UNIVERSIDADE FEDERAL FLUMINENSE

MAURÍCIO DA SILVA PIRES

Revisitando o Teorema de Courcelle

NITERÓI

2020

UNIVERSIDADE FEDERAL FLUMINENSE

MAURÍCIO DA SILVA PIRES

Revisitando o Teorema de Courcelle

Dissertação de Mestrado apresentada ao Programa de Pós-Graduação em Computação da Universidade Federal Fluminense como requisito parcial para a obtenção do Grau de Mestre em Computação. Área de concentração: Engenharia de Sistemas e Informação

Orientadores:
Bruno Lopes Vieira
Uéverton dos Santos Souza

NITERÓI

2020

Ficha catalográfica automática - SDC/BEE
Gerada com informações fornecidas pelo autor

P667r Pires, Maurício da Silva
Revisitando o Teorema de Courcelle / Maurício da Silva Pires ; Bruno Lopes Vieira, orientador ; Uéverton dos Santos Souza, coorientador. Niterói, 2020.
130 f.

Dissertação (mestrado)-Universidade Federal Fluminense, Niterói, 2020.

DOI: http://dx.doi.org/10.22409/PGC.2020.m.15169213727

1. Complexidade computacional. 2. Linguagem formal. 3. Teoria dos grafos. 4. Lógica computacional. 5. Produção intelectual. I. Vieira, Bruno Lopes, orientador. II. Souza, Uéverton dos Santos, coorientador. III. Universidade Federal Fluminense. Instituto de Computação. IV. Título.

CDD -

Bibliotecário responsável: Sandra Lopes Coelho - CRB7/3389

Maurício da Silva Pires

Revisitando o Teorema de Courcelle

Dissertação de Mestrado apresentada ao Programa de Pós-Graduação em Computação da Universidade Federal Fluminense como requisito parcial para a obtenção do Grau de Mestre em Computação. Área de concentração: Engenharia de Sistemas e Informação.

Aprovada em março de 2020.

BANCA EXAMINADORA

Prof. Bruno Lopes Vieira - Orientador, UFF

Prof. Uéverton dos Santos Souza - Orientador, UFF

Prof. Fábio Protti - UFF

Prof. Mario Benevides - UFF

Prof. Edward Hermann Hauesler - PUC-Rio

Pesquisador Mateus de Oliveira Oliveira - University of Bergen

Niterói

2020

Aos que compreendem o valor do conhecimento

Agradecimentos

Meus agradecimentos se iniciam pelos meus pais. Sem todo apoio e compreensão nada disso seria possível. Obrigado por acreditarem no meu potencial e estarem ao meu lado durante esses dois anos de mestrado. Sonhos foram feitos para serem compartilhados e fico feliz em poder dividi-los com vocês.

Aos professores Bruno Lopes e Uéverton Souza, deixo aqui minha admiração e respeito. Admiro-os demais pelo profissionalismo e me sinto privilegiado de ter sido orientado por vocês. Tenham certeza de que carregarei tudo que aprendi com vocês com muito carinho. Muito obrigado!

Aos meus amigos e colegas do Instituto de Computação da Universidade Federal Fluminense, não poderia deixar de agradecê-los. Sabemos o quão difícil é esta jornada, mas creio que todos compartilhamos do mesmo sentimento de satisfação. Fazemos pesquisa por acreditar num futuro melhor. Desejo que continuemos lutando pelo progresso e pela ciência. Muito obrigado por tudo! Sempre estarão em meu coração e em minha mente!

Por fim, meu muito obrigado à Coordenação de Aperfeiçoamento de Pessoal de Nível Superior (CAPES) pelo financiamento a esta pesquisa e também ao Conselho Nacional de Desenvolvimento Científico e Tecnológico (CNPq) e à Fundação Carlos Chagas Filho de Amparo à Pesquisa do Estado do Rio de Janeiro (Faperj) por financiarem o projeto no qual esta pesquisa está inserida.

Resumo

Grafos são estruturas matemáticas que possuem um alto potencial para modelar problemas do mundo real. Assim, estudos computacionais de problemas em grafos são um excelente ponto de partida na busca de soluções práticas a esses problemas. A Teoria da Complexidade Parametrizada apresenta um conjunto de ferramentas de projeto e análise de algoritmos que permite um estudo mais elaborado de problemas em grafos, visto que considera características das entradas ou dos problemas nas tarefas de projeto e análise. Um exemplo de parâmetro é a chamada *treewidth*, que é uma invariante do grafo que mensura sua semelhança com uma árvore.

O Teorema de Courcelle é um dos mais relevantes resultados parametrizados sobre estudos em grafos. O Teorema estabelece que problemas em grafos expressíveis em fórmulas de Lógica Monádica de Segunda Ordem de tipo 2 podem ser verificados de forma eficiente segundo a Complexidade Parametrizada em instâncias com *treewidth* limitada por um valor k pré-estabelecido. Este resultado possui duas importantes facetas. A primeira refere-se à tarefa de classificação de problemas parametrizados na classe dos problemas tratáveis de acordo com a teoria. A segunda, muito pouco explorada na literatura, refere-se à implementação das etapas sugeridas em sua demonstração. Isso significa que a prova do Teorema de Courcelle envolve processos passíveis de mecanização, e, portanto, podem ser utilizados como um algoritmo para resolver problemas em grafos.

Neste trabalho é apresentada uma descrição algorítmica completa e explícita para as etapas envolvidas na prova da versão clássica do Teorema de Courcelle. O resultado obtido é um algoritmo baseado em autômatos que permite verificar propriedades em grafos por meio da técnica conhecida como *model checking*.

Palavras-chave: *Model checking*, *Treewidth*, Complexidade parametrizada, Grafos, Linguagens formais, *Tree automata*

Abstract

Graphs are mathematical structures with a high potential to model real-world problems. Thus, computational studies of graph problems are an excellent starting point in searching for practical solutions to these problems. The Parameterized Complexity Theory presents a set of algorithm design and analysis tools that allows a more detailed study of problems in graphs since it considers characteristics of the inputs or problems in the design and analysis tasks. A well-known example of a parameter is treewidth, an invariant of the graph that measures its resemblance to a tree.

Courcelle's Theorem is one of the most relevant parameterized results in graph studies. This Theorem proves that graph problems expressed in formulas of type 2 Monadic Second-Order Logic are verifiable in an efficient manner concerning Parameterized Complexity in instances with treewidth limited by a pre-established k value. This result is twofold. According to the theory, the first refers to the task of classifying problems parameterized in the class of treatable problems. The second, very little explored in the literature, refers to implementing the steps suggested in its demonstration. It means that the proof of Courcelle's Theorem involves mechanizable processes, and therefore used as an algorithm to solve graph problems.

This work presents a complete and explicit algorithmic description for all of the steps involved in the proof of the classic version of Courcelle's Theorem. The result obtained is an algorithm based on Tree-Automata that allows verifying properties in graphs through the technique known as model checking.

Keywords: Model checking, Treewidth, Parameterized complexity, Graphs, Formal languages, Tree automata.

Lista de Figuras

2.1 Processo de derivação da palavra w_{3a} a partir de $\mathbf{G}_{3a}$ 9

2.2 Diferença entre transições determinísticas e não-determinísticas 10

2.3 Autômato não-determinístico para a linguagem de $\mathbf{G}_{3a}$ 11

2.4 Autômato determinístico para a linguagem de $\mathbf{G}_{3a}$ 11

2.5 Visualização hierárquica da árvore $\sigma_1[\sigma_3, \sigma_1[\sigma_2[\sigma_3], \sigma_3]]$ 12

2.6 Exemplo da execução de A_{bool} tomando t_{bool} como entrada 14

2.7 Processo de derivação da árvore t_{bool} a partir de $\mathbf{G}_{bool}$ 15

2.8 Exemplo de palavra e árvore denotada pelo alfabeto estendido para três conjuntos 26

2.9 Exemplo de árvore e sua codificação de posições 37

2.10 Visualização hierárquica da construção do autômato 50

2.11 Autômato para $sing(X)$ 51

2.12 Autômato para $sing(Y)$ 51

2.13 Autômato para $S(X, Y)$ 51

2.14 Autômato para $X \subseteq Q_0$ 52

2.15 Autômato para $Y \subseteq Q_0$ 52

2.16 Autômato para $sing(X) \wedge sing(Y)$ 53

2.17 Autômato para $(X \subseteq Q_0) \wedge (Y \subseteq Q_0)$ 53

2.18 Autômato para $((X \subseteq Q_0) \wedge (Y \subseteq Q_0)) \wedge S(X, Y)$ 54

2.19 Autômato para $sing(X) \wedge sing(Y) \wedge (X \subseteq Q_0) \wedge (Y \subseteq Q_0) \wedge S(X, Y)$ . . . 55

2.20 Autômato para primeira projeção em $sing(X) \wedge sing(Y) \wedge (X \subseteq Q_0) \wedge (Y \subseteq Q_0) \wedge S(X, Y)$ 56

2.21 Autômato para segunda projeção em $sing(X) \wedge sing(Y) \wedge (X \subseteq Q_0) \wedge (Y \subseteq Q_0) \wedge S(X,Y)$ 56

3.1 Exemplos de grafo. 59

3.2 Exemplo de um grafo G (a) e o grafo de incidência associado (b). 59

3.3 Notações gráficas para hiperarestas. 60

3.4 Exemplo de hipergrafo. 61

3.5 Exemplo de um grafo e uma *tree decomposition* para ele. 63

3.6 Processos de tradução de problemas envolvidos no algoritmo de Courcelle . 64

4.1 Exemplo de inconsistência de marcações em nó de junção 80

4.2 Exemplo de *tree expression* denotada em topologia de árvore e seu grafo associado 82

5.1 Exemplo de vértices representados por nós da árvore por meio da relação *inc* 87

Lista de Tabelas

2.1 Regras de produção para a gramática $\mathbf{G}_{3a}$. 9

2.2 Regras de produção para a gramática $\mathbf{G}_{bool}$ 14

2.3 Tabela-verdade para conjunção 16

2.4 Tabela-verdade para disjunção 16

2.5 Tabela-verdade para negação 16

2.6 Tabela-verdade para implicação 16

2.7 Tabela-verdade para $p_1 \wedge (p_2 \vee \neg p_1)$ 17

2.8 Sistema de transições para $sing(X) \wedge sing(Y) \wedge (X \subseteq Q_0) \wedge (Y \subseteq Q_0) \wedge S(X, Y)$ 55

2.9 Sistema de transições após a primeira projeção 55

2.10 Sistema de transições após a segunda projeção 56

Lista de Gramáticas

2.1 Sintaxe da Lógica Proposicional 15

2.2 Sintaxe da Lógica de Primeira Ordem 18

2.3 Sintaxe da Lógica Monádica de Segunda Ordem 19

Lista de Abreviaturas e Siglas

FOL	:	Lógica de Primeira Ordem;
FPT	:	Tratável por Parâmetro Fixo;
FTA	:	*Finite tree automaton*
FTA_B	:	*Bottom-up finite tree automaton*
FTA_T	:	*Top-down finite tree automaton*
HRG	:	*Hyperedge replacement graph grammar*;
MSOL	:	Lógica Monádica de Segunda Ordem;
MSOL_1	:	Lógica Monádica de Segunda Ordem tipo 1;
MSOL_2	:	Lógica Monádica de Segunda Ordem tipo 2;
NRG	:	*Node replacement graph grammar*;
RTG	:	*Regular tree grammar*;

Sumário

1 Contextualização **1**

1.1 Grafos e *treewidth* 3

1.2 O Teorema de Courcelle 3

1.3 Contribuições 5

1.4 Trabalhos Relacionados 5

1.5 Organização da dissertação 6

2 Linguagens, Lógica e Verificação Formal **7**

2.1 Linguagens formais 7

2.1.1 Linguagem de palavras 8

2.1.2 *Tree languages* 11

2.2 Lógica 15

2.2.1 Lógica Proposicional 15

2.2.2 Lógica de Primeira Ordem 17

2.2.3 Lógica Monádica de Segunda Ordem 19

2.3 *Model Checking* 20

2.4 Verificação de Propriedades MSOL-expressíveis 23

2.4.1 Palavras como modelos 27

2.4.2 Árvores como modelos 36

2.4.3 Exemplo para o caso de palavras 49

3 (Hiper)Grafos e Teorema de Courcelle **58**

3.1 Grafos e Hipergrafos 58

3.2 *Tree decompositions* e *Treewidth* 61

3.3 Preliminares no algoritmo de Courcelle 64

3.3.1 Formalização de problemas 65

3.3.2 Processamento de entradas 68

4 Grafos codificados em árvores 70

4.1 Gramáticas de grafos 71

4.2 *Tree language* para gramáticas de grafo 73

4.3 Tradução de modelos 75

4.4 Algoritmo de codificação de *tree decompositions* 77

4.4.1 Codificação via *extended nice tree decomposition* 77

4.4.2 Uma estratégia baseada em coloração 79

5 Autômatos e Fórmulas em $\mathbf{MSOL_2}$ 83

5.1 De grafos a árvores 84

5.2 Algoritmos para *tree automata* baseados em MSOL_2 88

5.2.1 *Tree automata* para predicados 89

5.2.2 Subrotinas para composição e projeção de *tree automata* 104

6 Considerações Finais e Trabalhos Futuros 110

Referências 113

Capítulo 1

Contextualização

O amplo uso de ferramentas computacionais em diferentes áreas da vida cotidiana e da indústria cria uma demanda cada vez maior por soluções que apresentem bom desempenho ao trabalhar com um grande volume de dados. Entende-se por uma solução com bom desempenho aquela que consegue manter uma relação tão uniforme quanto possível entre a massa de dados de entrada e o uso de recursos computacionais necessários ao seu processamento. A relação entre os dados de entrada e poder computacional requisitado para o processamento é melhor formalizada na chamada **Teoria da Complexidade Computacional Clássica**.

A Teoria da Complexidade Computacional Clássica [43, 17] (ou simplesmente complexidade clássica) é o *framework* mais tradicional para projeto e análise de algoritmos. Nesta teoria, os algoritmos são hierarquizados de acordo com o crescimento da curva que representa o número de passos necessários a sua execução ou à quantidade necessária de memória para executá-lo. Assim, uma solução de bom desempenho na complexidade clássica é aquela cujo comportamento é, no máximo, polinomial no tamanho da entrada, tal que o polinômio em questão possua um grau baixo. Quanto menor for o grau máximo desse polinômio, melhor o desempenho. Algoritmos que possuem um comportamento polinomial são ditos pertencentes à classe **P**.

Problemas estudados na Complexidade Clássica podem ser representados em alto nível pelos chamados **problemas de decisão**. Esses problemas consistem em avaliar a presença de uma propriedade, denotada por Π, em uma instância de entrada, denotada por I. A notação usual para um problema de decisão é a seguinte:

$Decide(I, \Pi)$

Entrada: Uma instância de entrada I e uma propriedade Π.

Questão: Π ocorre em I?

Apesar de muito utilizada, a Teoria da Complexidade Clássica possui uma limitação. Ao considerar exclusivamente o tamanho da entrada para analisar a curva que descreve o comportamento do algoritmo, o *framework* tradicional desconsidera características inerentes das instâncias de entrada. Deste modo, a análise produzida é de uma granularidade grossa, ocultando informações que podem ser cruciais para a construção de algoritmos com possível bom desempenho. Ainda que não se obtenha um algoritmo eficiente para instâncias genéricas, uma solução voltada para instâncias que compartilhem similaridades pode ser desejável do ponto de vista prático. Essa limitação demanda uma nova forma de estudar os algoritmos, permitindo assim uma investigação em uma granularidade mais fina, explorando melhor toda informação previamente obtida sobre os dados de entrada.

Proposta por Downey e Fellows, a **Teoria da Complexidade Parametrizada** [33, 28, 29] busca estudar problemas computacionais através de algoritmos que exploram diferentes características dos dados de entrada, sempre visando um resultado melhor que o tradicional. Tais características são denominadas **parâmetros**. Um parâmetro é um mapeamento computável que associa a cada instância de entrada do problema um número inteiro não-negativo, o qual está associado a algum aspecto da entrada ou da questão a ser respondida. Assim, denomina-se **problema parametrizado** todo par $\langle \Pi, \kappa \rangle$, onde Π é um problema de decisão no sentido usual e $\kappa : I(\Pi) \to \mathbb{Z}^+$ é o mapeamento representando o parâmetro.

De modo similar ao que acontece na complexidade clássica, os problemas parametrizados também são classificados de maneira hierárquica. Se na teoria clássica um algoritmo é visto como possuindo um bom desempenho se possui um comportamento polinomial em relação ao tamanho da entrada, na teoria parametrizada essa noção é um pouco mais flexível. A classe de **problemas tratáveis por parâmetro fixo**, ou simplesmente **FPT**, reúne problemas parametrizados para os quais é possível construir algoritmos cuja complexidade seja da forma $f(\kappa) \times n^c$, onde $f(\kappa)$ é qualquer função computável no parâmetro e c é uma constante. Não há restrição alguma sobre o tipo da função f, e isso não costuma representar um problema dado que o parâmetro é muito menor que o tamanho da entrada. Assim, em cenários onde se obtém um algoritmo **FPT** para o problema tal que $\kappa(I)$ e c sejam valores pequenos, obtém-se um algoritmo factível para problemas **NP**-completos ou **NP**-difíceis. Logo, a flexibilidade da Teoria da Complexidade Parametrizada se mostra

fundamental para o estudo dos clássicos problemas da classe **NP**, para os quais não se conhecem algoritmos de tempo polinomial que os solucionem, a menos que **P** = **NP**.

1.1 Grafos e *treewidth*

Um **grafo** é uma construção matemática composta de um conjunto de elementos, chamados de **vértices**, e uma relação binária sobre esse conjunto de elementos, onde cada par de elementos pertencente a essa relação é denominado **aresta**. Por sua representação simples e alto potencial para modelar situações reais, o estudo de problemas em grafos é um dos pontos de partida na busca de soluções eficientes para problemas. Tome como exemplo uma transportadora. Suponha que deseja-se definir as melhores rotas de entrega dentro de uma cidade. Recolhendo informações sobre o fluxo do trânsito, presença de semáforos, limite de velocidade e horários de pico de transporte, pode-se agregá-las a um grafo que represente a organização do trânsito da cidade. Nesse cenário, a solução do problema pode ser encontrada calculando caminhos mínimos no grafo, que representariam as rotas com as melhores combinações dos fatores citados anteriormente. Infelizmente, inúmeros problemas em grafos são sabidamente **NP**-difíceis. Portanto, soluções parametrizadas são bem-vindas nesses cenários.

Sob a ótica da complexidade parametrizada, os problemas em grafos podem ser estudados explorando invariantes estruturais das entradas. O parâmetro conhecido como *treewidth* mede a semelhança de um grafo qualquer com uma árvore. Essa medida é computada a partir de uma codificação do grafo original. A codificação considera uma árvore que terá seus nós associados a partes do grafo de entrada. Tal divisão do grafo permite que a avaliação de uma propriedade seja realizada de maneira controlada através programação dinâmica. Baseando-se em divisão e conquista, cada um dos nós da árvore de decomposição é processado para gerar um conjunto de soluções candidatas, que serão compostas de maneira adequada para obter a solução final do problema.

1.2 O Teorema de Courcelle

Enunciado em 1990, o Teorema de Courcelle [19, 21, 29] é um dos mais relevantes resultados da Teoria da Complexidade Parametrizada. Este resultado ajuda a estabelecer a classificação de diferentes problemas parametrizados em grafos [11, 13] considerando o parâmetro denominado *treewidth* (twd). O Teorema estabelece uma tratabilidade gené-

rica que, de certa forma, é "instanciada" para os problemas via representação lógica. Essa ideia de instanciação é associada a fórmula a ser verificada. Cada fórmula é mapeada em um dispositivo computacional específico que realizará o processamento final no algoritmo. Uma vez construído o dispositivo, ele pode ser reaproveitado para avaliar a mesma propriedade em qualquer instância que respeite a limitação da *treewidth*. Portanto, é natural encarar o Teorema de Courcelle como uma receita guiada pelo resultado desejado. Isso confere ao resultado o título de metateorema algorítmico [36, 44, 37], dado que a prova do Teorema envolve processos construtivos que podem ser reproduzidos mecanicamente.

Teorema (Courcelle 1990)**.** *Dados um grafo G que possui $twd(G) \leq k$ e uma fórmula φ em $MSOL_2$, pode-se verificar em tempo **FPT** se $G \models \varphi$.*

O trabalho de Courcelle apresenta uma solução para problemas em grafos quando a propriedade a ser verificada pode ser expressa por uma fórmula em um fragmento de Lógica Monádica de Segunda Ordem (MSOL_2) e a instância de entrada possua *treewidth* limitada por um valor k pré estabelecido. A *treewidth* é um parâmetro que quantifica a semelhança de um grafo com uma árvore. Assim, o autor explora uma estratégia de *model checking* parametrizado, derivada do trabalho de Büchi [12] e Tatcher e Wright [53], executando o seguinte protocolo:

Dado um grafo G com *treewidth* limitada por k, faça:

1. Obtenha uma *tree decomposition* para este grafo.
2. Codifique a *tree decomposition* em uma árvore.
3. Construa um autômato a partir da fórmula φ em MSOL_2.
4. Se o autômato reconhecer a árvore, então $G \models \varphi$ e a resposta do problema de decisão é **sim**. Senão, $G \not\models \varphi$ e a resposta do problema de decisão é **não**.

Além da característica de "instanciação" do algoritmo subjacente à prova do Teorema, ele possui ainda uma segunda faceta arquetípica. O limite k para o valor da *treewidth* serve como limitante máximo das classes de grafos reconhecidas pelo autômato construído. Isso quer dizer que qualquer grafo que tenha *treewidth* igual a k' e que possua a propriedade formalizada por φ será reconhecido pelo autômato construído a partir de φ, desde que $k' \leq k$. Se as instâncias de entrada do problema forem agrupadas pelo valor de sua *treewidth*, então o Teorema de Courcelle garante que, para cada k estipulado, o conjunto $\bigcup\{I \mid twd(I) = k'\}_{1 \leq k' \leq k}$ pode ser avaliado pelo autômato construído.

É importante ressaltar que a versão clássica do Teorema de Courcelle apresenta um procedimento de decisão. A resposta final do processo é sempre se a entrada possui ou não a propriedade desejada. Suponha, por exemplo, o problema de verificar se um grafo pode ter seu conjunto de vértices dividido em dois subconjuntos não vazios e disjuntos, tal que não exista uma aresta entre quaisquer pares de vértices pertencentes a um mesmo subconjunto. Esta propriedade é conhecida como bipartição de um grafo ou 2-coloração. O procedimento clássico descrito por Courcelle seria capaz de atestar se o grafo de entrada pode ser bipartido, não sendo, portanto, capaz de apresentar a solução para o problema, no caso, de exibir os subconjuntos que satisfaçam a propriedade.

1.3 Contribuições

Dada a importância do Teorema de Courcelle para o desenvolvimento de algoritmos parametrizados e a ausência de uma formalização algorítmica explícita e completa para este teorema na literatura, este trabalho dedica-se a apresentar tal formalização. Tomando por base o artigo de Courcelle de 1992 [19], o trabalho desenvolve os principais conceitos necessários ao projeto do algoritmo, buscando alternativas ao processo apresentado com a finalidade de produzir um material que seja acessível a leitores não familiarizados com conceitos de complexidade parametrizada [28, 33, 29], verificação de modelos [14, 15] e *graph grammars* [52, 32].

1.4 Trabalhos Relacionados

Courcelle ressalta sempre a possibilidade de explosão combinatória dos estados do autômato no momento de construí-lo a partir da fórmula em MSOL_2. Como solução a esse problema, ele mesmo propõe o uso de um autômato que seja construído em tempo de execução [24, 23, 22], o que reduziria o gasto de memória envolvido neste processo.

Kneis et al. [41, 42] apresentam uma abordagem prática ao teorema utilizando uma versão híbrida de teoria dos jogos e *model checking*. Todavia, este trabalho é restrito apenas à Lógica de Primeira Ordem. Uma versão ampliada dele é apresentada por Emily Black [6], que apresenta uma parcial formalização do teorema utilizando o assistente de prova Agda.

Um *survey* sobre métodos eficientes de verificação de propriedades MSOL expressíveis pode ser encontrado no trabalho de Langer et al. [45].

1.5 Organização da dissertação

Esta dissertação está organizada da seguinte maneira. O Capítulo 2 introduz o leitor aos conceitos básicos da verificação de modelos, focando na verificação de propriedades expressas em Lógica Monádica de Segunda Ordem (MSOL). Familiarizados com esses conceitos, o Capítulo 3 discute as transformações a serem realizadas nos grafos de entrada e as informações preliminares sobre o processo apresentado por Courcelle. Esclarecidos os passos iniciais do algoritmo, a discussão sobre a relação de linguagens formais com as *tree decompositions* é apresentada no Capítulo 4. A técnica para compilação de fórmulas em MSOL_2 em autômatos é o tema do Capítulo 5. O trabalho é concluído no Capítulo 6 com a apresentação das conclusões e oportunidades de trabalhos futuros.

Capítulo 2

Linguagens, Lógica e Verificação Formal

Descrever problemas computacionais por meio de fórmulas lógicas permite que tais problemas sejam abordados de maneira elegante e livre de eventual ambiguidade inerente da linguagem natural. Adicionalmente, a representação lógica é acrescida de um *framework* que permite estudar os problemas representados. Uma das ferramentas disponíveis para este estudo é a chamada verificação de modelos (*model checking*) [14, 3, 15], que consiste na tentativa de validação de propriedades de interesse sobre um modelo matemático. Portanto, os problemas de decisão devem ser traduzidos para problemas de *model checking*, que possuem o seguinte formato:

$Model(\mathcal{C}, \mathcal{L})$
Entrada: Uma estrutura $\mathcal{M} \in \mathcal{C}$ e uma fórmula φ da lógica $\mathcal{L}$.
Questão: $\mathcal{M}$ é um modelo para φ, ou seja, $\mathcal{M} \models \varphi$?

Este capítulo faz uma revisão dos principais conceitos necessários à formalização de problemas em alguma lógica $\mathcal{L}$ e ao uso de *model checking* dada esta lógica. A discussão é iniciada por conceitos essenciais de linguagens formais na Seção 2.1. Preliminares em lógica e *model checking* são discutidas nas Seções 2.2 e 2.3, respectivamente. A verificação de propriedades MSOL expressíveis é abordada na Seção 2.4, tomando por base modelos construídos a partir de palavras e árvores.

2.1 Linguagens formais

O termo **linguagem formal** refere-se a um conjunto de objetos construídos a partir de símbolos elementares e que são bem formados dado um conjunto de regras para sua

formação. A noção de bem formado refere-se a estar em pleno acordo com as regras de construção destes objetos. Ao conjunto de símbolos elementares dá-se o nome de **alfabeto**. O conjunto de regras que define como os objetos são gerados é um dos componentes da **gramática** sendo considerada. Neste capítulo serão abordadas as linguagens de palavras e as *tree languages*.

2.1.1 Linguagem de palavras

Uma **palavra** é uma sequência de símbolos de um alfabeto. Considere que Σ represente um alfabeto. Denota-se por Σ^* o conjunto de todas as sequências que podem ser construídas a partir dos símbolos deste alfabeto. Assim, uma **linguagem de palavras**, denotada por $\mathbf{L}$, é um subconjunto de todas as sequências possíveis de símbolos do alfabeto, i.e., $\mathbf{L} \subseteq \Sigma^\star$. Toda linguagem é gerada a partir do conjunto de regras definidas em sua gramática, apresentada na Definição 2.1.1.

Definição 2.1.1. *[Gramática] Uma* ***gramática*** *é uma quádrupla* $\mathbf{G} = \langle V, T, P, S \rangle$*, tal que*

V *é um conjunto de símbolos não terminais, também chamados de variáveis;*

T *é um conjunto de símbolos terminais;*

P *é um conjunto de regras de produção da forma* $\alpha \rightarrow \beta$*, tal que* $\alpha \in (V \cup T)^+$ *e* $\beta \in (V \cup T)^*$*;*

S *é o símbolo de início para o processo de geração das palavras.*

O conjunto mais simples de gramáticas de palavras é o das gramáticas lineares, apresentadas na Definição 2.1.2 [47], de sorte que qualquer gramática linear é uma **gramática regular** [35, 40]. As gramáticas regulares são responsáveis por gerar o conjunto mais simples de linguagens, chamadas **linguagens regulares**, amplamente utilizadas em processos de reconhecimento de *tokens* em compiladores, por exemplo.

Definição 2.1.2. *[Gramáticas Lineares] Sejam* $\mathbf{G} = \langle V, T, P, S \rangle$ *uma gramática (de palavras),* $A, B \in V$ *e* w *uma palavra de* T^**. Então,* $\mathbf{G}$ *é uma:*

(a) Gramática Linear à Direita (GLD) se todas as regras de produção são da forma

$$A \rightarrow wB \quad ou \quad A \rightarrow w$$

(b) *Gramática Linear à Esquerda (GLE) se todas as regras de produção são da forma*

$$A \rightarrow Bw \quad ou \quad A \rightarrow w$$

(c) *Gramática Linear Unitária à Direita (GLUD) se todas as regras de produção como na GLD e* $|w| \leq 1$

(d) *Gramática Linear Unitária à Esquerda (GLUE) se todas as regras de produção como na GLE e* $|w| \leq 1$

O processo de geração das palavras da linguagem, chamado de **derivação**, se dá por meio da aplicação finita de regras de produção a partir do símbolo de início da gramática de modo a obter uma cadeia de símbolos terminais. Estes símbolos terminais são exatamente os símbolos que compõem o alfabeto da linguagem. O Exemplo 2.1.1 ilustra o processo de derivação.

Exemplo 2.1.1. Considere uma gramática $\mathbf{G}_{3a} = \langle \{A, B\}, \{a, b\}, P, A \rangle$, tal que P seja o conjunto de regras de produção exibido na Tabela 2.1. A linguagem gerada por essa gramática é a linguagem das palavras que terminam com "aaa". O processo de derivação da palavra $w_{3a} = abaaa$ é apresentado na Figura 2.1. É importante ressaltar que a palavra vazia ϵ é o elemento neutro da operação de concatenação, que é a operação algébrica elementar encapsulada no processo de derivação. Portanto, para quaisquer palavras w_1 e w_2 sobre qualquer alfabeto Σ, é verdade que $\epsilon w_1 = w_1 = w_1\epsilon$ e $w_1w_2 = w_1\epsilon w_2 = \epsilon w_1 w_2 = w_1 w_2 \epsilon$.

Tabela 2.1: Regras de produção para a gramática $\mathbf{G}_{3a}$.

ID	Produção
01	$A \rightarrow Baaa$
02	$B \rightarrow Ba$
03	$B \rightarrow Bb$
04	$B \rightarrow \epsilon$

$$A \rightarrow^{01} Baaa \rightarrow^{03} Bbaaa \rightarrow^{02} Babaaa \rightarrow^{04} \epsilon abaaa = abaaa$$

Figura 2.1: Processo de derivação da palavra w_{3a} a partir de $\mathbf{G}_{3a}$

Ao passo que as gramáticas geram linguagens, o processo de *reconhecimento* delas é feito por algum dispositivo de computação. Estes dispositivos são modelos teóricos, e em sua maioria implementáveis, baseados em máquinas de estados. Estas máquinas

processam a entrada, símbolo a símbolo, e alteram seu estado interno. Este estado é a representação em alto nível do processamento sendo executado. Ao final da leitura é preciso verificar em qual estado a máquina parou. Caso ela tenha atingido um estado especial, denominado **estado de aceitação**, obtém-se a confirmação de que a entrada faz parte da linguagem que é reconhecida por esta máquina. Neste trabalho, por motivos didáticos, a atenção será voltada exclusivamente para os chamados **autômatos finitos**. Estes dispositivos podem ser de dois tipos: os **determinísticos**, apresentados na Definição 2.1.3, e os **não-determinísticos**.

Definição 2.1.3. *[Autômato Finito Determinístico] Um* ***autômato finito determinístico*** *é uma quíntupla* $A = \langle \Sigma, Q, \delta, q_0, F \rangle$*, tal que*

$\boldsymbol{\Sigma}$ *é o alfabeto;*

$\boldsymbol{Q}$ *é o conjunto de estados;*

$\boldsymbol{\delta}$ *é a função de transição da forma* $\delta : \Sigma \times Q \to Q$*;*

$\boldsymbol{q_0}$ *é o estado inicial;*

$\boldsymbol{F}$ *é o conjunto de estados de aceitação, logo* $F \subseteq Q$*.*

Autômatos finitos não-determinísticos são muito similares aos *determinísticos*, com exceção de sua função de transição e sua noção de aceitação de uma palavra. Enquanto nos autômatos determinísticos, a função de transição garante um comportamento previsível, i.e., dados um símbolo e um estado, a transição sempre será para um único novo estado, o mesmo não ocorre nos autômatos não-determinísticos. Neste caso, a função de transição tem forma $\delta : \Sigma \times Q \to 2^Q$, o que revela a capacidade da máquina alcançar mais de um estado simultaneamente. Com essa característica, a noção de aceitação passa a ser a de, ao final da leitura, a máquina atingir um estado de aceitação por algum caminho. A Figura 2.2 ilustra como são representadas graficamente transições determinísticas e não-determinísticas.

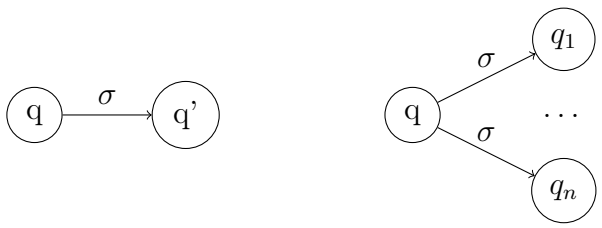


Figura 2.2: Diferença entre transições determinísticas e não-determinísticas

As diferenças técnicas entre os autômatos não-determinísticos e determinísticos não impactam na expressividade destes dispositivos. Ambos são equivalentes e reconhecem as linguagens regulares. É um fato que, em certos cenários, a construção de um autômato não-determinístico é mais imediata para seres humanos, porém o não-determinismo é um fenômeno não escalável e não pode ser implementado nos atuais modelos de computação. Portanto, usualmente, dado um autômato não-determinístico, aplica-se sobre ele um algoritmo que o torne determinístico.

Observe nas Figuras 2.3 e 2.4 exemplos de autômatos que reconhecem a linguagem gerada pela gramática $\mathbf{G}_{3a}$. Os círculos representam os estados do autômato. O estado com uma seta vinda de lugar nenhum é o estado inicial. Estados marcados com círculo duplo são estados de aceitação. As setas entre os círculos representam as transições de um estado a outro ao ler o símbolo que rotula a seta.

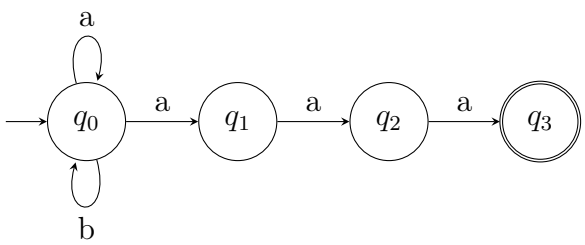


Figura 2.3: Autômato não-determinístico para a linguagem de $\mathbf{G}_{3a}$

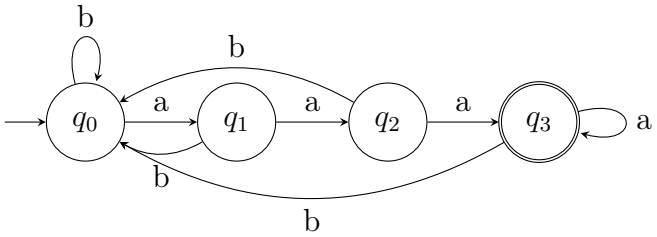


Figura 2.4: Autômato determinístico para a linguagem de $\mathbf{G}_{3a}$

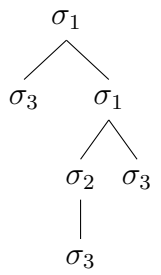
2.1.2 *Tree languages*

Uma ***tree language*** é uma linguagem cujos objetos são árvores. Em linguagens formais, as **árvores** são expressões especiais construídas a partir de um alfabeto ranqueado. Um **alfabeto ranqueado** $\Sigma_{rank} = \langle \Sigma, rank \rangle$ é composto por um conjunto de símbolos Σ e uma função $rank : \Sigma \to \mathbb{Z}^+$, que associa cada símbolo do alfabeto a um número inteiro não negativo. A construção indutiva de uma árvore é formalizada na Definição 2.1.4.

Definição 2.1.4. *[Árvore] Seja $\Sigma_{rank} = \langle \Sigma, rank \rangle$ um alfabeto rankeado. Uma **árvore**, denotada por t, é definida recursivamente como segue:*

- $t = \sigma$ *se* $\sigma \in \Sigma$ *e* $rank(\sigma) = 0$.
- $t = \sigma[t, \ldots, t]$ *se* $\sigma \in \Sigma$, $rank(\sigma) = r$, $r \geq 1$ *e* $t, \ldots, t$ *são* r *árvores quaisquer.*

Apesar de denotada como uma sequência de símbolos, toda árvore possui uma noção hierárquica que é determinada pelo *rank* do símbolo. Símbolos de *rank* igual a 0, denominados **folhas**, são a base dessa hierarquia. Considere uma árvore $t = \sigma[t_1, \ldots, t_r]$, na qual $t_1, \ldots, t_r$ representam outras r árvores. Dado que σ tem *rank* $r > 0$, dize-se que σ é *pai* das árvores $t_1, \ldots, t_r$ e que essas árvores são *filhos* de σ. Como σ não possui um pai, ele é denominado *raiz* da árvore. A Figura 2.5 mostra a representação hierárquica de uma árvore, na qual os símbolos σ_3 rotulam as folhas da árvore e a raiz é rotulada por σ_1.



$rank(\sigma_1) = 2$
$rank(\sigma_2) = 1$
$rank(\sigma_3) = 0$

Figura 2.5: Visualização hierárquica da árvore $\sigma_1[\sigma_3, \sigma_1[\sigma_2[\sigma_3], \sigma_3]]$

Uma ***tree grammar*** $\mathbf{G}_{tree}$ possui os mesmos componentes de uma gramática apresentada na Definição 2.1.1, diferindo essencialmente nos objetos gerados. *Tree grammars* possuem regras de produção da forma $\alpha \rightarrow \beta$, tal que α e β são árvores e α contém pelo menos um símbolo não terminal. Uma ***regular tree grammar*** (RTG) é uma *tree grammar* na qual o elemento α das regras de produção sempre é apenas um símbolo não terminal e todo símbolo não terminal é uma folha. Um exemplo de RTG é apresentado na Tabela 2.2.

As *regular tree languages* são as linguagens geradas a partir de RTG e são reconhecidas por *bottom-up finite tree automaton* (FTA_B) ou *top-down finite tree automaton* (FTA_T). Estes dispositivos realizam o processamento das árvores das folhas para a raiz e da raiz para as folhas respectivamente. Apesar de reconhecerem linguagens de uma mesma classe, os FTA_T são capazes de reconhecer um conjunto menor de linguagens que os FTA_B. Portanto, este trabalho utiliza apenas os FTA_B, apresentados formalmente na Definição 2.1.5. Ao longo do texto, qualquer menção aos FTA_B será feita apenas como *finite tree automaton* (FTA).

Definição 2.1.5. /Bottom-up Finite Tree Automaton/ *Seja* Σ_{rank} *um alfabeto ranqueado. Um* **bottom-up finite tree automaton** *é uma quádrupla* $\mathcal{A} = \langle \Sigma_{rank}, Q, Q_F, \delta \rangle$ *tal que*

$\boldsymbol{\Sigma_{rank}}$ *é o alfabeto*

$\boldsymbol{Q}$ *é o conjunto de estados*

$\boldsymbol{\delta}$ *é a função de transição,* $\delta : \Sigma_{rank} \times (A^Q \cup \{\emptyset\}) \rightarrow Q$*, onde* A^Q *denota todos os possíveis arranjos tomados dentre os elementos de* Q

$\boldsymbol{F}$ *é o conjunto de estados de aceitação*

Para ilustrar melhor os conceitos de árvore, *tree grammar* e *tree automaton*, considere o Exemplo 2.1.2 que explora uma representação da Álgebra Booleana por meio de uma *tree language*. A Álgebra Booleana é composta pelas constantes $true$ e $false$ e os operadores *and*, *or* e *not*. Pode-se denotar estes elementos por meio de um alfabeto ranqueado $\Sigma_{bool} = \langle\{or, and, not, true, false\},$
$\{(true, 0), (false, 0), (not, 1), (or, 2), (and, 2)\}\rangle$, de modo que o *rank* de cada símbolo seja a aridade desse operador.

Exemplo 2.1.2. Suponha que deseja-se construir um autômato que reconheça expressões da Álgebra Booleana cujo resultado das operações é $true$. Para isto, basta construir um autômato $A_{bool} = \langle\Sigma_{bool}, Q, Q_f, \delta\rangle$ com os seguintes componentes:

$Q = \{Q_{true}, Q_{false}\}$, representando os estados onde o valor da expressão é $true$ e $false$ respectivamente

$F = \{Q_{true}\}$, como o estado de aceitação desse autômato

$\delta = \{false \rightarrow Q_{false},\ true \rightarrow Q_{true},\ not(Q_{false}) \rightarrow Q_{true},\ not(Q_{true}) \rightarrow Q_{false},$
$and(Q_{false}, Q_{false}) \rightarrow Q_{false},\ and(Q_{false}, Q_{true}) \rightarrow Q_{false},$
$and(Q_{true}, Q_{false}) \rightarrow Q_{false},\ and(Q_{true}, Q_{true}) \rightarrow Q_{true},$
$or(Q_{false}, Q_{false}) \rightarrow Q_{false}, or(Q_{false}, Q_{true}) \rightarrow Q_{true},$
$or(Q_{true}, Q_{false}) \rightarrow Q_{true},\ or(Q_{true}, Q_{true}) \rightarrow Q_{true}\}$, codifica em transições os resultados das aplicações de *and*, *or* e *not* sobre as constantes $true$ e $false$.

Tome a árvore $t_{bool} = or[and[true, false], not[false]]$. É trivial verificar que o resultado dessa expressão é $true$, portanto uma árvore da *tree language* apresentada nos parágrafos anteriores. Para verificar formalmente a pertinência da árvore à linguagem apresentada, pode-se utilizar o processo de derivação em $\mathbf{G}_{bool}$ ou realizar o processamento com A_{bool}. A Figura 2.6 apresenta uma execução do autômato ao processar a

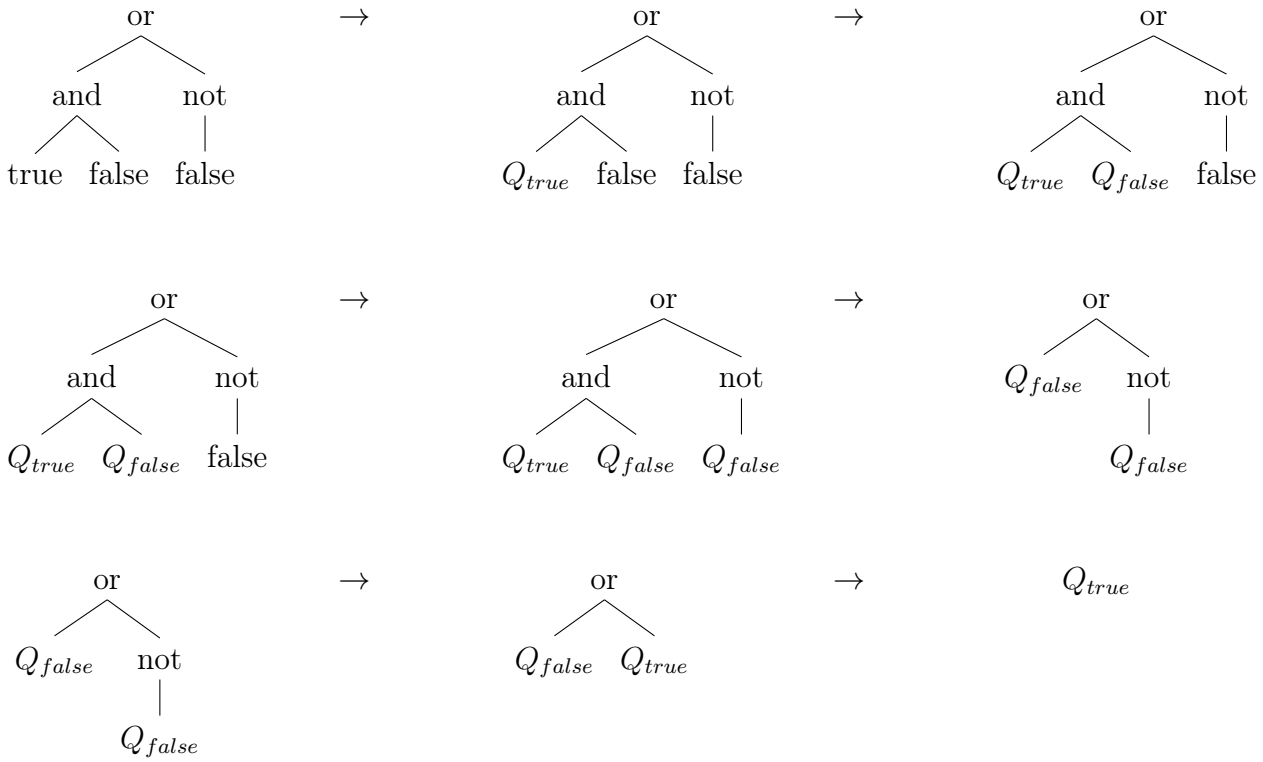


Figura 2.6: Exemplo da execução de A_{bool} tomando t_{bool} como entrada

árvore t_{bool}. Note que a imagem exibe a árvore de modo hierárquico apenas por motivos didáticos.

Esta mesma linguagem pode ser descrita por uma RTG. Considere a seguinte gramática $\mathbf{G}_{bool} = \langle \{T, F\}, \{true, false, and, or, not\}, T, P \rangle$ de modo que o conjunto P contenha as regras de produção apresentadas na Tabela 2.2. As produções, assim como as transições do autômato, representam os cálculos dos operadores sobre as constantes da álgebra.

Tabela 2.2: Regras de produção para a gramática $\mathbf{G}_{bool}$

ID	**Produção**	**ID**	**Produção**
01	$T \to true$	07	$F \to false$
02	$T \to and[T, T]$	08	$F \to or[F, F]$
03	$T \to not[F]$	09	$F \to not[T]$
04	$T \to or[T, F]$	10	$F \to and[T, F]$
05	$T \to or[F, T]$	11	$F \to and[F, T]$
06	$T \to or[T, T]$	12	$F \to and[F, F]$

A Figura 2.7 apresenta o resultados da uma derivação da árvore t_{bool} na gramática $\mathbf{G}_{bool}$. Note que a sequência de derivação usando as produções $05, 03, 10, 01, 07$ e 07 não é única.

$$S \rightarrow^{05} or[N, S] \rightarrow^{03} or[N, not[N]] \rightarrow^{10} or[and[S, N], not[N]]$$

$$or[and[S, N], not[N]] \rightarrow^{01} or[and[true, N], not[N]] \rightarrow^{07} or[and[true, false], not[N]]$$

$$or[and[true, false], not[N]] \rightarrow^{07} or[and[true, false], not[false]]$$

Figura 2.7: Processo de derivação da árvore t_{bool} a partir de $\mathbf{G}_{bool}$

2.2 Lógica

Uma **lógica** pode ser interpretada como uma linguagem formal equipada com um sistema dedutivo. Este sistema dedutivo é composto por axiomas e regras de inferência que codificam o raciocínio de uma maneira matemática.

Em lógica, a linguagem formal utilizada para a construção de fórmulas pode ser dividida em símbolos constantes e variáveis. As constantes dessa linguagem podem ainda ser dividas em outros dois grupos, os *símbolos lógicos* e os *símbolos não-lógicos*. Os **símbolos lógicos** são próprios da linguagem, possuindo assim uma semântica pré definida que não depende do universo de discurso. Por outro lado, os **símbolos não-lógicos** são aqueles acrescentados à linguagem para a construção e sua semântica dependente é dada dentro do universo de discurso.

2.2.1 Lógica Proposicional

A **Lógica Proposicional** é uma lógica que possui como elemento mínimo as chamadas **proposições**, que são enunciados verificáveis, como "Hoje está chovendo" ou "Acordei às 9h", por exemplo. Proposições podem ser avaliadas como verdadeiras ou falsas. Dessa forma, parte do conhecimento codificado na Lógica Proposicional consiste na manipulação do **valor verdade** dado às proposições utilizadas para construir suas fórmulas. A Gramática 2.1 mostra a forma de se construir fórmulas proposicionais, considerando que o símbolo p nesta gramática representa uma possível proposição dado um conjunto enumerável P de proposições previamente definidas.

$$\begin{array}{lcl} \langle formula \rangle & ::= & \text{true} \mid \text{false} \mid \text{p} \mid \langle formula \rangle \wedge \langle formula \rangle \mid \langle formula \rangle \vee \langle formula \rangle \mid \neg \langle formula \rangle \\ & \mid & \langle formula \rangle \rightarrow \langle formula \rangle \end{array}$$

Gramática 2.1: Sintaxe da Lógica Proposicional

Os conectivos das fórmulas da lógica proposicional são '$\wedge$' (conjunção), '$\vee$' (disjunção), '$\neg$' (negação) e '$\rightarrow$' (implicação). A semântica desses símbolos é expressa por meio de

tabelas-verdade, exibidas nas Tabelas 2.3 a 2.6, que são tabelas que apresentam a avaliação de cada um dos conectivos da lógica mediante a valoração das fórmulas por eles conectadas. Note que o conectivo $\neg$ é o único conectivo unário da lógica proposicional, ou seja, se aplica a apenas um argumento.

Tabela 2.3: Tabela-verdade para conjunção

φ_1	φ_2	$\varphi_1 \wedge \varphi_2$
$true$	$true$	$true$
$true$	$false$	$false$
$false$	$true$	$false$
$false$	$false$	$false$

Tabela 2.4: Tabela-verdade para disjunção

φ_1	φ_2	$\varphi_1 \vee \varphi_2$
$true$	$true$	$true$
$true$	$false$	$true$
$false$	$true$	$true$
$false$	$false$	$false$

Tabela 2.5: Tabela-verdade para negação

φ_1	$\neg\varphi_1$
$true$	$false$
$false$	$true$

Tabela 2.6: Tabela-verdade para implicação

φ_1	φ_2	$\varphi_1 \rightarrow \varphi_2$
$true$	$true$	$true$
$true$	$false$	$false$
$false$	$true$	$true$
$false$	$false$	$true$

De maneira simples, pode-se estudar a semântica da Lógica Proposicional com o auxílio de tabelas-verdade. Dada uma fórmula em Lógica Proposicional, basta substituir as proposições por um valor verdade e, com o auxílio das tabelas-verdade para os conectivos, aplicar os resultados até obter apenas um valor verdade, o qual representa o resultado da avaliação desta fórmula dada a **interpretação** para suas proposições. Uma fórmula é dita **satisfatível** quando uma interpretação para suas proposições resultam em uma

avaliação verdadeira. Em termos de tabela-verdade, diz-se que uma fórmula é satisfatível quando uma linha de sua tabela tem como resultado *true*. Observe a Tabela 2.7 para um exemplo de tabela verdade para a fórmula $p_1 \wedge (p_2 \vee \neg p_1)$. Note que na primeira linha da tabela o resultado da fórmula é *true*. Portanto, a fórmula $p_1 \wedge (p_2 \vee \neg p_1)$ é satisfatível quando p_1 e p_2 são proposições avaliadas como verdadeiras.

Tabela 2.7: Tabela-verdade para $p_1 \wedge (p_2 \vee \neg p_1)$

$\boldsymbol{p_1}$	$\boldsymbol{p_2}$	$\boldsymbol{\neg p_1}$	$\boldsymbol{p_2 \vee \neg p_1}$	$\boldsymbol{p_1 \wedge (p_2 \vee \neg p_1)}$
$true$	$true$	$false$	$true$	$true$
$true$	$false$	$false$	$false$	$false$
$false$	$true$	$true$	$true$	$false$
$false$	$false$	$true$	$true$	$false$

Uma limitação inerente da lógica proposicional é a incapacidade de expressar relações entre os elementos envolvidos nas proposições. Tome como exemplo os seguintes enunciados "$x < y$", "y < z" e "x < z". Pela aritmética clássica, a relação óbvia entre esses enunciados é a de que a veracidade dos dois primeiros, implica na veracidade do terceiro. Representar esse tipo de relação em lógica proposicional é impossível, dado que cada enunciado seria representado por um símbolo proposicional diferente. Assim, a relação de consequência lógica seria dada por $p, q \vDash r$, considerando p, q e r representando "$x < y$", "y < z" e "x < z", nesta ordem. Note que é impossível garantir a veracidade de r dada a veracidade de p e q.

2.2.2 Lógica de Primeira Ordem

A **Lógica de Primeira Ordem** (FOL) é extensão da lógica proposicional que permite lidar com relações e funções sobre elementos antes envolvidos nas proposições. Além disso, FOL permite ainda realizar quantificações sobre os elementos do universo de discurso. Toda relação ou função possui uma **aridade**, que determina o número de elementos envolvidos. Esses elementos podem ser representados por meio de constantes, que são interpretadas como funções de aridade 0, ou de variáveis, que permitem que qualquer elemento do domínio seja associado a elas. A construção de fórmulas em FOL é apresentada na Gramática 2.2, onde P representa um predicado/relação, F representa uma função, $\bar{x}$ representa uma tupla de variáveis (de entrada) e y representa uma variável (de retorno).

Sintaticamente, fórmulas em FOL são construídas a partir de componentes descritos em uma **assinatura**. Uma assinatura é um conjunto $sig = \langle S, arity \rangle$, onde S é um

$$\begin{array}{rcl} \langle formula \rangle & ::= & \text{true} \mid \text{false} \mid \text{P}(\bar{x}) \mid \text{F}(\bar{x},y) \mid \langle formula \rangle \wedge \langle formula \rangle \mid \langle formula \rangle \vee \langle formula \rangle \\ & \mid & \neg\langle formula \rangle \mid \langle formula \rangle \rightarrow \langle formula \rangle \mid \exists \text{x}.\langle formula \rangle \mid \forall \text{x}.\langle\ formula \rangle \end{array}$$

Gramática 2.2: Sintaxe da Lógica de Primeira Ordem

conjunto de símbolos e $arity : S \rightarrow \mathbb{N}$ é a função de aridade. Os símbolos de S podem ser divididos entre símbolos de relação e funções, que possuem aridade maior que 0, e os símbolos constantes, que possuem aridade igual a 0.

A semântica de FOL é denotada por meio de sig-estruturas, também chamadas de sig-modelos, onde sig é uma assinatura. Quando a assinatura sig é clara, pode-se falar unicamente em estruturas ou modelos. Um sig-modelo para FOL, denotado por $\mathcal{M} = \langle Dom(\mathcal{M}), I(sig)\rangle$, é um objeto matemático composto por um universo de discurso, denotado por $Dom(\mathcal{M})$, e um conjunto de relações sobre esse universo, denotado por $I(sig)$. Cada relação em $I(sig)$ representa os conjuntos de tuplas válidas de elementos do universo para cada símbolo contido na assinatura sig. As constantes são os elementos de sig utilizados para acessar unicamente cada um dos elementos do domínio $Dom(\mathcal{M})$. Símbolos para função que possuem aridade r são associados a relações $Dom(\mathcal{M})^{r+1}$, nas quais o último elemento de cada tupla é concebido como o retorno da função.

Quantificar sobre uma variável significa, de forma ingênua, iterar sobre o domínio do modelo avaliando os resultados da substituição da variável quantificada por cada um dos elementos do domínio. Há dois tipos de quantificação em FOL, a existencial e a universal. A quantificação existencial ocorre com sucesso quando ao menos um elemento do domínio satisfaz todos os predicados nos quais a variável está envolvida. Já a quantificação universal é bem sucedida se, e somente, nenhum elemento do domínio ou todos satisfazem os predicados e funções nos quais a variável quantificada está envolvida.

A **relação de satisfação** para a Lógica de Primeira Ordem, apresentada neste trabalho, não considera casos onde as fórmulas envolvem as chamadas **variáveis livres**, que são aquelas que não estão ligadas a um quantificador. As notações $\{\bar{x}/\bar{i}\}$ e $\{x/i\}$ denotam a substituição das variáveis da tupla $\bar{x}$ pelos elementos da tupla $\bar{i}$, na ordem em que eles aparecem, e a substituição da variável x pelo elemento i, respectivamente. Considere que as notações $I(P)_{\mathcal{M}}$ e $I(F)_{\mathcal{M}}$ representam as relações de elementos do domínio de um modelo $\mathcal{M}$ associadas aos símbolos de predicado P e de função F de uma assinatura sig, respectivamente. Assim, a relação de satisfação para FOL é dada por:

- $\mathcal{M} \models true$

- $\mathcal{M} \not\models false$
- $\mathcal{M}$,$\{\bar{x}/\bar{i}\} \models P(\bar{x})$ se, e somente se, $|\bar{x}| = |\bar{i}|$ e $\bar{i} \in I(P)_{\mathcal{M}}$
- $\mathcal{M}$,$\{\bar{x}/\bar{i}\} \not\models P(\bar{x})$ se, e somente se, $|\bar{x}| \neq |\bar{i}|$ ou $\bar{i} \notin I(P)_{\mathcal{M}}$
- $\mathcal{M}$,$\{\bar{x}/\bar{i}\} \models F(\bar{x}, y)$ se, e somente se, $|\bar{x}| = |\bar{i}|$ e $(\bar{i}, i') \in I(F)_{\mathcal{M}}$, para algum i'
- $\mathcal{M}$,$\{\bar{x}/\bar{i}\} \not\models F(\bar{x}, y)$ se, e somente se, $|\bar{x}| \neq |\bar{i}|$ e $(\bar{i}, i') \notin I(F)_{\mathcal{M}}$, para qualquer i'
- $\mathcal{M} \models \neg\varphi$ se, e somente se, $\mathcal{M} \not\models \varphi$
- $\mathcal{M} \models \varphi \wedge \psi$ se, e somente se, $\mathcal{M} \models \varphi$ e $\mathcal{M} \models \psi$
- $\mathcal{M} \models \varphi \vee \psi$ se, e somente se, $\mathcal{M} \models \varphi$ ou $\mathcal{M} \models \psi$
- $\mathcal{M} \models \varphi \rightarrow \psi$ se, e somente se, $\mathcal{M} \not\models \varphi$ ou $\mathcal{M} \models \psi$
- $\mathcal{M} \models \exists x.\varphi$ se, e somente se, para algum $i \in Dom(\mathcal{M})$, $\mathcal{M}$,$\{x/i\} \models \varphi$
- $\mathcal{M} \models \forall x.\varphi$ se, e somente se, para todo $i \in Dom(\mathcal{M})$, $\mathcal{M}$,$\{x/i\} \models \varphi$

2.2.3 Lógica Monádica de Segunda Ordem

A **Lógica Monádica de Segunda Ordem** (MSOL) é uma extensão da Lógica de Primeira Ordem. Nesta lógica, além da quantificação sobre elementos do domínio do modelo, é possível quantificar sobre predicados monádicos, ou seja, predicados de aridade um. Predicados monádicos, quando interpretados sobre um universo de discurso, são associados a relações unárias no domínio. Isso quer dizer que todo predicado monádico representa, na verdade, um subconjunto de elementos do domínio. A sintaxe de MSOL é apresentada na Gramática 2.3.

$$
\begin{array}{lcl}
\langle formula \rangle & ::= & \text{true} \mid \text{false} \mid \text{P}(\bar{x}) \mid \text{F}(\bar{x},y) \mid \text{X(x)} \mid \langle formula \rangle \wedge \langle formula \rangle \\
& \mid & \langle formula \rangle \vee \langle formula \rangle \mid \neg\langle formula \rangle \mid \langle formula \rangle \rightarrow \langle formula \rangle \\
& \mid & \exists \text{x}.\langle formula \rangle \mid \forall \text{x}.\langle\ formula \rangle \mid \exists \text{X}.\langle formula \rangle \mid \forall \text{X}.\langle formula \rangle
\end{array}
$$

Gramática 2.3: Sintaxe da Lógica Monádica de Segunda Ordem

Assim como acontece com FOL, a semântica de MSOL também é definida por meio de modelos. Dado que MSOL permite quantificação sobre conjuntos de elementos, a notação $\{\bar{X}/\bar{I}\}$ denotará a substituição da tupla de variáveis para conjuntos $\bar{X}$ pelos conjuntos da tupla $\bar{I}$ na ordem em que aparecem. A notação $\{X/I\}$ denota a substituição da variável

X pelo conjunto de elementos denotado por I. Portanto, a **relação de satisfação** para MSOL é definida por:

- $\mathcal{M} \models true$
- $\mathcal{M} \not\models false$
- $\mathcal{M}$,$\{\bar{x}/\bar{i}\} \models P(\bar{x})$ se, e somente se, $|\bar{x}| = |\bar{i}|$ e $\bar{i} \in I(P)_{\mathcal{M}}$
- $\mathcal{M}$,$\{\bar{x}/\bar{i}\} \not\models P(\bar{x})$ se, e somente se, $|\bar{x}| \neq |\bar{i}|$ ou $\bar{i} \notin I(P)_{\mathcal{M}}$
- $\mathcal{M}$,$\{\bar{x}/\bar{i}\} \models F(\bar{x}, y)$ se, e somente se, $|\bar{x}| = |\bar{i}|$ e $(\bar{i}, i') \in I(F)_{\mathcal{M}}$, para algum i'
- $\mathcal{M}$,$\{\bar{x}/\bar{i}\} \not\models F(\bar{x}, y)$ se, e somente se, $|\bar{x}| \neq |\bar{i}|$ e $(\bar{i}, i') \notin I(F)_{\mathcal{M}}$, para qualquer i'
- $\mathcal{M}$,$\{x/i,X/I\} \models X(x)$ se, e somente se, $i \in Dom(\mathcal{M})$, $I \subseteq Dom(\mathcal{M})$ e $i \in I$
- $\mathcal{M}$,$\{x/i,X/I\} \not\models X(x)$ se, e somente se, $i \notin Dom(\mathcal{M})$, $I \not\subseteq Dom(\mathcal{M})$ ou $i \notin I$
- $\mathcal{M} \models \neg\varphi$ se, e somente se, $\mathcal{M} \not\models \varphi$
- $\mathcal{M} \models \varphi \wedge \psi$ se, e somente se, $\mathcal{M} \models \varphi$ e $\mathcal{M} \models \psi$
- $\mathcal{M} \models \varphi \vee \psi$ se, e somente se, $\mathcal{M} \models \varphi$ ou $\mathcal{M} \models \psi$
- $\mathcal{M} \models \varphi \rightarrow \psi$ se, e somente se, $\mathcal{M} \not\models \varphi$ ou $\mathcal{M} \models \psi$
- $\mathcal{M} \models \exists x.\varphi$ se, e somente se, $\mathcal{M}$,$\{x/i\} \models \varphi$ para algum $i \in Dom(\mathcal{M})$
- $\mathcal{M} \models \exists X.\varphi$ se, e somente se, $\mathcal{M}$,$\{X/I\} \models \varphi$ para algum $I \subseteq Dom(\mathcal{M})$
- $\mathcal{M} \models \forall x.\varphi$ se, e somente se, $\mathcal{M}$,$\{x/i\} \models \varphi$ para todo $i \in Dom(\mathcal{M})$
- $\mathcal{M} \models \forall X.\varphi$ se, e somente se, $\mathcal{M}$,$\{X/I\} \models \varphi$ para todo $I \subseteq Dom(\mathcal{M})$

2.3 *Model Checking*

Garantir que um sistema comporte-se da maneira como foi idealizado, por vezes, é uma tarefa não trivial. Sistemas simples e que apresentam baixo risco financeiro e humano podem ser validados por meio de testes. Caso algum problema de especificação seja detectado, este poderá ser corrigido sem causar impacto negativo para clientes e desenvolvedores. Todavia, os chamados *sistemas críticos*, ou seja, aqueles que envolvem alto custo financeiro e vidas humanas, não podem ser submetidos a testes sem a possibilidade

da ocorrência de algum desastre. Um exemplo foram as mortes de seis pacientes portadores de câncer entre 1985 e 1987 graças a um erro nas máquinas Therac-25, as quais administraram doses inadequadas de radiação nos pacientes [3, 46].

A necessidade de formas seguras e viáveis para a verificação de sistemas críticos inspirou a criação de uma técnica conhecida por *model checking*. Proposta por Edmund Clarke, o ***model checking*** [14, 3, 15] é uma técnica que consiste na verificação automática de fórmulas em uma lógica de interesse sobre um modelo. Para fins de verificação de sistemas, o modelo é construído a partir da especificação do sistema e as fórmulas a serem verificadas codificam propriedades desejáveis no sistema. Quando uma propriedade não consegue ser validada, o resultado da verificação é um contra-exemplo para aquela fórmula. Dessa maneira, pode-se verificar sistemas críticos de forma mais barata e segura. É importante ressaltar que a qualidade da verificação é diretamente proporcional à qualidade do modelo construído. Portanto, erros e displicências na etapa de construção do modelo matemático podem resultar em verificações imprecisas.

Para ilustrar exemplos de verificação em algumas lógicas, tome o exemplo da lógica proposicional. Modelos em lógica proposicional são vistos como funções da forma $val : \mathrm{P} \rightarrow \{true, false\}$, que associam cada uma das proposições do conjunto P em um valor verdade. Assim, o *model checking* para lógica proposicional consiste em tomar uma associação de valores verdade para cada proposição, substituir na fórmula as proposições por seus valores associados e realizar o processo de valoração da fórmula, gradativamente, por meio das identidades estabelecidas nas tabelas-verdade dos conectivos da lógica.

Courcelle apresenta uma nova forma de se utilizar o *model checking* ao aplicá-lo em problemas de decisão em grafos. Tome por base um problema de decisão em grafos denotado por

Decide(G,P)
Entrada: Um grafo G e uma propriedade P.
Questão: G possui P?

A proposta de Courcelle, de um ponto de vista superficial, consiste em construir um modelo a partir de G e formalizar P por meio de uma fórmula lógica. A escolha da lógica a ser utilizada para representar a propriedade P depende dos tipos de elementos a serem representados. Por exemplo, não há como quantificar sobre conjuntos em FOL, portanto, propriedades como 3-coloração ou bipartição devem ser representadas utilizando MSOL. Porém, o trabalho de Vardi [55] aponta que o *model checking* de MSOL é PSPACE-

completo em qualquer classe de grafos não trivial. Assim, a proposta inicial é relaxar o conceito de complexidade em perspectiva. Vardi aponta a existência de três tipos de complexidade, a saber

Complexidade combinada que se refere à verificação em casos onde modelo e fórmula são dados como entrada;

Complexidade de dados que se refere à verificação considerando apenas o tamanho do modelo, dado que a fórmula a ser verificada é fixa;

Complexidade de expressão que se refere à verificação considerando apenas o tamanho da fórmula, dado que o modelo no qual a fórmula será verificada é fixo.

Dizer que um modelo ou uma fórmula é fixo significa que eles farão parte do projeto do algoritmo. Assim, o resultado da construção é um algoritmo que ou verifica se diferentes fórmulas são satisfeitas em um modelo ou verifica quais modelos satisfazem uma fórmula. Ao fixar um modelo ou fórmula, estudos apontam diferentes resultados em complexidade. O Teorema 2.3.1 é derivado dos estudos de Büchi [12] e Tatcher e Wright [53], estabelecendo um resultado parametrizado que se baseia no conceito de complexidade de dados. Em vez de fixar a fórmula, a proposta é utilizar o tamanho dela como parâmetro no algoritmo.

Teorema 2.3.1 (*Model checking* parametrizado para MSOL)**.** Model Checking *para Lógica Monádica de Segunda Ordem é solucionável em tempo*

$$f(\kappa) \times n$$

para alguma função computável f nas classes de estruturas construídas a partir de palavras e árvores, onde κ representa o parâmetro associado ao tamanho da fórmula de entrada.

Tendo por base o conhecimento do Teorema 2.3.1, Courcelle propõe simular o *model checking* para grafos por meio de *model checking* para árvores. Assim, definindo uma codificação para os grafos de entrada do problema, Courcelle mostra que é possível projetar algoritmos **FPT** com complexidade $f(twd, |\varphi|) \times n$, onde twd é a *treewidth* do grafo e $|\varphi|$ é o tamanho da fórmula φ a ser verificada. Todavia esse resultado é restrito a dois fragmentos de MSOL, os chamados Lógica Monádica de Segunda Ordem de tipo 1 (MSOL_1) e Lógica Monádica de Segunda Ordem de tipo 2 (MSOL_2).

Em MSOL_1 é possível quantificar sobre vértices e conjuntos de vértices. Logo, os modelos construídos a partir dos grafos são feitos de maneira imediata. O conjunto de

vértices se torna o domínio do modelo e as arestas se tornam um predicado binário. Por outro lado, se for necessário quantificar sobre arestas ou conjuntos de arestas, é indicado utilizar MSOL_2. Como as arestas são um predicado binário, elas não podem ser quantificadas em MSOL, a menos que componham o domínio do modelo. Dessa forma, gera-se o grafo de incidência do grafo de entrada e a partir dele o modelo será construído como no caso para MSOL_1. Deve-se ressaltar que os fragmentos MSOL_1 e MSOL_2 não são disjuntos. Tudo que pode ser feito com MSOL_1 pode ser feito com MSOL_2. Logo, usualmente considera-se MSOL_2 como uma extensão de MSOL_1. A Seção 2.4 discutirá a verificação de propriedades MSOL, porém nos Capítulos 4 e 5 a discussão será restrita exclusivamente a MSOL_2.

2.4 Verificação de Propriedades MSOL-expressíveis

A proposta de verificação de propriedades em MSOL baseada em autômatos começa com o trabalho de Büchi [12] ao estudar as relações de MSOL com autômatos finitos. As ideias de Büchi são então estendidas por Thatcher e Wright [53], que mostram uma relação similar entre MSOL e *tree automata*. Ambas as propostas são descritas de forma clara por Thomas em um *survey* de 1997 [54].

Essa relação entre MSOL e autômatos finitos/*tree automata* permite que se realize *model checking* de fórmulas em modelos por meio de palavras/árvores que representem os modelos a serem verificados. As palavras/árvores em questão são construídas por meio de gramáticas regulares, portanto são linguagens que possuem propriedades de fecho importantes para o processo de construção dos autômatos.

Para construir um autômato a partir de uma fórmula em MSOL, é necessário considerar alguns pontos. O primeiro deles é que o modelo sobre o qual será feita a verificação não contém nenhum símbolo funcional, ou seja, é modelo relacional. O segundo ponto se refere à assinatura do modelo. Ela deve ser bem conhecida, bem como a semântica de cada um dos predicados apresentados por ela. O terceiro ponto é que todas as quantificações sobre elementos deverão ser substituídas por quantificações sobre conjuntos unitários. Por fim, a fórmula em questão deve estar na **forma normal prenex**, ou seja, todos os quantificadores presentes na fórmula estão posicionados no início dela. Se os quantificadores de uma fórmula em forma normal prenex forem ignorados, a fórmula restante contém apenas variáveis livres. O algoritmo para colocar uma fórmula na forma normal prenex é dado pelos seguintes passos:

1. Eliminar todas as ocorrências de implicação e bi-implicação. Use as identidades:

$$\varphi \rightarrow \psi = \neg\varphi \vee \psi$$

$$\varphi \leftrightarrow \psi = (\neg\varphi \vee \psi) \wedge (\varphi \vee \neg\psi)$$

$$\varphi \leftrightarrow \psi = (\varphi \wedge \psi) \vee (\neg\varphi \wedge \neg\psi)$$

2. Mover todas as negações para o interior da fórmula. Use as identidades:

$$\neg\neg\varphi = \varphi$$

$$\neg\exists X.\varphi(X) = \forall X.\neg\varphi(X)$$

$$\neg\forall X.\varphi(X) = \exists X.\neg\varphi(X)$$

3. Renomear variáveis que apareçam ligadas a mais de um quantificador em diferentes pontos da fórmula. Por exemplo, considere a fórmula $\exists X.\varphi(X) \wedge \forall X.\psi(X)$. Apesar de um mesmo nome, a variável X está inserida em dois contextos diferentes. A solução é renomear uma delas.

4. Mover os quantificadores para frente da fórmula. Use as identidades:

$$\varphi \wedge \exists X.\psi(X) = \exists X.(\varphi \wedge \psi(X))$$

$$\varphi \wedge \forall X.\psi(X) = \forall X.(\varphi \wedge \psi(X))$$

$$\varphi \vee \exists X.\psi(X) = \exists X.(\varphi \vee \psi(X))$$

$$\varphi \vee \forall X.\psi(X) = \forall X.(\varphi \vee \psi(X))$$

quando X não ocorre em φ

$$Q'X.\varphi(X) \wedge Q''Y.\psi(Y) = Q'X.Q''Y.(\varphi(X) \wedge \psi(Y))$$

$$Q'X.\varphi(X) \vee Q''Y.\psi(Y) = Q'X.Q''Y.(\varphi(X) \vee \psi(Y))$$

para $Q', Q'' \in \{\exists, \forall\}$.

A relação de satisfação para MSOL em fórmulas com variáveis livres é definida mediante a atribuição de elementos e conjuntos de elementos às variáveis para elementos e para conjuntos. Uma simplificação existente na literatura [54] consiste em substituir MSOL pela chamada MSOL_0, que seria o fragmento de MSOL onde não existem quantificações sobre elementos. Assim, quantificações sobre elementos serão substituídas por

quantificações sobre conjuntos unitários. Os conjuntos atribuídos, no caso, consistem em posições da palavra/árvore, dado que esse é o domínio dos modelos construídos. Cada conjunto será codificado por meio de uma palavra/árvore de mesmo tamanho da palavra/árvore que codifica o modelo. Esse conjunto codificado será composto de 0's e 1's, onde 0 significa que aquela posição não pertence àquele conjunto e 1 significa sua pertinência. Agora, a palavra/árvore que codifica o modelo é lida considerando também uma *string* binária de tamanho c, onde c é o número de variáveis para conjuntos existentes na fórmula. Isso cria a noção do chamado **alfabeto estendido**, definido matematicamente como $\Sigma^X = \Sigma \times \{0,1\}^c$, onde Σ é um alfabeto. Uma visualização melhor destes conceitos é apresentada no Exemplo 2.4.1.

Exemplo 2.4.1. Suponha um problema onde a palavra de entrada é $w = aabba$ de modo que temos três conjuntos S_1, S_2 e S_3 a considerar em uma fórmula φ. Considere que a indexação das posições de w seja de 0 a 4 (w tem 5 caracteres) e que esses conjuntos recebem as seguintes atribuições $S_1 = \{3,4\}$, $S_2 = \{0\}$ e $S_3 = \{0,2,4\}$. A proposta se inicia com a transformação de cada conjunto em uma *string* binária de mesmo tamanho que w, de modo que o *bit* da posição i denote que aquela posição pertence ou não ao conjunto. Assim, o S_1 seria dado por 00011, o S_2 por 10000 e o S_3 por 10101. De posse da representação binária de cada conjunto, pode-se compor essas palavras, montando algo semelhante a uma matriz como mostrado na Figura 2.8. O autômato que irá processar essa palavra deverá ler cada coluna dessa matriz como se fosse um símbolo. Deste modo, obtém-se a representação estendida de w, que no exemplo apresentado seria $a011\ a000\ b001\ b100\ a101$.

Note que a Figura 2.8 também ilustra como seria a representação estendida para uma árvore e três conjuntos. Nesse caso, tem-se a árvore $t = b[b[a,a],a]$ e a representação das atribuições dos conjuntos é $S_1 = 0[1[0,0],0]$, $S_2 = 0[0[0,1],0]$ e $S_3 = 0[1[0,0],0]$. Faz-se a "composição matricial" das árvores, de modo que cada coluna de elementos da árvore seja lida como um único símbolo. Lembre-se que, na árvore, os símbolos '[', ']' e ',' são apenas marcações para leitura, não são elementos do alfabeto que serão considerados pelo *tree automaton*. Assim, o processo de composição gera a versão estendida de t considerando uma atribuição de S_1, S_2 e S_3, que é $t = b000\ [b101\ [a000,\ a010],\ a000]$.

O procedimento a ser aplicado é recursivo na estrutura da fórmula. De modo genérico, este procedimento pode ser definido da seguinte maneira. Tome uma fórmula em MSOL denotada por φ. A construção do autômato A_φ, a partir da fórmula φ, é dada pelos seguintes casos:

$$\begin{array}{rcccccc} \mathrm{w} & = & \mathrm{a} & \mathrm{a} & \mathrm{b} & \mathrm{b} & \mathrm{a} \\ S_1 & = & 0 & 0 & 0 & 1 & 1 \\ S_2 & = & 1 & 0 & 0 & 0 & 0 \\ S_3 & = & 1 & 0 & 1 & 0 & 1 \end{array} \qquad \begin{array}{rcccccccccc} \mathrm{t} & = & \mathrm{b} & [& \mathrm{b} & [& \mathrm{a} & , & \mathrm{a} &] & , & \mathrm{a} &] \\ S_1 & = & 0 & [& 1 & [& 0 & , & 0 &] & , & 0 &] \\ S_2 & = & 0 & [& 0 & [& 0 & , & 1 &] & , & 0 &] \\ S_3 & = & 0 & [& 1 & [& 0 & , & 0 &] & , & 0 &] \end{array}$$

$$\mathrm{w} = \mathrm{a011\ a000\ b001\ b100\ a101}$$

$$\mathrm{t} = \mathrm{b000\ [b101\ [a000\ ,\ a010]\ ,\ a000]}$$

Figura 2.8: Exemplo de palavra e árvore denotada pelo alfabeto estendido para três conjuntos

φ é um predicado : este é um caso onde o autômato deve ser construído baseado na semântica do predicado. Mais detalhes sobre isso serão apresentados a posteriori.

$\varphi = \neg\phi$: construa um autômato que reconheça a fórmula ϕ. Caso a função de transição dele não seja total, torne-a total. Após isso, aplique a operação de complemento no autômato e obterá o autômato que reconhece φ.

$\varphi = \phi \wedge \psi$: construa os autômatos que reconheça as fórmulas ϕ e ψ. Após construí-los, aplique sobre eles a operação de interseção de autômatos, pois se buscam as estruturas que atendam simultaneamente as fórmulas ϕ e ψ.

$\varphi = \phi \vee \psi$: construa os autômatos que reconheça as fórmulas ϕ e ψ. Após construí-los, aplique sobre eles a operação de união de autômatos, pois se buscam as estruturas que atendam as fórmulas ϕ, ψ ou ambas simultaneamente.

$\varphi = \exists X.\phi$: inicialmente constrói-se o autômato que reconheça a fórmula ϕ. Depois, do autômato construído, percorrem-se as transições eliminando de cada uma delas a posição referente à variável X, o que é equivalente a uma operação de projeção em autômatos.

Informações sobre união, interseção, complemento e projeção não serão explicadas neste momento. No Capítulo 5 serão apresentadas as rotinas necessárias para obter esses autômatos apenas para o caso de árvores e aplicando a *tree language* utilizada no algoritmo de Courcelle.

Dado que não foram encontrados na literatura algoritmos explícitos para a construção dos autômatos para predicados MSOL baseados em modelos construídos a partir de palavras e árvores, este texto propõe os algoritmos para as fórmulas atômicas apresentadas por Thomas [54] e apresenta suas provas de corretude.

2.4.1 Palavras como modelos

Considere a linguagem de palavras apresentada no Exemplo 2.1.1. Para representar esta linguagem por meio de uma fórmula lógica é necessário expressar a existência de três posições consecutivas na palavra que sejam rotuladas por "a" e a não existência de uma posição consecutiva a esta "terceira" posição.

Seja w uma palavra construída a partir de um alfabeto Σ qualquer. A construção do modelo $\underline{w}$ a partir de w é dada por

$$\underline{w} = \langle Dom(\underline{w}), S, <, (Q_\sigma)_{\sigma\in\Sigma}, \subseteq, sing\rangle$$

onde:

***Dom*($\underline{w}$)** é o domínio do modelo e representa as posições dos símbolos na palavra, portanto $Dom(\underline{w}) = \{0, 1, \ldots, n-1\}$, onde n é o tamanho da *string*

S é a relação binária definida sobre conjuntos unitários, onde o elemento do primeiro conjunto é o antecessor do elemento do segundo conjunto. Assim, denota-se o predicado como $S(X, Y)$, tal que $X = \{i\}$ e $Y = \{j\}$ para $j = i + 1$, $0 \leq i \leq n - 2$.

$<$ é a relação binária definida sobre conjuntos unitários de modo que o elemento do primeiro conjunto é estritamente menor que o elemento do segundo conjunto. Denota-se o predicado como $X < Y$, tal que $X = \{i\}$ e $Y = \{j\}$ para $0 \leq i < j$ e $i < j \leq n - 1$.

$\boldsymbol{Q_\sigma}$, para $\sigma \in \Sigma$, representa o conjunto das posições da palavra que estão associadas ao símbolo σ.

$\subseteq$ é a relação de inclusão de subconjuntos, denotada como $X \subseteq Y$.

sing é o predicado que testa se um determinado conjunto é unitário, denotado como $sing(X)$.

É importante ressaltar que, logicamente falando, 'S', '$<$', '$\subseteq$' e '$sing$' são definidos como

- $sing(X) \equiv \exists x.(X(x) \wedge \forall y.(X(y) \rightarrow (y = x)))$
- $S(X, Y) \equiv \exists x.(\exists y.(X(x) \wedge Y(y) \wedge (y = x + 1)))$
- $X < Y \equiv \exists x.(\exists y.(\exists k.(X(x) \wedge Y(y) \wedge (y = x + k) \wedge (k \geq 1))))$

- $X \subseteq Y \equiv \forall x.(X(x) \rightarrow Y(x))$

Deste modo, a linguagem do Exemplo 2.1.1 pode ser definida por meio da fórmula $\exists X.\exists Y.\exists Z.(sing(X) \wedge sing(Y) \wedge sing(Z) \wedge \neg(X = Y) \wedge \neg(X = Z) \wedge \neg(Y = Z) \wedge (X \subseteq Q_a) \wedge (Y \subseteq Q_a) \wedge (Z \subseteq Q_a) \wedge S(X, Y) \wedge S(Y, Z) \wedge \neg\exists W.(sing(W) \wedge \neg(W = X) \wedge \neg(W = Y) \wedge \neg(W = Z) \wedge S(Z, W)))$ expressa em MSOL_0. Esta fórmula diz que X, Y e Z devem ser conjuntos unitários distintos de modo que a posição contida em cada um deles seja rotulada por "a". Além disso, a posição de X deve ser sucedida pela posição de Y, que deve ser sucedida pela posição de Z, e não deve existir um conjunto unitário W cuja posição seja sucessora da posição de Z.

Tendo em vista o modelo $\underline{w}$, definimos as fórmulas atômicas para esse modelo como sendo $sing(X_i)$, $X_i \subseteq X_j$, $X_i \subseteq Q_\sigma$, $X_i < X_j$ e $S(X_i, X_j)$, onde X_i e X_j são variáveis para conjuntos, $i \neq j$ e i e j são índices do conjunto no bloco de quantificadores da fórmula em forma normal prenex sendo avaliada. O método de construção dos autômatos para essas fórmulas atômicas será explicado no restante desta subseção. Lembre-se que todos os autômatos consideram como entrada o alfabeto estendido Σ^X. Considere, em todos os algoritmos até o fim do capítulo, $sym \in \Sigma^X$ e $sym = (\sigma, bin)$, onde $\sigma \in \Sigma$ e $bin \in \{0, 1\}^c$.

Iniciando pela fórmula $sing(X_i)$, o autômato a ser construído deve avaliar se o conjunto X_i é um conjunto unitário. Essa verificação se dá ao processar toda a palavra de entrada para garantir que apenas um símbolo tem a i-ésima posição da *string* de controle como 1. Lembre-se que a pertinência de uma ou mais posições da palavra a um conjunto é denotada pelos *bits* da *string* de controle. O Algoritmo 2.1 apresenta um método de construção para o autômato para esta fórmula, assumindo que são dados como entrada o índice i do conjunto no bloco de quantificadores da fórmula e o alfabeto estendido. Observe que esta é uma fórmula que não considera em momento algum o símbolo que ocupa aquela posição na palavra. Apenas interessa se um único par sym possui a i-ésima posição do bin marcada com 1.

```
1  automaton-singleton(i,Σ^X)
2      Q ← {q_wait, q_unit, q_error};
3      F ← {q_unit};
4      q_0 ← q_wait;
5      δ ← ∅
6      FOR EACH sym ∈ Σ^X DO
7          IF (sym.bin[i] = 0) THEN
8              δ ← δ ∪ {(sym, q_wait, q_wait)}
9              δ ← δ ∪ {(sym, q_unit, q_unit)}
```

```
10                  δ ← δ ∪ {(sym, q_error, q_error)}
11              ELSE
12                  δ ← δ ∪ {(sym, q_wait, q_unit)}
13                  δ ← δ ∪ {(sym, q_unit, q_error)}
14                  δ ← δ ∪ {(sym, q_error, q_error)}
15          auto ← ⟨Σ^X, Q, δ, F, q_0⟩
16          RETURN auto
```

Algoritmo 2.1: Autômato para reconhecimento do predicado $sing(X_i)$

Teorema 2.4.1 (Corretude do Algoritmo 2.1)**.** *Dado que Σ^X é um alfabeto estendido e $1 \leq i \leq c$, onde c é o número de conjuntos mencionados na fórmula, o Algoritmo 2.1 constrói um autômato que aceita uma palavra quando o conjunto X_i é unitário.*

Demonstração. Suponha que o autômato construído aceite que o conjunto X_i não seja unitário. Portanto, dois cenários devem ser considerados.

1. $X_i = \emptyset$.

   Supondo X_i vazio, tem-se que todo símbolo lido terá o i-ésimo bit da *string* de controle em 0. Logo, o autômato executará a transição $(sym, q_{wait}, q_{wait})$ até terminar a leitura. O estado encontrado ao final do processo é q_{wait}, o qual não é um estado de aceitação. Isso contradiz a suposição inicial.

2. X_i possui pelo menos dois elementos.

   Sem perda de generalidade, assuma que X_i possui exatamente dois elementos. O autômato executará a transição $(sym, q_{wait}, q_{wait})$ até encontrar o primeiro elemento, quando então realizará a transição $(sym, q_{wait}, q_{unit})$. Após encontrar o primeiro elemento, ele executará a transição $(sym, q_{unit}, q_{unit})$ até encontrar o segundo, quando então executará $(sym, q_{unit}, q_{error})$. Observe que toda transição partindo do estado q_{error} permanece neste estado. Logo, o processo termina no estad q_{error}, que também não é um estado de aceitação. Novamente obtém-se uma contradição.

Logo, o autômato construído pelo Algoritmo 2.1 reconhece apenas casos onde o conjunto X_i é unitário. □

Diferentemente das demais fórmulas atômicas, a fórmula $X_i \subseteq Q_\sigma$, $\sigma \in \Sigma$, é a única que necessita de uma verificação no símbolo do alfabeto sendo lido com a *string* de controle. O Algoritmo 2.2 mostra como construir esse autômato. A cada posição que possua

o i-ésimo bit de *bin* marcado com 1, é necessário verificar se o símbolo que a rotula é σ, que é um parâmetro que determina o símbolo do alfabeto sendo verificado. Lembre-se que existe um conjunto Q_σ para cada um dos símbolos do alfabeto.

```
1   automaton-subset-of-Q-sigma(i,Σ^X,σ)
2       Q ← {q_sub, q_error};
3       F ← {q_sub};
4       q_0 ← q_sub;
5       δ ← ∅
6       FOR EACH sym ∈ Σ^X DO
7           IF (sym.bin[i] = 0) THEN
8               δ ← δ ∪ {(sym, q_sub, q_sub)}
9               δ ← δ ∪ {(sym, q_error, q_error)}
10          ELSE
11              IF (sym.σ = σ) THEN
12                  δ ← δ ∪ {(sym, q_sub, q_sub)}
13                  δ ← δ ∪ {(sym, q_error, q_error)}
14              ELSE
15                  δ ← δ ∪ {(sym, q_sub, q_error)}
16                  δ ← δ ∪ {(sym, q_error, q_error)}
17      auto ← ⟨Σ^X, Q, δ, F, q_0⟩
18      RETURN auto
```

Algoritmo 2.2: Autômato para reconhecimento do predicado $X_i \subseteq Q_\sigma$.

Teorema 2.4.2 (Corretude do Algoritmo 2.2)**.** *Dado que Σ^X é um alfabeto estendido, $\sigma \in \Sigma$ e $1 \leq i \leq c$, onde c é o número de conjuntos mencionados na fórmula e Σ é o alfabeto antes de ser estendido, o Algoritmo 2.2 constrói um autômato que aceita uma palavra sempre que $X_i \subseteq Q_\sigma$.*

Demonstração. Para a prova deste teorema, considere os seguintes cenários:

1. $X_i = \emptyset$

   Se $X_i = \emptyset$, então toda posição da palavra terá o i-ésimo bit da *string* de controle em 0. Logo, a transição (sym, q_{sub}, q_{sub}) executará até o fim da leitura. O autômato não deixará o estado q_{sub}, que é um estado de aceitação. O resultado não é afetado pela cardinalidade do conjunto Q_σ.

2. $X_i \neq \emptyset$ e $Q_\sigma = \emptyset$

   Suponha, por absurdo, que $X_i \neq \emptyset$ e $Q_\sigma = \emptyset$ seja uma situação aceita pelo autômato em questão. Dado que $Q_\sigma = \emptyset$, nenhuma posição da palavra lida é rotulada por σ

(lembre-se que o modelo é construído a partir da palavra considerada). Sem perda de generalidade, assuma que X_i é unitário. Logo, ao encontrar a posição cujo i-ésimo bit da *string* de controle está em 1, o autômato irá testar se tal posição é rotulada por σ. A transição a ser executada é $(sym, q_{sub}, q_{error})$, pois nenhuma posição da palavra é rotulada por σ. A execução termina no estado q_{error} (observe as transições que partem de q_{error}), que não é um estado de aceitação. Isso contradiz a suposição de que o autômato aceita o caso mencionado.

3. $X_i \neq \emptyset$ e $Q_\sigma \neq \emptyset$

   Considere $X_i \neq \emptyset$ e $Q_\sigma \neq \emptyset$. Suponha, sem perda de generalidade, a existência de um elemento de X_i que não é rotulado por σ e que ainda assim o autômato construído aceita $X_i \subseteq Q_\sigma$. Enquanto o autômato não encontrar elemento de X_i, ele executará a transição (sym, q_{sub}, q_{sub}). Ao encontrar símbolos que possuam o i-ésimo bit da *string* de controle em 1, o autômato irá testar se tal símbolo tem rótulo σ ou não. Independente do estado de origem, ao encontrar o símbolo que não é rotulado por σ, o autômato muda seu estado para q_{error}, no qual irá permanecer até o fim da leitura. Note que q_{error} não é um estado de aceitação. Isso contradiz a hipótese de que X_i é aceito como subconjunto de Q_σ quanto este possui algum elemento que não está rotulado por σ.

Portanto, o autômato construído pelo Algoritmo 2.2 aceita uma palavra sempre que $X_i \subseteq Q_\sigma$. □

A ideia apresentada para o caso de subconjuntos de Q_σ pode ser generalizada para conjuntos diferentes. Note que a fórmula $X_i \subseteq X_j$ também não considera o símbolo da posição lida como relevante. Garante-se que X_i é um subconjunto de X_j sempre que todo símbolo lido que possuir a i-ésima posição de sua *string* de controle marcada com 1 também possuirá a j-ésima posição marcada com 1. Encontrar um símbolo que possua a i-ésima posição de sua *string* de controle marcada com 1 e com a j-ésima posição marcada com 0 implica na existência de um elemento de X_i que não pertence a X_j, fazendo com que $X_i \not\subseteq X_j$. A construção é dada pelo Algoritmo 2.3.

```
1   automaton-subset(i,j,Σ^X)
2       Q ← {q_sub, q_error};
3       F ← {q_sub};
4       q_0 ← q_none;
5       δ ← ∅
6       FOR EACH sym ∈ Σ^X DO
```

```
7          IF (sym.bin[i] = 0) THEN
8              δ ← δ ∪ {(sym, q_sub, q_sub)}
9              δ ← δ ∪ {(sym, q_error, q_error)}
10         ELSE
11             IF (σ → bin[j] = 1) THEN
12                 δ ← δ ∪ {(sym, q_sub, q_sub)}
13                 δ ← δ ∪ {(sym, q_error, q_error)}
14             ELSE
15                 δ ← δ ∪ {(sym, q_sub, q_error)}
16                 δ ← δ ∪ {(sym, q_error, q_error)}
17     auto ← ⟨Σ^X, Q, δ, F, q_0⟩
18     RETURN auto
```

Algoritmo 2.3: Autômato para reconhecimento do predicado $X_i \subseteq X_j$

Teorema 2.4.3 (Corretude do Algoritmo 2.3)**.** *Dado que Σ^X é um alfabeto estendido, $1 \leq i \leq c$, $1 \leq j \leq c$ e $i \neq j$, onde c é o número de conjuntos mencionados na fórmula, o Algoritmo 2.3 constrói um autômato que aceita uma palavra quando o conjunto X_i é um subconjunto do conjunto X_j.*

Demonstração. A prova deste teorema é análoga à do Teorema 2.4.2, diferindo apenas nos conjuntos considerados. Enquanto no Teorema 2.4.2 era necessário verificar o símbolo que rotulava uma determinada posição sendo lida da palavra, neste caso a verificação é feita em cima do j-ésimo bit da *string* de controle. □

Os autômatos para as fórmulas $S(X_i, X_j)$, apresentado no Algoritmo 2.4, e $X_i < X_j$, apresentado no Algoritmo 2.5, são extremamente semelhantes. A diferença fundamental entre ela está na aceitação do momento em que o elemento do conjunto X_j é encontrado. Enquanto o caminho de sucesso de $S(X_i, X_j)$ é atingido ao encontrar o elemento de X_j imediatamente após encontrar o elemento de X_i, para $X_i < X_j$ não ocorre dessa forma. A única coisa necessária para atingir o sucesso na leitura deste último é encontrar o elemento de X_j *em algum momento* após encontrar o elemento de X_i. Não esqueça que estas fórmulas são definidas sobre conjuntos unitários. Leituras de conjuntos não unitários devem ser rejeitadas ao longo do processo.

```
1  automaton-successor(i,j,Σ^X)
2      Q ← {q_none, q_first, q_second, q_error};
3      F ← {q_second};
4      q_0 ← q_none;
5      δ ← ∅
```

```
6        FOR EACH sym ∈ Σ^X DO
7            IF (sym.bin[i] = 0) THEN
8                IF (sym.bin[j] = 1) THEN
9                    δ ← δ ∪ {(sym, q_none, q_error)}
10                   δ ← δ ∪ {(sym, q_first, q_second)}
11                   δ ← δ ∪ {(sym, q_second, q_error)}
12                   δ ← δ ∪ {(sym, q_error, q_error)}
13               ELSE
14                   δ ← δ ∪ {(sym, q_none, q_none)}
15                   δ ← δ ∪ {(sym, q_first, q_error)}
16                   δ ← δ ∪ {(sym, q_second, q_second)}
17                   δ ← δ ∪ {(sym, q_error, q_error)}
18           ELSE
19               IF (sym.bin[j] = 1) THEN
20                   δ ← δ ∪ {(sym, q_none, q_error)}
21                   δ ← δ ∪ {(sym, q_first, q_error)}
22                   δ ← δ ∪ {(sym, q_second, q_error)}
23                   δ ← δ ∪ {(sym, q_error, q_error)}
24               ELSE
25                   δ ← δ ∪ {(sym, q_none, q_first)}
26                   δ ← δ ∪ {(sym, q_first, q_error)}
27                   δ ← δ ∪ {(sym, q_second, q_error)}
28                   δ ← δ ∪ {(sym, q_error, q_error)}
29       auto ← ⟨Σ^X, Q, δ, F, q_0⟩
30       RETURN auto
```

Algoritmo 2.4: Autômato para reconhecimento do sucessor

Teorema 2.4.4 (Corretude do Algoritmo 2.4)**.** *Dado que Σ^X é um alfabeto estendido, $1 \leq i \leq c$, $1 \leq j \leq c$ e $i \neq j$, onde c é o número de conjuntos mencionados na fórmula, o Algoritmo 2.4 constrói um autômato que aceita uma palavra quando os conjuntos X_i e X_j quando ambos são conjuntos unitários e a posição pertencente a X_i é uma unidade menor que a posição pertencente a X_j.*

Demonstração. Suponha que o autômato construído não reconheça o caso citado. Portanto, deve-se avaliar os seguintes cenários:

1. $X_i = \emptyset$ e $X_j \neq \emptyset$

   Assuma, sem perda de generalidade, que X_j é unitário. Isso significa que todo símbolo lido possui o i-ésimo bit da *string* de controle em 0 e apenas um com o j-ésimo bit de controle em 1. Enquanto o autômato não encontrar o elemento associado ao conjunto X_j, a transição $(sym, q_{none}, q_{none})$ será executada. Ao encontrar a posição

referente ao conjunto X_j, o autômato executa a transição $(sym, q_{none}, q_{error})$ e permanece no estado q_{error} até o fim da execução. Dado que q_{error} não é um estado de aceitação, obtém-se uma contradição em relação a suposição inicial. Caso X_j não seja unitário, o raciocínio não se modifica. Observe que uma vez no estado q_{error}, o autômato não pode sair dele.

2. $X_i \neq \emptyset$ e $X_j = \emptyset$

   Similarmente, assuma que X_i é unitário. Até que o autômato encontre a posição pertencente a X_i, a transição $(sym, q_{none}, q_{none})$ será executada. Ao encontrar a posição mencionada, ele executa a transição $(sym, q_{none}, q_{first})$, para indicar que encontrou a posição contida em X_i. Dado que X_j é vazio, o símbolo lido na sequência possui o j-ésimo bit em 0. Portanto, o autômato irá executar a transição $(sym, q_{first}, q_{error})$, permanecendo no estado q_{error} até o término da execução. Portanto, obtém-se uma nova contradição em relação ao cenário proposto.

3. $X_i = X_j = \emptyset$

   Assumindo que X_i e X_j são vazios, o autômato executará a transição $(sym, q_{none}, q_{none})$ até atingir o final da palavra. Dado que o estado q_{none} não é um estado de aceitação, obtém-se uma nova contradição.

4. $X_i = X_j \neq \emptyset$

   Assuma que ambos os conjuntos X_i e X_j são unitários e iguais. A autômato executará a transição $(sym, q_{none}, q_{none})$ até encontrar a posição onde ocorrem, simultaneamente, os bits i e j da *string* de controle em 1. Nesse cenário, o autômato executa a transição $(sym, q_{none}, q_{error})$. Atingir o estado q_{error} significa contradizer a hipótese inicial, dado que ele não é um estado de aceitação.

5. $X_i \neq X_j \neq \emptyset$

   Para o caso em que ambos os conjuntos são não vazios e diferentes, deve-se observar os seguintes cenários:

   (a) X_i e X_j são unitários e a posição de X_j aparece antes da posição de X_i.

   Caso a posição de X_j seja menor que a de X_i e ambos sejam unitários, existe um símbolo na palavra que possui o i-ésimo bit da *string* de controle em 0 e o j-ésimo em 1. Logo, o autômato executará a transição $(sym, q_{none}, q_{error})$, pois o elemento do segundo conjunto apareceu antes do primeiro. Uma vez em q_{error}, não há saída que possa gerar uma aceitação, o que contradiz a hipótese.

(b) X_i e X_j são unitários e a posição de X_j aparece não consecutivamente à posição de X_i.

Neste caso, o autômato executa a transição $(sym, q_{none}, q_{none})$ até encontrar a posição pertencente ao conjunto X_i, quando a transição $(sym, q_{none}, q_{first})$ é disparada. Como a posição de X_j não é consecutiva de X_i, a transição $(sym, q_{first}, q_{error})$ é disparada porque o símbolo lido possui os i-ésimo e j-ésimo bits da *string* de controle em 0. Novamente, o autômato atinge o estado q_{error} e contradiz a hipóstese.

(c) Pelo menos um dos conjuntos não é unitário.

Caso um dos conjuntos não seja unitário, transições como $(sym, q_{second}, q_{error})$ ou $(sym, q_{first}, q_{error})$ serão disparadas quando mais de um elemento é encontrado com o i-ésimo ou j-ésimo bit da *string* de controle em 1. O autômato é levado para o estado q_{error}, de onde não há saída que possa gerar uma aceitação, novamente contradizendo a hipótese inicial.

Portanto, o autômato construído pelo Algoritmo 2.4 aceita uma palavra sempre que a posição de X_i for uma unidade menor que a posição de X_j. □

```
1   automaton-order(i,j,Σ^X)
2       Q ← {q_none, q_first, q_second, q_error};
3       F ← {q_second};
4       q_0 ← q_none;
5       δ ← ∅
6       FOR EACH σ ∈ Σ DO
7           IF (σ → bin[i] = 0) THEN
8               IF (σ → bin[j] = 1) THEN
9                   δ ← δ ∪ {(σ, q_none, q_error)}
10                  δ ← δ ∪ {(σ, q_first, q_second)}
11                  δ ← δ ∪ {(σ, q_second, q_error)}
12                  δ ← δ ∪ {(σ, q_error, q_error)}
13              ELSE
14                  δ ← δ ∪ {(σ, q_none, q_none)}
15                  δ ← δ ∪ {(σ, q_first, q_first)}
16                  δ ← δ ∪ {(σ, q_second, q_second)}
17                  δ ← δ ∪ {(σ, q_error, q_error)}
18          ELSE
19              IF (σ → bin[j] = 1) THEN
20                  δ ← δ ∪ {(σ, q_none, q_error)}
21                  δ ← δ ∪ {(σ, q_first, q_error)}
22                  δ ← δ ∪ {(σ, q_second, q_error)}
```

```
23                  δ ← δ ∪ {(σ, q_error, q_error)}
24              ELSE
25                  δ ← δ ∪ {(σ, q_none, q_first)}
26                  δ ← δ ∪ {(σ, q_first, q_error)}
27                  δ ← δ ∪ {(σ, q_second, q_error)}
28                  δ ← δ ∪ {(σ, q_error, q_error)}
29          auto ← ⟨Σ^X, Q, δ, F, q_0⟩
30          RETURN auto
```

Algoritmo 2.5: Autômato para reconhecimento da ordenação

Teorema 2.4.5 (Corretude do Algoritmo 2.5)**.** *Dado que Σ^X é um alfabeto estendido, $1 \leq i \leq c$, $1 \leq j \leq c$ e $i \neq j$, onde c é o número de conjuntos mencionados na fórmula, o Algoritmo 2.4 constrói um autômato que reconhece os conjuntos X_i e X_j quando ambos são conjuntos unitários e a posição pertencente a X_i é k unidades menor que a posição pertencente a X_j, para algum $k \geq 1$.*

Demonstração. A prova desse teorema é análoga à prova do Teorema 2.4.4, com excessão do Caso 5b. O caso em questão é aceito pelo autômato construído. Basta observar a presença da transição $(sym, q_{first}, q_{first})$, a qual não está presente no autômato gerado pelo Algoritmo 2.4. □

2.4.2 Árvores como modelos

A técnica para construção de *tree automata* baseados em árvores é semelhante ao processo feito por palavras. Por simplificação, os algoritmos e o modelo apresentados nesta subseção consideram unicamente árvores estritamente binárias como modelos. Essas árvores serão construídas por meio de um alfabeto binário onde os símbolos possuem *rank* igual a 2 ou *rank* igual a 0.

Semelhante ao que ocorreu com as palavras, as posições dos símbolos na árvore serão os elementos do domínio do modelo a ser construído. Thomas sugere o uso de uma codificação dessas posições por meio de *strings* binárias. Assim, toda posição p de um símbolo da árvore terá posição denotada por uma palavra da forma $\{0, 1\}^*$. Observe a Figura 2.9 para um exemplo de árvore construída sobre o alfabeto $\Sigma_{rank} = \langle \{a, b\}, \{(a, 0), (b, 2)\} \rangle$. A codificação é dada da seguinte maneira:

- A posição da raiz da árvore é sempre denotada por ϵ. É necessário mencionar que ϵ representa o elemento neutro da concatenação de palavras, podendo ser omitido em casos onde a palavra em questão é não-vazia.

- Uma posição denotada por $i0$ significa que esta posição é o filho esquerdo da posição denotada por i, com $i \in \{0,1\}^*$.
- Uma posição denotada por $i1$ significa que esta posição é o filho direito da posição denotada por i, com $i \in \{0,1\}^*$.

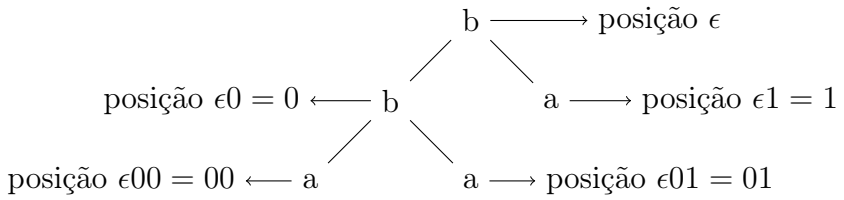


Figura 2.9: Exemplo de árvore e sua codificação de posições

Essa codificação induz uma espécie de ordenação nas árvores por meio da relação de prefixo próprio, dado que cada posição é representada por uma palavra. Um **prefixo próprio** w' de uma palavra w é palavra tal que exista uma palavra não-vazia w'', i.e., diferente de ϵ, de modo que $w = w'w''$. A relação de prefixo próprio permite percorrer a árvore alternando, possivelmente, passos para a esquerda com passos para a direita.

Considere uma árvore t construído a partir de um alfabeto ranqueado $\Sigma_{rank} = \langle \Sigma, rank \rangle$. A construção do modelo $\underline{t}$ para esta árvore é denotada por:

$$\underline{t} = \langle Dom(\underline{t}), S_1, S_2, <, (Q_\sigma)_{\sigma \in \Sigma}, \subseteq, sing \rangle$$

onde:

$\boldsymbol{Dom(\underline{t})}$ é o conjunto das posições dos nós da árvore codificados por meio de palavras de $\{0,1\}^*$.

$\boldsymbol{S_1}$ é a relação binária sobre conjuntos unitários onde o nó contido no segundo conjunto é o filho esquerdo do nó contido no primeiro conjunto. Denota-se esse predicado da forma $S_1(X,Y)$, onde $X = \{i\}$ e $Y = \{j\}$ tal que $j = i0$, para $i, j \in \{0,1\}^*$.

$\boldsymbol{S_2}$ é a relação binária sobre conjuntos unitários onde o nó contido no segundo conjunto é o filho direito do nó contido no primeiro conjunto. Denota-se esse predicado da forma $S_1(X,Y)$, onde $X = \{i\}$ e $Y = \{j\}$ tal que $j = i1$, para $i, j \in \{0,1\}^*$.

$<$ é a relação binária definida sobre conjuntos unitários, de modo que o nó contido no primeiro conjunto é um ancestral do nó contido no segundo conjunto. Denota-se esse predicado por $X < Y$, onde $X = \{i\}$ e $Y = \{j\}$ tal que $j = iw$, onde $w \in \{0,1\}^*$ e $w \neq \epsilon$.

$\boldsymbol{Q_\sigma}$, para $\sigma \in \Sigma$, é o conjunto das posições dos nós da árvore que são rotulados pelo símbolo σ.

$\subseteq$ é a relação de inclusão de conjuntos, denotada por $X \subseteq Y$.

sing é o predicado que testa se um determinado conjunto é unitário, denotado como $sing(X)$.

Outra forma de interpretar o símbolo '$<$' é por meio do conceito de fecho transitivo de uma relação. Considere uma relação binária R construída sobre um conjunto A. O **fecho transitivo** de R, denotado por R^t, é definido como $R \subseteq R^t$ e se $(a, a') \in R$ e $(a', a'') \in R$, então $(a, a'') \in R^t$. Considere que A seja o conjunto das posições codificadas de uma árvore, suc_0 seja a relação de sucessor à esquerda definida sobre A e suc_1 seja a relação de sucessor à direita definida sobre A. Repare que a aplicação do fecho transitivo sobre suc_0 gera uma relação que permite percursos na árvore apenas caminhando por ramos à esquerda. A cada novo nó alcançado, o próximo movimento, se necessário, obrigatoriamente é à esquerda. O resultado do fecho transitivo aplicado a suc_1 é análogo, porém resultando em movimentos à direita. Assim, a relação de prefixo próprio, denotada por '$<$' segundo Thomas, pode ser interpretada como o fecho transitivo da união dos fechos transitivos das relações suc_0 e suc_1.

Assim, as fórmulas atômicas construídas a partir do modelo $\underline{t}$ são $sing(X_i)$, $X_i \subseteq X_j$, $X_i \subseteq Q_\sigma$, $X_i < Y_i$, $S_1(X_i, X_j)$ e $S_2(X_i, X_j)$, onde X_i e X_j são variáveis para conjuntos. Lembre-se que todos os autômatos consideram como entrada o alfabeto estendido Σ^X, no qual cada elemento é da forma $sym = (\sigma, bin)$.

Para ilustrar o apresentado, considere a *tree language* para a Álgebra Booleana apresentada no Exemplo 2.1.2 da Subseção 2.1.2. Se o símbolo "not" for retirado do alfabeto, a *tree language* resultante pode gerar árvores estritamente binárias. Suponha que deseja-se validar um conjunto de árvores que representam expressões que contenham ao menos um aninhamento de "and" à esquerda, ou seja, árvores que contenham uma subárvore da forma $and[and[t, t'], t'']$, onde t, t' e t'' são árvores quaisquer para a *tree language* considerada. Este tipo de propriedade pode ser expressa por meio da fórmula $\exists X.\exists Y.(sing(X) \wedge sing(Y) \wedge (X \subseteq Q_{and}) \wedge (Y \subseteq Q_{and}) \wedge S_1(X, Y))$, assumindo que as posições rotuladas por *not* possuem apenas o sucessor à esquerda.

O Algoritmo 2.6 apresenta a forma de se construir o *tree automaton* para o reconhecimento de $sing(X_i)$. O funcionamento dele é semelhante ao do autômato para este mesmo predicado recebendo uma palavra como entrada. As diferenças aqui estão no fato de ser

necessário identificar o *rank* do símbolo lido e, em casos onde o *rank* é maior que 0, executar uma combinatória no espaço de estados. No mais, é necessário processar a árvore e garantir que apenas uma posição em toda a árvore possua um símbolo cuja *string* de controle associada possua o i-ésimo bit marcado como 1.

```
1   tree-automaton-singleton(i,Σ^X)
2       Q ← {q_wait, q_unit, q_error};
3       F ← {q_unit};
4       δ ← ∅
5       FOR EACH sym ∈ Σ^X DO
6           IF (rank(sym.σ) = 2) THEN
7               FOR EACH q ∈ Q DO
8                   FOR EACH q' ∈ Q DO
9                       IF (sym.bin[i] = 1) THEN
10                          IF (q = q_wait) AND (q' = q_wait) THEN
11                              δ ← δ ∪ {(sym, (q, q'), q_unit)}
12                          ELSE
13                              δ ← δ ∪ {(sym, (q, q'), q_error)}
14                      ELSE
15                          IF (q = q_error) OR (q' = q_error) THEN
16                              δ ← δ ∪ {(sym, (q, q'), q_error)}
17                          ELIF (q = q_unit) AND (q' = q_unit) THEN
18                              δ ← δ ∪ {(sym, (q, q'), q_error)}
19                          ELIF (q = q_wait) THEN
20                              δ ← δ ∪ {(sym, (q, q'), q')}
21                          ELSE
22                              δ ← δ ∪ {(sym, (q, q'), q_unit)}
23          ELSE
24              IF (σ → bin[i] = 1) THEN
25                  δ ← δ ∪ {(sym, ∅, q_unit)}
26              ELSE
27                  δ ← δ ∪ {(sym, ∅, q_wait)}
28      auto ← ⟨Σ^X, Q, δ, F⟩
29      RETURN auto
```

Algoritmo 2.6: *Tree automaton* para reconhecimento do predicado $sing(X_i)$

Teorema 2.4.6 (Corretude do Algoritmo 2.6)**.** *Dado que Σ^X é um alfabeto estendido e $1 \leq i \leq c$, onde c é o número de conjuntos mencionados na fórmula, o Algoritmo 2.6 constrói um* tree automaton *que reconhece uma árvore quando o conjunto X_i é um conjunto unitário.*

Demonstração. Suponha que o Algoritmo 2.6 constrói um autômato que reconhece uma

árvore quando o conjunto X_i não for unitário. Portanto, dois casos devem ser avaliados.

1. $X_i = \emptyset$

   Se o conjunto X_i é vazio, toda posição da árvore terá o i-ésimo bit da *string* de controle em 0. Portanto, as folhas da árvore dispararão transições da forma $(sym, \emptyset, q_{wait})$. Todos as posições que não são folhas dispararão transições da forma $(sym, (q_{wait}, q_{wait}), q_{wait})$. Assim, ao final da leitura da palavra, o autômato estará no estado q_{wait}, que não é um estado de aceitação. Isso contradiz a hipótese inicial.

2. $|X_i| \geq 2$

   Sem perda de generalidade, assuma que X_i tem exatamente dois elementos. Caso algum, senão os dois elementos do conjunto, for uma folha, o autômato executará uma transição da forma $(sym, \emptyset, q_{unit})$. Nas posições que não são folhas, o processamento será mediante a pertinência da posição lida no autômato e dos estados encontrados nos filhos daquela posição. Esta situação pode disparar alguma transição da forma descrita nas linhas 13, 16 ou 18. Assim, o autômato será conduzido ao estado q_{error}, de onde não conseguirá sair. Como q_{error} não é um estado de aceitação, a hipótese de que a cardinalidade de X_i é maior ou igual a 2 não é satisfeita.

Logo, o Algoritmo 2.6 reconhece palavras apenas quando o conjunto X_i é unitário. □

A construção do *tree automaton* para $X_i \subseteq Q_\sigma$ segue o mesmo padrão expresso pala palavras. Sempre que o autômato ler um símbolo que possui o i-ésimo bit da *string* de controle marcado como 1, ele deve avaliar se este símbolo é o símbolo σ procurado. Em caso positivo, a transição ocorre (ou permanece) para o estado que representa a confirmação do subconjunto. Em caso negativo, gera-se uma transição para o estado de erro. Caso o i-ésimo bit esteja marcado com 0, a transição a ser adicionada é neutra no caso das folhas ou processa os estados encontrados nos filhos e repassa a informação adequada. Observe a construção apresentada no Algoritmo 2.7.

```
1   tree-automaton-subset-Q-sigma(i,Σ^X,σ)
2       Q ← {q_sub, q_error};
3       F ← {q_sub};
4       δ ← ∅
5       FOR EACH sym ∈ Σ^X DO
6           IF (rank(sym.σ) = 2) THEN
7               FOR EACH q ∈ Q DO
8                   FOR EACH q' ∈ Q DO
```

```
9                       IF (sym.bin[i] = 1) THEN
10                          IF (sym.σ = σ) THEN
11                              IF (q = q_error) OR (q' = q_error) THEN
12                                  δ ← δ ∪ {(sym, (q, q'), q_error)}
13                              ELSE
14                                  δ ← δ ∪ {(sym, (q, q'), q_sub)}
15                          ELSE
16                              δ ← δ ∪ {(sym, (q, q'), q_error)}
17                      ELSE
18                          IF (q = q_error) OR (q' = q_error) THEN
19                              δ ← δ ∪ {(sym, (q, q'), q_error)}
20                          ELSE
21                              δ ← δ ∪ {(sym, (q, q'), q_sub)}
22          ELSE
23              IF (sym.bin[i] = 1) THEN
24                  IF (sym.σ = σ) THEN
25                      δ ← δ ∪ {(sym, ∅, q_sub)}
26                  ELSE
27                      δ ← δ ∪ {(sym, ∅, q_error)}
28              ELSE
29                  δ ← δ ∪ {(sym, ∅, q_sub)}
30      auto ← ⟨Σ^X, Q, δ, F⟩
31      RETURN auto
```

Algoritmo 2.7: *Tree automaton* para reconhecimento do predicado $X_i \subseteq Q_\sigma$

Teorema 2.4.7 (Corretude do Algoritmo 2.7)**.** *Dado que* $\Sigma^X = \Sigma \times \{0,1\}^c$ *é um alfabeto estendido,* $1 \leq i \leq c$*, onde* c *é o número de variáveis para conjuntos mencionadas na fórmula, e* $\sigma \in \Sigma$*, o Algoritmo 2.7 constrói um* tree automaton *que reconhece uma árvore quando o conjunto* X_i *é um subconjunto do conjunto* Q_σ*.*

Demonstração. Suponha que o *tree automaton* construído aceite uma árvore quando o conjunto X_i não for um subconjunto de Q_σ, para algum $\sigma \in \Sigma$, dado que $\Sigma^X = \Sigma \times \{0,1\}^c$. Assim, pode-se assumir, sem perda de generalidade a existência de um elemento pertencente a X_i que não pertence a Q_σ. Caso esse elemento seja uma folha, uma transição da forma $(sym, \emptyset, q_{error})$ será executada. Se o elemento em questão não for uma folha, então uma transição como a apresentada na linha 16 do algoritmo é executada, visto que o símbolo lido pertence ao conjunto X_i, mas não é rotulado por σ. Uma vez que um estado q_{error} é encontrado em algum lugar da árvore, transições das linhas 12, 16 e 19 serão responsáveis por manter o *tree automaton* neste estado até que a raiz seja lida. A execução termina no estado q_{error} que não é um estado de aceitação. Logo, o resultado encontrado contradiz a suposição apresentada.

É importante ressaltar que o resultado não se altera ainda que mais de um elemento de X_i não pertença a Q_σ. Observe que as condições das linhas 11 e 18 servem para garantir que, uma vez atingido algum estado de erro, o *tree automaton* não possa sair dele. □

O *tree automaton* equivalente ao predicado $X_i \subseteq X_j$, apresentado no Algoritmo 2.8, é semelhante ao apresentado para o predicado $X_i \subseteq Q_\sigma$. A diferença entre eles consiste em substituir a restrição $sym.\sigma = \sigma$ por $sym.bin[j] = 1$. Observe que este predicado visa testar se um conjunto X_i de posições da árvore é um subconjunto de um conjunto X_j também de posições da árvore. O símbolo lido nessas posições é irrelevante.

```
1   tree-automaton-subset(i,j,Σ^X)
2       Q ← {q_sub, q_error};
3       F ← {q_sub};
4       δ ← ∅
5       FOR EACH sym ∈ Σ^X DO
6           IF (rank(sym.σ) = 2) THEN
7               FOR EACH q ∈ Q DO
8                   FOR EACH q' ∈ Q DO
9                       IF (sym.bin[i] = 1) THEN
10                          IF (sym.bin[j] = 1) THEN
11                              IF (q = q_error) OR (q' = q_error) THEN
12                                  δ ← δ ∪ {(sym, (q, q'), q_error)}
13                              ELSE
14                                  δ ← δ ∪ {(sym, (q, q'), q_sub)}
15                          ELSE
16                              δ ← δ ∪ {(sym, (q, q'), q_error)}
17                      ELSE
18                          IF (q = q_error) OR (q' = q_error) THEN
19                              δ ← δ ∪ {(sym, (q, q'), q_error)}
20                          ELSE
21                              δ ← δ ∪ {(sym, (q, q'), q_sub)}
22          ELSE
23              IF (sym.bin[i] = 1) THEN
24                  IF (sym.bin[j] = 1) THEN
25                      δ ← δ ∪ {(sym, ∅, q_sub)}
26                  ELSE
27                      δ ← δ ∪ {(sym, ∅, q_error)}
28              ELSE
29                  δ ← δ ∪ {(sym, ∅, q_sub)}
30      auto ← ⟨Σ^X, Q, δ, F⟩
31      RETURN auto
```

Algoritmo 2.8: *Tree automaton* para reconhecimento do predicado $X_i \subseteq X_j$

Teorema 2.4.8 (Corretude do Algoritmo 2.8)**.** *Dado que* $\Sigma^X = \Sigma \times \{0,1\}^c$ *é um alfabeto estendido,* $1 \leq i \leq c$*,* $1 \leq j \leq c$ *e* $i \neq j$*, onde* c *é o número de variáveis para conjuntos mencionadas na fórmula, o Algoritmo 2.8 constrói um* tree automaton *que reconhece uma árvore quando o conjunto* X_i *é um subconjunto do conjunto* X_j*.*

Demonstração. A prova é análoga à prova do Teorema 2.4.7, a menos da restrição $sym.bin[j] = 1$ que desempenha o mesmo papel de $sym.\sigma = \sigma$, para algum $\sigma \in \Sigma$, dado que $\Sigma^X = \Sigma \times \{0,1\}^c$. □

A construção dos autômatos para os predicados $S_1(X_i, X_j)$ e $S_2(X_i, X_j)$ é exibida nos Algoritmos 2.9 e 2.10, respectivamente. A ideia central é semelhante à do predicado de sucessor para palavras: ao encontrar a posição contida no conjunto X_j, imediatamente após é necessário encontrar a posição contida no conjunto X_i, lembrando que ambos são unitários. Para palavras, o ocorrido era encontrar primeiro a posição de X_i seguida da posição de X_j. A diferença na árvore é dada pelo "sentido de crescimento" da estrutura. A palavra é uma estrutura linear, assumindo que ela sempre cresce da esquerda para a direita. Em contrapartida, a árvore possui um sentido de crescimento da raiz para as folhas. Portanto, dizer que uma posição p_1 é antecessora de uma posição p_2 (ou p_2 é sucessora de p_1) significa dizer que p_1 é pai de p_2 (ou que p_2 é filho de p_1). Nas árvores, assume-se uma certa ordenação nos filhos. Como estes algoritmos tratam de árvores estritamente binárias, observe que a diferença essencial entre os algoritmos para sucessor à esquerda e sucessor à direita consiste na substituição da restrição $((q = q_{son})$ AND $(q' = q_{none}))$ pela restrição $((q = q_{none})$ AND $(q' = q_{son}))$, onde o estado q_{none} significa nenhuma informação relevante e o estado q_{son} indica a localização de um elemento do conjunto X_j. Reforça-se aqui a ideia de que X_i e X_j devem ser unitários. Ao identificar qualquer ponto no processamento que identifique mais de um estado-filho ou estado-pai, deve-se reportar e propagar o erro no processamento.

```
1   tree-automaton-succ-left(i,j,Σ^X)
2       Q ← {q_none, q_son, q_father, q_error};
3       F ← {q_father};
4       δ ← ∅
5       FOR EACH sym ∈ Σ^X DO
6           IF (rank(sym.σ) = 2) THEN
7               FOR EACH q ∈ Q DO
8                   FOR EACH q' ∈ Q DO
9                       IF (sym.bin[i] = 1) THEN
10                          IF (sym.bin[j] = 1) THEN
```

```
11                                  δ ← δ ∪ {(sym, (q, q′), q_error)}
12                              ELSE
13                                  IF (q = q_son) AND (q′ = q_none) THEN
14                                      δ ← δ ∪ {(sym, (q, q′), q_father)}
15                                  ELSE
16                                      δ ← δ ∪ {(sym, (q, q′), q_error)}
17                          ELSE
18                              IF (sym.bin[j] = 1) THEN
19                                  IF (q = q_none) AND (q′ = q_none) THEN
20                                      δ ← δ ∪ {(sym, (q, q′), q_son)}
21                                  ELSE
22                                      δ ← δ ∪ {(sym, (q, q′), q_error)}
23                              ELSE
24                                  IF (q = q_none) AND (q′ = q_none) THEN
25                                      δ ← δ ∪ {(sym, (q, q′), q_none)}
26                                  ELIF ((q = q_father) AND (q′ = q_none)) OR ((q = q_none)
                                        AND (q′ = q_father)) THEN
27                                      δ ← δ ∪ {(sym, (q, q′), q_father)}
28                                  ELSE
29                                      δ ← δ ∪ {(sym, (q, q′), q_error)}
30              ELSE
31                  IF (sym.bin[i] = 1) THEN
32                      δ ← δ ∪ {(sym, ∅, q_error)}
33                  ELSE
34                      IF (sym.bin[j] = 1) THEN
35                          δ ← δ ∪ {(sym, ∅, q_son)}
36                      ELSE
37                          δ ← δ ∪ {(sym, ∅, q_none)}
38          auto ← ⟨Σ^X, Q, δ, F⟩
39          RETURN auto
```

Algoritmo 2.9: *Tree automaton* para reconhecimento do predicado de sucessor à esquerda

Teorema 2.4.9 (Corretude do Algoritmo 2.9)**.** *Dado que* $\Sigma^X = \Sigma \times \{0,1\}^c$ *é um alfabeto estendido,* $1 \leq i \leq c$*,* $1 \leq j \leq c$ *e* $i \neq j$*, onde* c *é o número de variáveis para conjuntos mencionadas na fórmula, o Algoritmo 2.9 constrói um* tree automaton *que reconhece uma árvore quando o elemento do conjunto unitário* X_j *é o filho esquerdo do elemento do conjunto unitário* X_i*.*

Demonstração. Para provar que o *tree automaton* construído satisfaz os requisitos, deve-se investigar duas situações.

1. Os conjuntos são de fato unitários.

(a) Conjunto(s) vazio(s).

Se ambos os conjuntos forem vazios, o autômato executará transições como $(sym, \emptyset, q_{none})$ a partir das folhas e transições $(sym, (q_{none}, q_{none}), q_{none})$ nos elementos não folhas até atingir a raiz. Logo, o autômato termina a execução em q_{none}, que não é de aceitação, e percebe-se que os conjuntos não devem ser vazios simultaneamente.

Se $X_i = \emptyset$ e $X_j \neq \emptyset$, o autômato irá encontrar um primeiro elemento de X_j, mudando o estado do *tree automaton* para q_{son}, mas numa transição posterior não encontrará o elemento de X_i, pois ele não existe. Logo, o autômato se encaminha para o estado q_{error} e permanece nele até o fim da execução. Assim, X_i não pode ser vazio. Note que, no momento, é indiferente discutir a cardinalidade de X_j, pois uma vez no estado q_{error}, a execução terminará obrigatoriamente neste estado.

Se $X_i \neq \emptyset$ e $X_j = \emptyset$, o autômato encontrará a posição de X_i, mas não encontrará nenhuma informação sobre a posição de X_j, pois ela não existe. Então o autômato se dirige ao estado q_{error} e mantém-se nele até o fim da execução. Portanto, X_j também não pode ser vazio.

(b) Conjuntos com dois ou mais elementos.

Caso a interseção dos conjuntos X_i e X_j seja não vazia, as condições das linhas 9 e 10 garantem que situações assim direcionem o autômato para o estado q_{error}. Portanto, os conjuntos devem ser disjuntos. Caso um dos conjuntos possua mais de um elemento, as transições das linhas 16, 21 e 28 tratam de combinações dos estados q_{son}, q_{father} e q_{error} que demonstram que um dos conjuntos, potencialmente, possui cardinalidade maior ou igual a dois.

Logo, os conjuntos X_i e X_j devem ser unitários.

2. O elemento de X_j é o filho esquerdo do elemento de X_i.

Um detalhe importante ocorre com a descrição do ocorrido no Item 1b. As transições das linhas 16, 21 e 28 são as principais responsáveis por propagar erros encontrados em partes das árvores. Um desses erros, como mencionado no item anterior, é pelo menos um dos conjuntos ter cardinalidade maior que um.

O outro erro frequente ocorre quando o elemento de X_j não é o filho esquerdo do elemento de X_i. Este cenário ocorre quando, por exemplo, o elemento de X_i é uma folha. Dado que as folhas não possuem filhos, uma transição da forma $(sym, \emptyset, q_{error})$ é executada e propagada até atingir a raiz da árvore.

Outra possibilidade é a de os elementos de X_i e X_j não estarem em uma mesma subárvore. Logo, transições da forma $(sym, (q_{error}, q_{son}), q_{error})$ ou $(sym, (q_{son}, q_{error}), q_{error})$ são executadas, deixando o *tree automaton* preso em q_{error}.

Outro cenário consiste nos elementos de X_i e X_j estarem numa mesma subárvore, mas não serem pai e filho. Se o elemento de X_i, por exemplo, for avô do elemento de X_j, o *tree automaton* executará a transição da linha 16, pois não encontrará X_i após encontrar X_j.

Por fim, se o elemento de X_j for o filho direito de X_i, a transição da linha 16 será executada com q_{son} vindo da direita. Novamente o *tree automaton* é dirigido para o estado q_{error} e não aceita a árvore sendo lida.

Logo, o *tree automaton* construído reconhece uma árvore quando o elemento do conjunto unitário X_j é o filho esquerdo do elemento do conjunto unitário X_i. □

```
1   tree-automaton-succ-right(i,j,Σ^X)
2       Q ← {q_none, q_son, q_father, q_error};
3       F ← {q_father};
4       δ ← ∅
5       FOR EACH sym ∈ Σ^X DO
6           IF (rank(sym.σ) = 2) THEN
7               FOR EACH q ∈ Q DO
8                   FOR EACH q' ∈ Q DO
9                       IF (sym.bin[i] = 1) THEN
10                          IF (sym.bin[j] = 1) THEN
11                              δ ← δ ∪ {(sym, (q, q'), q_error)}
12                          ELSE
13                              IF (q = q_none) AND (q' = q_son) THEN
14                                  δ ← δ ∪ {(sym, (q, q'), q_father)}
15                              ELSE
16                                  δ ← δ ∪ {(sym, (q, q'), q_error)}
17                      ELSE
18                          IF (sym.bin[j] = 1) THEN
19                              IF (q = q_none) AND (q' = q_none) THEN
20                                  δ ← δ ∪ {(sym, (q, q'), q_son)}
21                              ELSE
22                                  δ ← δ ∪ {(sym, (q, q'), q_error)}
23                          ELSE
24                              IF (q = q_none) AND (q' = q_none) THEN
25                                  δ ← δ ∪ {(sym, (q, q'), q_none)}
26                              ELIF ((q = q_father) AND (q' = q_none)) OR ((q = q_none)
                                    AND (q' = q_father)) THEN
```

```
27                                  δ ← δ ∪ {(sym, (q, q′), q_father)}
28                              ELSE
29                                  δ ← δ ∪ {(sym, (q, q′), q_error)}
30          ELSE
31              IF (sym.bin[i] = 1) THEN
32                  δ ← δ ∪ {(sym, ∅, q_error)}
33              ELSE
34                  IF (sym.bin[j] = 1) THEN
35                      δ ← δ ∪ {(sym, ∅, q_son)}
36                  ELSE
37                      δ ← δ ∪ {(sym, ∅, q_none)}
38      auto ← ⟨Σ^X, Q, δ, F⟩
39      RETURN auto
```

Algoritmo 2.10: *Tree automaton* para reconhecimento do predicado de sucessor à direita

Teorema 2.4.10 (Corretude do Algoritmo 2.10)**.** *Dado que* $\Sigma^X = \Sigma \times \{0,1\}^c$ *é um alfabeto estendido,* $1 \leq i \leq c$*,* $1 \leq j \leq c$ *e* $i \neq j$*, onde* c *é o número de variáveis para conjuntos mencionadas na fórmula, o Algoritmo 2.10 constrói um* tree automaton *que reconhece uma árvore quando o elemento do conjunto unitário* X_j *é o filho direito do elemento do conjunto unitário* X_i*.*

Demonstração. A prova deste teorema é análoga a do Teorema 2.4.9. A única diferença nos algoritmos é na linha 13, onde a condição $((q = q_{son})AND(q' = q_{none}))$ é substituída por $((q = q_{none})AND(q' = q_{son}))$, permitindo assim que o filho direito seja reconhecido. □

Como o predicado $X_i < X_j$ apresenta uma relação entre os nós da árvore, independente de ser um nó à esquerda ou à direita, a construção do autômato que o reconheça possui a flexibilidade de a informação ser repassada por qualquer um dos filhos de um determinado nó. Diferentemente das relações $S_1(X_i, X_j)$ e $S_2(X_i, X_j)$, $X_i < X_j$ só deve garantir que encontra a posição contida no conjunto X_i em algum momento após encontrar a posição contida no conjunto X_j, não importando se a informação é encontra à esquerda ou à direita. Por isso, utiliza-se a restrição $((q = q_{first})$ AND $(q' = q_{none}))$ OR $((q = q_{none})$ AND $(q' = q_{first}))$ para garantir que, além dos conjuntos em questão serem unitários, é irrelevante a origem da informação, desde que ela exista.

```
1   tree-automaton-prefix(i,j,Σ^X)
2       Q ← {q_none, q_first, q_second, q_error};
3       F ← {q_second};
```

```
4       δ ← ∅
5       FOR EACH sym ∈ Σ^X DO
6           IF (rank(sym.σ) = 2) THEN
7               FOR EACH q ∈ Q DO
8                   FOR EACH q' ∈ Q DO
9                       IF (sym.bin[i] = 1) THEN
10                          IF (sym.bin[j] = 1) THEN
11                              δ ← δ ∪ {(sym, (q, q'), q_error)}
12                          ELSE
13                              IF ((q = q_first) AND (q' = q_none)) OR ((q = q_none) AND (
                                    q' = q_first)) THEN
14                                  δ ← δ ∪ {(sym, (q, q'), q_second)}
15                              ELSE
16                                  δ ← δ ∪ {(sym, (q, q'), q_error)}
17                      ELSE
18                          IF (sym.bin[j] = 1) THEN
19                              IF (q = q_none) AND (q' = q_none) THEN
20                                  δ ← δ ∪ {(sym, (q, q'), q_first)}
21                              ELSE
22                                  δ ← δ ∪ {(sym, (q, q'), q_error)}
23                          ELSE
24                              IF (q = q_none) THEN
25                                  δ ← δ ∪ {(sym, (q, q'), q')}
26                              ELIF (q' = q_none) THEN
27                                  δ ← δ ∪ {(sym, (q, q'), q)}
28                              ELSE
29                                  δ ← δ ∪ {(sym, (q, q'), q_error)}
30          ELSE
31              IF (sym.bin[i] = 1) THEN
32                  δ ← δ ∪ {(sym, ∅, q_error)}
33              ELSE
34                  IF (sym.bin[j] = 1) THEN
35                      δ ← δ ∪ {(sym, ∅, q_first)}
36                  ELSE
37                      δ ← δ ∪ {(sym, ∅, q_none)}
38      auto ← ⟨Σ^X, Q, δ, F⟩
39      RETURN auto
```

Algoritmo 2.11: *Tree automaton* para reconhecimento do predicado de $X_i < X_j$

Teorema 2.4.11 (Corretude do Algoritmo 2.11)**.** *Dado que* $\Sigma^X = \Sigma \times \{0,1\}^c$ *é um alfabeto estendido,* $1 \leq i \leq c$, $1 \leq j \leq c$ *e* $i \neq j$, *onde* c *é o número de variáveis para conjuntos mencionadas na fórmula, o Algoritmo 2.11 constrói um* tree automaton *que reconhece uma árvore quando o elemento do conjunto unitário* X_i *é um ancestral do elemento do conjunto unitário* X_j.

Demonstração. A prova deste teorema é análoga a do Teorema 2.4.9, a menos da flexibilidade do predicado $X_i < X_j$. Na linha 13 as condições para aceitação de $S_1(X_i, X_j)$ e $S_2(X_i, X_j)$ são aceitas. Diferentemente dos predicados de sucessores, o predicado de prefixo permite que as condições das linhas 24 e 26 sejam flexibilizadas, permitindo que o elemento do conjunto X_i seja localizado em algum ancestral do elemento do conjunto X_j. □

2.4.3 Exemplo para o caso de palavras

A fim de complementar o conteúdo exposto neste capítulo, considere os seguintes exemplos pautados na linguagem dos números binários. É sabido que todo número binário cujos dois últimos dígitos são 0's é um múltiplo de 4. Assim, caso seja necessário validar que um número binário é um múltiplo de 4, basta verificar a seguinte propriedade, convenientemente escrita sem quantificações de primeira ordem,

$$\exists X.(\exists Y.(\neg\exists Z.(sing(X) \wedge sing(Y) \wedge sing(Z) \wedge (X \subseteq Q_0) \wedge (Y \subseteq Q_0) \wedge S(X,Y) \wedge S(Y,Z))))$$

onde Q_0, como visto na Subseção 2.4.1, representa o conjunto das posições da palavra que são rotuladas por 0.

Dado que a fórmula já se encontra em forma normal prenex, a construção do autômato que reconhece essa propriedade consiste na construção dos autômatos para as fórmulas atômicas e depois na combinação desses autômatos por meio das propriedades de fecho das linguagens regulares. Em particular, essa combinação será feita por meio do cálculo da interseção dos autômatos. Por fim, serão aplicados uma projeção, uma operação de complemento e duas projeções seguidas, como mostrado na Figura 2.10.

Apesar de sua aparência simples, este exemplo gera autômatos com um considerável número de padrões de transições. Um padrão de transições é um esquema da forma $((\star, 0\star\star), q, q')$, por exemplo, que indica que esta transição é válida para qualquer símbolo do alfabeto estendido que possua o primeiro bit da *string* de controle como 0. Logo, o padrão apresentado pode denotar transições como $((1, 000), q, q')$ e $((0, 011,), q, q')$. Possuir um elevado número de padrões de transições implica em um baixo nível de abstração, o que faz com que um modelo gráfico do autômato fique denso e, consequentemente, pouco compreensível. Portanto, por motivos de visualização, esta subseção irá adotar um exemplo mais simples de ser visualizado.

Tomando ainda por base a linguagem dos números binários, suponha que deseja-se

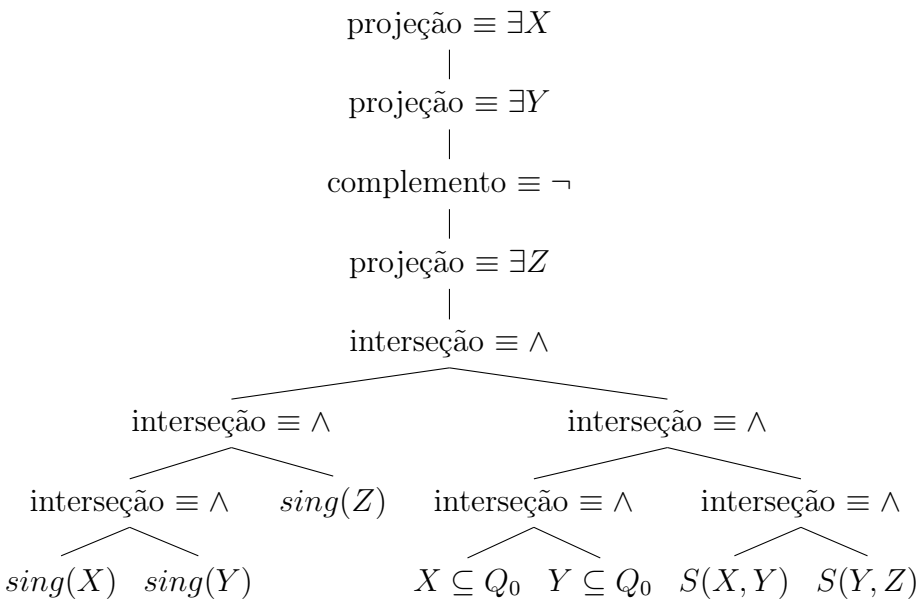


Figura 2.10: Visualização hierárquica da construção do autômato

verificar se um determinado número binário possui dois zeros consecutivos. O alfabeto sobre o qual os autômatos serão construídos é $\Sigma = \{0, 1\}$. Esta propriedade pode ser representada pela fórmula

$$\exists X.(\exists Y.(sing(X) \wedge sing(Y) \wedge (X \subseteq Q_0) \wedge (Y \subseteq Q_0) \wedge S(X, Y)))$$

Dado que a fórmula a ser trabalhada referencia dois conjuntos, o alfabeto estendido será da forma $\Sigma^X = \Sigma \times \{0, 1\}^2$. Para denotar um símbolo do alfabeto estendido, considere $(\sigma, control)$, onde $\sigma \in \Sigma$ e $control \in \{0, 1\}^2$. O processo de construção do autômato para a fórmula começa pela construção dos autômatos para as fórmulas atômicas, a saber $sing(X)$, $sing(Y)$, $X \subseteq Q_0$, $Y \subseteq Q_0$ e $S(X, Y)$. Observe que estes autômatos seguem os padrões apresentados na Subseção 2.4.1. Para tornar claro o padrão, observe as Figuras 2.11 e 2.12. Estas figuras trazem uma representação gráfica para os autômatos de $sing(X)$ e $sing(Y)$. Note que a única diferença entre eles é o bit da *string* de controle considerado, dado que esses bits representam os conjuntos na ordem em que eles aparecem quantificados na fórmula considerada. Assim, o primeiro bit, o mais à esquerda, representa o X e o outro representa o Y.

O próximo autômato a ser construído é o autômatos para $S(X, Y)$. Diferente dos autômatos para conjuntos unitários, esse autômato precisa considerar duas posições da *string* de controle. A semântica dele diz que o elemento do primeiro conjunto unitário é antecessor do elemento do segundo conjunto unitário. Logo, deve-se garantir tanto que os conjuntos são unitários quanto que a leitura de seus elementos seja consecutiva. A

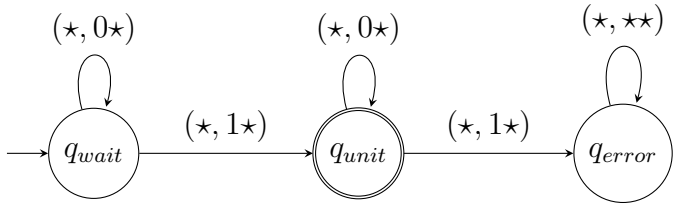


Figura 2.11: Autômato para $sing(X)$

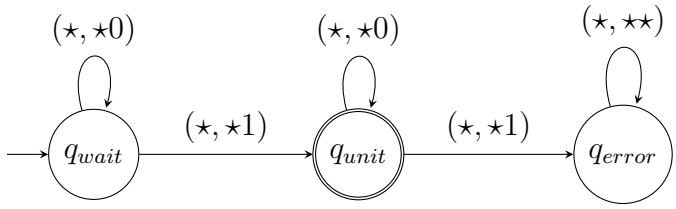


Figura 2.12: Autômato para $sing(Y)$

Figura 2.13 mostra graficamente o autômato para $S(X, Y)$.

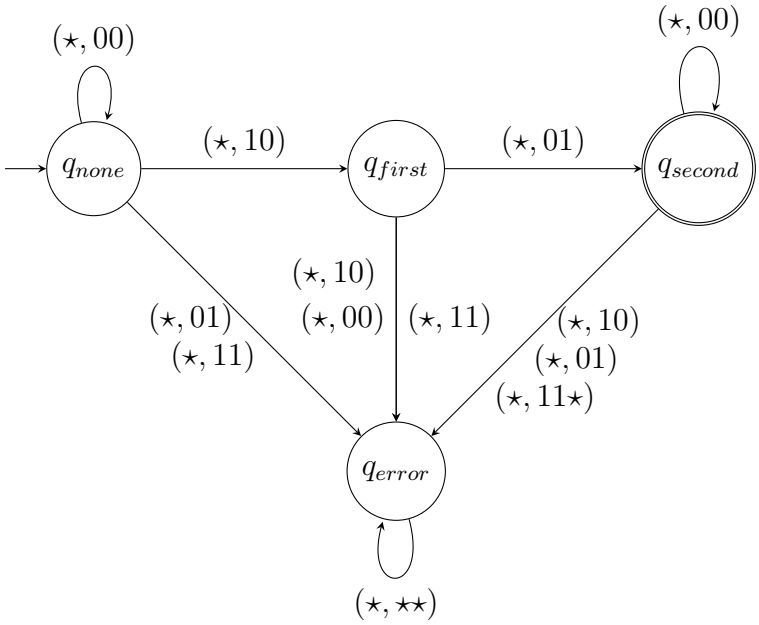


Figura 2.13: Autômato para $S(X, Y)$

Para finalizar os autômatos para fórmulas atômicas, resta apenas construir os autômatos para $X \subseteq Q_0$ e $Y \subseteq Q_0$. Diferentemente de todos os autômatos baseados em fórmulas atômicas para modelos construídos a partir de palavras, esses são os únicos autômatos que necessitam verificar o símbolo do alfabeto lido. Em geral, os autômatos trabalham apenas manipulando a pertinência das posições das palavras aos conjuntos. Isso não ocorre totalmente no caso de autômatos para fórmulas como $X \subseteq Q_0$. Nelas é necessário garantir que toda posição que pertença ao conjunto X, por exemplo, seja rotulada por 0. Observe a

construção apresentada nas Figuras 2.14 e 2.15 que retratam os autômatos para $X \subseteq Q_0$ e $Y \subseteq Q_0$ respectivamente.

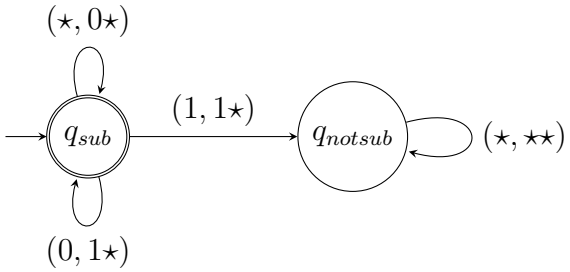


Figura 2.14: Autômato para $X \subseteq Q_0$

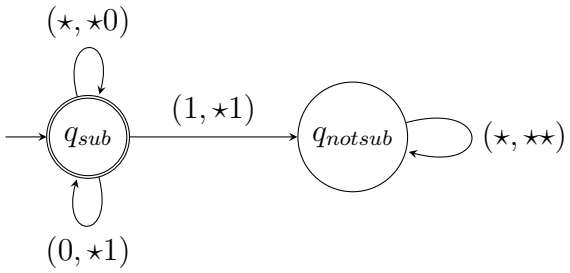


Figura 2.15: Autômato para $Y \subseteq Q_0$

Terminada a construção dos autômatos para as fórmulas atômicas, os autômatos para os símbolos da linguagem lógica (conectivos, no caso a conjunção, e quantificação existencial) são construídos aplicando as propriedades das linguagens regulares sobre estes autômatos. Como mencionado, toda conjunção $\varphi \wedge \psi$ é tratada como a interseção dos autômatos para φ e ψ, nesta ordem. Assim, considere que $A_\varphi = \langle \Sigma, Q, \delta, q_0, F \rangle$ e $A_\psi = \langle \Sigma, Q', \delta', q'_0, F' \rangle$ sejam os autômatos construídos a partir de φ e ψ respectivamente. O autômato para a interseção será denotado por $A_\cap = \langle \Sigma, Q_\cap, \delta_\cap, q_{0\cap}, F_\cap \rangle$ de modo que:

$\boldsymbol{\Sigma}$ é o mesmo alfabeto para cada um dos três autômatos

$$\boldsymbol{Q_\cap = Q \times Q'}$$

$$\boldsymbol{\delta_\cap = \{(sym, (q_1 q'_1), (q_2 q'_2)) \mid (sym, q_1, q_2) \in \delta \wedge (sym, q'_1, q'_2) \in \delta'\}}$$

$$\boldsymbol{q_{0\cap} = (q_0 q'_0)}$$

$$\boldsymbol{F_\cap = \{(q, q') \mid q \in F \wedge q' \in F'\}}$$

A primeira conjunção a ser explorada será $sing(X) \wedge sing(Y)$. A Figura 2.16 exibe graficamente o resultado da interseção dos autômatos para $sing(X)$ e $sing(Y)$. Um detalhe deve ser ressaltado. No autômato obtido pela interseção, muitos estados possuíam

q_{error} como uma das componentes. Esses estados, uma vez que q_{error} transita apenas para si mesmo, formavam um bloco de estados que transitavam entre si, formando assim um ciclo. Portanto, esses estados foram colapsados no estado conhecido como ERRO.

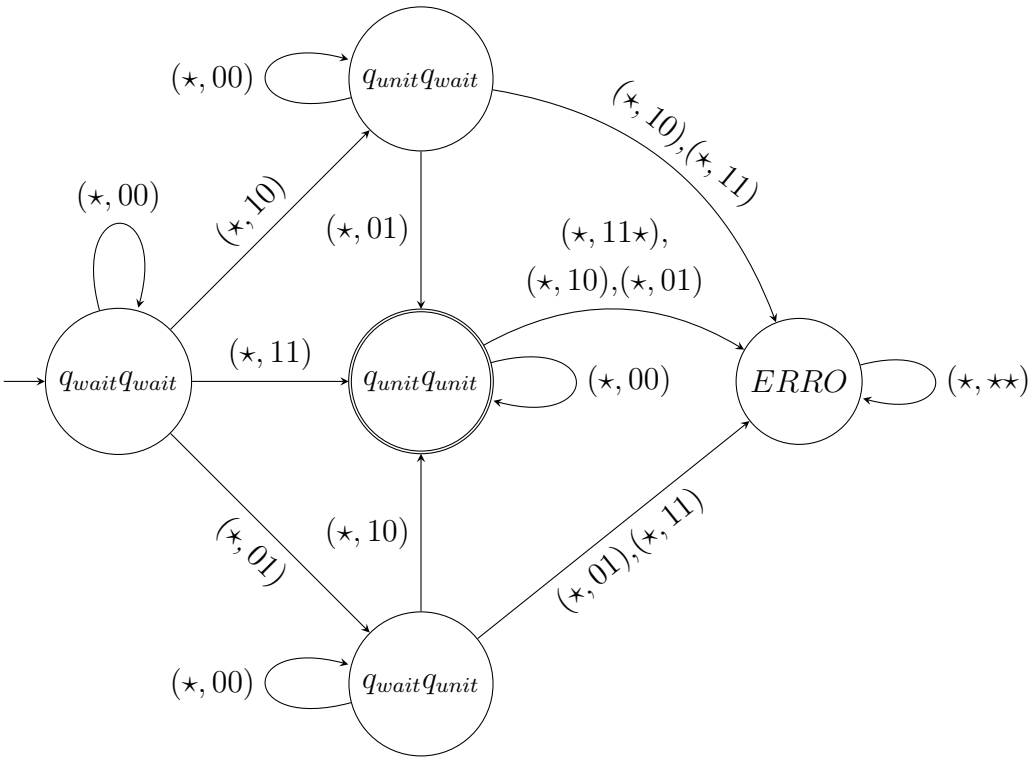


Figura 2.16: Autômato para $sing(X) \wedge sing(Y)$

A próxima conjunção a ser explorada será $(X \subseteq Q_0) \wedge (Y \subseteq Q_0)$, apresentada na Figura 2.17. Similar ao ocorrido para $sing(X) \wedge sing(Y)$, todo estado do produto cartesiano que possuía uma componente q_{notsub} foi colapsado em um estado ERRO.

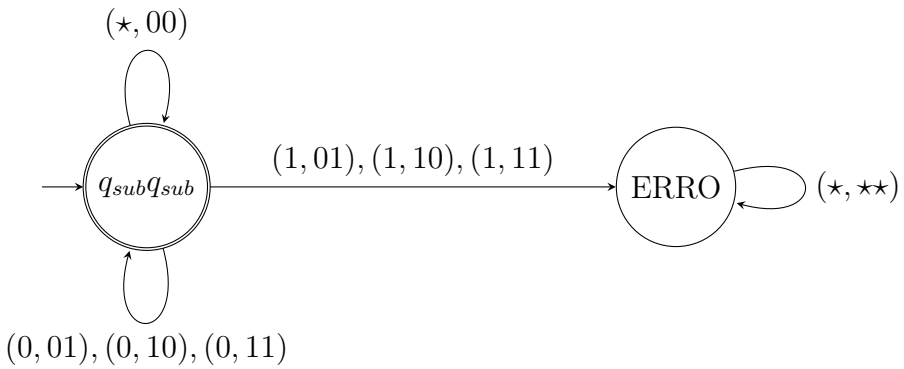


Figura 2.17: Autômato para $(X \subseteq Q_0) \wedge (Y \subseteq Q_0)$

A Figura 2.18 representa graficamente o autômato para $((X \subseteq Q_0) \wedge (Y \subseteq Q_0)) \wedge S(X, Y)$. Na figura, os nomes dos estados representam o produto cartesiano de alguns

estados, com exceção do estado ERRO, o qual é uma síntese de todo estado do produto cartesiano que possui pelo menos um q_{error} como componente. Por exemplo, SSN representa o estado $q_{sub}q_{sub}q_{none}$, que é o estado inicial do autômato.

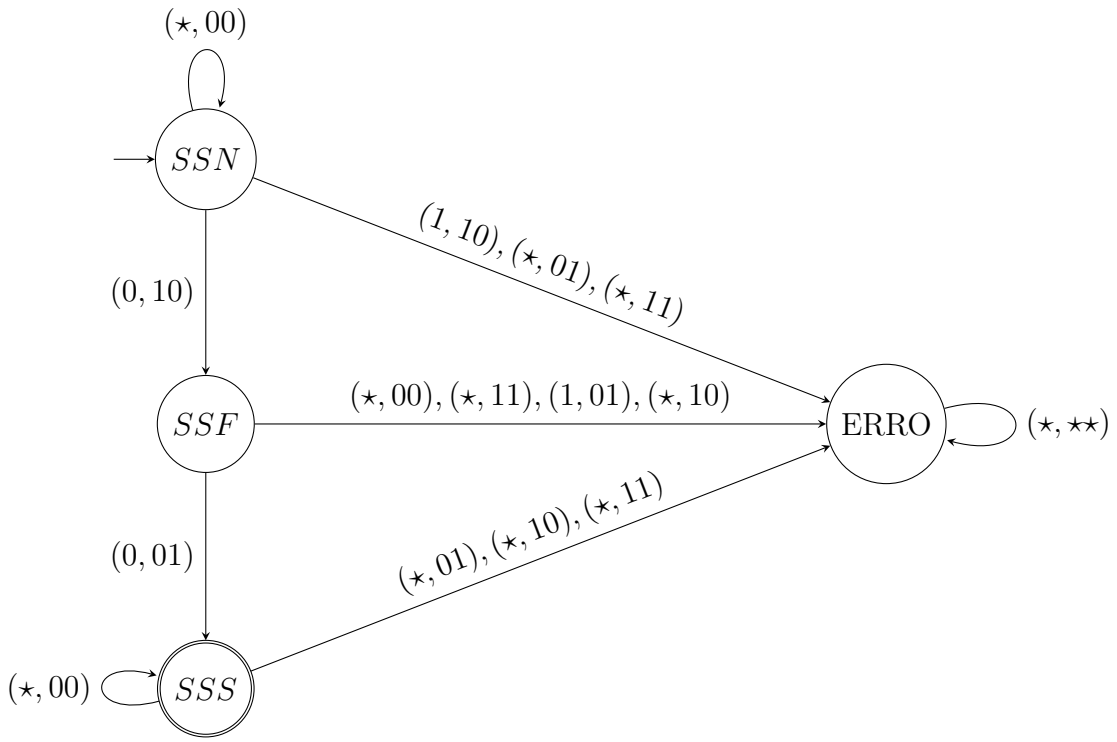


Figura 2.18: Autômato para $((X \subseteq Q_0) \wedge (Y \subseteq Q_0)) \wedge S(X, Y)$

Para finalizar o processo de cálculo de interseções dos autômatos, basta computar a interseção para os autômatos das Figuras 2.16 e 2.18. De modo semelhante ao ocorrido na Figura 2.18, os estados deste autômato também são nomeados de acordo com a primeira letra do subscrito do estado do autômato para a fórmula atômica. Assim, $WWSSN$ significa $q_{wait}q_{wait}q_{sub}q_{sub}q_{none}$. Em particular, o cálculo desta interseção apontou uma série de estados inalcançáveis, como o estado $q_{wait}q_{unit}q_{sub}q_{sub}q_{none}$. A Figura 2.19 mostra o esquema de transições do autômato sem os estados inalcançáveis e a Tabela 2.8 traz a lista completa das transições.

Após obter a interseção de todas as fórmulas atômicas, pela forma normal prenex, basta realizar duas projeções seguidas no autômato para obter o autômato para a fórmula $\exists X.(\exists Y.(sing(X) \wedge sing(Y) \wedge (X \subseteq Q_0) \wedge (Y \subseteq Q_0) \wedge S(X, Y)))$. O intuito principal da operação de projeção é, em alto nível, permitir que o autômato encontre uma atribuição que satisfaça a variável sendo esquecida na projeção. Como mencionado anteriormente, o fato de não serem tratadas fórmulas com variáveis livres e apenas na forma prenex, permite que esta operação seja encarada apenas como a remoção do último bit

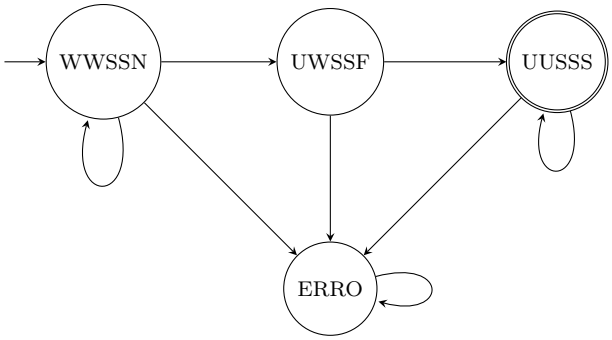


Figura 2.19: Autômato para $sing(X) \wedge sing(Y) \wedge (X \subseteq Q_0) \wedge (Y \subseteq Q_0) \wedge S(X, Y)$

Tabela 2.8: Sistema de transições para $sing(X) \wedge sing(Y) \wedge (X \subseteq Q_0) \wedge (Y \subseteq Q_0) \wedge S(X, Y)$

Símbolo \ Origem	WWSSN	UWSSF	UUSSS	ERRO
0,00	WWSSN	ERRO	UUSSS	ERRO
0,01	ERRO	UUSSS	ERRO	ERRO
0,10	ERRO	ERRO	ERRO	ERRO
0,11	ERRO	ERRO	ERRO	ERRO
1,00	WWSSN	ERRO	UUSSS	ERRO
1,01	ERRO	ERRO	ERRO	ERRO
1,10	ERRO	ERRO	ERRO	ERRO
1,11	ERRO	ERRO	ERRO	ERRO

da *string* de controle. A Tabela 2.9 retrata o resultado da projeção aplicada ao autômato descrito na Tabela 2.8. Observe a existência de não-determinismo no sistema de transições. Desta forma, faz-se necessário aplicar uma rotina para tornar este autômato determinístico para, depois, aplicar novamente a operação de projeção. A Figura 2.20 mostra a versão determinística do autômato da Tabela 2.9. Na figura, os estados I, II, III e IV são definidos como a classe de equivalência $[\{WWSSN\}, \{WWSSN, ERRO\}]$, o conjunto $\{UWSSF, ERRO\}$, o conjunto $\{ERRO\}$ e o conjunto $\{UUSSS, ERRO\}$, respectivamente.

Tabela 2.9: Sistema de transições após a primeira projeção

Símbolo \ Origem	WWSSN	UWSSF	UUSSS	ERRO
0,0	WWSSN, ERRO	UUSS,ERRO	UUSSS,ERRO	ERRO
0,1	ERRO	ERRO	ERRO	ERRO
1,0	WWSSN,ERRO	ERRO	UUSSS,ERRO	ERRO
1,1	ERRO	ERRO	ERRO	ERRO

Considerando o autômato exibido na Figura 2.20, basta aplicar uma nova projeção e tornar o autômato resultante determinístico para que seja possível implementá-lo para validar a propriedade de que uma *string* binária possui dois zeros consecutivos. A Ta-

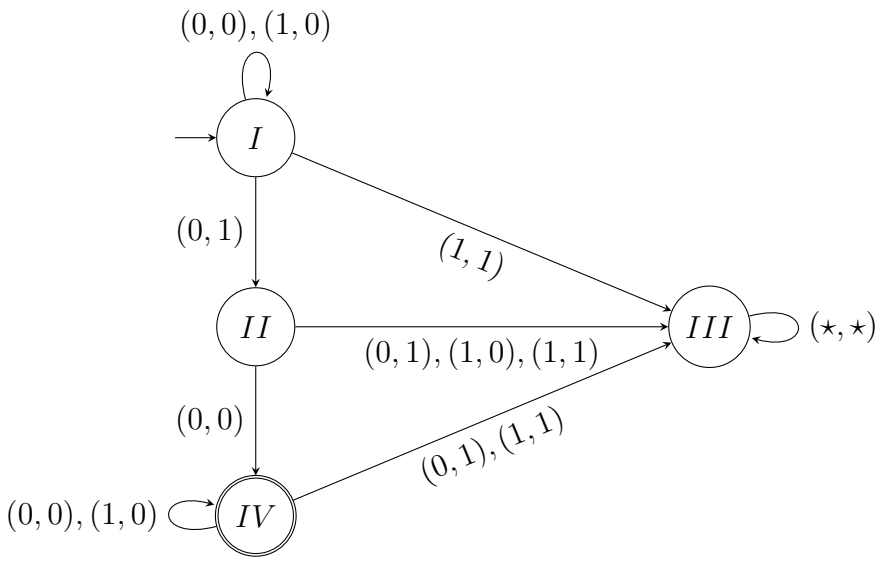


Figura 2.20: Autômato para primeira projeção em $sing(X) \wedge sing(Y) \wedge (X \subseteq Q_0) \wedge (Y \subseteq Q_0) \wedge S(X, Y)$

bela 2.10 mostra o resultado da projeção aplicada ao autômato da Figura 2.20. A Figura 2.21 mostra a versão determinística do autômato da Tabela 2.10, de sorte que os estados A, B, C, D, E e F são $\{I\}$, $\{I, II\}$, $\{I, III\}$, $\{I, II, III\}$, $\{I, II, III, IV\}$ e $\{I, III, IV\}$, respectivamente.

Tabela 2.10: Sistema de transições após a segunda projeção

Símbolo \ Origem	I	II	III	IV
0	I, II	III, IV	III	III, IV
1	I, III	III	III	III, IV

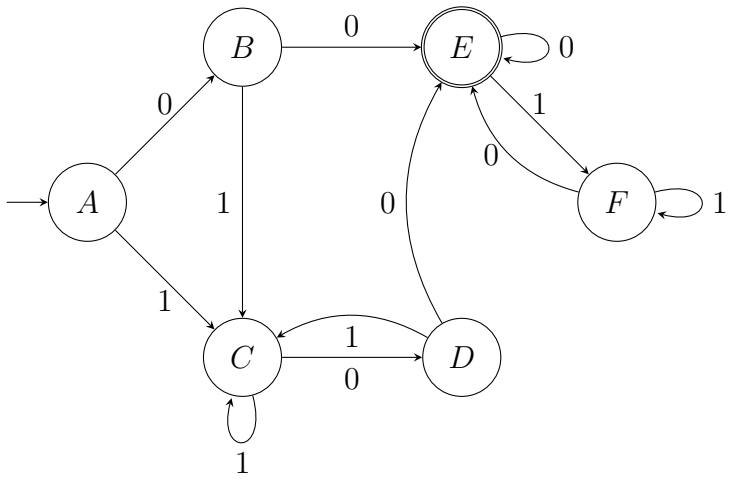


Figura 2.21: Autômato para segunda projeção em $sing(X) \wedge sing(Y) \wedge (X \subseteq Q_0) \wedge (Y \subseteq Q_0) \wedge S(X, Y)$

É necessário ressaltar que o autômato obtido não é necessariamente o autômato mínimo. A garantia dada pelo trabalho de Büchi é de que a linguagem reconhecida por este autômato possui exatamente a propriedade descrita pela fórmula em MSOL a partir da qual ele foi construído. Pode-se verificar que a linguagem aceita é a mesma expressa pela fórmula em MSOL por meio do processo de leitura das palavras desta linguagem pelo autômato. Para encerrar o exemplo, considere as entradas 100100, 1111 e 101000000. Supondo que o autômato da Figura 2.21 receba esses dados como entrada, a sequência de estados que representa a leitura destas entradas é

- 100100 : A, C, D, E, F, E, E. A execução termina no estado E, que é um estado de aceitação. O autômato reconhece a palavra.
- 1111 : A, C, C, C, C. A execução termina no estado C. O autômato não reconhece a palavra.
- 101000000 : A, C, D, C, D, E, E, E, E, E. A execução termina no estado E. O autômato reconhece a palavra.

Capítulo 3

(Hiper)Grafos e Teorema de Courcelle

Dotados de um enorme potencial para modelagem, grafos [11, 38, 13] podem ser utilizados para representar matematicamente uma série de elementos e suas relações. Redes sociais, ruas de uma cidade, linhas de energia, filamentos de DNA, moléculas e fórmulas lógicas são bons exemplos de objetos que podem ser representados através de grafos. Portanto, conhecer algoritmos eficientes para solucionar problemas em grafos pode ser o ponto de partida na busca de soluções para problemas complexos.

O metateorema de Courcelle [19, 21] oferece uma solução eficiente em termos de complexidade parametrizada para problemas representáveis na classe de grafos que possui *treewidth limitada*. A discussão deste capítulo é voltada para os elementos algorítmicos e estruturais utilizados na prova do metateorema de Courcelle: os grafos, apresentados na Seção 3.1, e sua relação com as chamadas *tree decompositions*, apresentadas na Seção 3.2. O capítulo também aborda os passos iniciais do Algoritmo de Courcelle na Seção 3.3, que envolvem a transformação de um problema de decisão em um problema de *model checking* e um pré-processamento da entrada e seus impactos.

3.1 Grafos e Hipergrafos

Um grafo [11, 13] é uma estrutura matemática composta de objetos e uma relação entre esses objetos. Suponha que deseja-se representar moléculas como grafos. Neste caso, os objetos desse grafo representariam os átomos dos elementos químicos e a relação entre eles representariam as ligações químicas. Observando uma rede social, os usuários dessa rede seriam codificados como os objetos enquanto a relação de amizade seria o fator que relaciona estes usuários.

Formalmente, um **grafo** $G = \langle V, E \rangle$ é uma dupla composta por um conjunto V de vértices e uma relação binária $E \subseteq V \times V$, na qual cada par é chamado de aresta. A Figura 3.1 apresenta três exemplos de grafos.

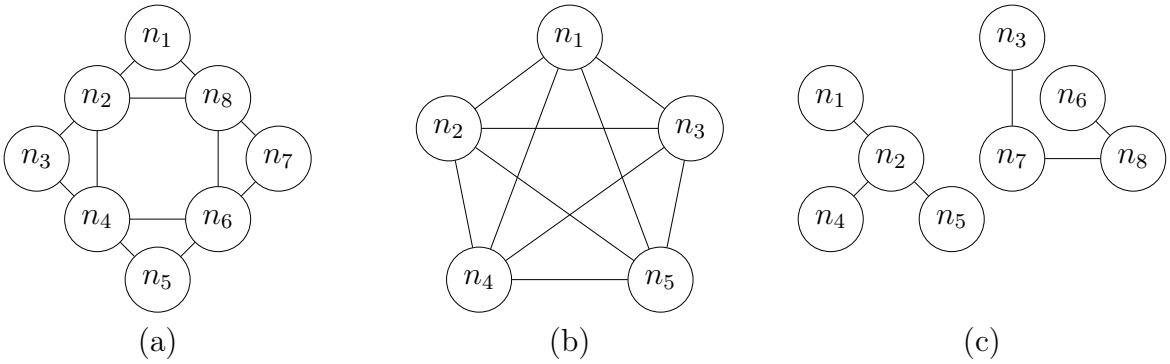


Figura 3.1: Exemplos de grafo.

Esta notação trata as arestas como sendo um elemento definido exclusivamente como pares vértices. Porém, suponha que se deseja poder identificar unicamente as relações entre os vértices. Isso gera o chamado **grafo de incidência**, denotado por $G_{inc} = \langle V \cup E, incidence \rangle$, no qual os objetos em questão são os vértices e as arestas, unicamente identificados de modo adequado, e a relação binária *incidence* associa um determinado vértice como um extremo de uma aresta, ou seja, $incidence \subseteq V \times E$. A Figura 3.2 exibe um grafo G e sua versão em grafo de incidência. Os vértices rotulados por e_i, $1 \leq i \leq 4$, são os vértices que representam as arestas de G.

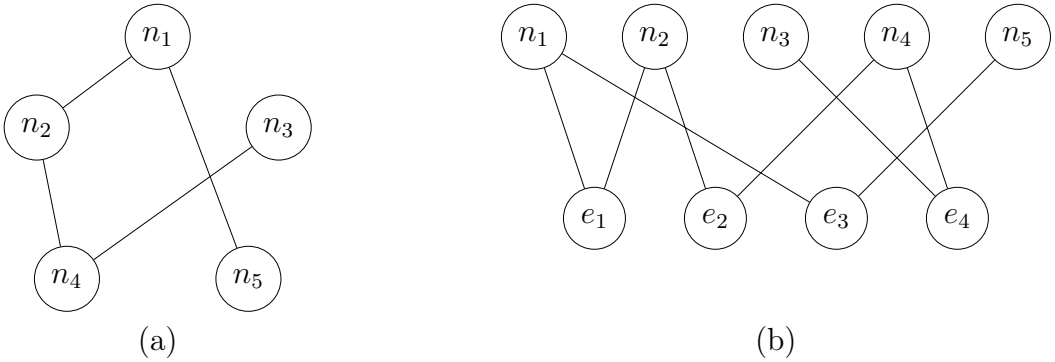
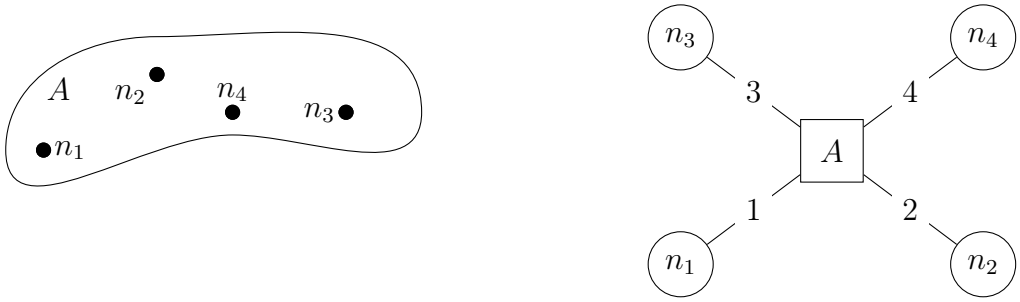


Figura 3.2: Exemplo de um grafo G (a) e o grafo de incidência associado (b).

A noção de aresta do grafo, usualmente concebida como uma relação binária, pode ser estendida para uma relação arbitrária no conjunto de vértices. Esse é o conceito das **hiperarestas** [5, 39], que podem incidir em tantos vértices quanto possível ou mesmo em nenhum deles. A literatura apresenta dois tipos de notação para hiperarestas, que são apresentadas na Figura 3.3. A notação **por campo** [19] assume que a hiperaresta incide

sobre os vértices de maneira não ordenada. Logo, a hiperaresta incide em um conjunto de vértices. A notação de **tentáculos** [30, 39] assume uma ordenação nos vértices incididos, ou seja, toda hiperaresta possui uma sequência de vértices associada a ela.



(a) Campo (b) Tentáculos

Figura 3.3: Notações gráficas para hiperarestas.

A flexibilidade das hiperarestas generalizam o conceito do grafo para os chamados **n-hipergrafos** [5, 30, 19]. A Definição 3.1.1 apresenta o conceito de n-hipergrafo e a Figura 3.4 exibe um exemplo de um n-hipergrafo considerando as hiperarestas como tentáculos. Todo n-hipergrafo é associado a um alfabeto ranqueado, como o apresentado na Seção 2.1, de modo que o *rank* do símbolo que rotula a hiperaresta determina o número de vértices que são incididos pela hiperaresta.

Definição 3.1.1 (n-Hipergrafo)**.** *Seja* $\Sigma_{rank} = \langle \Sigma, rank \rangle$ *um alfabeto ranqueado. Um* ***n-hipergrafo*** *é uma quintupla*

$$\mathcal{H} = \langle V, E, lab, vert, ext \rangle$$

de modo que:

V *é o conjunto de vértices*

E *é o conjunto de arestas*

$lab : E \to \Sigma$ *é a função de rotulação, que associa a cada aresta um símbolo de* Σ_{rank}

$vert : E \to 2^V$ *é a função de incidência, que associa a cada aresta uma sequência de vértices que deve ter a mesma cardinalidade do* rank *do símbolo que rotula essa aresta*

ext *é uma sequência de* n *de vértices destacados de* V*, denominados* ***vértices externos***

O termo hipergrafo é um equivalente ao termo 0-hipergrafo. O hipergrafo da Figura 3.4 é um 3-hipergrafo que é associado ao alfabeto ranqueado $\Sigma_{rank} = \langle (A, B, C, D, E), ((A, 2), (B, 5), (C, 2), (D, 3), (E, 0)) \rangle$ e os vértices externos são $ext = n_4, n_1, n_7$. A utilidade dos vértices destacados ficará evidente ao compreender o funcionamento das *graph grammars* no Capítulo 4. Note também que este hipergrafo possui, além de uma **hiperaresta vazia** (não possui vértices associados), um vértice **isolado**, ou seja, não é incidido por nenhuma (hiper)aresta.

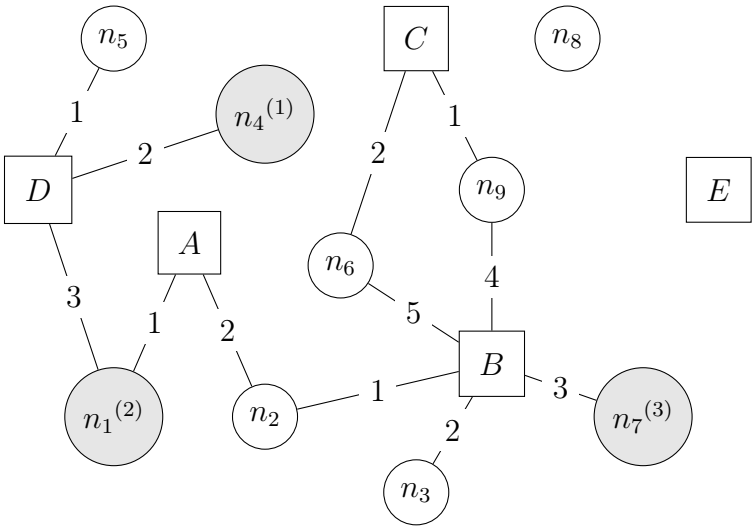


Figura 3.4: Exemplo de hipergrafo.

Alguns autores costumam empregar o termo grafo para se referir a hipergrafos que possuem apenas hiperarestas de tipo 2, ou seja, que incidem em apenas dois vértices. Assim, denota-se por $H(\Sigma_{rank})$ o conjunto de todos os hipergrafos sobre o alfabeto Σ_{rank}. Para falar de um conjunto de hipergrafos que possuam n vértices externos sobre um alfabeto ranqueado, costuma-se utilizar $H_n(\Sigma_{rank})$. Analogamente, as notações $G(\Sigma_{rank})$ e $G_n(\Sigma_{rank})$ são usadas para se referir ao conjunto de todos os grafos e o conjunto de todos os n-grafos sobre o alfabeto Σ_{rank}, respectivamente.

3.2 *Tree decompositions* e *Treewidth*

Em Teoria dos Grafos [11, 38, 13], **árvore** é a nomenclatura utilizada para denominar grafos conexos e acíclicos. Um grafo é dito **conexo** se, e somente se, é possível encontrar um caminho entre qualquer par de vértices. Um **caminho** é uma sequência de vértices adjacentes $v_0 v_1 \ldots v_n$ tal que $v_i \neq v_j$, $\forall i, j \in [0, n]$ e $i \neq j$, e $(v_i, v_{i+1}) \in E(G)$, $0 \leq i < n$. Um grafo é dito **acíclico** se, e somente se, não existe uma sequência de vértices adjacentes $v_0 v_1 \ldots v_n$ tal que $v_0 = v_n$ e $v_0 v_1 \ldots v_{n-1}$ seja um caminho.

Muitos problemas em árvores são resolvidos com algoritmos de tempo polinomial, o que torna essa classe de grafos muito atraente do ponto de vista de obtenção de um bom desempenho. Dado que nem todo grafo é uma árvore, pode-se imaginar que seria interessante uma maneira de codificar grafos que possuem ciclos em uma árvore. Outros questionamentos naturais seriam como realizar e se beneficiar da codificação proposta.

A **treewidth** [49, 51, 28], citada por alguns autores como largura arbórea, é um parâmetro proposto por Robertson e Seymour [49, 50, 51] para quantificar a semelhança de um grafo com uma árvore. Para calcular este parâmetro, é necessário realizar a codificação do grafo de entrada por meio da chamada ***tree decomposition*** [49, 50, 51, 28], formalizada na Definição 3.2.1. A ideia, apesar de muito simples, é bem poderosa. A *tree decomposition* associa a cada vértice de uma árvore um subconjunto de vértices e arestas do grafo G de entrada, de modo a satisfazer algumas condições adicionais. Dada essa decomposição, pode-se projetar algoritmos que visitam cada nó da árvore e exploram soluções parciais candidatas no subgrafo associado ao conjunto em questão. Assim, a obtenção da solução final é condicionada pela composição adequada das soluções parciais obtidas durante a visitação dos vértices da árvore. Apesar de tradicionalmente não ser obrigatório a definição de uma raiz, assuma sem perda de generalidade que a árvore de decomposição é enraizada. A definição de *tree decomposition* pode ser estendida para hipergrafos ao acrescentar a restrição de que todos os vértices externos do hipergrafo devem estar associados à raiz da árvore da decomposição.

Definição 3.2.1 (*Tree decomposition*)**.** *Seja G um grafo. Uma* **tree decomposition** *para G é uma dupla $\mathcal{T} = \langle T, f \rangle$ tal que:*

T é uma árvore enraizada de raiz $r(T)$;

$f : V(T) \to 2^{V(G)}$ é uma função que associa cada nó da árvore T a um subconjunto de vértices de G denominado bag*;*

$V(G) = \bigcup\{f(t) : t \in V(T)\}$, ou seja, todo vértice de G está em algum bag*;*

$\forall uv \in E(G), \exists t \in V(T) : \{u, v\} \subseteq f(t)$, ou seja, dada uma aresta de G, os vértices associados a ela estão contidos em um bag*;*

Sejam $t_1, t_2, t_3 \in V(T)$. Se t_2 pertence ao caminho não direcionado entre t_1 e t_3, então $f(t_1) \cap f(t_3) \subseteq f(t_2)$, ou seja, os nós associados a bags *que contém um dado vértice $v \in V(G)$ induzem uma subárvore de T;*

A mensuração da semelhança do grafo com uma árvore é feita explorando a largura da decomposição. A **largura de uma *tree decomposition***, denotada por $width(\mathcal{T})$, é a cardinalidade máxima de vértices dentre os subgrafos associados aos vértices da árvore menos uma unidade, i.e., $wid(\mathcal{T}) = \max\{f(t) \mid t \in V(T)\} - 1$. Esse decremento é feito para ajustar a definição de modo que as árvores sejam exatamente a classe de grafos conexos com *treewidth* igual a 1. Dado que a decomposição de um grafo não é única, a ***treewidth* de um grafo *G*** é calculada como a largura mínima dentre todas as *tree decompositions* possíveis para o grafo G, ou seja, $twd(G) = \min\{width(\mathcal{T})\}$. A Figura 3.5 apresenta um exemplo de um grafo com ciclos e uma *tree decomposition* para ele. Repare que a decomposição em questão possui largura igual a três.

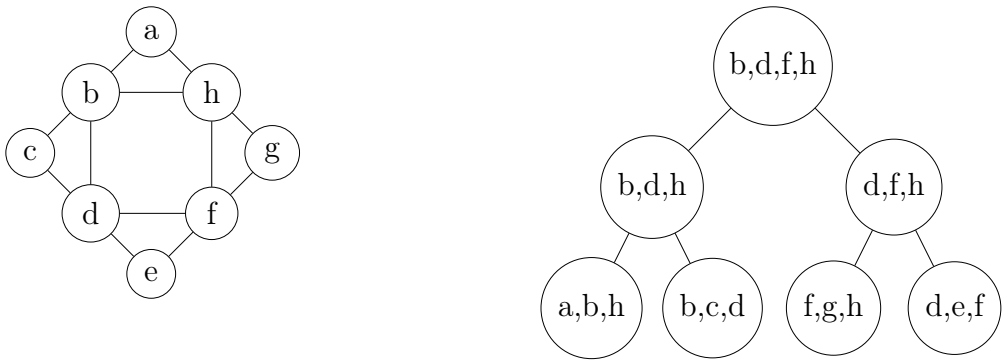


Figura 3.5: Exemplo de um grafo e uma *tree decomposition* para ele.

Outro resultado parametrizado muito importante é o Teorema de Bodlaender [8]. O problema de determinar se um grafo possui uma *tree decomposition* de largura no máximo k é NP-completo. Todavia, Bodlaender demonstra que esse problema pode ser solucionado em tempo **FPT**. Logo, qualquer algoritmo parametrizado pela *treewidth* não terá seu resultado impactado por esse cálculo. Consulte o trabalho de Bodlaender [7] para um *survey* de algoritmos de programação dinâmica baseados em *treewidth*.

Teorema (Bodlaender)**.** *Existe um algoritmo **FPT** que determina se um grafo G possui* treewidth *no máximo k.*

A obtenção de uma *tree decomposition* e cálculo da *treewidth*, apesar de uma etapa relevante para o procedimento de Courcelle, é tratada pelo autor como caixa-preta. Existe uma vasta literatura sobre algoritmos para cálculo de *treewidth*, portanto este tema não será abordado nesta dissertação. Para a resolução desta etapa, sugere-se consultar trabalhos como os de Bodlaender [8, 10, 9], Fomin et. al. [34] e Amir [2].

3.3 Preliminares no algoritmo de Courcelle

Ao rever a prova do teorema apresentada no artigo de 1992 [19], percebe-se que o procedimento proposto pelo autor realiza uma série de transformações de problemas, como mostrado na Figura 3.6.

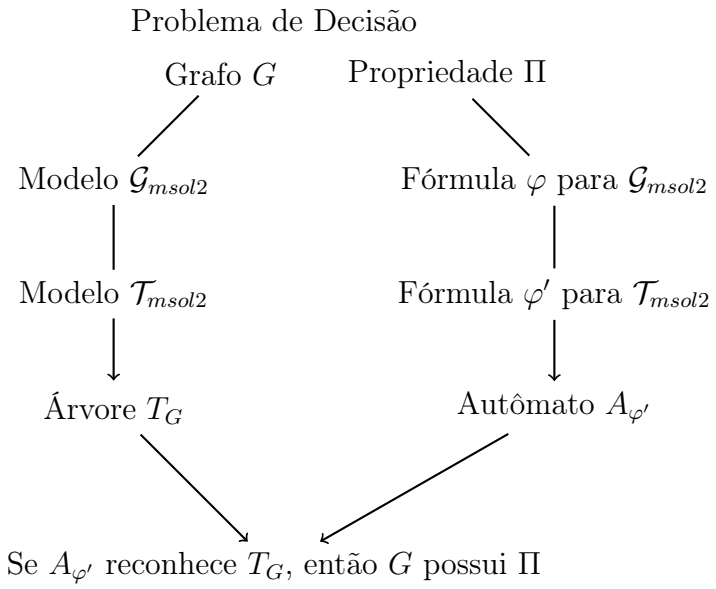


Figura 3.6: Processos de tradução de problemas envolvidos no algoritmo de Courcelle

A primeira etapa do procedimento é uma etapa manual e demanda certa expertise do usuário com as representações lógicas apresentadas na Seção 2.2. O procedimento é iniciado quando o usuário, de posse de um problema de decisão, precisa formalizá-lo em MSOL_2. Em geral essa formalização é bem próximo de um processo natural. O grafo de entrada é substituído pelo seu grafo de incidência. Dessa forma, o novo grafo de entrada possui uma bipartição entre os vértices que representam os vértices de G e os vértices que representam as arestas de G. Assim, o domínio da estrutura é o conjunto de vértices e arestas e inclui-se um predicado de incidência para representar a ligação entre os vértices e as arestas do grafo. Ainda que não possa ser considerada trivial, a construção dessa estrutura e definição de fórmulas é mais acessível para usuários sem um profundo conhecimento de *model checking* e lógica.

A segunda etapa é mais mecânica e depende apenas da escolha de como implementar certos algoritmos. Valendo-se de um resultado parametrizado de Tatcher e Wright [53], Courcelle transforma o problema de *model checking* para grafos, que é mais acessível para os usuários, em um problema de *model checking* para árvores. Essa tradução é feita abordando tanto o modelo como a fórmula a ser verificada. Como visto na Figura 3.6,

traduz-se o problema de *model checking* em termos do grafo denotado por $\mathcal{G}_{msol2}$ para um problema de *model checking* em termos da árvore denotada por $\mathcal{T}_{msol2}$. Uma vez que sintaxe e semântica dos modelos estão fixadas, a tradução da fórmula φ para a fórmula φ' pode ser feita algoritmicamente. Esta etapa é discutida com detalhes no Capítulo 4.

Neste momento, o leitor deve ter em mente apenas que a *tree decomposition* do grafo será processada e transformada em uma árvore, segundo o conceito de linguagens formais. A partir dessa árvore é que o novo modelo será construído. Já a tradução das fórmulas estará relacionada a uma tradução dos predicados utilizados no modelo para grafo em, possivelmente, uma fórmula que utiliza predicados no modelo para a árvore. Esse processo de tradução de fórmulas é discutido em detalhes no Capítulo 5 e é a etapa final do processo. Construídos a representação em árvore e o *tree automaton*, basta executar a leitura da árvore pelo *tree automaton* e receber o resultado.

3.3.1 Formalização de problemas

A primeira etapa envolvida no processo idealizado por Courcelle é a **formalização do problema**. Problemas de decisão usualmente estão descritos em alto nível por meio de linguagem natural. Formalizar o problema significa representá-lo de uma forma matemática. Portanto, para construir essa representação é necessário saber qual linguagem está disponível ao usuário e como ele deve usá-la.

Modelos construídos a partir de grafos $G = \langle V, E\rangle$ usualmente possuem forma $\mathcal{M} = \langle V(G), E^2\rangle$, onde $V(G)$ é igual ao domínio da estrutura e E^2 representa as arestas do grafo. Um adendo a ser feito é o fato de o modelo omitir qualquer símbolo constante necessário para acessar os elementos do domínio. Portanto, para representar propriedades em grafos, minimamente, este é o modelo necessário quando as propriedades são expressíveis em FOL ou em MSOL_1. Se a propriedade em questão envolver MSOL_2, o modelo mínimo para este estudo é $\mathcal{M} = \langle V(G) \cup E(G), incidence^2\rangle$. Ao abordar propriedades MSOL_2 expressíveis, o Teorema de Courcelle não perde expressividade, dado que FOL é um subconjunto que MSOL_1 que é um subconjunto de MSOL_2.

Assim, este trabalho propõe a assinatura $sig = (incidence^2, V^1, E^1, \in^2, \subseteq^2, =^2)$ como a componente sintática das estruturas para problemas em grafos expressos em MSOL_2. Logo, a estrutura completa, baseada em um grafo G, para MSOL_2 é denotada por

$$\mathcal{G}_{MSOL2} = \langle dom(\mathcal{G}_{MSOL2}), incidence, V, E, \in, \subseteq, =\rangle$$

de modo que a interpretação de cada símbolo é dada por:

$\boldsymbol{dom(\mathcal{G}_{MSOL2})} = V(G) \cup E(G)$;

$\boldsymbol{incidence} \subseteq V(G) \times E(G)$;

$\boldsymbol{V} = V(G)$, denotando o conjunto de elementos do domínio que são vértices;

$\boldsymbol{E} = E(G)$, detonando o conjunto de elementos do domínio que são arestas;

$\in$ é um subconjunto de $(V(G) \times 2^{V(G)}) \cup (E(G) \times 2^{E(G)})$;

$\subseteq$ é um subconjunto de $(2^{V(G)} \times 2^{V(G)}) \cup (2^{E(G)} \times 2^{E(G)})$

$=$ é um subconjunto de $(V(G) \times V(G)) \cup (E(G) \times E(G)) \cup (2^{V(G)} \times 2^{V(G)}) \cup (2^{E(G)} \times 2^{E(G)})$.

É importante ressaltar que os símbolos $\in$, $\subseteq$ e $=$ são definidos como

$\boldsymbol{x} \in \boldsymbol{V} \equiv V(x)$

$\boldsymbol{y} \in \boldsymbol{E} \equiv E(y)$

$\boldsymbol{x} \in \boldsymbol{X} \equiv X(x) \wedge \forall x'(X(x') \rightarrow V(x'))$

$\boldsymbol{y} \in \boldsymbol{Y} \equiv Y(y) \wedge \forall y'(Y(y') \rightarrow E(y'))$

$\boldsymbol{X} \subseteq \boldsymbol{V} \equiv \forall x'(X(x') \rightarrow V(x'))$

$\boldsymbol{Y} \subseteq \boldsymbol{E} \equiv \forall y'(Y(y') \rightarrow E(y'))$

$\boldsymbol{X_1} \subseteq \boldsymbol{X_2} \equiv \forall x'(X_1(x') \rightarrow X_2(x'))$

$\boldsymbol{Y_1} \subseteq \boldsymbol{Y_2} \equiv \forall y'(Y_1(y') \rightarrow Y_2(y'))$

$\boldsymbol{X} = \boldsymbol{V} \equiv \forall x'(V(x') \rightarrow X(x'))$

$\boldsymbol{Y} = \boldsymbol{E} \equiv \forall y'(E(y') \rightarrow Y(y'))$

$\boldsymbol{X_1} = \boldsymbol{X_2} \equiv (\forall x'(X_1(x') \rightarrow X_2(x')) \wedge (\forall x'(X_2(x') \rightarrow X_1(x'))$

$\boldsymbol{Y_1} = \boldsymbol{Y_2} \equiv (\forall y'(Y_1(y') \rightarrow Y_2(y')) \wedge (\forall y'(Y_2(y') \rightarrow Y_1(y'))$

para x, x', y e y' representando variáveis para elementos e X, X_1, X_2, Y, Y_1 e Y_2 representando variáveis para conjuntos. O uso de açúcar sintático visa deixar a fórmula mais próxima do alto nível dos problemas de decisão. Dessa forma, usuários não muito habituados à notação lógica podem representar seus problemas sem a necessidade de um conhecimento aprofundado sobre o assunto.

Para exemplificar o processo de formalização, considere o problema de decisão da 3-coloração denotado por

3-coloração
Entrada: Um grafo G.
Questão: É possível particionar os vértices de G em três conjuntos R, G e B tal que não exista aresta entre qualquer par de vértices pertencentes a um mesmo conjunto?

Para formalizar esse problema de decisão como um problema de *model checking*, é necessário construir um modelo usando a linguagem e a interpretação para a estrutura $\mathcal{G}_{MSOL2}$. A questão do problema deve ser traduzida em uma fórmula MSOL$_2$. Para isto, é necessário considerar que a fórmula em questão deve abranger os seguintes tópicos:

- R, G e B são conjuntos de vértices distintos;
- todo vértice do grafo pertence a um desses conjuntos;
- se um vértice pertence a um dos conjuntos, necessariamente ele não pode pertencer aos demais;
- entre vértices de um mesmo conjunto não há uma aresta.

Assim, essas ponderações resultam na seguinte fórmula:

$$\begin{aligned}
&\exists R \exists G \exists B \forall x \forall x' ((x \in V) \wedge (x' \in V) \wedge \neg(R = B) \wedge \neg(R = G) \wedge \neg(B = G) \wedge \\
&((x \in R) \rightarrow (\neg(x \in G) \wedge \neg(x \in B))) \wedge ((x \in G) \rightarrow (\neg(x \in R) \wedge \neg(x \in B))) \wedge \\
&((x \in B) \rightarrow (\neg(x \in G) \wedge \neg(x \in R))) \wedge \neg\exists y((y \in E) \wedge \\
&(((x \in R) \wedge (x' \in R)) \rightarrow (incidence(x, y) \wedge incidence(x', y))) \wedge \\
&(((x \in G) \wedge (x' \in G)) \rightarrow (incidence(x, y) \wedge incidence(x', y))) \wedge \\
&(((x \in B) \wedge (x' \in B)) \rightarrow (incidence(x, y) \wedge incidence(x', y)))))
\end{aligned}$$

É necessário enfatizar que a fórmula apresentada anteriormente não é a que será usada de fato para a execução do *model checking*. Segundo Grohe [36], o problema de *model*

checking de MSOL em qualquer classe de grafos não triviais é PSPACE-Hard. Esse passo de formalização do problema de decisão como um problema de *model checking* em grafos é apenas para tornar o processo mais amigável para os usuários. Pensar em um problema como o apresentado quando o modelo de entrada é uma árvore que codifica um grafo não é uma tarefa trivial. Para essa tradução será apresentado um algoritmo no Capítulo 5.

3.3.2 Processamento de entradas

Em seu artigo de 1992, Courcelle apresenta seus resultados assumindo que a entrada do procedimento é um n-hipergrafo, provando que o resultado pode ser estendido sem perder sua validade. Porém, ainda que a entrada seja um n-hipergrafo, Courcelle apresenta um pré-processamento que reduz o n-hipergrafo a um 0-grafo, que tradicionalmente chamamos apenas de grafo. Esse pré-processamento, chamado de K-map (Definição 3.3.1), apesar de alterar alguns componentes estruturais da entrada, não altera a *treewidth* do n-hipergrafo de entrada.

Definição 3.3.1 (K-map)**.** *Seja $\Sigma_{rank} = \langle \Sigma, rank \rangle$ um alfabeto ranqueado. Constrói-se um novo alfabeto $\Sigma'_{rank} = \langle \Sigma', rank' \rangle$, o qual contém todos os símbolos σ de Σ que possuam $rank(\sigma) = 2$, também chamados de símbolos de rank 2. Novos símbolos de rank 2 podem ser acrescentados a Σ'_{rank} de acordo com a necessidade. Logo, assume-se $\Sigma'_{rank} \neq \emptyset$.*

Assim,

$$K : H(\Sigma_{rank})_n \to H(\Sigma'_{rank})_0$$

onde $H(\Sigma_{rank})_n$ e $H(\Sigma'_{rank})_0$ representam a classe dos n-hipergrafos definidos sobre Σ_{rank} e a classe dos 0-hipergrafos definidos sobre Σ'_{rank}, respectivamente.

Para todo $G \in H(\Sigma_{rank})_n$ constrói-se um $\bar{K}(G) \in G(\Sigma'_{rank})_m$ utilizando duas operações:

1. *Eliminar todas as hiperarestas de rótulo com* rank 0 *e* rank 1*, mas manter os vértices associados a hiperarestas de* rank 1*.*
2. *Substituir hiperarestas com rótulos de* rank *maior ou igual a 3 por grafos completos.*

Note que as operações acima podem eliminar alguns vértices marcados, por isso dize-se que $\bar{K}(G) \in G(\Sigma'_{rank})_m$. Assim, $\bar{K}(G)$ é um m-grafo, pois só restaram hiperarestas que incidem em pares de vértices. O processo de geração de $K(G) \in G(\Sigma'_{rank})_0$ a partir de $\bar{K}(G)$ se dá ao colocar uma aresta entre todo par de vértices marcados, gerando uma clique.

Além da definição do K-map, Courcelle prova que a *treewidth* do grafo é preservada, portanto, $twd(H) = twd(\bar{K}(H)) = twd(K(H))$, sendo H um n-hipergrafo tomado como entrada. É natural sentir algum desconforto com a alteração da estrutura do hipergrafo de entrada e aparentar não existir nenhum impacto na fórmula que expressa a propriedade de entrada. Na verdade, há sim um impacto na fórmula, já que a estrutura foi alterada. Por exemplo, suponha que o problema estudado seja o de encontrar um conjunto independente no grafo. Um **conjunto independente** é um subconjunto de vértices de um grafo tal que não exista uma aresta entre qualquer par de vértices do conjunto no grafo. É notório que remover vértices isolados da entrada podem impactar no resultado da avaliação da propriedade, dado que esses vértices são naturalmente parte da solução. Logo, para se manter a consistência do processo é sim necessário alterar a propriedade a ser verificada. Entretanto, as alterações a serem feitas serão relacionadas à quantidade de vértices ou arestas que serão descartadas por serem parte natural da solução ou por não impactarem a solução. Assim, a expressividade da propriedade a ser verificada não é alterada. Uma vez que não há alteração de expressividade, o processo enunciado por Courcelle pode transcorrer normalmente, a menos de ajustes na propriedade desejada e, por consequência, na fórmula a ser verificada. Mediante essa informação, esta dissertação pode prosseguir normalmente considerando que a entrada a ser recebida é um grafo, não necessariamente conexo, mas qualquer que seja a quantidade de componentes conexas no grafo, elas sempre serão pelo menos um par de vértices ligado por uma aresta.

Capítulo 4

Grafos codificados em árvores

A segunda etapa essencial no processo elucidado por Courcelle refere-se à transformação do problema de *model checking* para grafos em um problema de *model checking* para árvores. Esta tradução vai ao encontro do resultado do trabalho de Thatcher e Wright [53], que mostrou que o problema de *model checking* parametrizado pelo tamanho da fórmula de entrada é solucionável em tempo **FPT** na classe das árvores.

Teorema (Büchi (1960) / Thatcher, Wright (1968))**.** Model Checking *para Lógica Monádica de Segunda Ordem é solucionável em tempo*

$$f(\kappa) \times n$$

para alguma função computável f nas classes de estruturas construídas a partir de palavras ou árvores, quando κ é o tamanho da fórmula e n é o tamanho do modelo.

Courcelle então propõe uma forma de codificar o grafo de entrada em uma árvore para garantir a solução por meio do resultado de Thatcher e Wright. O processo iniciado com a obtenção de uma *tree decomposition* para o grafo de entrada é seguido de uma etapa de codificação desta decomposição em uma árvore, o que representa a tradução do modelo construído a partir de um grafo para um modelo construído a partir de uma árvore no problema de *model checking*.

Este capítulo apresenta na Seção 4.1 o conceito das *graph grammars*, que são uma categoria de gramáticas formais para gerar grafos. O estudo das *graph grammars* e as operações executadas nos hipergrafos levam à proposta de uma álgebra sobre hipergrafos, que pode ser vista como uma *tree language*. Este assunto é discutido na Seção 4.2. Codificando grafos por meio desta álgebra, propõe-se então um modelo para o problema de *model checking* para árvores na Seção 4.3, ressaltando as relações entre o modelo

proposto e o modelo para grafos apresentado no Capítulo 3. O capítulo se encerra na Seção 4.4 discutindo o método para construir então uma árvore que codifique os grafos de entrada dos problemas tratados pelo Teorema de Courcelle.

4.1 Gramáticas de grafos

Assim como palavras e árvores podem ser estudadas sobre a ótica das linguagens formais, grafos também podem ser estudados utilizando este *framework*. Grafos são gerados pelas *graph grammars*, que possuem uma sutil diferença em relação às gramáticas para palavras e *tree grammars*. Para o caso das palavras, a aplicação de regras de produção sobre os símbolos não-terminais é tal que o resultado sempre seja uma palavra, que pode conter símbolos terminais e não-terminais (caso a derivação não tenha terminado). Com as árvores, os símbolos não-terminais, quando presentes, ocupam as folhas da árvore para que, ao aplicar uma produção, seja possível garantir a conexidade e a aciclicidade do objeto gerado. Ao trabalhar com grafos, de uma maneira geral, não há nenhuma necessidade em manter a linearidade, como para as palavras, ou a conexidade e a aciclicidade, como para as árvores. Assim, grafos podem ser gerados segundo as abordagens de substituição de vértices e a de substituição de hiperarestas.

Uma *graph grammar* que utilize a abordagem de substituição de vértices é denominada ***node replacement graph grammar***, ou simplesmente **NRG**. Nessas gramáticas, os símbolos não terminais serão utilizados para rotular vértices. Ao aplicar uma produção sobre este símbolo, as arestas incidentes ao vértice e o próprio vértice são removidos do grafo. Após a remoção, o lado direito da produção é então unido de forma disjunta ao grafo resultante da remoção do vértice. Por fim, dado um conjunto de regras próprio de cada gramática, o grafo resultante da remoção e o recém adicionado serão unidos por meio de arestas.

As ***hyperedge replacement graph grammars*** (**HRG**) são *graph grammars* que abordam a geração de grafos por meio da substituição das hiperarestas. Os símbolos não terminais da gramática rotulam hiperarestas que são removidas ao aplicar uma produção, porém mantendo no grafo os vértices incididos por esta hiperaresta. O lado direito da regra de produção é unido de forma disjunta com o grafo resultante da remoção. Por último, alguns vértices do grafo recém adicionado e os vértices antigamente incididos pela aresta removida são fundidos, dadas as regras da **HRG** considerada.

O uso de hiperarestas denotadas como tentáculos simplifica o processo de derivação

de grafos por meio das **HRG**. Nesse caso, a regra de fusão dos vértices é feito por meio da posição do vértice na sequência de vértices externos e do tentáculo incidindo sobre os vértices. Matematicamente falando, a regra para a fusão dos vértices é da forma $ext_H[i] = vert(e)[i]$, $1 \leq i \leq n$, para uma hiperaresta e e um n-hipergrafo H com $n = rank(lab(e))$. Isso quer dizer que, ao substituir uma hiperaresta e por um n-hipergrafo H, o i-ésimo vértice externo de H deve ser fundido com o vértice incidido pelo i-ésimo tentáculo de e.

A Definição 4.1.1 apresenta formalmente uma **HGR**. Estas gramáticas possuem aspectos semelhantes aos das gramáticas de palavras e árvores, mas a diferença essencial está no símbolo de início. O símbolo de início de uma **HRG** é o chamado *handle*. Um *handle* é um hipergrafo que possui uma única hiperaresta rotulada por um símbolo não-terminal e todo vértice incidido por essa hiperaresta é um nó externo desse hipergrafo. Usualmente, denota-se por $(S, n)^\bullet$ o *handle* cuja hiperaresta é rotulada por S e incide em n vértices. O rótulo dessa hiperaresta é um símbolo não terminal. Assim, denota-se por $\mathcal{H}_C$ a classe dos n-hipergrafos que possuem arestas rotuladas por C, sendo que $C = N \cup T$, onde N é um conjunto de rótulos não terminais e T é um conjunto de rótulos terminais. O processo de derivação utilizando uma **HRG** é terminado quando o grafo construído não possui nenhuma hiperaresta rotulada por um símbolo não terminal.

Definição 4.1.1 (Gramática **HRG**). *Considere um conjunto de rótulos C. Uma* **hyperedge replacement graph gammar** ***(HRG)*** *é uma quádrupla $\mathcal{G}_{HRG} = \langle N, T, P, Z \rangle$ onde:*

N, $N \subseteq C$, é um conjunto de rótulos não terminais;

T, $T \subseteq C$, é um conjunto de rótulos terminais;

P é um conjunto de regras de produção da forma $\alpha \rightarrow \beta$, onde $\alpha \in N$ e $\beta \in \mathcal{H}_C$ tais que $|vert(lab(\alpha))| = |ext_\beta|$;

Z é um handle.

O processo de geração dos grafos por meio das **HRG**'s, independente das regras de produção da gramática considerada, segue um padrão muito claro de operações sobre hipergrafos. Este padrão de operações é apresentado por Drewes [31]. A primeira dessas operações é a **remoção de uma hiperaresta**. Esta operação apenas apaga uma hiperaresta e não remove os vértices associados a ela. A chamada **união disjunta** recebe um n-hipergafo e um m-hipergrafo como entrada e retorna um (n+m)-hipergrafo. Assumindo

que os hipergrafos não possuem interseção no conjunto de vértices ou arestas, o hipergrafo retornado é um hipergrafo desconexo. Para conectar grafos disjuntos é necessário utilizar a operação de **fusão de nós**. Esta operação utiliza um mapeamento (função) para identificar vértices em diferentes componentes conexas do hipergrafo com o objetivo de torná-lo conexo. Essa é a ideia de "colagem" de vértices por muitas vezes mencionada por Courcelle. Após a aplicação desta operação, o novo hipergrafo conexo pode ter mais do que o conjunto de vértices externos originais. Para corrigir esse efeito utiliza-se a operação de **redefinição de nós**. Utilizando outro mapeamento, a operação pode incluir ou excluir vértices do conjunto de vértices externos.

4.2 *Tree language* para gramáticas de grafo

Autores como Drewes [31], Habel [39] e Engelfriet [32] explicam de diferentes formas as operações principais encapsuladas nas *graph grammars*. O trabalho de Mezei e Wright [48] apresentou um caminho para a caracterização algébrica de conjuntos livres de contexto, o que impacta diretamente no uso das **HRG**'s que são tidas como gramáticas livres de contexto. Baseando-se no trabalho de Mezei e Wright, Bauderon e Courcelle [4] propõem a primeira abordagem baseada em RTG para *graph grammars* livres de contexto.

Os escritos de Courcelle costumam trazer o termo *source* para se referir aos vértices marcados nos hipergrafos. Nesta dissertação, o termo vértice externo é equivalente ao *source* de Courcelle. A adoção da mesma nomenclatura de Drewes e Habel para os vértices marcados visa evitar vícios relacionados ao termo *source*. Não há nenhuma noção de fluxo envolvida neste processo. Em seu artigo de 1992, Courcelle [19] apresenta a seguinte álgebra para hipergrafos:

$\boldsymbol{n}$ é a constante que representa um hipergrafo sem hiperarestas e com n vértices, todos marcados como externos;

$\boldsymbol{a}$ é a constante que representa um hipergrafo com n vértices, todos eles marcados como externos, e uma única hiperaresta rotulada por a que incide em todos os vértices do hipergrafo;

$\oplus_{\boldsymbol{n,m}}$ é a operação binária de união disjunta de um hipergrafo com n vértices externos e um hipergrafo com m vértices externos, assumindo estes hipergrafos não possuam nenhum vértice ou aresta em comum;

$\boldsymbol{\sigma_\alpha}$ é a operação unária de redefinição dos vértices externos do hipergrafo, segundo o mapeamento α que pode aumentar ou diminuir a sequência de vértices externos;

$\boldsymbol{\theta_\delta}$ é a operação unária de fusão de vértices externos do hipergrafo segundo o mapeamento δ, que possivelmente colapsa vértices gerando classes de equivalência como vértices do grafo.

A álgebra apresentada causa uma explosão combinatória no conjunto de símbolos terminais da RTG por conta das operações dependentes de relações. Para estas operações, para cada possível relação sobre o conjunto de vértices do hipergrafo, seria necessário um símbolo que a identificasse unicamente. Como alternativa simplificada à álgebra de 1992, em seu livro de 2012, Courcelle [21] apresenta uma versão revisada de sua álgebra utilizando um conjunto de marcações. As **marcações**, denotadas por $marked = \{a, b, \dots\}$, são um conjunto de símbolos usados para destacar os vértices externos do grafo. A álgebra revisada é denotada por:

0 é a constante que representa um hipergrafo vazio.

1 é a constante que representa um hipergrafo com um único vértice que não é externo.

$\boldsymbol{a^l}$ é a constante que representa um hipergrafo composto de um vértice externo marcado com a, $a \in marked$, e um *loop* neste vértice;

$\boldsymbol{ab}$ é a constante que denota um hipergrafo composto por dois vértices externos marcados com a e b, $a, b \in marked$, e uma hiperaresta, denotada por ab conectando os vértices;

$\boldsymbol{//}$ é a operação binária de composição paralela sobre hipergrafos. Esta operação realiza, inicialmente, uma união disjunta dos hipergrafos. Caso existam vértices nos hipergrafos que possuam uma mesma marcação, estes vértices serão fundidos em um só, mantendo sua marcação;

$\boldsymbol{fg_a}$ é a operação unária de esquecimento de um vértice externo. Este esquecimento é representado por meio da remoção da marcação a, $a \in marked$, utilizada no vértice;

$\boldsymbol{ren_{a \to b}}$ é a operação unária de substituição de marcações. Sejam $a, b \in marked$. O símbolo $ren_{a \to b}$ substitui toda marcação a por b, ou seja, todo vértice marcado com a passa a ser marcado com b.

Observando a versão sintetizada da álgebra para **HRG**'s e as características do grafo resultante do pré-processamento proposto por Courcelle, e apresentado no Capítulo 3, é

necessário destacar alguns pontos. Dado que nenhuma componente conexa do grafo de entrada é um vértice isolado, por conta do K-map, a inclusão de um símbolo para um grafo composto por um vértice isolado torna-se supérflua. Dado que *loops* não ocorrem, a rigor, em grafos, um símbolo que represente *loops* também parece supérfluo. Por codificar o grafo de entrada utilizando uma *tree decomposition*, tem-se que o número máximo de vértices manipulados por nó da árvore é $k+1$, dado que a *treewidth* do grafo é igual a k. Portanto, considerar estes vértices como vértices externos não é um problema. O controle desse conjunto de marcações é exatamente a propriedade desejada, pois irá gerar um alfabeto quadrático na *treewidth* do grafo. Assim, a linguagem proposta para este trabalho consiste apenas nos símbolos $//$, fg_a e ab, para $a, b \in marked$ onde $|marked| = k + 1$. A remoção dos símbolos 0 e $ren_{a \to b}$ tem razões algorítmicas e ficarão mais claras na Seção 4.4. A princípio o leitor deve ter em mente que o símbolo 0 também pode ser descartado sem causar danos à representação dos grafos de entrada. Já a eliminação do símbolo $ren_{a \to b}$ ocorre pela garantia de que existe uma atribuição de marcações aos vértices que dispensa correções, a qual será apresentada na Subseção 4.4.2.

4.3 Tradução de modelos

Considere a *tree language* sobre o alfabeto $\Sigma_{HR} = \langle \{//, (fg_a)_{a \in marked}, (ab)_{a,b \in marked}\}, \{(//, 2), ((fg_a)_{a \in marked}, 1), ((ab)_{a,b \in marked}, 0)\}\rangle$ como sendo a ***tree language* HR**. Árvores que pertencem à *tree language* HR serão chamadas de ***tree expressions***.

As *tree expressions* serão a forma de codificar o grafo de entrada como uma árvore. Para tornar este processo mais natural, Courcelle sugere a utilização das *tree decompositions*, principalmente para conseguir controlar a quantidade de informação manipulada por vez, sem causar uma explosão combinatória. Assim, o problema de *model checking* para árvores a ser utilizado consiste em construir um modelo a partir de uma *tree expression* e traduzir a fórmula formalizada no modelo para grafos em uma fórmula no modelo para *tree expressions*.

O modelo, construído a partir de uma árvore, apresentado na Seção 2.4 não se mostra adequado para lidar com *tree expressions* por conta das particularidades delas. Em uma *tree expression* os símbolos de *rank* 0 são os símbolos que representam as arestas do grafo. Portanto, dado que certas propriedades consistem em verificar informações ligadas às arestas, independente de qual seja o rótulo dela, não há sentido em gerar uma série de conjuntos que diferencie estas arestas.

Para lidar com as particularidades da *tree language* HR, este trabalho propõe a assinatura $sig = (<^2, \subseteq^2, sing^1, leaves^1, forgNodes^1, =^2, inc^2)$ como componente sintático para os modelos baseados em árvores que codificam grafos para problemas em MSOL_2. Assim, o modelo proposto é denotado por

$$\mathcal{T}_{MSOL2} = \langle dom(\mathcal{T}_{MSOL2}), <, \subseteq, sing, leaves, forgNodes, =, inc \rangle$$

onde:

$\boldsymbol{dom(\mathcal{T}_{MSOL2})}$ é o conjunto das posições dos nós da árvore codificados por meio de palavras de $\{0,1\}^*$. Observe que a *tree language* HR não é estritamente binária. Nesse caso, filhos de nós que só possuem um único filho serão codificados de modo semelhante à codificação do filho esquerdo de um nó que possui dois filhos.

$<$ é a relação binária definida sobre conjuntos unitários, de modo que o nó contido no primeiro conjunto é um ancestral do nó contido no segundo conjunto. Denota-se esse predicado por $X < Y$, onde $X = \{i\}$ e $Y = \{j\}$ tal que $j = iw$, onde $w \in \{0,1\}^*$ e $w \neq \epsilon$.

$\subseteq$ é a relação de inclusão de conjuntos, denotada por $X \subseteq Y$.

sing é o predicado que testa se um determinado conjunto é unitário, denotado como $sing(X)$.

leaves é o conjunto de todas as posições da árvore que são folhas.

forgNodes é o conjunto de todas as posições da árvore que são rotuladas por algum símbolo fg_a, $a \in marked$.

$=$ é a relação de igualdade entre conjuntos definidos sobre o domínio, denotada por $X = Y$.

inc é a relação binária entre conjuntos unitários tal que o elemento do primeiro conjunto seja posição da árvore rotulada com algum símbolo fg_a, $a \in marked$ e o elemento do segundo conjunto seja uma folha da árvore.

A existência do predicado $forgNodes$ é uma consequência da forma como a *tree expression* é construída. Enquanto as folhas representam as arestas do grafo de entrada, os nós para esquecimento representarão os vértices. Este modelo deve ser construído após

a obtenção da *tree expression* e dele também serão retiradas as informações para traduzir as fórmulas do modelo para grafos para fórmulas no modelo para árvores.

4.4 Algoritmo de codificação de *tree decompositions*

O método geral para algoritmos baseados em programação dinâmica sobre *tree decompositions* consiste em explorar os nós da decomposição, gerando soluções candidatas a cada nó visitado e passando adiante as melhores soluções parciais. Deste modo, a melhor solução é computada de forma fracionada através dos nós da árvore.

No artigo de 1992, Courcelle apresenta uma primeira estratégia para a geração da *tree expression* usando o conceito de operação combinada. Em particular, ele executava um algoritmo de binarização na *tree decomposition* e, após concluir esta etapa, ia aplicando recursivamente uma operação composta pelas operações definidas em sua primeira versão da álgebra para *graph grammars*. Em seu livro de 2012, Courcelle apresenta uma técnica para a codificação por meio da *tree expression* usando uma variação da *tree decomposition* na qual cada aresta do grafo só pode estar associada a um e apenas um nó da árvore. De modo a tornar este algoritmo mais atual, uma das propostas desta dissertação é utilizar as *extended nice tree decompositions* para a realização desse processo.

4.4.1 Codificação via *extended nice tree decomposition*

Uma ***extended nice tree decomposition*** [27] é uma *tree decomposition* que, além das restrições usuais apresentadas na Definição 3.2.1, possui seus nós divididos entre as seguintes categorias:

Nó folha : nó que não possui filhos. Em geral, não há informações sobre vértices associadas a esses nós.

Nó de inserção de vértice : nó que possui um único filho. Seja t um nó de inserção de vértice e t' o filho de t. Assim, $f(t') = f(t) \backslash \{v\}$, para algum $v \in V(G)$ onde G é o grafo de entrada.

Nó de inserção de aresta : nó que possui um único filho. Estes nós inserem uma aresta uv ao grafo associado ao nó, de sorte que $\{u, v\} \in f(t)$ onde t é um nó de inserção de aresta.

Nó de esquecimento : nó que possui um único filho. Seja t um nó de esquecimento e t' o filho de t. Assim, $f(t) = f(t')\backslash\{v\}$, para algum $v \in V(G)$ onde G é o grafo de entrada.

Nó de junção : nó que possui dois filhos e representam a união de soluções parciais construídas em diferentes ramos da árvore. Sejam t, t' e t'' nós da árvore. Se t é um nó de junção e t' e t'' são filhos de t, então os *bags* associados a t, t' e t'' são iguais, i.e., $f(t) = f(t') = f(t'')$.

O que torna as *extended nice tree decompositions* atraentes para o processo de geração das *tree expressions* é a distinção que ela faz dos nós que inserem uma aresta. Saber o momento de inserir a aresta e qual aresta está sendo inserida evita que muitas operações sejam realizadas para identificar o símbolo que representa adequadamente tal aresta do grafo de entrada.

Assim, a *tree expression* associada ao grafo G, denotada por $t\text{-}exp(G)$, é igual à *tree expression* associada à raiz da árvore de decomposição de G, denotada por $exp(r(T))$. O algoritmo para encontrar a *tree expression* associada à raiz da *extended nice tree decomposition* é baseado em percurso em pós ordem da árvore de decomposição, de modo que:

- Se o nó corrente t é uma folha, faça $exp(t) = 0$.
- Se o nó corrente t é do tipo inserção de vértice, faça $exp(t) = exp(t')$ para t' filho de t.
- Se o nó corrente t é do tipo inserção de aresta, faça $exp(t) = //[ab, exp(t')]$ para t' filho de t e ab o símbolo que representa a aresta inserida, cujas extremidades estão marcadas com a e b, para quaisquer $a, b \in marked$.
- Se o nó corrente t é do tipo esquecimento, faça $exp(t) = fg_a[exp(t')]$ para t' filho de t e a a marcação associada ao vértice sendo esquecido na decomposição.
- Se o nó corrente t é um nó de junção, faça $exp(t) = //[exp(t'), exp(t'')]$ para t' e t'' filhos de t.

Neste algoritmo, alguns pontos importantes devem ser ressaltados. O primeiro deles se refere ao símbolo para grafo vazio, o 0. Ao final da Seção 4.2 foi dito que o símbolo para grafo vazio poderia ser eliminado sem representar nenhuma perda de expressividade

ao algoritmo. Isso acontece porque 0 é o elemento neutro da operação de composição paralela. Aqui há duas possibilidades de implementação. A primeira é considerar que as folhas contêm um único vértice e, ao atingir uma folha, nada é processado. Neste cenário, a *tree expression* só começa a ser computada quando a primeira aresta é inserida em um dado ramo da árvore de decomposição. Portanto, ao programar esta estratégia, é necessário tomar cuidado com os retornos da função. A segunda estratégia consiste em computar a *tree expression* considerando o símbolo para grafo vazio e, após terminada a computação, realizar um procedimento de limpeza dessa *tree expression*. O procedimento de limpeza deve, em uma única passada, substituir todas as partes da *tree expression* que sejam da forma $//[tree, 0]$ ou $//[0, tree]$ apenas pela árvore denotada por $tree$.

Outro ponto importante a ser relatado é o uso das operações de esquecimento de marcação, denotadas por fg_a, $\forall a \in marked$. Pela estrutura das *tree decompositions*, todo vértice do grafo é esquecido uma única vez. Portanto, o número de símbolos de esquecimento na *tree expression* é exatamente a cardinalidade do conjunto de vértices do grafo. Assim como se identificam as arestas do grafo por meio dos símbolos contidos nas folhas, os vértices do grafo podem ser identificados por meio dos símbolos de esquecimento.

Observando atentamente o algoritmo descrito é possível observar uma limitação que pode gerar problemas a cada aplicação da operação de composição paralela, principalmente nos nós de junção. Esta limitação pode ser resolvida por meio de uma fixação prévia das marcações dos vértices do grafo. Para isto, este trabalho apresenta um procedimento, não relatado por Courcelle, para garantir uma construção consistente das *tree expressions*.

4.4.2 Uma estratégia baseada em coloração

Os trabalhos de Courcelle não tratam da construção dos autômatos para os predicados adicionados à MSOL_2, principalmente por não definir um padrão de modelo a ser seguido. Entretanto, para formalizar o algoritmo completamente, além de definir tal modelo, é necessário entender como aproveitar as características da construção proposta.

A geração da *tree expression* utilizando *extended nice tree decompositions*, apesar de natural e didática, possui uma limitação. Se a atribuição de marcações aos vértices do grafo for arbitrária, é natural que um mesmo vértice, em diferentes pontos da decomposição, possa receber marcações diferentes. Esse cenário é comum em vértices de junção. Lembre-se que um vértice de junção é aquele que possui dois filhos cujos *bags* associados são iguais, ou seja, se t é um nó de junção e t' e t'' são filhos de t, então

$f(t) = f(t') = f(t'')$, onde f é a função que associa um nó da árvore de decomposição a um *bag*. Tome, por exemplo, a Figura 4.1. Observe que o vértice v_3 está marcado com o símbolo a no filho esquerdo e com f no filho direito. A aplicação de uma composição paralela nesse caso resultaria em um erro, dado que, por estar com uma marcação diferente em cada ramo, o vértice v_3 seria considerado como dois vértices distintos pela operação. Esse ainda é um dos cenários mais simples. É possível que mais de um vértice do *bag* esteja com marcações inconsistentes. A solução deste problema se dá por meio dos símbolos $ren_{a\to b}$, $\forall a, b \in marked$ e $a \neq b$, que são responsáveis por modificar marcações. Voltando ao exemplo da Figura 4.1, considere que exp' e exp'' sejam as *tree expressions* associadas aos filhos esquerdo e direito do nó de junção, respectivamente. Logo, a *tree expression* associada ao nó de junção t seria denotada por $exp(t) = //[ren_{a\to f}[exp'], exp'']$ ou $exp(t) = //[exp', ren_{f\to a}[exp'']]$, assumindo que a operação $ren_{a\to f}$ ou $ren_{f\to a}$ não desencadeia alguma outra inconsistência. Em um cenário mais crítico, a inconsistência de marcações nos filhos de um determinado nó de junção é tal que a alteração de qualquer marcação gera inconsistência na *tree expression* e, consequentemente, no grafo representado por ela. Naturalmente, o leitor é induzido a acreditar que um conjunto com $k + 1$ marcações não é o suficiente para lidar com essa situação. Contudo, isso não é verdade.

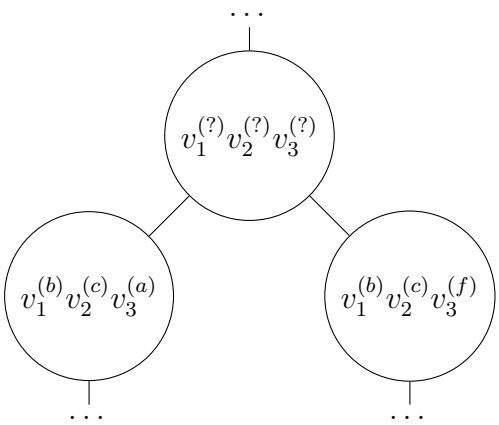


Figura 4.1: Exemplo de inconsistência de marcações em nó de junção

De modo a manter um controle de como o processo de construção da *tree expression* ocorre, propõe-se uma abordagem baseada em coloração, de modo a existir uma bijeção entre as cores utilizadas para colorir o grafo e as marcações utilizadas para codificar o grafo. A estratégia, apesar da aparência simples, é muito eficaz e se baseia em grafos cordais [11]. Um grafo é dito **cordal** se, e somente se, todo ciclo induzido possui tamanho três. Grafos cordais são caracterizados por um esquema de eliminação perfeito. Um **esquema de eliminação perfeito** é uma ordenação de vértices do grafo de modo que, ao ignorar os vértices anteriores ao vértice v na ordenação, v e seus vizinhos formam uma

clique.

Grafos que possuem *treewidth* limitada por um valor k são também chamados de k-árvores parciais. Uma **k-árvore parcial** é um subgrafo de uma k-árvore, que é apresentada na Definição 4.4.1. *Tree decompositions* de k-árvores são tais que cada nó da árvore é associado a uma clique. Além disso, k-árvores também são grafos cordais. Portanto, toda k-árvore admite um esquema de eliminação perfeito.

Definição 4.4.1 (k-árvore)**.** *Seja k um número inteiro. Uma **k-árvore** é um grafo G tal que ou G é um grafo completo com $k+1$ vértices ou G é um grafo composto por sucessivas adições vértices a partir de um grafo completo inicial de tamanho $k+1$ vértices, de modo que cada vértice inserido seja vizinho de exatamente k vértices do grafo.*

Em uma k-árvore é possível montar um esquema de eliminação perfeito apenas percorrendo sua *tree decomposition* em pós ordem e enfileirando os vértices na ordem em que eles são esquecidos. Ao remover um vértice em um determinado ponto da árvore, a vizinhança restante dele permanece sendo uma clique. Como essa vizinhança será esquecida posteriormente, pelo tipo de percurso em árvore escolhido, tem-se que, na ordenação construída, o vértice v sendo esquecido e seus vizinhos que aparecem após ele, juntos, induzem uma clique. Para colorir um grafo cordal, gasta-se tempo polinomial no número de vértices do grafo G.

A estratégia proposta, simulando que o grafo de entrada é uma k-árvore, é dada pelos seguintes passos:

1. Percorra a árvore de decomposição em pós ordem e enfileire os vértices que são esquecidos na ordem em que seus respectivos nós de esquecimento são visitados.

2. Após obter a ordenação dos vértices, percorra-a da direita para a esquerda, colorindo um determinado vértice com a menor cor disponível para ele, evitando as cores que foram usadas no nó de esquecimento deste vértice.

Utilizando este algoritmo garante-se que o grafo G foi colorido propriamente com $k+1$ cores, como consequência do algoritmo guloso de coloração baseado no esquema de eliminação perfeito. Além disso, em cada nó da árvore de decomposição os vértices associados a eles estão coloridos com cores distintas entre si, ainda que eles não sejam adjacentes em G. Esta propriedade é consequência da simulação de um grafo cordal usada para gerar o esquema de eliminação. Logo, tais garantias são suficientes para evitar o uso

de símbolos de alteração de marcação para corrigir eventuais problemas de uma atribuição de marcação de maneira indiscriminada.

Esta proposta não altera o resultado assintótico do Teorema de Courcelle, visto que cada uma das etapas enunciadas possui tempo polinomial. Além disso, obtém-se uma expressão mais simples de ser processada, dado que todo vértice carregará sua marcação adequada até que ele seja esquecido, como é mostrado na Figura 4.2.

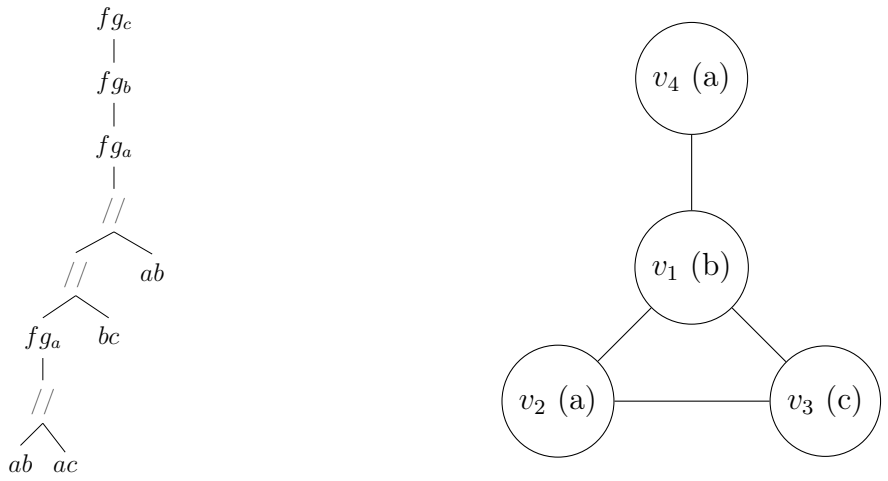


Figura 4.2: Exemplo de *tree expression* denotada em topologia de árvore e seu grafo associado

Capítulo 5

Autômatos e Fórmulas em MSOL_2

O elemento principal do algoritmo envolvido no Teorema de Courcelle é a aplicação do *model checking*. Pelo resultado de Tatcher e Wright, esta verificação pode ser feita por meio de um *tree automaton* construído a partir de uma fórmula em MSOL. No Capítulo 4 foi apresentada a teoria por trás da construção do modelo para esse *model checking*. A construção consiste em representar uma *tree decomposition* do grafo de entrada como uma árvore construída a partir de uma álgebra que define operações para gerar grafos.

Teorema (Thatcher e Wright (1968) / Doner (1970))**.** *Um conjunto finito de árvores é reconhecível por meio de um* tree automata *se, e somente se, é MSOL-definível.*

Em linhas gerais, o processo a ser aplicado para construir um *tree automaton*, peça chave do algoritmo proposto por Courcelle, a partir de uma fórmula em MSOL_2 é similar ao Algoritmo de Thompson [1], o qual constrói autômatos finitos não-determinísticos a partir de expressões regulares. Dada uma fórmula φ no modelo das árvores na forma normal prenex, a construção segue o seguinte protocolo:

- Se φ é uma fórmula atômica, então constrói-se um *tree automaton* que reconheça esta fórmula. Este processo é similar ao estudado na Subseção 2.4.2.

- Se $\varphi = \neg\phi$, então constrói-se o complemento do *tree automaton* que aceita ϕ.

  Seja $\mathcal{A}_\phi = \langle \Sigma, Q_\phi, Q_{F\phi}, \delta \rangle$ o *tree automaton* que aceita expressões que denotam grafos que possuem a propriedade expressa por ϕ. O *tree automaton* que reconhece φ será $\mathcal{A}_\varphi = \langle \Sigma, Q_\phi, (Q_\phi \backslash Q_{F\phi}), \delta \rangle$.

- Se $\varphi = \phi \wedge \psi$, então constrói-se a interseção dos *tree automata* que reconhecem ϕ e ψ.

Sejam $\mathcal{A}_\phi = \langle \Sigma, Q_\phi, Q_{F\phi}, \delta \rangle$ e $\mathcal{A}_\psi = \langle \Sigma, Q_\psi, Q_{F\psi}, \delta' \rangle$ os *tree automata* que aceitam expressões que denotam grafos que possuem as propriedades ϕ e ψ, respectivamente. O *tree automaton* que reconhece φ será $\mathcal{A}_\varphi = \langle \Sigma, (Q_\phi \times Q_\psi), (Q_{F\phi} \times Q_{F\psi}), \delta'' \rangle$.

- Se $\varphi = \phi \vee \psi$, então constrói-se a união dos *tree automata* que reconhecem ϕ e ψ.

  Sejam $\mathcal{A}_\phi = \langle \Sigma, Q_\phi, Q_{F\phi}, \delta \rangle$ e $\mathcal{A}_\psi = \langle \Sigma, Q_\psi, Q_{F\psi}, \delta' \rangle$ os *tree automata* que aceitam expressões que denotam grafos que possuem as propriedades ϕ e ψ, respectivamente. O *tree automaton* que reconhece φ será $\mathcal{A}_\varphi = \langle \Sigma, (Q_\phi \times Q_\psi), ((Q_\phi \times Q_{F\psi}) \cup (Q_{F\phi} \times Q_\psi)), \delta'' \rangle$.

- Se $\varphi = \exists X.\phi$, o *tree automaton* a ser construído será obtido através de projeção [16] no *tree automaton* para ϕ, o que implicará na eliminação do bit da *string* de controle que representa o conjunto X. Dado que neste trabalho não são estudadas fórmulas com variáveis livres e a fórmula de entrada do processo está na forma normal prenex, pode-se assumir, sem perda de generalidade, que esta operação apaga o último bit da *string* de controle. Uma consequência imediata dessa operação é o surgimento de transições não-determinísticas. Portanto, o *tree automaton* gerado deve ser transformado em determinístico e, em seguida, ter sua função de transição tornada total.

Neste capítulo serão apresentados os conceitos necessários para a construção do *tree automaton* para reconhecer propriedades expressas em MSOL_2. Esses autômatos consideram os elementos sintáticos do modelo apresentado no Capítulo 4. A discussão se inicia com a finalização do processo de tradução do problema de *model checking* para grafos em um problema de *model checking* para árvores. Este processo consiste em traduzir modelos e fórmulas. Na Seção 4.3 foi apresentada a tradução de modelos. Para fechar esta etapa, a Seção 5.1 explica a técnica de tradução das fórmulas. Por fim, a Seção 5.2 mostra os algoritmos para os predicados definidos para MSOL_2 e detalha procedimentos genéricos usados na construção dos *tree automata* baseados nos conectivos e quantificação existencial.

5.1 De grafos a árvores

Na Seção 3.3 foram discutidos os preparativos iniciais para o desenvolvimento do algoritmo para o Teorema de Courcelle. A formalização do problema é feita usando um modelo construído a partir do grafo e definindo uma fórmula em MSOL_2 que representa

a propriedade sendo verificada. O modelo utilizado nessa fase inicial é da forma

$$\mathcal{G}_{MSOL2} = \langle dom(\mathcal{G}_{MSOL2}), incidence, V, E, \in, \subseteq, = \rangle$$

e resulta em uma fórmula construída por meio dos conectivos $\wedge$, $\vee$, $\rightarrow$, $\neg$, das quantificações existenciais e universais e das fórmulas atômicas $x \in V$, $y \in E$, $x \in X$, $X_1 \subseteq X_2$, $X \subseteq V$, $Y \subseteq E$, $x_1 = x_2$, $X_1 = X_2$ e $incidence(x, y)$.

Todavia, o processo de *model checking* é executado em cima de uma árvore. O modelo para essa árvore foi apresentado na Seção 4.3 e tem a forma

$$\mathcal{T}_{MSOL2} = \langle dom(\mathcal{T}_{MSOL2}), <, \subseteq, sing, leaves, forgNodes, =, inc \rangle$$

A diferença nos predicados utilizados para representar a propriedade gera uma necessidade de um método automático eficiente que traduza uma fórmula para grafos em uma fórmula para árvores.

A técnica para essa tradução consiste em, recursivamente, processar a fórmula do modelo para grafos e substituir seus predicados por fórmulas construídas no modelo para árvores. O Algoritmo 5.1 descreve essa ideia. Observe que o algoritmo substitui variáveis para elementos por variáveis para conjuntos. Qualquer menção a variáveis para elemento devem ser substituídas por conjuntos unitários ao executar o algoritmo $translate\text{-}atom(\psi)$. Este procedimento consome tempo $\mathcal{O}(|\varphi|)$, onde $|\varphi|$ denota o tamanho da fórmula φ de MSOL_2. O procedimento apresentado assume que a fórmula de entrada já está na forma normal prenex.

```
1   translate-formula(φ)
2       IF (φ = ∃x.ψ) OR (φ = ∃X.ψ)THEN
3           formula ← translate-formula(ψ)
4           new ← ∃X.formula
5           RETURN new
6       ELIF (φ = ∀x.ψ) OR (φ = ∀X.ψ) THEN
7           formula ← translate-formula(ψ)
8           new ← ¬∃X.¬formula
9           RETURN new
10      ELIF (φ = ¬ψ) THEN
11          formula ← translate-formula(ψ)
12          new ← ¬formula
13          RETURN new
14      ELIF (φ = ϕ ∧ ψ) THEN
15          formula1 ← translate-formula(ϕ)
16          formula2 ← translate-formula(ψ)
```

```
17          new ← formula1 ∧ formula2
18          RETURN new
19      ELIF (φ = ϕ ∨ ψ) THEN
20          formula1 ← translate-formula(ϕ)
21          formula2 ← translate-formula(ψ)
22          new ← formula1 ∨ formula2
23          RETURN new
24      ELSE
25          new ← translate-atom(φ)
26          RETURN new
```

Algoritmo 5.1: Algoritmo para tradução de fórmulas de $\mathcal{G}_{MSOL2}$.

A tradução de fórmulas atômicas do modelo para grafos, representada pela função *translate-atom*(ψ) no Algoritmo 5.1, deve cobrir todos os elementos sintáticos do modelo $\mathcal{G}_{MSOL2}$. O algoritmo em questão deve identificar a fórmula atômica de $\mathcal{G}_{MSOL2}$ e transforma-la em uma fórmula de $\mathcal{T}_{MSOL2}$. Qualquer menção a variáveis para elementos devem gerar uma menção a um conjunto unitário. O procedimento é descrito no Algoritmo 5.2 e consome tempo $\mathcal{O}(1)$.

```
1   translate-atom(φ)
2       IF (x ∈ V) THEN
3           RETURN sing(X) ∧ (X ⊆ forgNodes)
4       ELIF (y ∈ E) THEN
5           RETURN sing(Y) ∧ (Y ⊆ leaves)
6       ELIF (x ∈ X) THEN
7           RETURN sing(X′) ∧ (X′ ⊆ X)
8       ELIF (X1 ⊆ X2) THEN
9           RETURN X1 ⊆ X2
10      ELIF (X ⊆ V) THEN
11          RETURN X ⊆ forgNodes
12      ELIF (Y ⊆ E) THEN
13          RETURN Y ⊆ leaves
14      ELIF (x1 = x2) THEN
15          RETURN X1 = X2
16      ELIF (X1 = X2) THEN
17          RETURN X1 = X2
18      ELIF (incidence(x, y)) THEN
19          RETURN
                sing(X) ∧ (X ⊆ forgNodes) ∧ sing(Y) ∧ (Y ⊆ leaves) ∧ X < Y ∧ inc(X, Y)
```

Algoritmo 5.2: Algoritmo para tradução de fórmulas atômicas de $\mathcal{G}_{MSOL2}$

O predicado $inc(X, Y)$ existe para garantir a consistência da relação $X < Y$. En-

quanto $X < Y$ garante que a posição do conjunto Y está na subárvore enraizada pela posição do conjunto X, $inc(X, Y)$ garante que a posição de X representa um vértice do grafo e a posição de Y representa uma aresta que incide nesse vértice. Observando a álgebra definida para gerar hipergrafos e a *tree expression* construída a partir da *tree decomposition*, percebe-se um padrão que relaciona os símbolos para arestas e os símbolos de esquecimento de marcação. Veja o exemplo da Figura 5.1. Cada subárvore enraizada em um símbolo fg_a, para todo $a \in marked$, possui um conjunto de folhas que denotam as arestas incidentes ao vértice cuja marcação está sendo esquecida. Os nós relacionados por *inc* aparecem circulados na figura. Voltando a atenção para o caso retratado pela Figura 5.1(i), é natural questionar se as folhas rotuladas por ab e ac também não estariam relacionadas com o fg_a destacado. A resposta é não, porque há outro símbolo fg_a entre o destacado na figura e as folhas em questão. Esse é conceito de *proximidade* que permite o predicado garantir a consistência do processo. Considera-se que uma folha é próxima de um nó de esquecimento quando a folha é rotulada por m_1m_2 e o nó de esquecimento é rotulado por fg_{m_1} ou fg_{m_2}, para quaisquer $m_1, m_2 \in marked$. Além disso, no caminho entre m_1m_2 e fg_{m_1} não existe outro símbolo fg_{m_1}. A restrição é análoga para fg_{m_2}.

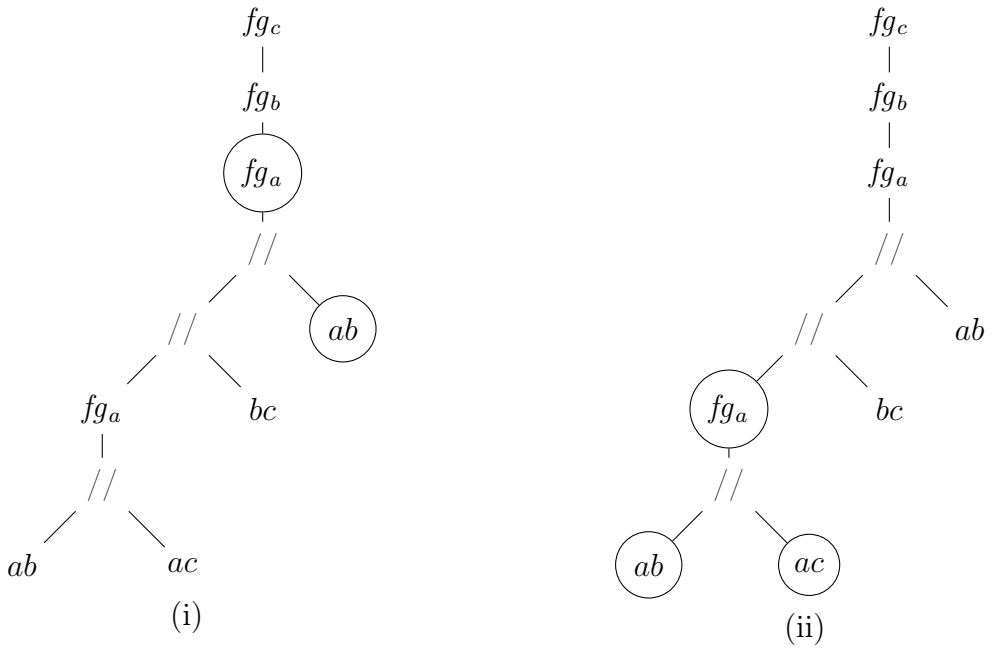


Figura 5.1: Exemplo de vértices representados por nós da árvore por meio da relação *inc*

O vértice colorido com a cor associada à marcação a nos níveis mais baixos da árvore não é o mesmo vértice colorido com a mesma cor nos níveis mais altos. Basta conferir a Figura 4.2 para observar o mencionado. É possível visualizar a proximidade por meio dos graus dos vértices no grafo gerado a partir da *tree expression*. Olhando para

o grafo da Figura 4.2, percebe-se a existência de dois nós coloridos com a, um de grau 1 e outro de grau 2. Os destaques feitos na Figura 5.1 representam exatamente esses vértices, respectivamente. Lembre-se que cada vértice é representado por um símbolo de esquecimento.

É necessário ressaltar que este processo de tradução pode causar redundâncias no uso de certas fórmulas atômicas de $\mathcal{T}_{MSOL2}$ e também uma possibilidade de aparecimento de duplas negações. Assim, é necessário processar a fórmula traduzida para remover duplas negações por meio da identidade $\neg\neg\varphi \equiv \varphi$ e verificar possíveis redundâncias, como $sing(X) \wedge \psi \wedge sing(X)$ e $sing(X) \vee \psi \vee sing(X)$, por exemplo. Esse procedimento consome tempo $\mathcal{O}(|\varphi|^2)$, onde $|\varphi|$ é o tamanho da fórmula.

5.2 Algoritmos para *tree automata* baseados em MSOL$_2$

Como apresentado na Seção 2.4, a construção de um autômato a partir de uma fórmula em MSOL é feita a partir de propriedades de linguagens regulares e autômatos que reconhecem os predicados utilizados para construir as fórmulas. Até um certo ponto, todos os *tree automata* a serem construídos durante esse processo irão compartilhar o mesmo alfabeto. Portanto, o Algoritmo 5.3 traz o método de construção da versão estendida do alfabeto, mencionada na Seção 2.4. A alfabeto básico Σ é composto pelo símbolo // de composição paralela, os símbolos fg_m, para todo $m \in marked$, para o esquecimento da marcação m e dos símbolos de arestas do grafo, que são gerados via justaposição dos elementos do conjunto de marcações $marked$. A função *binarize* é uma função que retorna uma *string* binária que representa o valor inteiro não-negativo passado como entrada. Em situações de implementação é extremamente importante garantir que o resultado dessa função seja sempre uma string com qr bits. Lembre-se que qr é o valor do *quantifier rank* da fórmula em forma normal prenex, que pode ser calculado via Algoritmo 5.4.

```
1   extend-alphabet(Σ, qr)
2       Σ' ← ∅
3       FOR EACH σ ∈ Σ DO
4           FOR control = 0 TO 2^qr DO
5               bin = binarize(control)
6               Σ' ← Σ' ∪ {(σ, bin)}
7       RETURN Σ'
```

Algoritmo 5.3: Algoritmo para construir o alfabeto estendido com *strings* de controle

```
1   quantifier-rank(φ)
2       IF (φ = ∃X.ψ) THEN
3           qr ← 1+quantifier-rank(ψ)
4           RETURN qr
5       ELIF (φ = ¬ψ) THEN
6           qr ← quantifier-rank(ψ)
7           RETURN qr
8       ELIF (φ = φ ∧ ψ) THEN
9           qr ← max{quantifier-rank(φ),quantifier-rank(ψ)}
10          RETURN qr
11      ELIF (φ = φ ∨ ψ) THEN
12          qr ← max{quantifier-rank(φ),quantifier-rank(ψ)}
13          RETURN qr
14      ELSE
15          RETURN 0
```

Algoritmo 5.4: Algoritmo para cálculo do *quantifier rank*

Todos os *tree automata* construídos para os predicados propostos para MSOL_2 utilizarão como entrada o alfabeto Σ'. Todavia, este alfabeto não permanecerá o mesmo durante todo o processo de construção do autômato para a fórmula por conta da operação de projeção em autômatos necessária à construção de *tree automata* para a quantificação existencial. O algoritmo essa construção é similar ao algoritmo genérico apresentado na Seção 2.4. Entretanto, aquele algoritmo abstraía algumas operações, principalmente a construção para os predicados da linguagem. Portanto, nas próximas subseções serão tratados os algoritmos para se construir os *tree automata* para os predicados da linguagem MSOL_2 proposta na Seção 5.1 e um detalhamento dos algoritmos abstraídos para fórmulas compostas por conectivos e quantificação existencial.

5.2.1 *Tree automata* para predicados

Como mencionado na Seção 2.4, a maior parte dos autômatos construídos para MSOL trabalha sempre observando o valor de bits chave na *string* de controle. Para MSOL_2 a ideia se mantém, porém, os autômatos a serem construídos não serão os mesmos da referida seção. Os predicados utilizados para expressar uma propriedade numa árvore que codifica um grafo são $sing(X_i)$, $X_i \subseteq X_j$, $X_i \subseteq leaves$, $X_i \subseteq forgNodes$, $X_i < X_j$, $X_i = X_j$ e $inc(X_i, X_j)$.

Para iniciar a discussão pela construção mais simples, tome o predicado de igualdade $X_i = X_j$. O algoritmo toma como entrada os índices dos bits a serem verificados durante

toda a leitura da árvore. É imprescindível que os bits nas posições i e j sejam sempre iguais. Este *tree automaton* pode ser construído como o expresso no Algoritmo 5.5. Lembre-se que todo elemento de Σ' é um par $sym = (\sigma, bin)$. Ainda que seja necessário fazer alguma combinatória no espaço de estados do autômato, esse espaço é finito e fixo. O número de combinações é constante. A única iteração relevante aqui é sobre o alfabeto Σ'. Logo, a complexidade deste algoritmo é $O(|\Sigma'|)$, sendo que $|\Sigma'| = O(\kappa^2)$.

```
1   equality-automaton(Σ', i, j)
2       Q ← {q_equal, q_not}
3       Q_f ← {q_equal}
4       δ ← ∅
5       FOR EACH sym ∈ Σ' DO
6           IF (sym.σ = //) THEN
7               IF (sym.bin[i] = sym.bin[j]) THEN
8                   FOR EACH q ∈ Q DO
9                       FOR EACH q' ∈ Q DO
10                          IF (q = q_not) or (q' = q_not) THEN
11                              δ ← δ ∪ {(sym, (q, q'), q_not)}
12                          ELSE
13                              δ ← δ ∪ {(sym, (q, q'), q_equal)}
14              ELSE
15                  FOR EACH q ∈ Q DO
16                      FOR EACH q' ∈ Q DO
17                          δ ← δ ∪ {(sym, (q, q'), q_not)}
18          ELIF (sym.σ = fg_k) THEN
19              IF (sym.bin[i] = sym.bin[j]) THEN
20                  δ ← δ ∪ {(sym, (q_equal), q_equal)}
21                  δ ← δ ∪ {(sym, (q_not), q_not)}
22              ELSE
23                  δ ← δ ∪ {(sym, (q_equal), q_not)}
24                  δ ← δ ∪ {(sym, (q_not), q_not)}
25          ELSE
26              IF (sym.bin[i] = sym.bin[j]) THEN
27                  δ ← δ ∪ {(sym, ∅, q_equal)}
28              ELSE
29                  δ ← δ ∪ {(sym, ∅, q_not)}
30      automaton ← ⟨Σ', Q, Q_f, δ⟩
31      RETURN automaton
```

Algoritmo 5.5: Algoritmo para *tree automaton* para o predicado de igualdade

Teorema 5.2.1 (Corretude do Algoritmo 5.5)**.** *Dado que Σ' é um alfabeto estendido, $1 \leq i \leq c$ e $1 \leq j \leq c$, onde c é o número de variáveis para conjuntos mencionadas na*

fórmula, o Algoritmo 5.5 constrói um tree automaton *que reconhece uma* tree expression *quando os conjuntos X_i e X_j forem iguais.*

Demonstração. Suponha que o *tree automaton* construído aceite uma *tree expression* quando os conjuntos forem diferentes. Sem perda de generalidade, assuma que existe apenas um elemento de X_i que não seja também elemento de X_j. Três cenários devem ser avaliados.

1. O elemento diferente é uma folha.

   Se o único elemento diferente dos conjuntos for uma folha, a transição $(sym, \emptyset, q_{not})$ é executada. Após essa execução, as transições das linhas 11 e 21 serão responsáveis por garantir que o autômato termine sua execução no estado q_{not}. Dado que q_{not} não é um estado de aceitação, a árvore é rejeitada pelo autômato e isso contradiz a suposição.

2. O elemento diferente é um símbolo para esquecimento.

   Se o único elemento diferente dos conjuntos for um símbolo para esquecimento, uma transição $(sym, (q_{equal}), q_{not})$ é executada. Novamente, as transições das linhas 11 e 21 serão responsáveis por propagar a informação do erro até a raiz, mantendo o *tree automaton* no estado q_{not}. Novamente a suposição é contrariada.

3. O elemento diferente é um símbolo de composição paralela.

   Para o caso de o elemento diferente ser um símbolo '//', $(sym, (q_{equal}, q_{equal}), q_{not})$ é a transição executada. Uma vez no estado q_{not}, as transições das linhas 11 e 21 irão impedir que o *tree automaton* saia desse estado. Portanto, o *tree automaton* rejeita a árvore, o que contradiz a suposição.

Observe que não faz diferença se os conjuntos possuírem mais de um elemento diferente. As transições das linhas 11, 21 e 24 mostram que basta aparecer um estado q_{not} no processamento para que o *tree automaton* não possa escapar deste estado. Portanto, o *tree automaton* construído aceita uma *tree expression* apenas quando $X_i = X_j$. □

O Algoritmo 5.6 representa a construção do autômato para o predicado $sing(X_i)$. Seguindo as convenções apresentadas na Seção 2.4, é utilizado o alfabeto estendido para essa construção. Este *tree automaton* possui três estados, a saber

$\boldsymbol{q_{wait}}$ que representa um estado de neutralidade onde uma possível solução não foi encontrada;

$\boldsymbol{q_{unit}}$ que representa o estado onde foi encontrado um elemento do conjunto X_i;

$\boldsymbol{q_{error}}$ que representa o estado no qual mais de uma posição da árvore é dita pertencer a um conjunto que deveria ser unitário.

O algoritmo itera sobre o alfabeto, gerando uma *string* de controle a ser anexada ao símbolo considerado. Após a criação do símbolo estendido, o processo verifica o i-ésimo bit do controle. Caso o bit esteja setado como 0, reconhece-se o tipo do símbolo para gerar suas transições considerando que a posição não pertence ao conjunto mencionado. Caso o bit esteja setado como 1, as transições são geradas considerando a posição como parte do conjunto. Portanto, leituras anteriores que atingiram ao menos um estado q_{unit} deverão retornar o estado q_{error}. Em tempo, o símbolo fg_k significa que algum k do conjunto de marcações foi esquecido.

```
1   singleton-automaton(Σ', i)
2       Q ← {q_wait, q_unit, q_error}
3       Q_f ← {q_unit}
4       δ ← ∅
5       FOR EACH sym ∈ Σ' DO
6           IF (sym.σ = //) THEN
7               FOR EACH q ∈ Q DO
8                   FOR EACH q' ∈ Q DO
9                       IF (sym.bin[i] = 1) THEN
10                          IF (q = q_wait) or (q' = q_wait) THEN
11                              δ ← δ ∪ {(sym, (q, q'), q_unit)}
12                          ELSE
13                              δ ← δ ∪ {(sym, (q, q'), q_error)}
14                      ELSE
15                          IF (q = q_error) or (q' = q_error) THEN
16                              δ ← δ ∪ {(sym, (q, q'), q_error)}
17                          ELIF (q = q_unit) and (q = q_unit) THEN
18                              δ ← δ ∪ {(sym, (q, q'), q_error)}
19                          ELIF (q = q_wait) THEN
20                              δ ← δ ∪ {(sym, (q, q'), q')}
21                          ELSE
22                              δ ← δ ∪ {(sym, (q, q'), q_unit)}
23          ELIF (sym.σ = fg_k) THEN
24              IF (sym.bin[i] = 1) THEN
25                  δ ← δ ∪ {(sym, (q_wait), q_unit)}
26                  δ ← δ ∪ {(sym, (q_unit), q_error)}
27                  δ ← δ ∪ {(sym, (q_error), q_error)}
28              ELSE
29                  δ ← δ ∪ {(sym, (q_wait), q_wait)}
```

```
30                  δ ← δ ∪ {(sym, (q_unit), q_unit)}
31                  δ ← δ ∪ {(sym, (q_error), q_error)}
32          ELSE
33              IF (sym.bin[i] = 1) THEN
34                  δ ← δ ∪ {(sym, ∅, q_unit)}
35              ELSE
36                  δ ← δ ∪ {(sym, ∅, q_wait)}
37      automaton ← ⟨Σ', Q, Q_f, δ⟩
38      RETURN automaton
```

Algoritmo 5.6: Algoritmo para *tree automaton* para o predicado $sing(X_i)$

Teorema 5.2.2 (Corretude do Algoritmo 5.6)**.** *Dado que Σ' é um alfabeto estendido e $1 \leq i \leq c$, onde c é o número de conjuntos mencionados na fórmula, o Algoritmo 5.6 constrói um* tree automaton *que reconhece uma* tree expression *quando o conjunto X_i é um conjunto unitário.*

Demonstração. A prova deste teorema é análoga à prova apresentada para o Teorema 2.4.6, a menos da presença de símbolos de *rank* 1 no alfabeto. Tais elementos não representam nenhum empecilho às ideias apresentadas. □

O Algoritmo 5.7 mostra como construir o *tree automaton* para o predicado $X_i \subseteq leaves$. Este predicado é utilizado para identificar conjuntos compostos por folhas da árvore, nas quais estão representadas as arestas do grafo. Este *tree automaton* é composto de 3 estados que representam a neutralidade da parte verificada (q_{none}), a confirmação de que um elemento é uma folha da árvore (q_{sub}) e a localização de algum elemento pertencente ao conjunto que não é uma folha (q_{error}). Neste *tree automaton* as transições mais importantes são as que ocorrem nas folhas da árvore. Se algum símbolo que não for uma folha estiver alocado no conjunto X_i, independente dos estados alcançados até aquele momento, o algoritmo deve gerar uma transição para o estado de erro.

```
1   leaves-automaton(Σ', i)
2       Q ← {q_sub, q_error}
3       Q_f ← {q_sub}
4       δ ← ∅
5       FOR EACH sym ∈ Σ' DO
6           IF (sym.σ = //) THEN
7               FOR EACH q ∈ Q DO
8                   FOR EACH q' ∈ Q DO
9                       IF (sym.bin[i] = 1) THEN
10                          δ ← δ ∪ {(sym, (q, q'), q_error)}
```

```
11                      ELSE
12                          IF (q = q_error) or (q' = q_error) THEN
13                              δ ← δ ∪ {(sym, (q, q'), q_error)}
14                          ELSE
15                              δ ← δ ∪ {(sym, (q, q'), q_sub)}
16          ELIF (sym.σ = fg_k) THEN
17              IF (sym.bin[i] = 1) THEN
18                  δ ← δ ∪ {(sym, (q_sub), q_error)}
19                  δ ← δ ∪ {(sym, (q_error), q_error)}
20              ELSE
21                  δ ← δ ∪ {(sym, (q_sub), q_sub)}
22                  δ ← δ ∪ {(sym, (q_error), q_error)}
23          ELSE
24              δ ← δ ∪ {(sym, ∅, q_sub)}
25      automaton ← ⟨Σ', Q, Q_f, δ⟩
26      RETURN automaton
```

Algoritmo 5.7: Algoritmo para *tree automaton* para o predicado $X_i \subseteq leaves$

Teorema 5.2.3 (Corretude do Algoritmo 5.7)**.** *Dado que Σ' é um alfabeto estendido e $1 \leq i \leq c$, onde c é o número de conjuntos mencionados na fórmula, o Algoritmo 5.7 constrói um* tree automaton *que reconhece uma* tree expression *quando o conjunto X_i é um subconjunto das folhas da árvore, denotadas pelo conjunto leaves.*

Demonstração. Suponha que o *tree automaton* aceite uma *tree expression* quando o conjunto $X_i \not\subseteq leaves$. Sem perda de generalidade, pode-se assumir que existe apenas um elemento no conjunto X_i que não é uma folha da árvore. Logo, dois cenários devem ser avaliados.

1. O elemento diferente é um símbolo para esquecimento.

   Neste caso, uma transição $(sym, (q_{sub}), q_{error})$ é executada. A partir desse resultado, transições como as das linhas 13 e 22 serão executadas para que o autômato chegue no estado q_{error} na raiz da árvore. Dado que q_{error} não é um estado de aceitação, a suposição apresentada está incorreta.

2. O elemento diferente é um símbolo para composição paralela.

   Neste cenário, uma transição com o formato da transição da linha 10 é executada. Novamente, as transições das linhas 13 e 22, por exemplo, desempenham o papel de impedir que o *tree automaton* termine a execução em um estado diferente de q_{error}. Logo, a suposição é contrariada.

Ainda que o conjunto pudesse possuir mais de um símbolo diferente de uma folha, as transições das linhas 10, 13, 18, 19 e 22 garantem que o *tree automaton* termine sua execução em q_{error}. Logo, o *tree automaton* construído reconhece uma *tree expression* apenas quando $X_i \subseteq leaves$. □

O Algoritmo 5.8 mostra o método de construção de um *tree automaton* para o predicado $X_i \subseteq forgNodes$. A ideia deste algoritmo é similar ao *tree automaton* para $X_i \subseteq leaves$. A diferença entre eles está no tipo de símbolo a ser considerado. Enquanto no anterior verificavam-se as folhas, que contém as arestas, neste se verificam os nós do tipo fg_k, para todo k no conjunto de marcações. Apenas símbolos fg_k com i-ésimo bit 1 devem gerar transições de sucesso. Símbolos com o i-ésimo bit em 0 devem apenas repassar as informações de modo adequado ou gerar transições neutras.

```
1   forget-automaton(Σ', i)
2       Q ← {q_sub, q_error}
3       Q_f ← {q_sub}
4       δ ← ∅
5       FOR EACH sym ∈ Σ' DO
6           IF (sym.σ = //) THEN
7               FOR EACH q ∈ Q DO
8                   FOR EACH q' ∈ Q DO
9                       IF (sym.bin[i] = 1) THEN
10                          δ ← δ ∪ {(sym, (q, q'), q_error)}
11                      ELSE
12                          IF (q = q_error) or (q' = q_error) THEN
13                              δ ← δ ∪ {(sym, (q, q'), q_error)}
14                          ELSE
15                              δ ← δ ∪ {(sym, (q, q'), q_sub)}
16          ELIF (sym.σ = fg_k) THEN
17              δ ← δ ∪ {(sym, (q_sub), q_sub)}
18              δ ← δ ∪ {(sym, (q_error), q_error)}
19          ELSE
20              IF (sym.bin[i] = 1) THEN
21                  δ ← δ ∪ {(sym, ∅, q_error)}
22              ELSE
23                  δ ← δ ∪ {(sym, ∅, q_sub)}
24      automaton ← ⟨Σ', Q, Q_f, δ⟩
25      RETURN automaton
```

Algoritmo 5.8: Algoritmo para *tree automaton* para o predicado $X_i \subseteq forgNodes$

Teorema 5.2.4 (Corretude do Algoritmo 5.8)**.** *Dado que Σ' é um alfabeto estendido e*

$1 \leq i \leq c$, onde c é o número de conjuntos mencionados na fórmula, o Algoritmo 5.8 constrói um tree automaton *que reconhece uma* tree expression *quando o conjunto X_i é um subconjunto dos símbolos para esquecimento, denotados pelo conjunto $forgNodes$.*

Demonstração. Suponha que o *tree automaton* aceite uma *tree expression* quando o conjunto $X_i \not\subseteq forgNodes$. Sem perda de generalidade, pode-se assumir que existe apenas um elemento no conjunto X_i que não é um símbolo de esquecimento da árvore. Logo, dois cenários devem ser avaliados.

1. O elemento diferente é uma folha.

   Caso o elemento diferente de X_i seja uma folha, uma transição $(sym, \emptyset, q_{error})$ é executada. Assim, as transições das linhas 13 e 18 desempenharão o papel de manter o *tree automaton* retido no estado q_{error} até o fim da execução. O fato de q_{error} não ser um estado de aceitação contradiz a suposição inicial.

2. O elemento diferente é um símbolo para composição paralela.

   Neste cenário, uma transição com o formato da transição da linha 10 é executada. Novamente, as transições das linhas 13 e 18 desempenham o papel de impedir que o *tree automaton* termine a execução em um estado diferente de q_{error}. Logo, a suposição é contrariada.

Ainda que o conjunto pudesse possuir mais de um símbolo diferente de um símbolo de esquecimento, as transições das linhas 10, 13, 18 e 21 garantem que o *tree automaton* termine sua execução em q_{error}. Logo, o *tree automaton* construído reconhece uma *tree expression* apenas quando $X_i \subseteq forgNodes$. □

O Algoritmo 5.9 mostra a construção do tradicional *tree automaton* para predicados do tipo $X_i \subseteq X_j$. Estes *tree automaton* devem sempre estar verificando os i-ésimos e j-ésimos bits das *strings* de controle, gerando uma transição de erro sempre que o i-ésimo bit estiver setado em 1 e o j-ésimo bit estiver setado em 0. Esta configuração representa que existe um elemento em X_i que não pertence a X_j, o que acarretaria $X_i \not\subseteq X_j$.

```
1   inclusion-automaton(Σ', i, j)
2       Q ← {q_sub, q_error}
3       Q_f ← {q_sub}
4       δ ← ∅
5       FOR EACH sym ∈ Σ' DO
6           IF (sym.σ = //) THEN
```

```
7            FOR EACH q ∈ Q DO
8                FOR EACH q' ∈ Q DO
9                    IF (sym.bin[i] = 1) THEN
10                       IF (sym.bin[j] = 1) THEN
11                           IF (q = q_error) or (q' = q_error) THEN
12                               δ ← δ ∪ {(sym, (q, q'), q_error)}
13                           ELSE
14                               δ ← δ ∪ {(sym, (q, q'), q_sub)}
15                       ELSE
16                           δ ← δ ∪ {(sym, (q, q'), q_error)}
17                   ELSE
18                       IF (q = q_error) or (q' = q_error) THEN
19                           δ ← δ ∪ {(sym, (q, q'), q_error)}
20                       ELSE
21                           δ ← δ ∪ {(sym, (q, q'), q_sub)}
22       ELIF (sym.σ = fg_k) THEN
23           IF (sym.bin[i] = 1) THEN
24               IF (sym.bin[j] = 1) THEN
25                   δ ← δ ∪ {(sym, (q_sub), q_sub)}
26                   δ ← δ ∪ {(sym, (q_error), q_error)}
27               ELSE
28                   δ ← δ ∪ {(sym, (q_sub), q_error)}
29                   δ ← δ ∪ {(sym, (q_error), q_error)}
30           ELSE
31               δ ← δ ∪ {(sym, (q_sub), q_sub)}
32               δ ← δ ∪ {(sym, (q_error), q_error)}
33       ELSE
34           IF (sym.bin[i] = 1) THEN
35               IF (sym.bin[j] = 1) THEN
36                   δ ← δ ∪ {(sym, ∅, q_sub)}
37               ELSE
38                   δ ← δ ∪ {(sym, ∅, q_error)}
39           ELSE
40               δ ← δ ∪ {(sym, ∅, q_sub)}
41   automaton ← ⟨Σ', Q, Q_f, δ⟩
42   RETURN automaton
```

Algoritmo 5.9: Algoritmo para *tree automaton* para o predicado $X_i \subseteq X_j$

Teorema 5.2.5 (Corretude do Algoritmo 5.9)**.** *Dado que Σ' é um alfabeto estendido, $1 \leq i \leq c$, $1 \leq j \leq c$ e $i \neq j$, onde c é o número de conjuntos mencionados na fórmula, o Algoritmo 5.9 constrói um* tree automaton *que reconhece uma* tree expression *quando o conjunto X_i é um subconjunto do conjunto X_j.*

Demonstração. A prova deste teorema é análoga à prova apresentada para o Teorema 2.4.8,

a menos da presença de símbolos de *rank* 1 no alfabeto. Tais elementos não representam nenhum empecilho às ideias apresentadas. □

O predicado $X_i < X_j$ é um predicado que representa um caminho ascendente na árvore. Um **caminho ascendente** numa árvore é uma sequência de posições da árvore, denotada por $p_1 p_2 \ldots p_n$, tal que p_{i+1} é pai de p_i e nenhum p_i é a raiz da árvore, para $1 \leq i < n$. O *tree automaton* para esse predicado consiste em encontrar a posição com o j-ésimo bit da *string* de controle marcado como 1 na subárvore cuja raiz é a posição com o i-ésimo bit da *string* de controle marcado como 1.

```
1   prefix-automaton(Σ', i, j)
2       Q ← {q_wait, q_first, q_success, q_error}
3       Q_f ← {q_success}
4       δ ← ∅
5       FOR EACH sym ∈ Σ' DO
6           IF (sym.σ = //) THEN
7               FOR EACH q ∈ Q DO
8                   FOR EACH q' ∈ Q DO
9                       IF (sym.bin[i] = 1) THEN
10                          IF (sym.bin[j] = 1) THEN
11                              δ ← δ ∪ {(sym, (q, q'), q_error)}
12                          ELSE
13                              IF (q = q_wait) AND (q' = q_first) THEN
14                                  δ ← δ ∪ {(sym, (q, q'), q_error)}
15                              ELIF (q = q_wait) AND (q' = q_first) THEN
16                                  δ ← δ ∪ {(sym, (q, q'), q_error)}
17                              ELSE
18                                  δ ← δ ∪ {(sym, (q, q'), q_error)}
19                      ELSE
20                          IF (sym.bin[j] = 1) THEN
21                              IF (q = q_wait) AND (q' = q_wait) THEN
22                                  δ ← δ ∪ {(sym, (q, q'), q_first)}
23                              ELSE
24                                  δ ← δ ∪ {(sym, (q, q'), q_error)}
25                          ELSE
26                              IF (q = q_wait) THEN
27                                  δ ← δ ∪ {(sym, (q, q'), q')}
28                              ELIF (q = q_first) AND (q' = q_wait) THEN
29                                  δ ← δ ∪ {(sym, (q, q'), q_first)}
30                              ELIF (q = q_succes) AND (q' = q_wait) THEN
31                                  δ ← δ ∪ {(sym, (q, q'), q_success)}
32                              ELSE
33                                  δ ← δ ∪ {(sym, (q, q'), q_error)}
```

```
34          ELIF (sym.σ = fg_k) THEN
35              IF (sym.bin[i] = 1) THEN
36                  IF (sym.bin[j] = 1) THEN
37                      δ ← δ ∪ {(sym, (q_wait), q_error)}
38                      δ ← δ ∪ {(sym, (q_first), q_error)}
39                      δ ← δ ∪ {(sym, (q_success), q_error)}
40                      δ ← δ ∪ {(sym, (q_error), q_error)}
41                  ELSE
42                      δ ← δ ∪ {(sym, (q_wait), q_error)}
43                      δ ← δ ∪ {(sym, (q_first), q_success)}
44                      δ ← δ ∪ {(sym, (q_success), q_error)}
45                      δ ← δ ∪ {(sym, (q_error), q_error)}
46              ELSE
47                  IF (sym.bin[j] = 1) THEN
48                      δ ← δ ∪ {(sym, (q_wait), q_first)}
49                      δ ← δ ∪ {(sym, (q_first), q_error)}
50                      δ ← δ ∪ {(sym, (q_success), q_error)}
51                      δ ← δ ∪ {(sym, (q_error), q_error)}
52                  ELSE
53                      δ ← δ ∪ {(sym, (q_wait), q_wait)}
54                      δ ← δ ∪ {(sym, (q_first), q_first)}
55                      δ ← δ ∪ {(sym, (q_success), q_success)}
56                      δ ← δ ∪ {(sym, (q_error), q_error)}
57          ELSE
58              IF (sym.bin[i] = 1) THEN
59                  δ ← δ ∪ {(sym, ∅, q_error)}
60              ELSE
61                  IF (sym.bin[j] = 1) THEN
62                      δ ← δ ∪ {(sym, ∅, q_first)}
63                  ELSE
64                      δ ← δ ∪ {(sym, ∅, q_wait)}
65      automaton ← ⟨Σ', Q, Q_f, δ⟩
66      RETURN automaton
```

Algoritmo 5.10: Algoritmo para *tree automaton* para o predicado $X_i < X_j$

Teorema 5.2.6 (Corretude do Algoritmo 5.10)**.** *Dado que Σ' é um alfabeto estendido, $1 \leq i \leq c$, $1 \leq j \leq c$ e $i \neq j$, onde c é o número de conjuntos mencionados na fórmula, o Algoritmo 5.10 constrói um* tree automaton *que reconhece uma* tree expression *quando o elemento do conjunto unitário X_i é um ancestral do elemento do conjunto unitário X_j.*

Demonstração. A prova deste teorema é análoga à prova apresentada para o Teorema 2.4.11, a menos da presença de símbolos de *rank* 1 no alfabeto. Tais elementos não representam nenhum empecilho às ideias apresentadas. □

Como explicado anteriormente, o predicado $inc(X_i, X_j)$ serve para garantir que a aresta representada na folha contida no conjunto X_j é incidente ao vértice codificado no nó de esquecimento do conjunto X_i. Portanto, a verificação precisa garantir que entre as posições de X_i e de X_j não há um símbolo de esquecimento igual. Para realizar corretamente a verificação, é preciso ter estados que diferenciem marcações procuradas e encontradas durante o processamento. Para isso, é necessário passar para o autômato o conjunto de marcações e iterar sobre ele para criar estes estados. Estados da forma $q_{s-a-s-b}$, por exemplo, denotam um estado onde o símbolo para aresta do conjunto X_j é ab e estão sendo buscados símbolos que esqueçam as marcações a ou b. Ao passar por um símbolo que esqueça uma dessas marcações, o estado transita para $q_{f-a-s-b}$, $q_{s-a-f-b}$ ou $q_{f-a-f-b}$ que denotam o encontro de um fg_a, o encontro de um fg_b e o encontro de ambos os símbolos, respectivamente. O *tree automaton* só reconhece sucesso quando a marcação denotada por k no nó de esquecimento do conjunto X_i é igual a uma marcação sendo procurada. Por exemplo, se o *tree automaton* ler um símbolo fg_a e o estado do seu filho for $q_{s-a-f-b}$ ou $q_{s-a-s-b}$, então ele retorna o estado q_{incid}, que detona uma incidência. Por outro lado, se nesse mesmo cenário o estado do filho fosse $q_{f-a-f-b}$ ou $q_{f-a-s-b}$, o autômato retornaria um erro, pois isso indica que entre a folha e o nó de esquecimento apontado existe pelo menos outro símbolo fg_a.

```
1   incidence-automaton(Σ', i, j, marked)
2       Q ← {q_wait, q_incid, q_error}
3       FOR EACH m_1 ∈ marked DO
4           FOR EACH m_2 ∈ marked DO
5               IF (m_1 ≠ m_2) THEN
6                   IF (m_1 < m_2) THEN
7                       Q ← Q ∪ {q_{s-m_1-s-m_2}}
8                       Q ← Q ∪ {q_{f-m_1-s-m_2}}
9                       Q ← Q ∪ {q_{s-m_1-f-m_2}}
10                      Q ← Q ∪ {q_{f-m_1-f-m_2}}
11                  ELSE
12                      Q ← Q ∪ {q_{s-m_2-s-m_1}}
13                      Q ← Q ∪ {q_{f-m_2-s-m_1}}
14                      Q ← Q ∪ {q_{s-m_2-f-m_1}}
15                      Q ← Q ∪ {q_{f-m_2-f-m_1}}
16      Q_f ← {q_success}
17      δ ← ∅
18      FOR EACH sym ∈ Σ' DO
19          IF (sym.σ = //) THEN
20              FOR EACH q ∈ Q DO
21                  FOR EACH q' ∈ Q DO
```

```
22                      IF (sym.bin[i] = 1) OR (sym.bin[j] = 1) THEN
23                          δ ← δ ∪ {(sym, (q, q'), q_error)}
24                      ELSE
25                          IF (q = q_wait) THEN
26                              δ ← δ ∪ {(sym, (q, q'), q')}
27                          ELIF (q' = q_wait) THEN
28                              δ ← δ ∪ {(sym, (q, q'), q)}
29                          ELSE
30                              δ ← δ ∪ {(sym, (q, q'), q_error)}
31
32          ELIF (sym.σ = fg_k) THEN
33              IF (sym.bin[j] = 1) THEN
34                  FOR EACH q ∈ Q DO
35                      δ ← δ ∪ {(sym, (q), q_error)}
36              ELSE
37                  IF (sym.bin[i] = 1) THEN
38                      FOR EACH q ∈ Q DO
39                          IF (q = q_{s−m1−s−m2}) THEN
40                              IF (k = m1) THEN
41                                  δ ← δ ∪ {(sym, (q), q_incid)}
42                              ELIF (k = m2) THEN
43                                  δ ← δ ∪ {(sym, (q), q_incid)}
44                              ELSE
45                                  δ ← δ ∪ {(sym, (q), q_error)}
46                          ELIF (q = q_{f−m1−s−m2}) THEN
47                              IF (k = m2) THEN
48                                  δ ← δ ∪ {(sym, (q), q_incid)}
49                              ELSE
50                                  δ ← δ ∪ {(sym, (q), q_error)}
51                          ELIF (q = q_{s−m1−f−m2}) THEN
52                              IF (k = m1) THEN
53                                  δ ← δ ∪ {(sym, (q), q_incid)}
54                              ELSE
55                                  δ ← δ ∪ {(sym, (q), q_error)}
56                          ELSE
57                              δ ← δ ∪ {(sym, (q), q_error)}
58                  ELSE
59                      FOR EACH q ∈ Q DO
60                          IF (q = q_{s−m1−s−m2}) THEN
61                              IF (k = m1) THEN
62                                  δ ← δ ∪ {(sym, (q), q_{f−m1−s−m2})}
63                              ELIF (k = m2) THEN
64                                  δ ← δ ∪ {(sym, (q), q_{s−m1−f−m2})}
65                              ELSE
66                                  δ ← δ ∪ {(sym, (q), q)}
67                          ELIF (q = q_{f−m1−s−m2}) THEN
```

```
68                              IF (k = m_2) THEN
69                                  δ ← δ ∪ {(sym, (q), q_{f−m_1−f−m_2})}
70                              ELSE
71                                  δ ← δ ∪ {(sym, (q), q)}
72                          ELIF (q = q_{s−m_1−f−m_2}) THEN
73                              IF (k = m_1) THEN
74                                  δ ← δ ∪ {(sym, (q), q_{f−m_1−f−m_2})}
75                              ELSE
76                                  δ ← δ ∪ {(sym, (q), q)}
77                          ELSE
78                              δ ← δ ∪ {(sym, (q), q)}
79            ELSE
80                IF (sym.bin[i] = 0) THEN
81                    IF (sym.bin[j] = 1) THEN
82                        m_1 ← sym.σ[1]
83                        m_2 ← sym.σ[2]
84                        δ ← δ ∪ {(sym, ∅, q_{s−m_1−s−m_2})}
85                    ELSE
86                        δ ← δ ∪ {(sym, ∅, q_wait)}
87                ELSE
88                    δ ← δ ∪ {(sym, ∅, q_error)}
89        automaton ← ⟨Σ′, Q, Q_f, δ⟩
90        RETURN automaton
```

Algoritmo 5.11: Algoritmo para *tree automaton* para o predicado de incidência

Teorema 5.2.7 (Corretude do Algoritmo 5.11)**.** *Dado que Σ' é um alfabeto estendido, marked é o conjunto de marcações utilizadas, $1 \leq i \leq c$, $1 \leq j \leq c$ e $i \neq j$, onde c é o número de conjuntos mencionados na fórmula, o Algoritmo 5.11 constrói um* tree automaton *que reconhece uma* tree expression *quando o elemento do conjunto unitário X_j é uma folha rotulada por um símbolo para aresta que é incidente ao símbolo de esquecimento do conjunto unitário X_i, o qual denota um vértice.*

Demonstração. Para garantir que o algoritmo se comporta como o previsto no teorema, deve-se garantir três coisas:

1. Os conjuntos X_i e X_j são unitários.

   Se ambos os conjuntos forem vazios, as transições das linhas 26, 28, 78 e 86 permitem que o autômato termine a execução no estado q_{wait}, o qual não é um estado de aceitação.

   Assuma $X_i = \emptyset$, $X_j \neq \emptyset$ e $|X_j| = 1$. Assim, uma transição da linha 84 será executada. Em algum momento futuro, duas transições dentre as das linhas 62, 64,

69 e 74 serão executadas, indicando que foram esquecidas as marcações relacionadas aos extremos daquela aresta. As transições das linhas 26, 28 e 78 vão propagar as informações encontradas até que o *tree automaton* termine a execução num estado da forma $q_{f-m_1-f-m_2}$. Tal estado não é um estado de aceitação.

Assuma $X_i \neq \emptyset$, $|X_i| = 1$ e $X_j = \emptyset$. Transições da linha 86 serão executadas. Em algum momento no futuro, o *tree automaton* executará uma transição da linha 57, colocando o dispositivo no estado q_{error}. As transições das linhas 30 e 78 garantem que o *tree automaton* termine a execução no estado q_{error}, que também não é um estado de aceitação.

Se ao menos um deles possuir cardinalidade maior ou igual a dois, as transições das linhas 23, 30, 35, 45, 50, 55 e 57 se encarregam de direcionar a execução para o estado q_{error}, de onde o *tree automaton* não consegue sair.

Logo, todas as suposições apresentadas conduzem o *tree automaton* a estados que não são de aceitação. Portanto, os conjuntos são unitários.

2. Os conjuntos X_i e X_j são adequados.

   Se o elemento de X_i for uma folha, uma transição da forma $(sym, \emptyset, q_{error})$ é executada (linha 88).

   Se o elemento de X_j for um símbolo de esquecimento, uma transição da forma descrita na linha 35 é executada.

   Se o elemento de X_i e/ou o elemento de X_j for uma composição paralela, uma transição da forma descrita na linha 23 é executada.

   Dado que os três cenários apresentados conduzem o *tree automaton* para o estado q_{error}, que não é o estado de aceitação, garante-se que os conjuntos X_i e X_j são adequados.

3. Os conjuntos X_i e X_j estão relacionados.

   Garantir que os conjuntos estão relacionados quer dizer que uma das marcações que compõem o símbolo da folha contida em X_j é esquecida no símbolo de esquecimento contido em X_i, sem ter passado por outro esquecimento relacionado a mesma marcação. Suponha que o *tree automaton* aceita a *tree expression* quando os conjuntos não estão relacionados. Isso implica em dois cenários.

   (a) A marcação esquecida não está presente na aresta.

Este cenário implica estar num estado da forma $q_{s-m_1-s-m_2}$, para $m_1, m_2 \in marked$ e lendo um símbolo fg_k, de modo que $k \neq m_1 \neq m_2$. Este estado satisfaz a condição da linha 39, porém dado que $k \neq m_1 \neq m_2$, a transição da linha 45 será executada, levando o *tree automaton* para o estado q_{error}. Logo, esse cenário contradiz a suposição apresentada.

(b) A marcação sendo esquecida já foi esquecida anteriormente.

Este caso indica que o dispositivo se encontra em $q_{f-m_1-s-m_2}$, $q_{s-m_1-f-m_2}$ ou $q_{f-m_1-f-m_2}$, que representa que durante a execução o *tree automaton* encontrou um símbolo fg_{m_1}, fg_{m_2} ou ambos, respectivamente, mas que este achado não pertencia ao conjunto X_i. Assim,

- se o estado for $q_{f-m_1-s-m_2}$ e o elemento de X_i for fg_{m_1}, o *tree automaton* executa a transição da linha 50, indo para q_{error}.
- se o estado for $q_{s-m_1-f-m_2}$ e o elemento de X_i for fg_{m_2}, o *tree automaton* executa a transição da linha 55, indo para q_{error}.
- se o estado for $q_{f-m_1-f-m_2}$ e o elemento de X_i for fg_{m_1} ou fg_{m_2}, o *tree automaton* executa a transição da linha 57, indo para q_{error}.

Portanto, o *tree automaton* termina a execução em um estado não final, pois não poderá se livrar do estado q_{error}. Isso também contradiz a suposição apresentada.

Deste modo, *tree automaton* construído reconhece uma *tree expression* quando o elemento do conjunto unitário X_j é uma folha rotulada por um símbolo para aresta que é incidente ao símbolo de esquecimento do conjunto unitário X_i, o qual denota um vértice. □

5.2.2 Subrotinas para composição e projeção de *tree automata*

Observando o algoritmo para normalizar a fórmula em MSOL apresentado na Seção 2.4, nota-se que a fórmula gerada pelo processo fica expressa em termos de conjunções, disjunções, negações e quantificações existenciais e universais. Todavia, quantificações universais serão substituídas por quantificações existenciais ao aplicar a identidade $\forall X.\varphi \equiv \neg\exists X.\neg\varphi$ por meio do algoritmo que traduz fórmulas. Por isso, nesta subseção serão explorados apenas os algoritmos para a construção de *tree automata* para conjunção, disjunção, negação e quantificação universal, que representam um conjunto básico de elementos que pode expressar outros conectivos e a quantificação universal.

Construir o *tree automaton* para uma fórmula $\exists X_i.\varphi$ utiliza o conceito de **projeção** [16] em autômato. Os símbolos do alfabeto estendido são visualizados como uma sobreposição de símbolos. Observe que a construção da forma estendida assume que, para cada conjunto X_i, existe uma *string* binária de mesmo tamanho da árvore de entrada, a qual codifica se uma determinada posição da árvore pertence ou não àquele conjunto. Portanto, essa operação de combinar o modelo com as atribuições das variáveis livres contidas na fórmula compõem o processo de **cilindrificação** [16]. A justificativa para o uso da cilindrificação é dada pela recursividade do algoritmo. Para processar uma fórmula da forma $\exists X_i.\varphi$, primeiro é necessário construir um autômato que reconheça a fórmula φ, a qual terá a variável X_i livre. Após a construção deste autômato, apaga-se da *string* de controle do alfabeto e das transições desse autômato a i-ésima posição, que corresponde a X_i. Apagar essa posição da *string* de controle significa encontrar não-deterministicamente, se possível, uma atribuição que garanta o reconhecimento da entrada. Esta operação de projeção potencialmente torna as transições do autômato não-determinísticas.

Assumindo que as fórmulas que representam as propriedades em grafos estudadas com o Teorema de Courcelle não possuem variáveis livres e que todas serão colocadas na forma normal prenex, a operação de projeção pode, simplesmente, ser vista como o ato de apagar o último elemento da *string* de controle. Note que se não há variáveis livres, o número de projeções a serem realizadas é igual ao *quantifier rank* da fórmula. Portanto, toda a *string* de controle será apagada ao final da construção do *tree automaton*. Além disso, pela estrutura recursiva do algoritmo, os bits da *string* de controle serão apagados de trás para frente. O Algoritmo 5.12 apresenta o método de projeção no alfabeto e nas transições de um *tree automaton*. A função $truncate(\sigma)$ apaga o último bit da *string* de controle de um símbolo σ do alfabeto estendido. Assuma que cada transição é uma tupla da forma $(sym, son\text{-}states, target)$, onde sym é um símbolo do alfabeto estendido.

```
1   projection(auto)
2       newAlphabet ← ∅
3       FOR EACH σ ∈ auto.Σ DO
4           newAlphabet ← truncate(σ)
5       newTransitions ← ∅
6       FOR EACH transition ∈ auto.δ DO
7           symbol ← truncate(transition → sym)
8           sources ← transition.son-states
9           target ← transition.target
10          newTransitions ← (symbol, sources, target)
11      auto → Σ ← newAlphabet
12      auto → δ ← newTransitions
```

```
13      RETURN auto
```

Algoritmo 5.12: Algoritmo para projeção em um *tree automaton*

As construções dos *tree automata* para fórmulas do tipo $\varphi \wedge \psi$ e $\varphi \vee \psi$ são muito semelhantes. Suponha que se conheçam os *tree automata* $\mathcal{A}_\varphi = \langle \Sigma, Q_\varphi, Q_{F\varphi}, \delta \rangle$ e $\mathcal{A}_\psi = \langle \Sigma, Q_\psi, Q_{F\psi}, \delta' \rangle$ que reconhecem as fórmulas φ e ψ. A construção do *tree automaton* para $\varphi \wedge \psi$ resulta em $\mathcal{A}_{inter} = \langle \Sigma, (Q_\varphi \times Q_\psi), (Q_{F\varphi} \times Q_{F\psi}), \delta'' \rangle$, que é o *tree automaton* que representa a interseção das linguagens reconhecidas por $\mathcal{A}_\varphi$ e $\mathcal{A}_\psi$. A construção do *tree automaton* para $\varphi \vee \psi$ resulta em $\mathcal{A}_{union} = \langle \Sigma, (Q_\varphi \times Q_\psi), ((Q_\varphi \times Q_{F\psi}) \cup (Q_{F\varphi} \times Q_\psi)), \delta'' \rangle$, que é o *tree automaton* que representa a união das linguagens reconhecidas por $\mathcal{A}_\varphi$ e $\mathcal{A}_\psi$. O espaço de estados dos *tree automata* $\mathcal{A}_{inter}$ e $\mathcal{A}_{union}$ é o mesmo, que é construído como o produto cartesiano dos espaços de estados de $\mathcal{A}_\varphi$ e $\mathcal{A}_\psi$. Os conjuntos de estados de aceitação destes *tree automaton* estão definidos de maneira clara pela notação matemática. Já o produto cartesiano das transições é dado no Algoritmo 5.13. Assuma que cada transição é uma tupla da forma $(sym, son\text{-}states, target)$, onde sym é um símbolo do alfabeto estendido e $son\text{-}states$ é uma tupla de estados de cardinalidade 1 ou 2.

```
1   compound-transitions(transitions1,transitions2)
2       compoundedTransitions ← ∅
3       FOR EACH trans1 ∈ transitions1 DO
4           FOR EACH trans2 ∈ transitions2 DO
5               IF (trans1.sym = trans2.sym) THEN
6                   IF (|trans1.son-states| = 2) THEN
7                       source1 ← (trans1.son-states[1] × trans2.son-states[1])
8                       source2 ← (trans1.son-states[2] × trans2.son-states[2])
9                       sources ← (source1, source2)
10                  ELSE
11                      sources ← (trans1.son-states[1] × trans2.son-states[1])
12                  target ← (trans1.target × trans2.target)
13                  compoundedTransitions ← (trans1.sym, sources, target)
14      RETURN compoundedTransitions
```

Algoritmo 5.13: Algoritmo para composição de transições de *tree automaton* para MSOL_2

A construção de um *tree automaton* pra uma fórmula do tipo $\neg\varphi$ é dada pelo complemento do *tree automaton* para φ. O complemento de um *tree automaton* é definido matematicamente por $\bar{\mathcal{A}} = \langle \Sigma, Q_\varphi, (Q_\varphi \backslash Q_{F\varphi}), \delta \rangle$ assumindo a existência do *tree automaton* $\mathcal{A} = \langle \Sigma, Q_\varphi, Q_{F\varphi}, \delta \rangle$. A construção simplesmente transforma estados que não eram de aceitação em estados de aceitação e vice-versa. Um ponto crítico nessa construção é

que a função de transição do *tree automaton* deve ser total. Para totalizar a função de transição de um *tree automaton*, utilize o Algoritmo 5.14 em um autômato determinístico. O símbolo "$-$" nas linhas 8, 14 e 19 do Algoritmo 5.14 denotam a irrelevância do estado destino da transição. O símbolo "fg_m" na linha 12 do Algoritmo 5.14 denota qualquer símbolo de esquecimento. O algoritmo busca identificar transições faltantes e completa-las sem alterar a linguagem reconhecida, por isso a necessidade do estado denominado *stuck*.

```
1   total-transitions(auto)
2       newStates ← auto → Q ∪ {stuck}
3       totalTransitions ← ∅
4       FOR EACH σ ∈ auto.Σ DO
5           IF (σ = //) THEN
6               FOR EACH q1 ∈ newStates DO
7                   FOR EACH q2 ∈ newStates DO
8                       IF ∃t : (t ∈ auto.δ) ∧ (t = (σ, {q1, q2}, −)) THEN
9                           totalTransitions ← totalTransitions ∪ {t}
10                      ELSE
11                          totalTransitions ← totalTransitions ∪ {(t, {q1, q2}, stuck)}
12          ELIF (σ = fg_m) THEN
13              FOR EACH q ∈ newStates DO
14                  IF ∃t : (t ∈ auto.δ) ∧ (t = (σ, {q}, −)) THEN
15                      totalTransitions ← totalTransitions ∪ {t}
16                  ELSE
17                      totalTransitions ← totalTransitions ∪ {(t, {q}, stuck)}
18          ELSE
19              IF ∃t : (t ∈ auto.δ) ∧ (t = (σ, ∅, −)) THEN
20                  totalTransitions ← totalTransitions ∪ {t}
21              ELSE
22                  totalTransitions ← totalTransitions ∪ {(t, ∅, stuck)}
23      auto → Q ← newStates
24      auto → δ ← totalTransitions
25      RETURN auto
```

Algoritmo 5.14: Algoritmo para completar a função de transição de um *tree automaton* para MSOL_2

A última etapa a ser tratada refere-se à determinização de um *tree automaton*. Por mais que o não-determinismo gere autômatos enxutos, ele não é algo implementável idealmente. Portanto, ao final do processo de projeções da fórmula MSOL_2 denotada por φ, o *tree automaton* obtido é não-determinístico. Assim, o Algoritmo 5.15 apresenta o processo de determinização de um *tree automaton* para MSOL_2. O algoritmo assume que transições não-determinísticas estejam comprimidas, ou seja, se existirem quaisquer

transições $(sym, (q1, q2), q')$ e $(sym, (q1, q2), q'')$, elas estarão compactadas numa transição da forma $(sym, (q1, q2), \{q', q''\})$, por exemplo. Considere toda transição da forma $(sym, sources, target)$, onde $sources$ é um conjunto de estados. Lembre-se que o conjunto $newStates$ é um conjunto de conjuntos de estados do *tree automaton* de entrada.

```
1   Determinize-automaton-msol2(auto)
2       newAlphabet ← auto.Σ
3       newStates ← ∅
4       finalStates ← ∅
5       δ ← ∅
6       recent ← ∅
7
8       REPEAT
9           FOR EACH σ ∈ newAlphabet DO
10              IF (σ = //) THEN
11                  IF (newStates ≠ ∅) THEN
12                      FOR EACH state1 ∈ newStates DO
13                          FOR EACH state2 ∈ newStates DO
14                              reached ← ∅
15                              FOR EACH s1 ∈ state1 DO
16                                  FOR EACH s2 ∈ state2 DO
17                                      FOR EACH t ∈ auto → δ DO
18                                          IF (t → sym = σ) AND (t.sources = {s1, s2})
                                             THEN
19                                              reached ← reached ∪ t.target
20                              recent ← recent ∪ {reached}
21                              δ ← δ ∪ {(σ, {state1, state2}, reached)}
22
23              ELIF (σ = fg_m) THEN
24                  IF (newStates ≠ ∅) THEN
25                      FOR EACH state ∈ newStates DO
26                          reached ← ∅
27                          FOR EACH s ∈ state DO
28                              FOR EACH t ∈ auto.δ DO
29                                  IF (t → sym = σ) AND (t.sources = s) THEN
30                                      reached ← reached ∪ t.target
31                          recent ← recent ∪ {reached}
32                          δ ← δ ∪ {(σ, state, reached)}
33
34              ELSE
35                  IF (newStates = ∅) THEN
36                      FOR EACH t ∈ auto.δ DO
37                          IF (t → sym = σ) THEN
38                              δ ← δ ∪ {t}
39                              recent ← recent ∪ {t.target}
```

```
40          recent ← recent − newStates
41          IF (recent ≠ ∅) THEN
42              change ← TRUE
43              newStates ← newStates ∪ recent
44          ELSE
45              change ← FALSE
46      UNTIL (change = FALSE)
47
48      FOR EACH state ∈ newStates DO
49          FOR EACH s ∈ states DO
50              IF (s ∈ auto → Q_f) THEN
51                  finalStates ← finalStates ∪ {states}
52                  BREAK
53
54      determinized ← ⟨newAlphabet, newStates, finalStates, δ⟩
55      RETURN determinized
```

Algoritmo 5.15: Algoritmo para determinização de *tree automaton* para MSOL_2

Ao obter a versão determinística do *tree automaton* para a fórmula MSOL_2 que expressa a propriedade desejada, basta utilizar as transições do *tree automaton* para processar a *tree expression* que codifica o grafo de entrada. Se o estado encontrado após a leitura da raiz da *tree expression* for um estado de aceitação, então o grafo de entrada possui a propriedade verificada. Caso contrário, o grafo não possui tal propriedade.

Capítulo 6

Considerações Finais e Trabalhos Futuros

As características arquetípicas do Teorema de Courcelle ficam melhor evidenciadas ao ter uma visão completa do processo. Dado que o processo utiliza uma estratégia de *model checking* parametrizado pelo tamanho da fórmula (fórmula fixa), o algoritmo desenvolvido a partir do teorema é adaptável a qualquer problema expressível em MSOL_2. Portanto, cria-se uma sensação de que o Teorema de Courcelle é um padrão para algoritmos que resolvam problemas distintos. Por outro lado, como o alfabeto do autômato a ser construído para a execução do *model checking* é dependente do limitante da *treewidth*, diferentes estratos de *treewidth* são tratados por um mesmo autômato. Suponha que seja construído um autômato para um problema Π assumindo um limitante da *treewidth* igual a 5. Este mesmo autômato pode reconhecer qualquer grafo que possua a propriedade em questão, mesmo que esse grafo possua *treewidth* menor que o limitante. Os autômatos que leriam os grafos considerando *treewidth* igual a 5 e os grafos que possuem *treewidth* igual a 3, por exemplo, não diferem em nada senão no alfabeto e nas transições. Em particular, o alfabeto e as transições do autômato para *treewidth* 3 são subconjuntos do alfabeto e das transições do autômato para *treewidth* 5. Tais características confirmam a grande importância do Teorema de Courcelle como uma ferramenta na resolução de problemas computacionais.

A formalização algorítmica proposta nas páginas desta dissertação cobrem completamente o processo enunciado por Courcelle, principalmente quando comparado com o estado da arte dos trabalhos de Black [6] e Kneis [41, 42]. A abstração dos métodos para cálculo de *tree decompositions* e *treewidth* se deve à vasta literatura sobre o assunto, podendo então ser utilizado o método que o leitor julgar mais adequado. De fato, o algoritmo proposto difere em alguns detalhes tanto com a descrição do artigo de Courcelle de 1992 [19] quanto com a descrição contida no livro do Courcelle de 2012 [21]. As dife-

renças introduzidas na formalização se devem, principalmente, a fatores didáticos. O uso de *extended nice tree decompositions* para a construção da *tree expression* associada ao grafo torna o processo mais natural e de fácil compreensão por parte de leitores pouco familiarizados com esses conceitos. A estratégia baseada em coloração, apresentada no Capítulo 4 torna o processo mais controlado, impactando diretamente na descrição algorítmica da construção do autômato para o predicado $inc(X_i, X_j)$. Dado que a marcação associada a um vértice não muda, evita-se definir transições que lidem com essa alteração para garantir a corretude do autômato. Mesmo essa etapa sendo uma etapa não existente no processo original, foi demonstrado que ela não altera o resultado do Teorema de Courcelle, produzindo ainda um algoritmo **FPT**.

O conteúdo desta dissertação provê oportunidades de trabalhos ainda não exploradas. Uma vez que esta dissertação explora apenas a versão clássica do Teorema, é interessante que sejam estudadas alterações dela para lidar com problemas de otimização. Nesse contexto, MSOL é substituída por variantes mais adequadas a essa tarefa, como a *Counting Monadic Second-Order Logic* (CMSOL) [18, 26] e a Lógica Monádica de Segunda Ordem com uma função de avaliação linear (linEMSOL) [20, 25]. Desse modo, pode-se propor o desenvolvimento de um *solver* baseado em *model checking*. Dado que o Teorema de Courcelle é bastante usado na literatura, uma ferramenta que implemente suas ideias soa como uma boa alternativa para aplicação em cenários reais, incluindo aqueles que envolvem otimização. Neste ponto, surge uma nova vertente de investigação.

A alternância de quantificadores nas fórmulas em MSOL_2 podem causar uma certa degeneração na função $f(\kappa)$ tornando-a uma torre exponencial. As palavras "torre" e "exponencial" naturalmente causam pânico em qualquer acadêmico da área de otimização de algoritmos, principalmente no momento histórico em que o Teorema de Courcelle foi proposto. Todavia, mais de 25 anos se passaram desde a apresentação do Teorema e avanços significativos foram obtidos no que diz respeito ao *hardware* dos computadores. Investigar o desempenho da abordagem apresentada para o Teorema de Courcelle pode revelar dados interessantes. Dada uma maior disponibilidade de memória, quais os limites que uma implementação deste algoritmo pode atingir? Qual o tamanho máximo de instâncias podem ser processado em tempo factível? Qual o comportamento da curva de crescimento da função $f(\kappa)$ em cenários reais de problemas de otimização? Todas essas questões oferecem um bom ponto de partida para um estudo empírico sobre um algoritmo baseado em *model checking* em comparação com as abordagens tradicionais da literatura para problemas de otimização.

Para encerrar, o parâmetro *treewidth* é muito relacionado com outro parâmetro, proposto por Courcelle, chamado de *clique width*. Esses parâmetros estão relacionados não só do ponto de vista estrutural na Teoria da Complexidade Parametrizada, visto que há uma razão de proporcionalidade entre eles, mas também do ponto de vista formal das *graph grammars*. Enquanto as gramáticas de substituição de hiperarestas são tradicionalmente associadas a *treewidth*, as gramáticas de substituição de vértices são normalmente relacionadas com a *clique width*. Ambos os tipos de gramática também estão relacionados, dado que algumas HRG podem ser simuladas através de NRG. Portanto, uma terceira investigação interessante seria como aproveitar conceitos explorados para *treewidth* em cenários que consideram a *clique width*. Dado que projetar algoritmos baseados em programação dinâmica para *clique width* é complexo e, por vezes, trabalhoso, a ideia de um *framework* baseado em *model checking* para problemas parametrizados pela *clique width* se mostra como uma alternativa promissora e atraente.

Referências

[1] Aho, A. V.; Sethi, R.; Ullman, J. D. Compilers: Principles, techniques, and tools addison-wesley. *Reading, Mass* (1986).

[2] Amir, E. Approximation algorithms for treewidth. *Algorithmica 56*, 4 (2010), 448–479.

[3] Baier, C.; Katoen, J.-P. *Principles of Model Checking (Representation and Mind Series)*. The MIT Press, 2008.

[4] Bauderon, M.; Courcelle, B. Graph expressions and graph rewritings. *Mathematical Systems Theory 20*, 1 (1987), 83–127.

[5] Berge, C. *Hypergraphs: combinatorics of finite sets*, vol. 45. Elsevier, 1984.

[6] Black, E. Towards a formal verification of courcelle's theorem.

[7] Bodlaender, H. L. Dynamic programming on graphs with bounded treewidth. In *International Colloquium on Automata, Languages, and Programming* (1988), Springer, pp. 105–118.

[8] Bodlaender, H. L. A linear-time algorithm for finding tree-decompositions of small treewidth. *SIAM Journal on computing 25*, 6 (1996), 1305–1317.

[9] Bodlaender, H. L.; Drange, P. G.; Dregi, M. S.; Fomin, F. V.; Lokshtanov, D.; Pilipczuk, M. A $c^k n$ 5-approximation algorithm for treewidth. *SIAM Journal on Computing 45*, 2 (2016), 317–378.

[10] Bodlaender, H. L.; Fomin, F. V.; Koster, A. M.; Kratsch, D.; Thilikos, D. M. On exact algorithms for treewidth. In *European Symposium on Algorithms* (2006), Springer, pp. 672–683.

[11] Bondy, J. A.; Murty, U. S. R. *Graph theory with applications*. Macmillan London, 1976.

[12] Büchi, J. R. Weak second-order arithmetic and finite automata. *Mathematical Logic Quarterly 6*, 1-6 (1960), 66–92.

[13] Chartrand, G.; Zhang, P. *A First Course in Graph Theory*. Dover books on mathematics. Dover Publications, 2012.

[14] Clarke, E. M. Model checking. In *International Conference on Foundations of Software Technology and Theoretical Computer Science* (1997), Springer, pp. 54–56.

[15] Clarke, E. M.; Henzinger, T. A.; Veith, H.; Bloem, R. *Handbook of model checking*. Springer, 2018.

[16] Comon, H.; Dauchet, M.; Gilleron, R.; Löding, C.; Jacquemard, F.; Lugiez, D.; Tison, S.; Tommasi, M. Tree automata techniques and applications. Available on: `http://www.grappa.univ-lille3.fr/tata`, 2007. release October, 12th 2007.

[17] Cormen, T. H.; Leiserson, C. E.; Rivest, R. L.; Stein, C. *Introduction to Algorithms, Third Edition*, 3rd ed. The MIT Press, 2009.

[18] Courcelle, B. The monadic second-order logic of graphs v: On closing the gap between definability and recognizability. *Theoretical Computer Science 80*, 2 (1991), 153–202.

[19] Courcelle, B. The monadic second-order logic of graphs iii : tree-decompositions, minors and complexity issues. *RAIRO - Theoretical Informatics and Applications - Informatique Théorique et Applications 26*, 3 (1992), 257–286.

[20] Courcelle, B. The monadic second-order logic of graphs x: Linear orderings. *Theoretical Computer Science 160*, 1-2 (1996), 87–143.

[21] Courcelle, B. *Graph Structure and Monadic Second-Order Logic: A Language-Theoretic Approach.* Cambridge University Press, 2012.

[22] Courcelle, B. Fly-automata for checking monadic second-order properties of graphs of bounded tree-width. *Electronic Notes in Discrete Mathematics 50* (2015), 3–8.

[23] Courcelle, B.; Durand, I. Automata for the verification of monadic second-order graph properties. *J. Applied Logic 10* (2012), 368–409.

[24] Courcelle, B.; Durand, I. A. Fly-automata, their properties and applications. In *International Conference on Implementation and Application of Automata* (2011), Springer, pp. 264–272.

[25] Courcelle, B.; Makowsky, J. A.; Rotics, U. Linear time solvable optimization problems on graphs of bounded clique width. In *International Workshop on Graph-Theoretic Concepts in Computer Science* (1998), Springer, pp. 1–16.

[26] Courcelle, B.; Makowsky, J. A.; Rotics, U. On the fixed parameter complexity of graph enumeration problems definable in monadic second-order logic. *Discrete Applied Mathematics 108*, 1-2 (2001), 23–52.

[27] Cygan, M.; Fomin, F. V.; Kowalik, L.; Lokshtanov, D.; Marx, D.; Pilipczuk, M.; Pilipczuk, M.; Saurabh, S. *Parameterized Algorithms*, 1st ed. Springer Publishing Company, Incorporated, 2015.

[28] Downey, R. G.; Fellows, M. R. *Parameterized complexity.* Springer Science & Business Media, 2012.

[29] Downey, R. G.; Fellows, M. R. *Fundamentals of Parameterized Complexity.* Springer Publishing Company, Incorporated, 2013.

[30] Drewes, F.; Kreowski, H.-J. A note on hyperedge replacement. In *International Workshop on Graph Grammars and Their Application to Computer Science* (1990), Springer, pp. 1–11.

[31] Drewes, F.; Kreowski, H.-J. A note on hyperedge replacement. In *Graph Grammars and Their Application to Computer Science* (Berlin, Heidelberg, 1991), H. Ehrig, H.-J. Kreowski, and G. Rozenberg, Eds., Springer Berlin Heidelberg, pp. 1–11.

[32] Engelfriet, J. Handbook of formal languages, vol. 3. Springer-Verlag New York, Inc., New York, NY, USA, 1997, ch. Context-free Graph Grammars, pp. 125–213.

[33] Flum, J.; Grohe, M. *Parameterized Complexity Theory (Texts in Theoretical Computer Science. An EATCS Series).* Springer-Verlag, Berlin, Heidelberg, 2006.

[34] Fomin, F. V.; Kratsch, D.; Todinca, I. Exact (exponential) algorithms for treewidth and minimum fill-in. In *International Colloquium on Automata, Languages, and Programming* (2004), Springer, pp. 568–580.

[35] Greenlaw, R.; Hoover, J. H. *Fundamentals of the Theory of Computation: Principles and Practice.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1998.

[36] Grohe, M. Logic, graphs, and algorithms. *Electronic Colloquium on Computational Complexity (ECCC) 14* (2007).

[37] Grohe, M. Algorithmic meta theorems for sparse graph classes. In *Computer Science - Theory and Applications* (Cham, 2014), E. A. Hirsch, S. O. Kuznetsov, J.-É. Pin, and N. K. Vereshchagin, Eds., Springer International Publishing, pp. 16–22.

[38] Gross, J. L.; Yellen, J. *Handbook of graph theory.* CRC press, 2004.

[39] Habel, A. *Hyperedge replacement: grammars and languages*, vol. 643. Springer Science & Business Media, 1992.

[40] Hopcroft, J. E.; Motwani, R.; Ullman, J. D. *Introduction to Automata Theory, Languages, and Computation.* Addison-Wesley, 3rd Edition, 2006.

[41] Kneis, J.; Langer, A. A practical approach to courcelle's theorem. *Electronic Notes in Theoretical Computer Science 251* (2009), 65–81.

[42] Kneis, J.; Langer, A.; Rossmanith, P. Courcelle's theorem – a game-theoretic approach. *Discrete Optimization 8*, 4 (2011), 568–594.

[43] Knuth, D. E. *Selected Papers on Analysis of Algorithms.* CSLI Publications, Stanford, CA, USA, 2000.

[44] Kreutzer, S. Algorithmic meta-theorems. In *Parameterized and Exact Computation* (Berlin, Heidelberg, 2008), M. Grohe and R. Niedermeier, Eds., Springer Berlin Heidelberg, pp. 10–12.

[45] Langer, A.; Reidl, F.; Rossmanith, P.; Sikdar, S. Practical algorithms for mso model-checking on tree-decomposable graphs. *Computer Science Review 13-14* (2014), 39–74.

[46] Leveson, N. G.; Turner, C. S. An investigation of the therac-25 accidents. *Computer 26*, 7 (July 1993), 18–41.

[47] Menezes, P. B. *Linguagens formais e autômatos*, 6^a ed. 2011.

[48] Mezei, J.; Wright, J. B. Algebraic automata and context-free sets. *Information and control 11*, 1-2 (1967), 3–29.

[49] Robertson, N.; Seymour, P. D. Graph minors. ii. algorithmic aspects of tree-width. *Journal of algorithms 7*, 3 (1986), 309–322.

[50] Robertson, N.; Seymour, P. D. Graph minors: X. obstructions to tree-decomposition. *J. Comb. Theory Ser. B 52*, 2 (June 1991), 153–190.

[51] Robertson, N.; Seymour, P. D. Graph minors. xiii: The disjoint paths problem. *J. Comb. Theory Ser. B 63*, 1 (Jan. 1995), 65–110.

[52] Rozenberg, G. Handbook of graph grammars and computing by graph transformation.

[53] Thatcher, J. W.; Wright, J. B. Generalized finite automata theory with an application to a decision problem of second-order logic. *Mathematical systems theory 2*, 1 (1968), 57–81.

[54] Thomas, W. Languages, automata, and logic. In *Handbook of formal languages*. Springer, 1997, pp. 389–455.

[55] Vardi, M. Y. The complexity of relational query languages. In *Proceedings of the fourteenth annual ACM symposium on Theory of computing* (1982), ACM, pp. 137–146.