

REVISITANDO O TEOREMA DE COURCELLE

Maurício da Silva Pires

Instituto de Computação, Universidade Federal Fluminense - UFF
Av. Gal. Milton Tavares de Souza s/n, Gragoatá
24210-346 – Niterói, RJ
mspires@id.uff.br

Bruno Lopes Vieira

Instituto de Computação, Universidade Federal Fluminense - UFF
Av. Gal. Milton Tavares de Souza s/n, Gragoatá
24210-346 – Niterói, RJ
bruno@ic.uff.br

Uéverton dos Santos Souza

Instituto de Computação, Universidade Federal Fluminense - UFF
Av. Gal. Milton Tavares de Souza s/n, Gragoatá
24210-346 – Niterói, RJ
ueverton@ic.uff.br

RESUMO

A Complexidade Parametrizada é uma teoria que propõe análise e projeto multivariado de algoritmos. O Teorema de Courcelle é um dos resultados parametrizados mais famosos, estabelecendo que problemas expressíveis em Lógica Monádica de Segunda Ordem podem ser resolvidos eficientemente em classes de grafos que possuem *treewidth* limitada. O teorema está definido matematicamente e, apesar de haver parcial algoritmização, não foi encontrada na literatura uma versão totalmente algorítmica e/ou implementação do procedimento. Este trabalho busca fornecer uma descrição algorítmica para o Teorema de Courcelle, permitindo que este seja usado para resolução efetiva de problemas em vez de apenas provar a existência de soluções eficientes.

PALAVRAS CHAVE. Metateoremas. Complexidade Parametrizada. Verificação de Modelos.

Tópicos. TAG - Teoria e Algoritmos em Grafos.

ABSTRACT

Parametrized Complexity is a theory that proposes multivariate analysis and design of algorithms. Courcelle's Theorem is one of the most famous parametrized results, stating that problems expressible in Monadic Second-Order Logic can be solved efficiently in classes of graphs that have bounded treewidth. The theorem is defined mathematically and, although there is partial algorithmization, a completely algorithmic version and/or implementation of the procedure was not found in the literature. This work seeks to provide an algorithmic description for Courcelle's Theorem, allowing it to be used for effective problem solving rather than just proving the existence of efficient solutions.

KEYWORDS. Metatheorems. Parameterized Complexity. Model Checking.

Paper topics. TAG - Theory and Algorithms in Graphs.

1. Introdução

A **Teoria da Complexidade Parametrizada** [Flum e Grohe, 2006; Downey e Fellows, 2012, 2013] propõe uma nova forma de projetar e analisar algoritmos. Classicamente, algoritmos são construídos considerando apenas o tamanho da entrada e são avaliados quanto ao número de passos a serem executados e à quantidade de memória necessária a essa execução. Nesta teoria, os algoritmos devem considerar diferentes aspectos dos dados de entrada. Isto significa que um problema, por exemplo a 3-coloração, pode ser estudado sobre diferentes pontos de vista.

Um problema parametrizado é representado por um par (Π, κ) , onde Π é um problema de decisão em seu sentido usual e κ é o chamado parâmetro, que é um mapeamento computável em tempo polinomial que associa uma instância de Π a um número inteiro não-negativo. Considerando que Π seja um problema em grafos, parâmetros utilizados para projetar um algoritmo podem ser o tamanho da solução procurada, o tamanho da cobertura por vértices ou o quão próximo de uma árvore é aquele grafo, por exemplo.

Assim, a Teoria da Complexidade Parametrizada apresenta ferramentas para análise em uma granularidade mais fina. Isto implica em uma análise mais acurada dos problemas NP-difíceis e NP-completos, dado que diferentes aspectos desses problemas podem contribuir mais ou menos para sua intratabilidade.

Como o parâmetro normalmente é muito menor que o tamanho da entrada, algoritmos que apresentem exponencialidade no parâmetro ainda podem fornecer boas soluções. Um problema parametrizado (Π, κ) é dito **tratável por parâmetro fixo** [Flum e Grohe, 2006; Downey e Fellows, 2012, 2013] quando é possível projetar um algoritmo cuja complexidade é da forma $f(k) \cdot n^c$, onde c é uma constante e $f(k)$ é qualquer função computável no parâmetro. Problemas parametrizados que possuem algoritmos com essa complexidade são ditos pertencentes à classe **FPT**. Para valores pequenos de κ , algoritmos **FPT** são resolvidos eficientemente.

Courcelle [1992] enunciou um teorema que relaciona problemas que podem ser expressados em Lógica Monádica de Segunda Ordem com quantificação sobre vértices, arestas e conjuntos de vértices e arestas (MSOL_2) com a existência de um algoritmo **FPT** dado um parâmetro que mede o quão próximo de uma árvore é um grafo, a *treewidth*. Este resultado vai além de uma única instância de problema, permitindo ser aplicado a todos os casos onde o grafo de entrada possui *treewidth* limitada e o problema pode ser expresso por uma fórmula em MSOL_2 . Isso faz com que este metateorema seja muito utilizado na literatura para provar a pertinência de problemas à classe **FPT**, porém pouco se explora dos processos algorítmicos envolvidos nele.

O objetivo deste trabalho é revisitar apresentando uma interpretação algorítmica do Teorema de Courcelle a fim de aproveitar os aspectos práticos deste metateorema. Cada uma das etapas será explicada ressaltando os aspectos mecanizáveis do processo a fim de obter uma estruturação algorítmica passível de implementação. Assim, problemas que são expressíveis em MSOL_2 poderão ser solucionados por este algoritmo quando o grafo de entrada tiver *treewidth* limitada.

Este artigo é organizado como se segue. A Seção 2 apresenta os conceitos básicos necessários à compreensão do trabalho desenvolvido por Courcelle. As Seções 3 e 4 discutem as etapas de maior complexidade do algoritmo proposto. Por fim, a Seção 5 contém as considerações finais e oportunidades de trabalhos futuros.

2. Preliminares

A *treewidth* é um parâmetro introduzido por Robertson e Seymour [1986], visando capturar o quão similar a uma árvore é um grafo G qualquer. Muitos algoritmos são eficientes em árvores, portanto, o uso de *tree decompositions* provê uma nova maneira de se projetar algoritmos baseados em programação dinâmica que possivelmente pertencem à classe **FPT**.

Uma *tree decomposition* [Downey e Fellows, 2013; Cygan et al., 2015], apresentada na Definição 1, é uma reinterpretação de um grafo a fim de avaliar sua similaridade com uma árvore. Denomina-se **bag** um subconjunto de vértices de um grafo G . A **largura de uma tree decomposition**, denotada por $wid(\mathcal{T})$, é a cardinalidade máxima dentre os *bags* associados aos nós da sua árvore, ou seja, $wid(\mathcal{T}) = \max\{|t| : t \in V(T)\}$. Portanto, define-se a **treewidth** de um grafo G como sendo $twd(G) = \min\{wid(\mathcal{T}) - 1\}$, ou seja, é a menor largura menos uma unidade dentre **todas** as *tree decompositions* possíveis para G . A Figura 1 apresenta um exemplo de um grafo e uma de suas *tree decompositions* com largura 3.

Definição 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 de raiz $r(T)$;

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

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

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

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 $f(t_1) \cap f(t_3) \subseteq f(t_2)$;

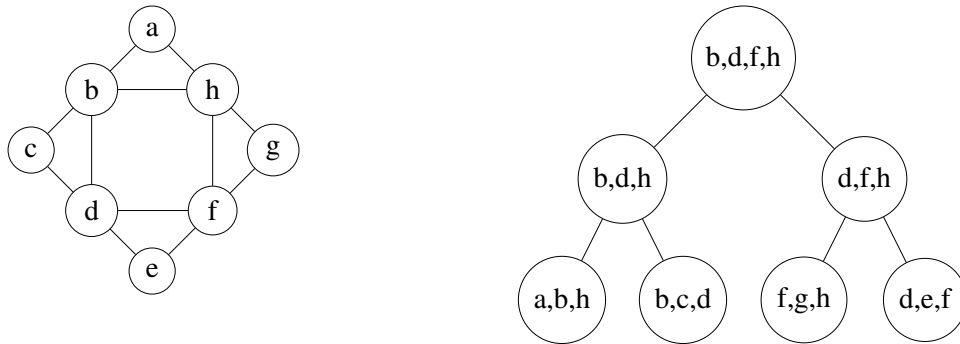


Figura 1: Exemplo de um grafo e uma *tree decomposition* para ele.

Cygan et al. [2015] definiu uma extensão da *tree decomposition* chamada *extended nice tree decomposition*. Considere $\mathcal{T}' = \langle T', f' \rangle$ como a notação para uma *extended nice tree decomposition*. Essa decomposição assume que tanto a raiz quanto as folhas da árvore possuem *bags* vazios, ou seja, $f'(r(T')) = \emptyset$ e $f'(t) = \emptyset$, para toda folha t de T' . Todos os nós não-folhas da árvore T' são de algum dos tipos abaixo:

Inserção de vértice : Um nó que possui apenas um filho. Seja t um nó deste tipo e t' seu filho em T' . Assim, $f'(t) = f'(t') \cup \{v\}$, para algum $v \notin f'(t')$.

Inserção de aresta : Um nó que possui apenas um filho. Seja t este nó e t' seu filho em T' . Considera-se que a aresta uv foi introduzida quando $f'(t) = f'(t')$, $\{u, v\} \subseteq f'(t)$ e t é rotulado pela aresta uv .

Esquecimento de vértice : Um nó que possui apenas um filho. Seja t um nó deste tipo e t' seu filho em T' . Assim, $f'(t) = f'(t') \setminus \{v\}$, para algum $v \in f'(t')$.

Junção : Um nó que possui exatamente dois filhos. Sejam t um nó deste tipo e t' e t'' seus filhos em T' . Logo, $f'(t) = f'(t') \cup f'(t'')$.

Este trabalho não assume o processo de encontrar uma *tree decomposition* para um grafo G como uma etapa do Teorema de Courcelle, visto que ele foi provado sendo solucionável em tempo **FPT** por Bodlaender [1996]. Portanto, o Teorema 1 é usado como processamento inicial necessário à aplicação do Teorema 2 propriamente dito.

Teorema 1 (Bodlaender). *Existe um algoritmo **FPT** que determina se um grafo G possui treewidth no máximo k .*

A **Lógica Monádica de Segunda Ordem** é uma extensão da **Lógica de Primeira Ordem** que permite realizar quantificações sobre predicados monádicos, ou seja, predicados de um único argumento. Predicados monádicos, quando interpretados em um modelo, definem conjuntos. Portanto, esta lógica pode expressar muitos problemas em grafos, como 3-coloração e conjunto dominante por exemplo. Usualmente são destacados dois fragmentos de MSOL para tratar problemas em grafos. O primeiro, chamado $MSOL_1$, permite que sejam feitas quantificações sobre vértices e conjuntos de vértices. O segundo, denominado $MSOL_2$, é uma extensão de $MSOL_1$ que inclui quantificação sobre arestas e conjuntos de arestas. Deste modo, fica claro que há problemas que são expressíveis em $MSOL_2$ que não conseguem ser expressos em $MSOL_1$, como o *k-Path Problem* por exemplo. O trabalho de Courcelle revisitado neste artigo considera fórmulas em $MSOL_2$.

O Teorema 2 é enunciado por Courcelle [1992] sobre a noção de hipergrafos com *sources*. O uso de hipergrafos simplifica os processos intermediários da prova e do algoritmo proposto. É importante ressaltar que todo grafo, no sentido usual da literatura, pode ser visto como um hipergrafo sem *sources* onde toda hiperaresta incide em exatamente dois vértices. Notações mais recentes do Teorema 2 desconsideram a entrada como um sendo um hipergrafo, devido às rotinas de pré processamento enunciadas por Courcelle [Courcelle, 1992] que efetivamente reduzem o hipergrafo a um grafo.

Teorema 2 (Courcelle). *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 processo encapsulado pelo Teorema 2 pode ser descrito pelo Algoritmo 1. Este algoritmo é uma versão simplificada, encapsulando processos complexos em um único passo. Tais processos serão explorados nas seções posteriores, com exceção da obtenção da *tree decomposition*.

Entrada: um grafo G e uma fórmula φ .

Parâmetro: um inteiro k .

Questão: $G \models \varphi$?

```

twd  $\leftarrow$  treewidth do grafo  $G$ ;
if  $twd \leq k$  then
    phi-automaton  $\leftarrow$  autômato construído a partir de  $\varphi$ ;
    tree-dec  $\leftarrow$  uma tree decomposition para  $G$ ;
    g-expression  $\leftarrow$  a expressão HR que denota tree-dec;
    processar g-expression com phi-automaton;
    if g-expression é aceita then
        |  $G \models \varphi$ ;
    else
        |  $G \not\models \varphi$ ;
    end
else
    | abortar;
end

```

Algoritmo 1: Algoritmo simplificado para o Teorema de Courcelle.

3. Expressões definindo grafos

A primeira etapa a ser solucionada pelo Teorema de Courcelle é a tradução do grafo de entrada em uma expressão que deverá ser processada posteriormente por um *tree automaton*. Esta tradução deve considerar a relação entre a expressão e o autômato a ser gerado. A construção deve ser tal que um número finito de símbolos seja utilizado, uma vez que estes símbolos serão considerados o alfabeto do *tree automaton*.

Definir grafos por meio de um número finito de símbolos não é uma tarefa trivial. Como demonstrado por Courcelle, o uso de *tree decompositions* auxilia no controle dos símbolos, visto que, em cada nó da árvore que representa a decomposição, gasta-se uma quantidade linear na $twd(G)$ de símbolos para realizar o processamento da expressão.

Um **alfabeto** é um conjunto de símbolos. Um **alfabeto ranqueado** é uma dupla $\langle A, r \rangle$, onde A é um alfabeto e $r: A \rightarrow \mathbb{N}$ é a chamada função de *ranking*. Este último conceito é essencial para a Definição 2, pois cada hipergrafo deve estar associado a um alfabeto ranqueado. Apesar do emprego do termo *source* é importante ressaltar a inexistência de qualquer noção de fluxo envolvida.

Definição 2 (n-Hipergrafo com *sources*). Um **n-hipergrafo com sources** é uma quintupla

$$\mathcal{H} = \langle V, E, lab_{\mathcal{H}}, vert_{\mathcal{H}}, src_{\mathcal{H}} \rangle$$

onde:

V é o conjunto de vértices;

E é o conjunto de arestas;

$lab_{\mathcal{H}}: E \rightarrow A$, é a função que associa a cada aresta um rótulo;

$vert_{\mathcal{H}}: E \rightarrow 2^V$, é a função que associa a cada aresta e um conjunto de vértices de cardinalidade $r(e)$;

src_H é uma sequência de n vértices destacados de V , chamados de *sources*.

Os *sources* [Courcelle, 1992, 2012] mencionados são apenas vértices destacados que serão utilizados para o processo de montagem do hipergrafo por meio da “colagem” dos vértices. Esta noção de “colagem” é proveniente das chamadas **gramáticas de grafos de substituição de hiperarestas** (*hyperedge replacement graph grammars*) [Rozenberg, 1997], ou simplesmente gramáticas **HR**. Tais gramáticas baseiam-se no princípio de que grafos podem ser construídos a partir de um elemento básico: hiperarestas. Assim, grafos mais complexos são vistos como um aglutinado de arestas unidas por seus vértices (pontos de colagem). Portanto, podemos definir uma álgebra para construir hipergrafos a partir das hiperarestas e algumas operações. Este trabalho segue os padrões apresentados por Courcelle [2012] na Definição 3.

Definição 3 (Operações sobre hipergrafo). Considere as seguintes operações:

$\emptyset$ é a constante que representa um hipergrafo vazio.

ab é a constante que representa um hipergrafo composto por uma única aresta incidente em dois vértices que são *sources* marcados com a e b , onde $a, b \in \mathcal{L}$ que é um conjunto de marcações especiais.

$//$ é a composição paralela. Dados dois hipergrafos com *sources*, este operador pode possuir dois comportamentos. Sejam G e H hipergrafos com *sources*. Se $src_G \cap src_H = \emptyset$, então o operador se comporta como uma união disjunta que apenas concatena as sequências de *sources*. Caso contrário, o operador fará uma “união disjunta”, porém eliminando as redundâncias da sequência de *sources* resultante. Isso implica colapsar alguns *sources*, unindo assim os hipergrafos.

fg_a é o esquecimento do *source* a . Como o nome sugere, esta operação retira um *source* da sequência do hipergrafo e remove a marcação a do hipergrafo.

$ren_{a \rightarrow b}$ é a substituição do *source* a por b . Esta operação causa uma reorganização na sequência de *sources* do hipergrafo. O vértice marcado por a passa a ser marcado por b e a marcação a é removida.

O processo de *parsing* do grafo de entrada em uma expressão segundo a álgebra apresentada pode ser feito de diferentes maneiras a partir de *tree decompositions*. Courcelle apresenta o processo fazendo uso de uma versão binária de uma *tree decomposition* arbitrária [Courcelle, 1992] e uma variação da *tree decomposition* na qual as arestas devem estar contidas em um **único bag** [Courcelle, 2012]. Neste trabalho, o processo de *parsing* será apresentado sobre o conceito de *extended nice tree decomposition* [Cygan et al., 2015].

A construção apresentada a seguir considera um algoritmo de programação dinâmica para o processamento da expressão. Inicialmente, o leitor deve assumir que as operações realizadas em cada nó da *extended nice tree decomposition* tem custo $O(1)$. Uma discussão detalhada sobre estes custos e operações é feita em seguida.

Considere exp a função de construção da expressão de G . Sejam t, t' e t'' nós da *extended nice tree decomposition*. Para cada um dos tipos de nós da decomposição de entrada, é definido um processamento parcial da expressão apresentado a seguir:

Nós folha : Como as folhas da decomposição não possuem informações de vértices de G associadas, a função que explora esse nó deve retornar um símbolo que represente um grafo vazio.

De um ponto de vista efetivo, este símbolo pode ser eliminado da expressão posteriormente. Logo, $exp[t] = 0$.

Nós de inserção de vértice : Em gramáticas HR, os vértices não são considerados elementos básicos, portanto esses nós não possuem nenhuma função relevante para esta construção. Assim, $exp[t] = exp[t']$, onde t' é filho de t .

Nós de inserção de aresta : Nós deste tipo possuem apenas um filho. Basta usar um símbolo para a aresta inserida e utilizar uma composição paralela com a expressão parcial contida no filho deste nó. Se ab é a aresta inserida, então $exp[t] = ab/exp[t']$, onde t' é filho de t .

Nós de esquecimento de vértice : Este tipo de nó permite eliminar vértices cuja vizinhança já tenha sido completamente processada. Portanto, se v é o vértice a ser esquecido e ele está marcado com a na sequência de *sources* da solução parcial, basta fazer $exp[t] = fg_a(exp[t'])$, onde t' é filho de t .

Nós de junção : Este tipo de nó possui dois filhos, de modo que ambos os filhos possuem os mesmos vértices associados que o pai. Logo, basta fazer $exp[t] = exp[t']/exp[t'']$.

A construção apresentada considera uma série de simplificações. Primeiro, toda aresta inserida é considerada como um 2-grafo, ou seja, um hipergrafo com 2 vértices marcados e uma aresta que incide nesses vértices. Segundo, se uma aresta é denotada por ab , significa que a e b são as marcações colocadas nos *sources* deste hipergrafo parcial. Por fim, essas considerações implicam em um controle maior do alfabeto da linguagem que será processada por um *tree automata* (Definição 4). Este alfabeto possui tamanho $O(k^2)$, onde $twd(G) = k$.

Nós de junção podem apresentar um pequeno problema. Suponha que um (possivelmente mais de um) vértice esteja com marcações diferentes nos filhos do nó de junção. Realizar uma composição paralela indiscriminada acarreta uma expressão que não representa o grafo de entrada. Este tipo de problema pode ser facilmente corrigido incluindo na função um vetor com as marcações e aplicar a operação $ren_{a \rightarrow b}$, onde a e b são marcações, até que todos os vértices marcados possuam marcações consistentes. Em contrapartida, operações de alteração de marcação demandam mais transições no autômato para garantir um processamento correto. Assim, o ideal seria que, previamente, os vértices do grafo de entrada pudessem ser associados exclusivamente a marcações, de modo que elas não sejam alteradas durante o *parsing* do grafo na expressão. A estratégia sugerida é baseada em coloração de grafos cordais. Dado que um grafo G com $twd(G) = k$ é subgrafo de uma k -árvore (uma subclasse de grafos cordais), uma $k + 1$ -coloração própria dos vértices de G pode ser computada em tempo polinomial.

O objetivo é fazer com que cada nó da árvore de decomposição esteja associada à uma clique, o que fará com que os vértices do grafo sejam coloridos com $twd(G) + 1$ cores que estarão associadas às marcações usadas para construir a expressão que denota o grafo. Cada um dos processos necessários a essa estratégia (cordalização, coloração de grafos cordais e remoção das arestas artificiais) demanda tempo polinomial. Logo, a complexidade assintótica do Algoritmo 1 permanece **FPT**.

4. Fórmulas MSOL₂ compiladas em *tree automata*

A segunda etapa encapsulada no Teorema 2 é a compilação da fórmula em MSOL₂ em um autômato que irá ler a expressão construída na seção anterior. Essa construção é idealizada a partir da técnica de *model checking* [Clarke, 1997; Clarke et al., 2018]. O autômato a ser construído deve

aceitar apenas os grafos que possuem a propriedade representada pela fórmula φ . Se a expressão que representa o grafo de entrada for reconhecida pelo autômato, tem-se a garantia de que este grafo possui a propriedade desejada. Como o autômato a ser construído é determinístico, o processo de verificação é executado em tempo **FPT**.

Uma **tree language** [Comon et al., 2007] é uma generalização das linguagens de *strings* presentes na Hierarquia de Chomsky [Hopcroft et al., 2006]. Essas linguagens são construídas sobre **alfabetos ranqueados**, de modo que o *rank* de um determinado símbolo determina quantos filhos o nó da árvore rotulado por ele deve ter. Desta forma, *strings* são árvores construídas sobre um alfabeto no qual todo símbolo possui *rank* no máximo um.

Da mesma forma, **tree automata** generalizam os autômatos a fim de processar árvores [Comon et al., 2007]. Estes autômatos podem ser construídos em duas abordagens. A abordagem utilizada por Courcelle [1992, 2012] é a *bottom-up*, que considera a computação da árvore das folhas para a raiz, como apresentada na Definição 4.

Definição 4 (Bottom-up finite tree automaton). Seja $\Sigma = \langle A, r \rangle$ um alfabeto ranqueado. Um **(bottom-up finite) tree automaton** é uma quádrupla $\mathcal{A} = \langle \Sigma, Q, Q_F, \delta \rangle$, onde

Q é o conjunto de estados do autômato.

$Q_F \subseteq Q$ é o conjunto de estados de aceitação (ou finais) do autômato.

δ é o conjunto de transições do autômato, onde $\delta : \Sigma \times 2^Q \rightarrow Q$. Em particular, se $(\sigma, S) \in \text{Dom}(\delta)$, onde $\sigma \in \Sigma$ e $S \subseteq 2^Q$, então $|S| = r(\sigma)$.

Para ilustrar o funcionamento de um *tree automaton*, considere o autômato cuja linguagem aceita é aquela cujas expressões montadas com *true*, *false*, *not*, *and* e *or* retornam *true*. Seja $\mathcal{A}_{TF} = \langle \Sigma, Q, Q_F, \delta \rangle$ onde:

$$\Sigma = \langle \{or, and, not, true, false\}, \{(true, 0), (false, 0), (not, 1), (or, 2), (and, 2)\} \rangle$$

$$Q = \{Q_{true}, Q_{false}\}$$

$$Q_f = \{Q_{true}\}$$

$$\begin{aligned} \delta = \{ & false \rightarrow Q_{false}, true \rightarrow Q_{true}, not(Q_{false}) \rightarrow Q_{true}, not(Q_{true}) \rightarrow Q_{false}, \\ & and(Q_{false}, Q_{false}) \rightarrow Q_{false}, and(Q_{false}, Q_{true}) \rightarrow Q_{false}, \\ & and(Q_{true}, Q_{false}) \rightarrow Q_{false}, and(Q_{true}, Q_{true}) \rightarrow Q_{true}, \\ & or(Q_{false}, Q_{false}) \rightarrow Q_{false}, or(Q_{false}, Q_{true}) \rightarrow Q_{true}, \\ & or(Q_{true}, Q_{false}) \rightarrow Q_{true}, or(Q_{true}, Q_{true}) \rightarrow Q_{true} \}. \end{aligned}$$

Considere como entrada para este autômato a expressão $or(and(true, false), not(false))$. A execução do *tree automaton* começa, de modo não determinístico, nas folhas da árvore. O resultado é obtido após o processamento da raiz. O processamento de $or(and(true, false), not(false))$ por $\mathcal{A}_{TF}$ é exibido na Figura 2. É importante ressaltar que $\mathcal{A}_{TF}$ é um autômato determinístico. *Tree automata* não determinísticos são aqueles onde há pelo menos duas transições com o mesmo lado esquerdo e lados direitos diferentes.

De modo semelhante, as operações apresentadas na Definição 3 serão utilizadas como alfabeto para a construção do autômato. A operação de composição paralela, denotada por $//$, será considerada um símbolo de *rank* 2. O símbolo de grafo vazio, denotado por 0, é uma constante, i.e.,

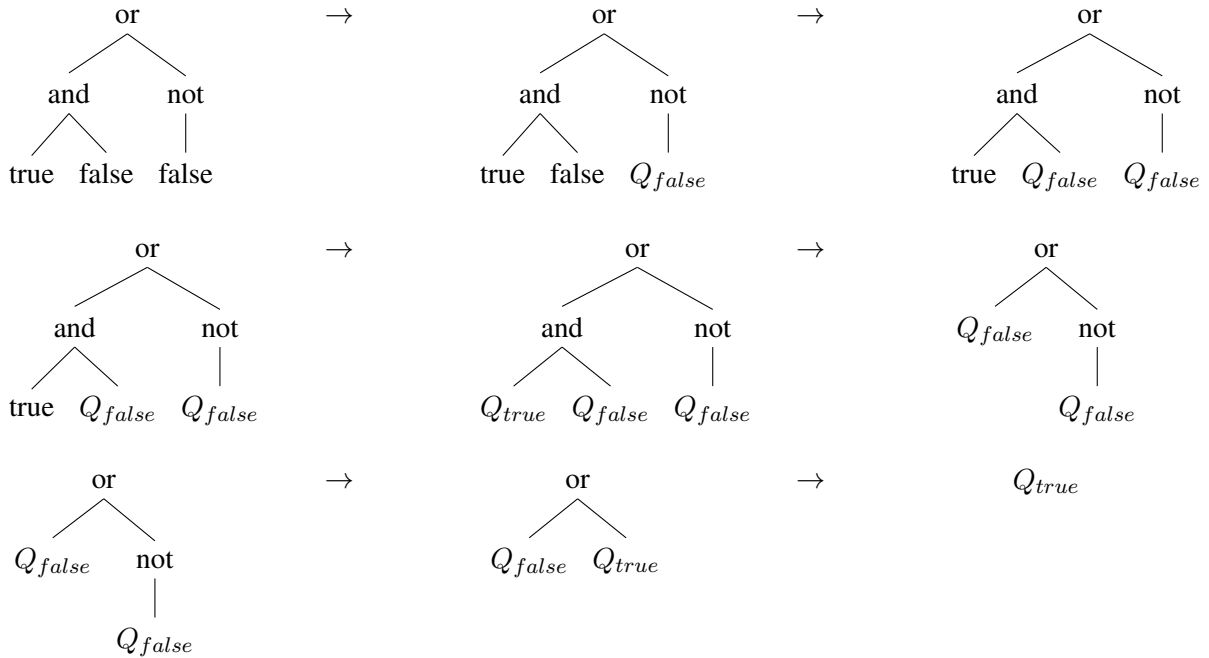


Figura 2: Exemplo da execução de $\mathcal{A}_{TF}$.

tem *rank* 0. Dado que a *treewidth* do grafo é limitada por k , assumindo a princípio a inexistência de excesso de marcações para os *sources*, serão necessários $k + 1$ símbolos de esquecimento, um por marcação, de *rank* 1. Lembre-se que, se $twd(G) = k$, então o maior *bag* de G contém $k + 1$ vértices. Os símbolos denotando os hipergrafos de uma aresta e dois vértices marcados devem ser tais que cubram toda combinação de marcações tomadas em pares não ordenados. Isso demanda um número quadrático na $twd(G)$ de símbolos para as arestas. Cada um destes símbolos deve ser de *rank* 0. Note que as arestas e o grafo vazio serão as folhas da árvore a ser processada.

A fórmula em Lógica Monádica de Segunda Ordem que representa o problema a ser solucionado deve ser compilado em um *tree automaton*. A existência deste autômato é uma consequência do *Weak Recognizability Theorem* [Courcelle, 1990, 2012].

Courcelle [2012] sugere uma normalização da fórmula em MSOL. Esta normalização utiliza alguns artifícios a fim de facilitar a construção do autômato por meio de uma ideia semelhante à proposta para a tradução de expressões regulares em autômatos finitos [Aho et al., 1986]. Por exemplo, quantificações de primeira ordem são substituídas por quantificações de segunda ordem com a restrição de que tais conjuntos sejam unitários. Determinadas partes da fórmula em MSOL podem ser substituídas por determinados predicados, para os quais é “fácil” construir um autômato que reconheça tal propriedade, como verificar se um conjunto é ou não unitário ou se existe uma aresta entre dois vértices. Tais predicados são denominados **fórmulas atômicas**.

Os autômatos serão construídos sobre a noção de **alfabeto expandido**. Este alfabeto expandido é obtido ao acoplar uma *string* binária de comprimento igual ao *quantifier rank* da fórmula φ a cada símbolo do alfabeto. Note que este novo alfabeto continua limitado pela *treewidth* do grafo de entrada, dado que a fórmula a ser verificada é fixa. Por exemplo, se uma fórmula φ em forma normal prenex (todos os quantificados foram posicionados à frente da fórmula) possui cinco variáveis ligadas a quantificadores, o *quantifier rank* de φ será igual a cinco. Essas *strings* binárias

serão uma codificação para indicar que uma determinada posição da expressão, que está associada a algum símbolo, pertence ou não pertence a um determinado conjunto. Suponha que, em um dado momento, o autômato leia um símbolo da forma $(ab, 00110)$. Isso significa que a posição lida está rotulada por ab e que ela pertence aos conjuntos 2 e 3, assumindo uma ordenação da esquerda para a direita das variáveis quantificadas em φ que está na forma normal prenex. Lembre-se que a normalização que Courcelle sugere usa apenas quantificações sobre conjuntos.

De modo semelhante ao Algoritmo de Thompson [Aho et al., 1986], o *tree automaton* resultante é obtido por uma construção indutiva na estrutura da fórmula normalizada. Assim, o processo de construção do *tree automaton* para uma fórmula φ é feito da seguinte maneira:

- Se φ é uma fórmula atômica, construa um autômato que reconheça esta fórmula.

A construção desses autômatos depende das variáveis envolvidas nas fórmulas atômicas em questão. Em geral, será necessário sempre observar os bits da string binária. Por exemplo, se a fórmula atômica for $X_i \subseteq X_j$, as transições do autômato devem conduzir à aceitação sempre que se o símbolo lido possuir o i -ésimo bit da *string* binária igual a 1, o j -ésimo bit também seja 1.

- Se $\varphi = \neg\phi$, o autômato a ser construído é o complemento do autômato que aceita ϕ .

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

- Se $\varphi = \phi \wedge \psi$, o autômato a ser construído é a interseção dos autômatos 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 autômatos que aceitam expressões que denotam grafos que possuem as propriedades ϕ e ψ , respectivamente. O autômato que reconhece φ será $\mathcal{A}_\varphi = \langle \Sigma, (Q_\phi \times Q_\psi), (Q_{F\phi} \times Q_{F\psi}), \delta'' \rangle$.

- Se $\varphi = \phi \vee \psi$, o autômato a ser construído é a união dos autômatos 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 autômatos que aceitam expressões que denotam grafos que possuem as propriedades ϕ e ψ , respectivamente. O autômato 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 autômato a ser construído será obtido por meio da projeção [Comon et al., 2007] do bit da *string* binária que representa conjunto X . Assim, o bit que denota esse conjunto será apagado das transições, o que implicará na aparição de transições não-determinísticas. O autômato obtido deve ser determinizado, como apresentado por Aho et al. [1986].

O processo de construção deste autômato demanda tempo $O(\text{size}(G))$ [Courcelle, 1992], onde G é o hipergrafo de entrada. O processo de execução do autômato para a fórmula φ sobre a expressão definindo o hipergrafo é **FPT** segundo Grohe [2008] e Grohe e Kreutzer [2011]. Note que todas as partes do processo destacado na Seção 2 podem ser solucionadas em tempo **FPT**. Portanto, o algoritmo de Courcelle é **FPT** em relação a *treewidth* do hipergrafo de entrada quando a fórmula de entrada φ é fixa.

5. Considerações Finais

Desmistificado o processo algorítmico idealizado por Courcelle, é possível identificar diferentes formas de se construir este algoritmo. Ao usar *extended nice tree decompositions* para realizar o processo de *parsing* do grafo em uma expressão HR, agrega-se a literatura um novo método de construção, não só eficiente como didático. Assim, esta primeira etapa encapsulada do algoritmo se torna mais amistosa para leitores que não estão acostumados com a notação complexa usualmente utilizada por Courcelle. É importante ressaltar que este método continua não alterando o resultado principal, que é a obtenção de um algoritmo **FPT** para o problema.

Em contrapartida, a etapa de construção de um autômato a partir de uma fórmula em MSOL_2 continua presa a complexidade de se construir um autômato para as chamadas **fórmulas atômicas**. Em geral, essas fórmulas estarão intimamente associadas à fórmula φ que representa a propriedade procurada. Ainda que Courcelle [2012] apresente exemplos de autômatos que verificam algumas dessas fórmulas, como o teste de um conjunto ser unitário, são construções pouco claras e intuitivas. Nem toda propriedade pode ser construída baseada nos exemplos apresentados.

Assim, o trabalho apresentado se mostra como uma luz para o processo de *model checking* usado por Courcelle para resolver problemas expressíveis em MSOL_2 em grafos que possuem *treewidth* limitada. Isso implica em dois novos direcionamentos. O primeiro se concentra numa investigação mais aprofundada sobre a forma de construir autômatos baseados em MSOL_2 , de modo a fornecer uma construção mais acessível para leitores que queiram iniciar nessa área de pesquisa. O segundo direcionamento se dá na exploração do uso de *model checking* para resolução de problemas algorítmicos, explorando novas lógicas em combinação com novos parâmetros. Este tipo de estudo resultará na classificação de diferentes problemas parametrizados dada a escolha de uma lógica e um parâmetro.

Em tempo, vale ressaltar que dado a limitação sobre o número de páginas, esse trabalho não visa apresentar todos os detalhes inerentes à demonstração do Teorema de Courcelle, sendo, no entanto, um material introdutório para pesquisadores interessados no tema. O teorema em sua completude pode ser encontrado no livro do Courcelle [2012].

Agradecimentos

Os autores agradecem a CAPES - Coordenação de Aperfeiçoamento de Pessoal de Nível Superior, ao CNPq - Conselho Nacional de Desenvolvimento Científico e Tecnológico, e a FAPERJ - Fundação Carlos Chagas Filho de Amparo à Pesquisa do Estado do Rio de Janeiro pelo apoio financeiro a essa pesquisa.

Referências

- Aho, A. V., Sethi, R., e Ullman, J. D. (1986). *Compilers: Principles, techniques, and tools* addison-wesley. Reading, Mass.
- Bodlaender, H. L. (1996). A linear-time algorithm for finding tree-decompositions of small treewidth. *SIAM Journal on computing*, 25(6):1305–1317.
- Clarke, E. M. (1997). Model checking. In *International Conference on Foundations of Software Technology and Theoretical Computer Science*, p. 54–56. Springer.
- Clarke, E. M., Henzinger, T. A., Veith, H., e Bloem, R. (2018). *Handbook of model checking*. Springer.

- Comon, H., Dauchet, M., Gilleron, R., Löding, C., Jacquemard, F., Lugiez, D., Tison, S., e Tommasi, M. (2007). Tree automata techniques and applications. Available on: <http://www.grappa.univ-lille3.fr/tata>. release October, 12th 2007.
- Courcelle, B. (1992). 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):257–286.
- Courcelle, B. (2012). *Graph Structure and Monadic Second-Order Logic: A Language-Theoretic Approach*. Cambridge University Press.
- Courcelle, B. (1990). The monadic second-order logic of graphs. i. recognizable sets of finite graphs. *Inf. Comput.*, 85(1):12–75. ISSN 0890-5401.
- Cygan, M., Fomin, F. V., Kowalik, L., Lokshtanov, D., Marx, D., Pilipczuk, M., Pilipczuk, M., e Saurabh, S. (2015). *Parameterized Algorithms*. Springer Publishing Company, Incorporated, 1st edition. ISBN 3319212745, 9783319212746.
- Downey, R. G. e Fellows, M. R. (2013). *Fundamentals of parameterized complexity*, volume 4. Springer.
- Downey, R. G. e Fellows, M. R. (2012). *Parameterized complexity*. Springer Science & Business Media.
- Flum, J. e Grohe, M. (2006). *Parameterized Complexity Theory (Texts in Theoretical Computer Science. An EATCS Series)*. Springer-Verlag, Berlin, Heidelberg. ISBN 3540299521.
- Grohe, M. (2008). Logic, graphs, and algorithms. In *Logic and automata*, p. 357–422.
- Grohe, M. e Kreutzer, S. (2011). Methods for algorithmic meta theorems. *Model Theoretic Methods in Finite Combinatorics*, 558:181–206.
- Hopcroft, J. E., Motwani, R., e Ullman, J. D. (2006). Introduction to automata theory, languages, and computation.
- Robertson, N. e Seymour, P. D. (1986). Graph minors. ii. algorithmic aspects of tree-width. *Journal of algorithms*, 7(3):309–322.
- Rozenberg, G. (1997). Handbook of graph grammars and computing by graph transformation.