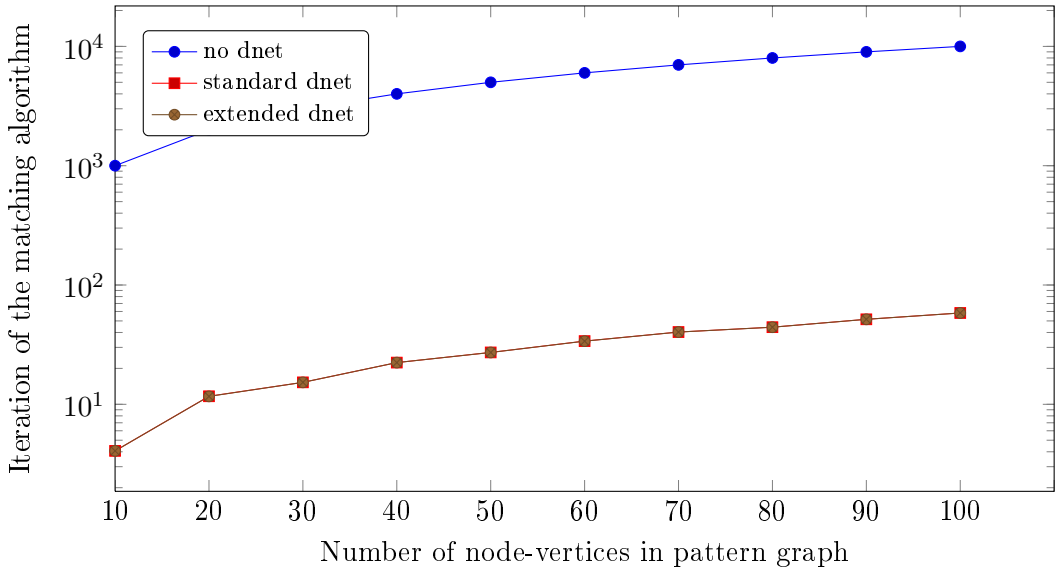


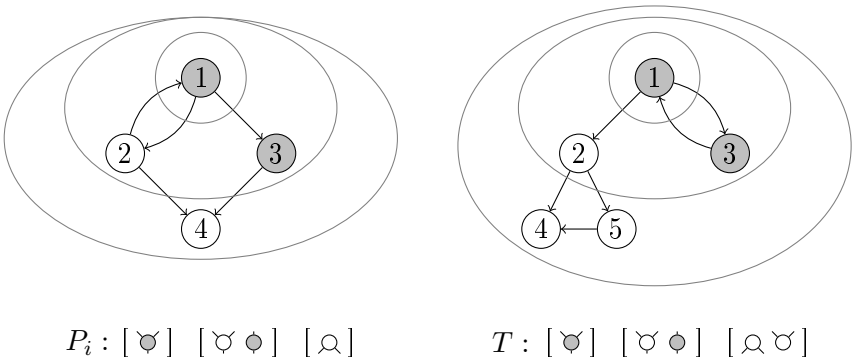
Figure 4.1: Number of times the matching algorithm must be run as a function of the number of node-vertices and the number of edges of the pattern graphs.
In this simulation we kept the parameters of the pattern graphs constant (*number of graphs = 100, number of node-vertices = 30, number of edges = 30, number of !-boxes = 0, maximum number of node-vertices per !-box = 0, number of boundary node-vertices = 0*); we varied the number of node-vertices and the number of edges of the target graph (*number of node-vertices = number of edges*) between 10 and 100.

These few cases account for the slightly better performance of the improved algorithm over the standard algorithm.

The next tests were designed to work with !-graphs. From this point on, we can not use the standard algorithm as a reference anymore, since the standard algorithm is not designed to work with !-boxes. In the following test we will compare the performance only of the algorithms we have proposed in the previous chapter.

The second test evaluates how the introduction of !-boxes affects the performance of the algorithms. In this case we evaluated the performance of the extended algorithm (section 3.4) against the performance of the weak version of the extended algorithm, that is the algorithm which uses only the notion of weak compatibility to prune the topography tree. Keeping the parameters of the target graph and the parameters of the pattern graph (except the number of !-boxes) constant, we evaluated the performance as a function of the number of !-boxes (see figure 4.3).

The results show an increasing gap between the performance of the extended algorithm and the performance of the weaker version of the extended algorithm. As the number of !-boxes increases, the weaker version of the extended algorithm becomes less and less



P_i : [ ] [ ] [ ] T : [ ] [ ] [ ]

Figure 4.2: Example of a pattern graph and a target graph being weakly compatible but not strongly compatible.

efficient. This is due to the fact that using only the notion of weak compatibility we are always looking for a matching between the regular expression of the contour of a pattern graph and a substring of the contour of the target graph; as we are looking for matching on a substring, all the !-boxes can be ignored and will not be used to discriminate. Instead when we use strong compatibility we are looking for a matching between the regular expression of the contour of a pattern graph and the entire string of the contour of the target graph; in this case !-boxes can not be ignored as they may be used to find a correct matching.

In presence of !-boxes, the extended algorithm can then offer a significantly better performance.

A third test was implemented to see how boundary node-vertices affect the performance of the extended algorithm against the performance of the weaker version of the extended algorithm. Keeping the parameters of the target graph and the parameters of the pattern graph (except the number boundary node-vertices) constant, we evaluated the performance as a function of the number of boundary node-vertices (see figure 4.4).

At the beginning, for low values of the number of boundary node-vertices, the performance of the extended algorithm and the performance of the weaker version of the extended algorithm are separated by the same gap we assessed during the previous test. As the number of boundary node-vertices increases, the performances of the two algorithms get closer to each other. This is due to the fact that having more boundary node-vertices, it is more likely that the extended algorithm will incur in one of these boundary node-vertices during the first steps; but as soon as the extended algorithm finds a boundary node-vertex, it switches from strong compatibility to weak compatibility, thus behaving as the weaker version of the extended algorithm.

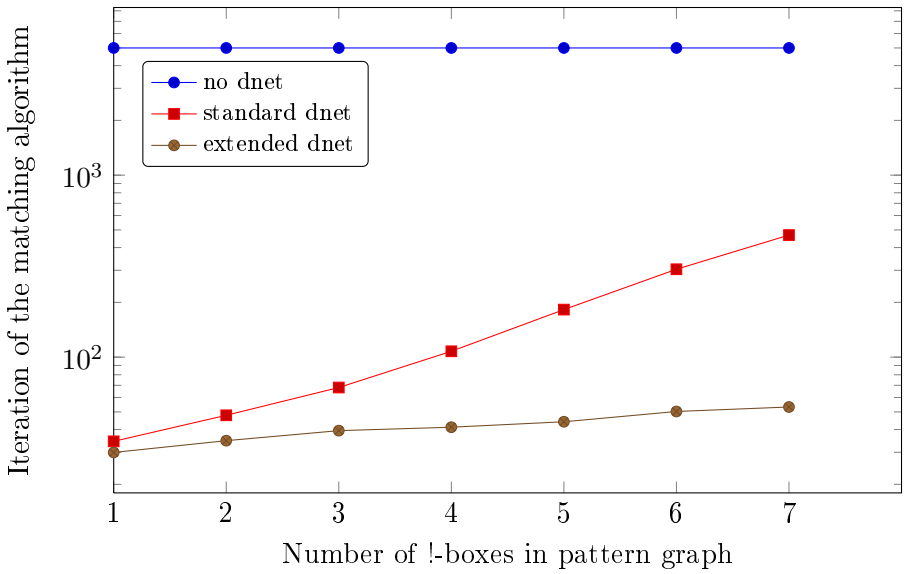


Figure 4.3: Number of times the matching algorithm must be run as a function of the number of !-boxes in the pattern graphs.
In this simulation we kept the parameters of the target graph constant (*number of node-vertices = 50, number of edges = 50*) and we kept all but one of the parameters of the pattern graphs constant (*number of graphs = 100, number of node-vertices = 10, number of edges = 10, maximum number of node-vertices per !-box = 1, number of boundary node-vertices = 0*); we varied the number of !-boxes in the pattern graph (*number of !-boxes*) between 1 and 7.

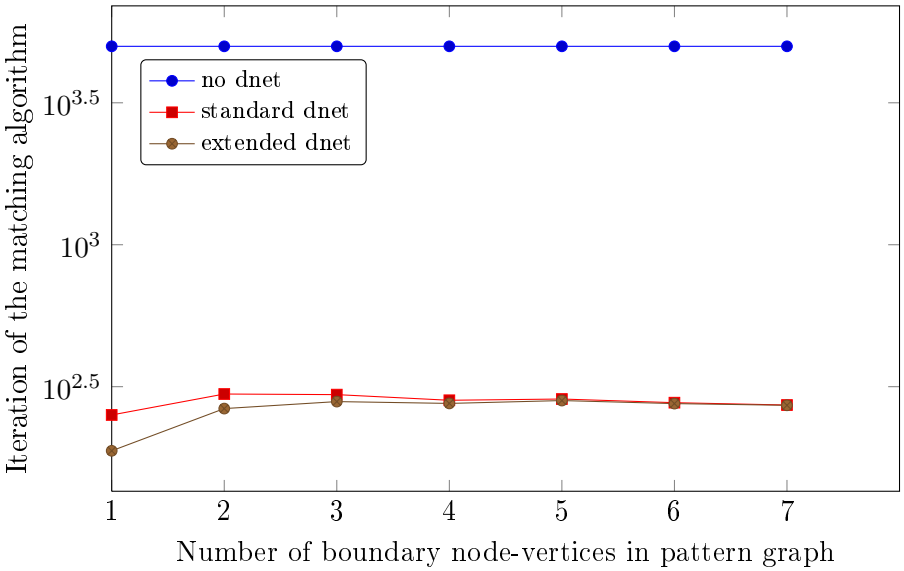


Figure 4.4: Number of times the matching algorithm must be run as a function of the number of boundary node-vertices in the pattern graphs.
In this simulation we kept the parameters of the target graph constant (*number of node-vertices = 50, number of edges = 50*) and we kept all but one of the parameters of the pattern graphs constant (*number of graphs = 100, number of node-vertices = 10, number of edges = 10, number of !-boxes = 5, maximum number of node-vertices per !-box = 1*); we varied the number of boundary node-vertices in the pattern graph (*number of boundary node-vertices*) between 1 and 7.

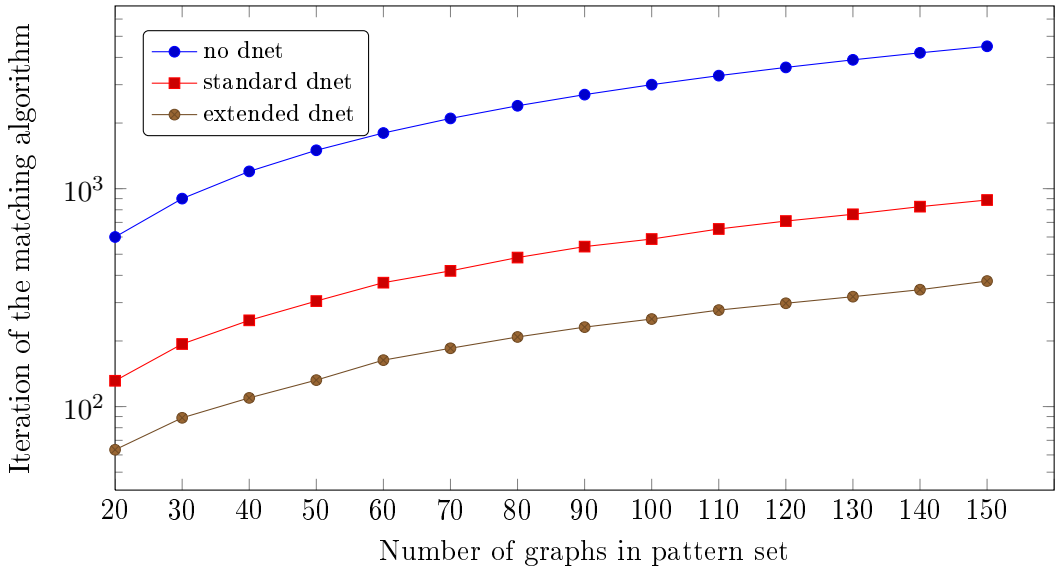


Figure 4.5: Number of times the matching algorithm must be run as a function of the number of pattern graphs in the pattern set.
In this simulation we kept the parameters of the target graph constant (*number of node-vertices = 30, number of edges = 30*) and some of the parameters of the pattern graphs constant (*number of node-vertices = 4, number of edges = 4, number of !-boxes = 3, maximum number of node-vertices per !-box = 1, number of boundary node-vertices = 1*); we varied the number of pattern graphs instantiated (*number of graphs*) between 20 and 150.

4.3.3 Results of Concrete Simulations

Lastly, we implemented simulations in which we set the parameters with values resembling the real values which are normally found in CQM scenarios.

We set pattern graphs to be small graphs having 4 node-vertices, 4 edges, from 2 to 4 !-boxes containing 1 node-vertex and 1 boundary node-vertex. The set of pattern graphs is expected to have a size ranging from 20 to 150 graphs. For the target graph, we instantiated a large graph having a number of node-vertices (and a number of edges) between 10 and 50.

We evaluated the performance of the extended algorithm and the performance of the weaker version of the extended algorithm as a function of the number of graphs in the pattern set (see figure 4.5), as a function of the number of node-vertices in the target graph (see figure 4.6) and as a function of the !-boxes in the pattern graphs (see figure 4.7).

The first two real-world cases show that the extended discrimination net algorithm can offer a better performance than the weaker version based only on the concept of weak compatibility. On average, the extended discrimination net requires approximately only half of the iterations of the full matching algorithm compared to the weaker algorithm. For the last real-world simulation we allowed a higher number of node-vertices and edges in order

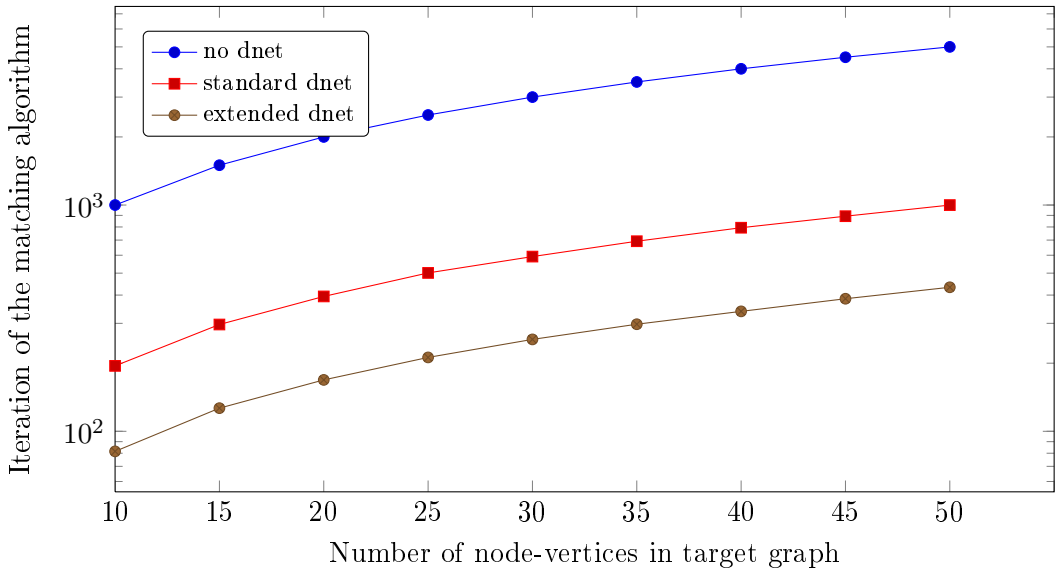


Figure 4.6: Number of times the matching algorithm must be run as a function of the number of node-vertices in the target graph.
In this simulation we kept the parameters of the pattern graph constant (*number of graphs = 100, number of node-vertices = 4, number of edges = 4, number of !-boxes = 3, maximum number of node-vertices per !-box = 1, number of boundary node-vertices = 1*); we varied the number of node-vertices and the number of edges of the target graph between 10 and 50 (*number of node-vertices = number of edges*).

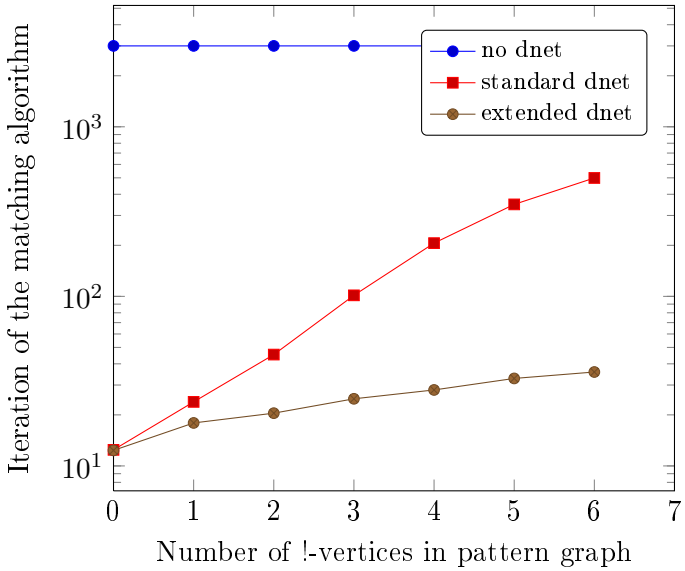


Figure 4.7: Number of times the matching algorithm must be run as a function of the number of !-boxes in the pattern graph.
In this simulation we kept the parameters of the target graph constant (*number of node-vertices = 30, number of edges = 30*) and some of the parameters of the pattern graphs constant (*number of graphs = 100, number of node-vertices = 7, number of edges = 7, maximum number of node-vertices per !-box = 1, number of boundary node-vertices = 0*); we varied the number of !-boxes (*number of !-boxes*) between 0 and 6.

to instantiate more !-boxes and collect more data points. This case confirms that the better performance of the extended algorithm is connected to the number of !-boxes in the pattern graphs; for a low number of !-boxes the performance of the extended algorithm and the performance of the weaker version of it tend to be very similar, but as the number of !-boxes increases the performances diverge, with the extended algorithm obtaining a significant edge over the weaker one.

Chapter 5

Conclusion and Future Work

5.1 Summary of Results

In this dissertation we have studied and analyzed how discrimination nets can be used to speed up the process of graph matching when working with string graphs and !-graphs.

Starting from the standard algorithm for discrimination nets used when working with string graphs, we have proposed a new improved version of this algorithm. Introducing the concepts of strong compatibility and weak compatibility we were able to write an algorithm which guarantees a stricter discrimination compared to the standard algorithm. The correctness of our improved algorithm has been formally proved.

Moving beyond string graphs, we have extended our improved algorithm to !-graphs. Redefining the formalism of the discrimination net and adapting the concepts of strong compatibility and weak compatibility we were able to define an algorithm which can work both with string graphs and !-graphs. The correctness of our extended algorithm has been formally proved, too.

The next step was the implementation of our algorithms within the Quantomatic framework. This allowed us to run a set of simulations to evaluate empirically the performance of our algorithms. When working with string graphs, the simulations showed that the improved algorithm performs indeed better than the standard algorithm; however the conceptual improvement of having two different types of compatibility translate in a very limited practical improvement. When working with !-graphs, the simulations showed all the importance of the concepts of weak and strong compatibility; as the number of !-boxes in the target graphs increased, the extended algorithm guaranteed better and better results compared to a discrimination net algorithm based only on the concept of weak compatibility; in other words, our extended algorithm proved to perform significantly better than a simple extension of the standard algorithm applied to !-graphs.

5.2 Future Work

Future developments on graph matching and discrimination nets could be done both at a theoretical level and at a practical level.

On the theoretical side, a major improvement could be a merge of the discrimination net algorithm and the graph matching algorithm; as the two algorithms use common data and common procedures, it could be possible to devise and to define a single algorithm carrying out discrimination and matching at the same time. Right now, in our implementation, no information is shared by the algorithms, but part of the information generated by the discrimination net algorithm could be forwarded to the graph matching algorithm without the need of being recomputed. Beyond working on the integration of the algorithms, it is also possible to consider different types of discrimination algorithms and evaluate their discriminatory power by comparing their performance with the results obtained using our extended discrimination net algorithm.

On the practical side, we can suggest some improvements that can be done to our implementation. The main improvement could be a deeper integration within Quantomatic and within the module QuantoCoSy; the aim would be to exploit discrimination nets every time the user runs the algorithm for graph matching from the graphical interface of QuantoGUI. Smaller improvements can be made to the code to increase its efficiency, for example: atomic expressions could have a cardinality field, so that, instead of having more different atomic expressions for equivalent node-vertices we could have a single atomic expression having a cardinality greater than one; the targeting function could be redefined; a hash-based search could be implemented for searches within a list of atomic expressions or a list of contours; contours could be built only on-the-fly, when required.

Appendix A

Documentation for the Code

In this appendix, we give the documentation of the implementation of the algorithms for the topography-based discrimination net using Poly/ML language. We first define the structures implemented in the code (section A.1) and then we give an outline of the ideal workflow of the code (section A.2); finally we explain the tests we performed on the code to check its correctness (section A.3) and we describe the script we used to run the simulations to evaluate the performance of the algorithms (section A.4).

All the code is available at:

$$https://github.com/Quantomatic/quantomatic/tree/dnets.$$

The code specific to the algorithms we discussed in chapter 3 is contained in the folder */core/dnets*.

This documentation refers to the version of the code committed on GitHub on the 19th August 2012.

A.1 Structures

Being integrated in Quantomatic, the algorithms we implemented rely on the structures already defined by Quantomatic; in particular we use the structure *BANG_GRAPH* to represent and to manipulate !-graphs and to get access to all the data about a graph.

The implementation of the algorithm for the discrimination net is based on the definition of the following entities:

(1) LITERAL (*Literal.ML*): a functor defining literals and atomic expressions (see signature in algorithm 5). LITERAL defines a types for multiplicity, literal and boundary.

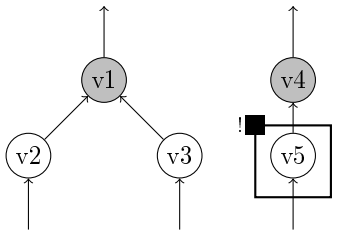
Algorithm 4 Literal signature

```
signature LITERAL =
sig
  type T
  type multiplicity

  structure G : BANG_GRAPH

  val boundary : G.VData.data

  (* CONSTRUCTORS *)
  val mk : G.T -> V.name -> T
  val build : V.name * G.VData.data * multiplicity *
              int * multiplicity * int * multiplicity -> T

  (* COMPARISON FUNCTIONS *)
  val eq : T * T -> bool
  val equiv : T * T -> bool
  val match : T * T -> bool

  (* GETTERS *)
  val get_name : T -> V.name
  val get_kind : T -> G.VData.data
  val get_kind_mult : T -> multiplicity
  val get_input_arity : T -> int
  val get_input_mult : T -> multiplicity
  val get_output_arity : T -> int
  val get_output_mult : T -> multiplicity

  val get_adj : G.T -> T -> T list
  val get_pred : G.T -> T -> T list
  val get_succ : G.T -> T -> T list

  val is_boundary : T -> bool

  (* MULTIPLICITY FUNCTIONS*)
  val mult_none : multiplicity
  val mult_star : multiplicity
  val mult_qm : multiplicity
  val is_kind_mult_none : T -> bool
  val is_kind_mult_star : T -> bool
  val is_kind_mult_qm : T -> bool
  val mult_eq : multiplicity * multiplicity -> bool

  (* PRINT FUNCTION *)
  val printout : T -> string

end
```

Multiplicity is implemented as a datatype having three possible values (none, star and question mark). Literal represents an atomic expression and it is implemented as a record storing all the information about a node-vertex (name of the generating node-vertex, kind of node-vertex, multiplicity of the node-vertex, input arity, multiplicity of the input arity, output arity, multiplicity of the output arity). Boundary defines the kind for boundary node-vertices.
The generation process of an atomic expression is implemented as a two-steps process: in the first step a literal is generated from a graph and a node-vertex; in the second step, the multiplicities of the atomic expression are evaluated considering the contour to which the atomic expression belongs.
LITERAL provides the following subsets of functions to work with atomic expressions:

Constructors: a set of functions to generate an atomic expression; an atomic expression can be built automatically starting from a string graph and a node-vertex (*mk*) or it can be manually built passing all the parameters required to generate an atomic expression (*build*);

Comparison functions: a set of functions to compare atomic expressions between each other; three types of comparison between atomic expressions are possible (see figure A.1):



a_1 = v1, a_2 = v2, a_3 = v2
a_4 = Lit(name="v2", kind="white", kind-mult="none", input-arity=2, ..)
a_5 = v3, a_6 = v4, a_7 = v5

a_2 and a_3 are equal, equivalent and matching as they have the same generating node and the same data (kind, arities and multiplicities);
a_2 and a_4 are equal but not equivalent and not matching as they have the same generating node but different data (different input-arity);
a_2 and a_5 are not equal but they are equivalent and matching as they have a different generating node but equal data (kind, arities and multiplicities);
a_1 and a_6 are not equal, not equivalent but matching as they have a different generating node and different but matching data (input arities can be matched as $input\_arity(a_1) = 2$ and $input\_arity(a_6) = 1^*$).

Figure A.1: Comparisons between atomic expressions.

- Equality (*eq*): two atomic expressions are considered equal if they are generated by the same node-vertex. When comparing two atomic expressions for equality, only their generating node-vertex is compared (no other information about the atomic expressions is compared). Notice that two atomic expressions having the same generating node-vertex but containing different data (such as a different input multiplicity) will be evaluated as equal. This is useful when comparing an atomic expression a_1 in the first step of the generation process and an atomic expressions a_2 in the second step of the generation process; a_1 and a_2 may represent the same generating node-vertex but they may differ on their multiplicities; by comparing only the generating node-vertex we save the computation required to perform the second step of the generating process on a_1 to make a_1 having the same data as a_2. Equality comparison is used during the generation of contours and atomic expressions to guarantee that a single atomic expression for each node-vertex is instantiated;
- Equivalence (*equiv*): two atomic expressions are considered equivalent if their kind, their arities and their multiplicities are the same. Two equivalent atomic expressions may be generated by different node-vertices. When comparing two atomic expressions for equivalence, the two atomic expressions are compared on all the information available except their generating node-vertex. This type of comparison is used during the generation of the topography tree to guarantee that contours containing equivalent node-vertices generate a single tree node in the tree;
- Matching (*match*): two atomic expressions matches if, considering their multiplicities, kind and arities can be made equivalent (e.g.: a boundary kind can be made equivalent to any other kind, an input arity equal to 1^* can be made equivalent to an input arity of 2). This type of comparison is used during the pruning of the topography tree to prevent pruning those contours whose node-vertices match the contours of the target graph;

Getters: a set of functions to get information about atomic expressions; there are three groups of getters: getters to obtain information about the data stored in a given atomic expression (*get_name, get_kind, get_kind_mult, get_input_arity, get_input_mult, get_output_arity, get_output_mult*), getters to retrieve the neighbourhood of a given atomic expression (*get_adj, get_pred, get_succ*) and a getter to query if an atomic expression represents a boundary node-vertex (*is_boundary*);

Multiplicity functions: a set of functions to work with multiplicity; there are three groups of multiplicity functions: functions to expose the multiplicity datatype (*mult_none, mult_star, mult_qm*), functions to test the kind multiplicity of an atomic expression (*is_kind_mult_none, is_kind_mult_star, is_kind_mult_qm*) and a function to compare multiplicities for equality (*mult_eq*);

Print function a function to print an atomic expression for feedback or debug (*printout*).

(2) CONTOUR (*Contour.ML*): a functor defining contours (see signature in algorithm 5). CONTOUR implements the concept of a contour as a list of atomic expressions. CONTOUR provides the following subsets of functions to work with contours:

Constructors: a set of functions to generate a contour; a contour can be generated as an empty list (*empty*) or it can be built given a graph and another contour (*mk*). Two starting constructor to build the 0^{th} contour are also provided: one builds the 0^{th} contour using the target function (*target_function*), the other one generates the 0^{th} contour given a graph and a node-vertex (*mk_first_contour*). Finally a function to add an atomic expression to a contour is provided too (*add_literal*);

Getters: a set of functions to get information about contours; there are three groups of getters: a getter to obtain an atomic expression in the contour (*get_first_literal*), getters to retrieve a contour containing all the atomic expressions with a specified multiplicity (*get_contour_mult_none, get_contour_mult_star, get_contour_mult_qm, get_contour_mult_star_or_qm*) and a getter to query if the contour contains any boundary node-vertex (*contains_boundary*);

Comparison functions: a set of functions to compare contours between each other; the same kind of comparisons explained above have been implemented: equality (*eq*), checking if all the atomic expressions in two contours are equal; equivalence (*equiv*), checking if all the atomic expressions in two contours are equivalent; matching (*check_strong_compatibility, check_weak_compatibility*), checking for matching between the atomic expressions in two contours;

Comparison-equality functions: a set of functions to perform set-like operations on contours using the notion of equality (*subtract_eq_contour, intersect_eq_contours, complement_eq_contour, remove_eq_duplicate, is_eq_literal_contained*);

Comparison-matching functions: a function to perform a set-like operation on contours using the notion of matching (*is_matching_contour_contained*);

Print function: a function to print a contour for feedback or debug (*printout*).

Algorithm 5 Contour signature

```
signature CONTOUR =
sig
        type T
        structure G : BANG_GRAPH
        structure L : LITERAL

        (* CONSTRUCTORS *)
        val empty : T
        val mk : G.T -> T -> T
        val mk_first_contour : G.T -> V.name -> T
        val target_function : G.T -> T
        val add_literal : T -> L.T -> T

        (* GETTERS *)
        val get_first_literal : T -> L.T

        val get_contour_mult_none : T -> T
        val get_contour_mult_star : T -> T
        val get_contour_mult_qm : T -> T
        val get_contour_mult_star_or_qm : T -> T

        val contains_boundary : T -> bool

        (* COMPARISON FUNCTIONS *)
        val eq : T * T -> bool
        val equiv : T * T -> bool
        val check_strong_compatibility : T * T -> bool
        val check_weak_compatibility : T * T -> bool

        (* COMPARISON-EQUALITY FUNCTIONS *)
        val subtract_eq_contour: T * T -> T
        val intersect_eq_contours: T * T -> T
        val complement_eq_contour: T * T -> T
        val remove_eq_duplicate : T -> T
        val is_eq_literal_contained : L.T * T -> bool

        (* COMPARISON-MATCHING FUNCTIONS *)
        val is_matching_contour_contained : T * T -> bool

        (* PRINT FUNCTION *)
        val printout : T -> string

end
```

Algorithm 6 Contour list signature

```
signature CONTOUR_LIST =
sig
        type T
        structure G : BANG_GRAPH
        structure C : CONTOUR

        (* CONSTRUCTORS *)
        val empty : T
    val mk : G.T -> T
        val mk_from : G.T -> V.name -> T

        (* COMPARISON FUNCTION *)
        val equiv : T * T -> bool

        (* PRINT FUNCTION *)
        val printout : T -> string
end
```

(3) CONTOUR LIST (*ContourList.ML*): a functor defining contour lists (see signature in algorithm 6). CONTOUR LIST implements the concept of a contour list as a list of contours.

CONTOUR LIST provides the following subsets of functions to work with contour lists:

Constructors: a set of functions to generate a contour list; a contour list can be generated as an empty list (*empty*) or it can be built giving only a graph, and relying then on the target function (*mk*), or it can be generated giving a graph and specifying a starting node-vertex for the 0^{th} contour (*mk_from*);

Comparison functions: a function to check for equivalence (*equiv*) between the contours comprising two contour lists;

Print function: a function to print a contour list for feedback or debug (*printout*).

(4) TOP_DNET (*Top_ DNet.ML*): a functor defining a topography tree (see signature in algorithm 7). TOP_DNET implements the concept of a topography tree defining a datatype for a tree; every node of the tree contains a contour and a list of children, that is a list of all the children tree nodes; the leaves of the tree contain the name of the graph which is defined by the list of all the contours encountered going from the root to the leaf itself. The information stored in the leaves is useful as, once pruned, the remaining leaves correspond to those pattern graphs on which we have to run

Algorithm 7 Topography discrimination net signature

```
signature TOP_DNET =
sig
        type T
        type tree
        structure G : BANG_GRAPH
        structure CL : CONTOUR_LIST

        (* CONSTRUCTORS *)
        val empty : T
        val mk : G.T GraphName.NTab.T -> T
        val add_cl_to_dnet : T -> CL.T ->
                          -> GraphName.name -> T
        val add_cl_list_to_dnet : T -> CL.T list ->
                               -> GraphName.name list -> T

        (* PRUNING FUNCTIONS *)
        val extended_pruning : T -> G.T ->
                            -> (V.name * GraphName.name list) list
        val standard_pruning : T -> G.T ->
                            -> (V.name * GraphName.name list) list
        val extended_prune : V.name -> G.T -> T -> T
        val standard_prune : V.name -> G.T -> T -> T
        val graphs : T -> GraphName.name list

        (* FOLD FUNCTION *)
        val fold : (('a * 'b) ->
                -> G.T GraphName.NTab.T -> G.T GraphName.NTab.T)
                -> 'a list -> 'b list -> G.T GraphName.NTab.T ->
                -> G.T GraphName.NTab.T

        (* GETTERS *)
        val is_node : tree -> bool
        val get_contour : tree -> CL.C.T
        val get_children : tree -> tree list
        val get_graph : tree -> GraphName.name

        (* COMPARISON FUNCTION *)
        val is_eq_graphs : GraphName.name list *
                         * GraphName.name list -> bool

        (* PRINT FUNCTION*)
        val printout : tree list -> string
end
```

the exact matching algorithm. The information contained in the leaves can be easily modified by changing the definition of this datatype in TOP_DNET.
TOP_DNET implements the two types of pruning algorithm described in section 4.2.
TOP_DNET provides the following subsets of functions to work with topography trees:

Constructors: a set of functions to generate a topography tree; a topography tree can be generated as an empty tree containing only the root node (*empty*) or it can be built starting from a set of graphs (*mk*); the topography tree can also be built incrementally by adding a contour list to the tree (*add_cl_list_to_dnet*) or by adding single contours one at a time (*add_cl_to_dnet*);

Pruning functions: a set of functions to prune the tree and collect the result. Two functions (*extended_pruning, standard_pruning*) run the complete extended algorithm or the weaker version of the extended algorithm starting from a topography tree and a target graph and returning the list of the names of all the graphs which were not pruned. Two functions (*extended_prune, standard_prune*) execute only one step of the extended algorithm or the weaker version of the extended algorithm starting from a topography tree, a target graph and a node-vertex to build the contour of the target graph and returning a pruned topography tree; these two function are called inside the complete algorithm. A last function (*graphs*) collect all the names of the graphs contained in the leaves of a pruned tree.

Fold function: a folding function used to generate the pattern set containing all the pattern graphs and their names (*fold*);

Getters: a set of functions to get information about the tree; there is a function to query if a tree node is a node or a leaf (*is_node*) and a set of getters to retrieve information stored in a node or in a leaf of the tree (*get_contour, get_children, get_graph*);

Comparison functions: a function to check if two lists of graph names (ideally collected from a pruned tree) are equal (*is_eq_graphs*);

Print function: a function to print a topography tree for feedback or debug (*printout*).

(5) TDNET LIBRARY (*DNetsLib.ML*): a structure collecting functions which are shared by the other functors (see signature in algorithm 8). TDNET LIBRARY implements generic library functions which can be called by the other functors.
TDNET LIBRARY provides the following subsets of functions:

Algorithm 8 DNet library list signature

```
signature TDNET_LIBRARY =
sig
        (* GENERIC FUNCTIONS *)
        val maps2 : ('a -> 'b -> 'b list) ->
                  ->'a list -> 'b list -> 'b list
        val maps3 : ('a -> 'b -> 'c list) ->
                  -> 'a -> 'b list -> 'c list

        (* LIST FUNCTIONS *)
        val is_contained : ('a * 'a -> bool) ->
                         -> 'a -> 'a list -> bool
        val rm_duplicates : ('a * 'a -> bool) ->
                          -> 'a list -> 'a list
        val rm_element : ('a * 'a -> bool) ->
                         -> 'a -> 'a list -> 'a list
        val sub_x_y : ('a * 'a -> bool) ->
                      -> 'a list -> 'a list -> 'a list
end
```

Generic functions: a set of generic functions to work with functions and lists (*maps2, maps3*);

List functions: a set of generic functions to work with lists; these functions receive a boolean function to make comparisons between the elements of a list and perform basic set-like operations on the lists (*is_ contained, rm_ duplicates, rm_ element, sub_ x_ y*).

Notice that all the functions we have described are only the functions defined in the signatures, that is those functions which are exposed to allow the user to work with discrimination nets or those functions which have been exposed for testing purposes. The actual implementation of the functors contains several other helper functions. Refer to the code and the inline comments for an explanation of these functions.

A.2 Workflow

We now explain how the main operations for building and pruning topography trees are carried out.

A.2.1 Generating the Topography Tree

Given a list of !-graphs graphs, the topography tree is generated by the following call:

```
val dnet = TOP_DNET.mk graphs
```

TOP_DNET.mk then invokes:

```
val cl_lists = map CONTOUR_LIST.mk graphs
val dnet = TOP_DNET.empty
val dnet = add_cl_list_to_dnet dnet cl_list graph_names
```

where the first instruction generates a list of contours by applying the function *CONTOUR_LIST.mk* to each element of the list of pattern graphs graphs; the second instruction produces an empty topography tree; the third instruction adds all the generated contours to the topography tree; *TOP_DNET.add_cl_list_to_dnet* works by invoking *TOP_DNET.add_cl_to_dnet* for each generated contour.

A.2.2 Generating a Contour List

Given a !-graph graph, its contour list is generated by the following call:

```
val clist = CONTOUR_LIST.mk graph
```

CONTOUR_LIST.mk then invokes:

```
val clist = CONTOUR_LIST.empty

val first_contour = CONTOUR_LIST.target_function graph
val clist = first_contour :: clist

val remaining = (*get literals*)

build_contours(graph, first_contour, clist, remaining)
```

where the first instruction produces an empty contour list; the second instruction generates the first contour for the graph; the third instruction inserts the first contour in the contour list; the fourth instruction retrieves the list of all the atomic expressions in the graph except the one in the first contour (for detailed implementation see the code); the

fifth instruction generates recursively all the other contours taking into account the previous contour and the list of remaining atomic expressions; *CONTOUR_LIST.build_contours* invokes:

```
val new_contour = CONTOUR.mk graph contour

val new_contour = CONTOUR.intersect_eq_contours(new_contour, remaining)
val remaining = CONTOUR.subtract_eq_contour(remaining, new_contour)

build_contours(g, new_contour, new_contour :: clist, remaining)
```

where the first instruction generates a new contour given the graph and the previous contour; the second instruction filters the generated contour keeping only those atomic expressions which are not in any previous contour; the third instruction updates the list of remaining atomic expressions; the fourth instruction recursively calls *CONTOUR_LIST.build_contours* until the list of remaining atomic expressions is empty or all the connected nodes have been processed.

A.2.3 Generating a Contour

Given a !-graph graph and a contour contour, a new contour is generated by the following call:

```
val new_contour = CONTOUR.mk graph contour
```

CONTOUR.mk then invokes:

```
val new_contour = maps2 L.get_adj graph_contour

val new_contour = remove_eq_duplicate new_contour
val new_contour = complement_eq_duplicate contour new_contour
val new_contour = rebuild_literals graph contour new_contour
val new_contour = rebuild_boundaries new_contour
```

where the first instruction retrieves a list containing all the atomic expressions which are adjacent to the atomic expressions in contour; the second instruction removes duplicate

atomic expressions in new_contour; the third instruction removes from new_contour the atomic expressions which are contained in contour; the fourth instruction recomputes the multiplicity of the atomic expressions in new_contour; the fifth instruction evaluates the multiplicity of boundary atomic expressions in new_contour.

A.2.4 Adding a Contour List to the Topography Tree

Given a topography tree dnet and a contour list clist, the contour list is added to the topography tree by the following call:

```
val dnet = TOP_DNET.add_cl_to_dnet dnet clist
```

TOP_DNET.add_cl_to_dnet then traverses the discrimination net dnet and adds the tree nodes and the leaf representing the current contour list clist to dnet (for the detailed implementation see the code).

A.2.5 Pruning the Tree

Given a topography tree dnet and a target graph target, the pruning of the topography tree using the extended algorithm is executed by the following call:

```
val dnet = TOP_DNET.extended_pruning dnet target
```

For every node-vertex v in target, *TOP_DNET.extended_pruning* then invokes:

```
val dnet = TOP_DNET.extended_prune v dnet target
val graphs = TOP_DNET.graphs dnet
```

where first instruction generates the contour list of the target graph target starting at the node-vertex v and then traverses and prune the topography tree dnet (for the detailed implementation see the code); the second instruction collects the name of all the graphs which have not been pruned.

A.3 Testing

A set of tests has been written during the development of the code to guarantee the correctness of the algorithm against the test cases we devised. All the tests are contained in a single file *(test.ML)* which verifies all the functionalities of the code. The test file is divided in two main sections:

- The first part (*Building tests*) builds the pattern graphs and then test all the functions related to the creation of a topography tree; this section is divided in the following subsections: tests checking basic set operations, tests checking atomic expression functions, tests checking contour functions, tests checking contour list functions, tests checking topography tree generation functions;

- The second part (*Pruning tests*) builds the target graphs and then checks all the functions related to the pruning of a topography tree; this section is composed of a single subsection testing both the extended algorithm and the weaker version of the extended algorithm.

Beside testing the correct working of the algorithms in the general case we also performed a set of test on limit cases of particular interest, for example:

(1) The case depicted in figure 3.8 allowed us to test if the algorithms can correctly map two boundary node-vertices to a single concrete node-vertex;

(2) The case depicted in figure 4.2 allowed us to verify that the extended algorithm does indeed provide a stricter discrimination than the weaker version of the extended algorithm;

(3) The case depicted in figure A.2 allowed us to assess if the extended algorithm correctly switches from strong compatibility to weak compatibility when encountering a boundary node-vertex.

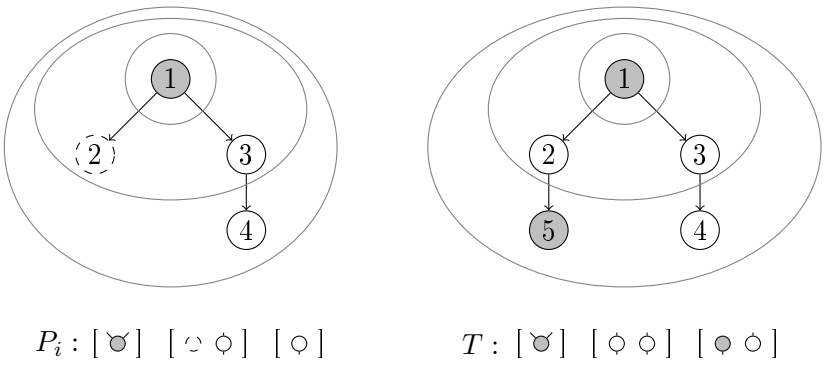


Figure A.2: Example of a case in which the extended algorithm is required to switch correctly between strong compatibility and weak compatibility to recognize that the pattern P_i is contained in the target T.

A.4 Simulation

In order to collect statistical data on the performance of the algorithms we implemented, a script (*perf.ML*) has been written.

The core of this script instantiates random generated pattern graphs and target graphs using a set of parameters (number of node-vertices, number of edges, number of kinds, number of !-boxes, maximum number of node-vertices per !-box, number of boundary node-vertices) specified by the user. After generating the graphs, the script computes the topography tree for the pattern graphs and prunes it, first using the extended algorithm and then the weaker version of the extended algorithm.

The script iterates this procedure a number of times defined by the user; at the end, the script returns statistical data (sample mean and sample variance of the number of non-pruned graphs) about the performance using the extended discrimination net algorithm, using the weaker version of the extended algorithm or using no algorithm at all.

Bibliography

[1] Samson Abramsky and Bob Coecke. A categorical semantics of quantum protocols. *Proceedings of the 19th Annual IEEE Symposium on Logic in Computer Science*, 415:425, 2004.

[2] Steve Awodey. *Category Theory*. Oxford University Press, 2006.

[3] Franz Baader and Tobias Nipkow. *Term Rewriting and All That*. Cambridge University Press, 1999.

[4] Jim Christian. Flatterms, discrimination nets, and fast term rewriting. *Journal of Automated Reasoning*, 10:95–113, 1993.

[5] Bob Coecke. Quantum picturialism. 2009.

[6] J. J. Dick. An introduction to knuth-bendix completion. *The Computer Journal*, 34, 1991.

[7] Lucas Dixon and Ross Duncan. Graphical reasoning in compact closed categories for quantum computation. 2008.

[8] Lucas Dixon and Aleks Kissinger. Monoidal categories, graphical reasoning, and quantum computation. 2009.

[9] Lucas Dixon and Aleks Kissinger. Open graphs and monoidal theories. 2010.

[10] Joe Douglas. Discrimination nets for faster quantum reasoning. Master's thesis, University of Edinburgh, 2010.

[11] Ross Duncan. Lectures on categorical quantum mechanics. 2010.

[12] Chris Heunen and James Vicary. Lectures on categorical quantum mechanics. 2012.

[13] Moa Johansson, Lucas Dixon, and Alan Bundy. Conjecture synthesis for inductive theories. *Journal of Automated Reasoning*, 2010.

[14] André Joyal and Ross Street. The geometry of tensor calculus i. *Advances in Mathematics*, 88:55–112, 1991.

[15] Aleks Kissinger. *Pictures of Processes: Automated Graph Rewriting for Monoidal Categories and Applications to Quantum Mechanics.* PhD thesis, University of Oxford, 2011.

[16] Aleks Kissinger. Synthesising graphical theories. 2012.

[17] Aleks Kissinger, Alex Merry, Lucas Dixon, Ross Duncan, Matvey Soloviev, and Ben Frot. Quantomatic, 2011.

[18] Aleks Kissinger, Alex Merry, and Matvey Soloviev. Pattern graph rewrite systems. 2012.

[19] J.W. Klop. *Term Rewriting Systems*, pages 2–116. Oxford University Press, 1987.

[20] D. E. Knuth and P. B. Bendix. Simple word problems in universal algebras. *Computational Problems in Abstract Algebra*, pages 263–297, 1970.

[21] Saunders Mac Lane. *Categories for the Working Mathematician.* Springer, 1971.

[22] William McCune. Experiments with discrimination-tree indexing and path indexing for term retrieval. 1990.

[23] L. C. Paulson. Isabelle: A generic theorem prover. *Springer*, 2004.

[24] Lawrence C. Paulson. *ML for the Working Programmer.* Cambridge University Press, 1996.

[25] Roger Penrose. Applications of negative dimensional tensors. *Combinatorial Mathematics and its Applications*, pages 221–244, 1971.

[26] Peter Selinger. Dagger compact closed categories and completely positive maps (extended abstract). 2005.

[27] Peter Selinger. *New Structures for Physics*, pages 289–355. Springer, 2011.