Oberlin
Digital Commons at Oberlin

Honors Papers | Student Work

2020

Courcelle's Theorem: Overview and Applications

Samuel Frederic Barr
Oberlin College

Follow this and additional works at: https://digitalcommons.oberlin.edu/honors

Part of the Computer Sciences Commons

Repository Citation

Barr, Samuel Frederic, "Courcelle's Theorem: Overview and Applications" (2020). *Honors Papers*. 679.
https://digitalcommons.oberlin.edu/honors/679

This Thesis is brought to you for free and open access by the Student Work at Digital Commons at Oberlin. It has been accepted for inclusion in Honors Papers by an authorized administrator of Digital Commons at Oberlin. For more information, please contact megan.mitchell@oberlin.edu.

Courcelle's Theorem: Overview and Applications

Sam Barr
Advisor: Robert Geitz

Contents

I Courcelle's Theorem **1**

1 Parameterized Complexity **2**
1.1 Tree Decompositions . . . 3

2 Algebra **5**
2.1 F-Magmas . . . 5
2.2 Homomorphism and Automata . . . 7
2.3 Graphs as Algebraic Objects . . . 8
2.4 Tree Decompositions and Graph Expressions . . . 10

3 Logic **11**
3.1 Graphs as Logical Objects . . . 11
3.2 Logical Languages . . . 12
3.2.1 First Order Logic . . . 12
3.2.2 Second Order Logic . . . 12
3.2.3 Monadic Second Order Logic . . . 13

4 Courcelle's Theorem **14**
4.1 Overview of Proof . . . 14
4.1.1 Preliminaries . . . 14
4.1.2 The Algorithm . . . 16
4.2 The Conclusion, or Maybe, the Punchline . . . 17

II Expressing Graph Properties in Monadic Second Order Logic **18**

5 Basic Properties **19**
5.1 Vertex Cover of Size K . . . 19
5.2 Dominating Set of size K . . . 19
5.3 Domatic Number K . . . 19
5.4 K-colorability . . . 20
5.5 Achromatic Number K . . . 20
5.6 K-edge-colorability . . . 20
5.7 Minimum Maximal Matching . . . 20
5.8 Supergraph of H . . . 21

6 Advanced Techniques **22**
6.1 Reflexive, Transitive Closure . . . 22
6.2 Transductions . . . 23

7 Advanced Properties **25**
7.1 Subgraph with property P . . . 25
7.2 Partition into K subgraphs with property P . . . 25
7.3 Supergraph with K extra edges with property P . . . 26
7.4 Hamiltonian Cycle . . . 26
7.5 K-connected . . . 26
7.6 Monochromatic Triangle . . . 26
7.7 Perfect Graph . . . 27
7.8 Complement has property P . . . 28
7.9 Feedback Vertex Set of size K . . . 28

Part I

Courcelle's Theorem

Chapter 1

Parameterized Complexity

In the study of algorithms, and complexity theory in general, we are primarily concerned with how the time to compute some property of an input scales with the *size* of the input. When speaking about graphs, we want to know how the time it takes to decide whether a graph has a certain property scales with the number of vertices and edges of the graph. To give a well known example, the Edmonds-Karp algorithm decides whether a flow network has a maximum flow greater than some integer k in time $O(|V||E|^2)$. In particular, it can be computed in *polynomial time* with respect to size of the inputs. Famously, many graph problems are not known to have polynomial time algorithms. It is still an open question whether deciding if a graph can be colored with k colors or has a vertex cover of size k can be computed efficiently.

In Parameterized Complexity Theory, we consider how difficult it is to compute a property in terms of the size of the input, and in terms of some *parameter* of the input. How does the difficulty of computing a vertex cover of size k scale as we increase k? How does the difficulty of deciding if a graph can be k-colored scale as we increase k? From a theoretical standpoint, this allows us to better understand where the difficulty of a problem is. The problem might not get that much more difficult as the input increases in size, but some other structural component of the input may be increasing the required time. From a practical point of view, parameterized complexity theory can help in designing more efficient algorithms for NP hard problems. For instance, consider the following algorithm for the Vertex Cover problem:

function VC($k, G = (V, E)$)
  if $E = \emptyset$ **then**
    return True
  else if $k = 0$ **then**
    return False
  else
    Arbitrarily pick $(u, v) \in E$
    $G_u \leftarrow (V \setminus \{u\}, \{e \in E : u \notin e\})$
    $G_v \leftarrow (V \setminus \{v\}, \{e \in E : v \notin e\})$
    return VC($k-1, G_u$) **or** VC($k-1, G_v$)
  end if
end function

This algorithm can compute whether a graph G has a vertex cover of size k in time $O(2^k|E|)$ (proof is left as an exercise for the reader). From this algorithm we can see that, for fixed k, vertex cover can be solved in polynomial time! Have we solved P=NP? Do I get to cash in my million dollar check? Not quite — the complexity classes P and NP are defined based off running time in terms of the size of the input. Since our result is dependent on the parameter k, it has no bearing on whether Vertex Cover belongs to the complexity class P.

In the realm of parameterized complexity, however, this algorithm demonstrates that Vertex Cover, when parameterized by k, is *Fixed Parameter Tractable.* A parameterized problem is a problem which, given some input x, can be decided in time $O(f(\kappa(x)) \cdot p(x))$, where $\kappa(x)$ is the parameter of the problem, $|x|$ is the size of the input, p is some polynomial, and f is some computable function. Referred to as FPT, this parameterized complexity class contains problems which, from a parameterized point of view, are efficient to compute. In

this paper, we will demonstrate how a large class of parameterized graph problems can be shown to be fixed parameter tractable.

1.1 Tree Decompositions

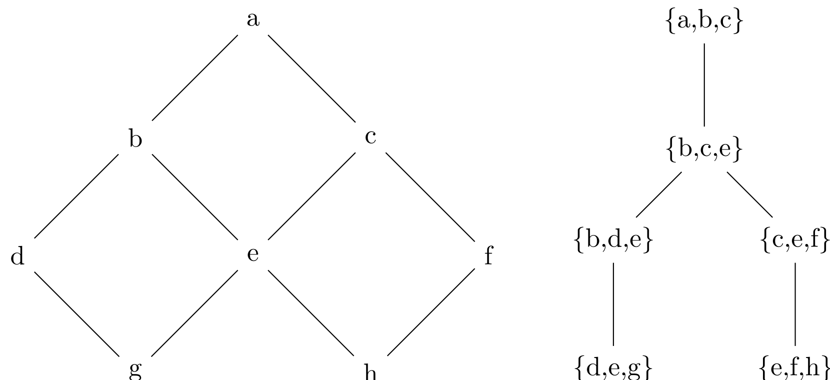
One particularly powerful way to parameterize a problem is by *treewidth*. Intuitively, the treewidth of a graph describes how closely a graph resembles a tree. The treewidth of a graph is determined by the size of the tree decompositions of a graph, a way to define a graph with a tree structure.

Definition 1.1.1 (Tree Decomposition)**.** Let G be a graph A *tree decomposition* of G is a pair $\mathcal{T} := \left\langle T, (B_t)_{t\in V(T)} \right\rangle$, where

- T is a tree
- Each B_t is a nonempty subset of $V(G)$ with the following three properties:
  - For every $v \in V(G)$, there is some $t \in V(T)$ such that $v \in B_t$
  - For every $xy \in E(G)$, there is some $t \in V(T)$ such that $x, y \in V(T)$
  - Let $s, t, u \in V(T)$ such that t is on the unique path between s and u. Then $B_t \supseteq B_s \cap B_u$.

The subsets B_t are referred to as the *bags* of the decomposition.

Example 1.1.2. We give an example of a graph and one of its possible tree decompositions. It is easy to check for yourself that the decomposition follows the three properties.



Notice how the structure of the tree visually resembles the corresponding graph. In this way, tree decompositions capture the tree structure of a given graph, and the more a graph "resembles a tree", the smaller its treewidth will be. It is worth noting that tree decompositions are not unique. For example, any graph G has the trivial tree decomposition consisting of a single node whose corresponding bag is $V(G)$. This would not be a useful decomposition, however, as it would lack any useful structure. From a decomposition for a graph, we can determine its treewidth.

Definition 1.1.3 (Treewidth)**.** Let $\mathcal{T} := \left\langle T, (B_t)_{t\in V(T)} \right\rangle$ be a tree decomposition. Then the *width* of T is $\max_{t\in V(T)} |B_t| - 1$. Then for any graph G, the *treewidth* of G is the minimum width of any decomposition of G, and we denote by $\mathbf{twd}(G)$ the treewidth of G.

The tree decomposition shown above demonstrates that the treewidth of the graph is at most two (and in fact, it is exactly two since a graph has treewidth one exactly when it is a forest). An important fact in the realm of tree decompositions is that, regardless of the width of the decomposition, the size of the tree decomposition can be bounded linearly by the size of G. Tree decomposition algorithms wouldn't be useful from a parameterized complexity viewpoint if, to get a decomposition of small width, the tree needed to be exponentially big.

Theorem 1.1.4. *Let G be a graph with n vertices. Let $\mathcal{T}$ be a decomposition of G of width k. Then there is a decomposition $\mathcal{T}'$ of G with width k whose tree has at most n vertices, and this decomposition can be computed efficiently from $\mathcal{T}$.*

Proof. We say that a tree decomposition is *nice* if for every $s, t \in V(T)$, $B_s \not\subseteq B_t$. It is clear that if a decomposition is nice, then $|V(T)| \leq n$, since each bag must have one vertex of G not present in any other bag. We present the following procedure to turn an arbitrary decomposition into a nice decomposition.

```
procedure MakeNice(T)
    if V(T) = ∅ then
        return
    end if
    Pick r ∈ V(T)
    for all neighbor s of r do
        T_s ← subtree of T rooted at S
        MakeNice(T_s)
        if B_s ⊆ B_r or B_r ⊆ B_s then
            identify vertices s and r, choosing the larger bag
        end if
    end for
end procedure
```

It is clear that this procedure runs in linear time, and that the tree decomposition properties are maintained throughout the procedure. It remains to show that this procedure transforms a tree decomposition into a nice decomposition. Suppose that, after this procedure, there are vertices u, v such that $B_u \subseteq B_v$. If u and v are not neighbors, let w be a neighbor of v on the path from u to v. By tree decomposition properties, we have that $B_w \supseteq B_u \cap B_v = B_v$. Therefore, without loss of generality, we assume that u and v are neighbors.

At some point in the procedure, some neighbor w was identified with v which caused v's bag to be a superset of u's. Then there must have been some vertex x of G such that $x \in B_u$ and $x \in B_w$ but $x \notin B_v$. This contradicts the tree decomposition properties. □

Chapter 2

Algebra

2.1 F-Magmas

In algebra, we often consider some set, and some operations on that set which obey certain laws. We use this to describe the properties of the set and the properties of the operation. For example, a Group consists of some set G and an operation $(\cdot) : (G \times G) \to G$ which is associative, has an identity, and has an inverse. A Ring consists of some set R and two operations $(+)$ and $(\cdot)$, which obey the associated Ring laws. While in some sense Groups and Rings are very different objects, both with rich theories, we can see that they have a similar structure. To capture this similarity in structure across algebras, we introduce F-Magmas.

Definition 2.1.1 (F-Magma)**.** Let F be a set of *signatures* of functions, describing a function f which accepts $n_f \geq 0$ arguments. An F-Magma is a pair $\mathbb{M} := \langle \mathbb{M}, (f_{\mathbb{M}})_{f \in F} \rangle$, where $f_{\mathbb{M}}$ is some function $\mathbb{M}^{n_f} \to \mathbb{M}$.

Magmas will prove useful for talking about larger algebras, and demonstrating properties shared by different algebras. We demonstrate how we can use a Magmas can be used to describe groups:

Example 2.1.2. (Groups) We define $F_{Group} := \{(\cdot), inv, e\}$, where

- $n_{(\cdot)} := 2$
- $n_{inv} := 1$
- $n_e := 0$

The $(\cdot)$ signature describes the group operation, the inv signature describes the inverse of each object in the group, and e is the identity of the $(\cdot)$ operation. The additive group of integers is $\langle \mathbb{Z}, ((\cdot)_{\mathbb{Z}}, inv_{\mathbb{Z}}, e_{\mathbb{Z}}) \rangle$, where

- $(a \cdot_{\mathbb{Z}} b) := a + b$
- $inv_{\mathbb{Z}}(a) := -a$
- $e_{\mathbb{Z}} := 0$

Another common algebraic object is a vector space, with some set of scalars F over a set of vectors V and two operations: vector addition: $(V \times V) \to V$ and scalar multiplication: $(F \times V) \to V$. For brevity we omit identities, zeros, and inverses in this example. If we wanted to describe this with an F-Magma as we did with groups, we'd run into a bit of an issue. We'd have a set of signatures F_{Vector} containing two operations $(+)$ and $(\cdot)$ with $n_{(+)} = n_{(\cdot)} = 2$. Let us try to specify this Magma for the vector space of real numbers $\mathbb{R}$ over $\mathbb{R}^{17}$. As per our definition, we have a pair $\langle \mathbb{R}^{17}, ((+)_{17}, (\cdot)_{17} \rangle)$, where

$$(+)_{17} : (\mathbb{R}^{17} \times \mathbb{R}^{17}) \to \mathbb{R}^{17}$$

as desired. However, we'd also be forced to have

$$(\cdot)_{17} : (\mathbb{R}^{17} \times \mathbb{R}^{17}) \to \mathbb{R}^{17}$$

Our definition of F-Magmas can't account for an algebra which operates over more than one set. One possible solution would be to, instead of having a single scalar multiple operator $(\cdot)$, include a scalar multiple operator $(r\cdot)$ for each $r \in \mathbb{R}$. This would work, but could become unwieldy, especially in more complicated Magmas. Instead, we introduce a slight generalization of an F-Magma which more elegantly solves this problem.

Definition 2.1.3 (Many Sorted Magma)**.** Let $\mathcal{S}$ be a set of *sorts*. An $\mathcal{S}$-signature is a set F, where each $f \in F$ is equipped with

- $\alpha(f) \in \mathcal{S}^*$, describing the inputs of f, and
- $\sigma(f) \in \mathcal{S}$, describing the output of f

An $\mathcal{S}$-sorted F-Magma is a pair $\mathbb{M} := \langle (\mathbb{M}_s)_{s\in\mathcal{S}}, (f_\mathbb{M})_{f\in F}\rangle$, where

- each $\mathbb{M}_s \neq \emptyset$ is the *domain* of sort s
- for $f \in F$ with $\alpha(f) = (s_1, \ldots, s_{n_f})$ and $\sigma(f) = s$, $f_\mathbb{M}$ is a mapping $(\mathbb{M}_{s_1} \times \ldots \times \mathbb{M}_{s_{n_f}}) \to M_s$.

We say that $\mathbb{M}$ is *locally finite* if $\mathbb{M}_s$ is finite for each $s \in \mathcal{S}$.

We demonstrate how a sorted Magma can be used to describe vector spaces. For brevity, we again only consider scalar multiplication and vector addition.

Example 2.1.4. We let $\mathcal{S}_{Vector} := \{s, v\}$, $F_{Vector} := \{(+), (\cdot)\}$, with

- $\alpha(+) := (v, v)$, $\sigma(+) := v$
- $\alpha(\cdot) := (s, v)$, $\sigma(\cdot) := v$

Now we can adequately describe our vector field with a Magma. We can define our Magma as the pair $R := \langle (R_s, R_v), ((+)_R, (\cdot)_R)\rangle$, where

- $R_s := \mathbb{R}$
- $R_v := \mathbb{R}^{17}$
- $\mathbf{v} +_R \mathbf{u} := \mathbf{v} + \mathbf{u}$
- $c \cdot_R \mathbf{v} := c\mathbf{v}$

When working with some F-Magma $\mathbb{M}$, we can consider each element as an expression written over the elements in $\mathbb{M}$ and F. Considering our description of the additive group of integers, we can write $4 = ((1 \cdot_\mathbb{Z} e_Z) \cdot_Z 2) \cdot_Z 1$. We can conceptualize this expression as an abstract syntax tree, and use it to perform computations on the elements of a Magma. Certain well behaved operations on the Magma will act as bottom-up computations on these expression trees, allowing for easy description of these operations and ensuring that they can be computed efficiently. Here we give one such operation, boolean predicates on some algebra, whose structure will prove useful in proving Courcelle's theorem. We fix a set of sorts $\mathcal{S}$.

For some set of signatures F, we can consider the set of expressions written over F as an F-Magma. We let $\mathbb{M}(F)$ be such a Magma. For any other F-Magma $\mathbb{A}$, there is a clear homomorphism from $\mathbb{M}(F)$ to $\mathbb{A}$ (Magma homomorphisms are defined in the next section).

Definition 2.1.5 (Family of Predicates)**.** Let $\mathbb{M}$ be an F-Magma. A *family of predicates* on $\mathbb{M}$ is a set P, where

- Each $p \in P$ has a sort $\sigma(p)$.
- Each $p \in P$ has an associated predicate $\hat{p} : \mathbb{M}_{\sigma(p)} \to \{\mathbf{true}, \mathbf{false}\}$.

For each $p \in P$, we define the subset of $\mathbb{M}_{\sigma(p)}$ where $\hat{p}$ is true $L_p := \{x \in \mathbb{M}_{\sigma(p)} : \hat{p}(x) = \mathbf{true}\}$. We say that P is locally finite if $\{p \in P : \sigma(p) = s\}$ is finite for each $s \in \mathcal{S}$.

Definition 2.1.6 (Inductive Family of Predicates)**.** Let $\mathbf{M}$ be an F-Magma, and P be a family of predicates on $\mathbf{M}$. We say that P is *F-inductive* if for each $f \in F$ with $\alpha(f) = (s_1, \ldots, s_n)$ and $\sigma(f) = s$ and $p \in P$ with $\sigma(p) = s$, there is

- A sequence of natural numbers $m_1, \ldots, m_n$
- A sequence $(p_{1,1}, \ldots, p_{1,m_1}, p_{2,1}, \ldots, p_{2,m_2}, \ldots, p_{n,1}, \ldots, p_{n,m_n})$ in P.
- A boolean expression B which accepts $m_1 + \ldots + m_n$ arguments

such that

- $\sigma(p_{i,j}) = s_i$ for each $1 \leq j \leq m_i$
- For every $x_1 \in \mathbb{M}_{s_1}, \ldots, x_n \in \mathbb{M}_{s_n}$,

$$\hat{p}(f_{\mathbb{M}}(x_1, \ldots, x_n)) = B(\hat{p}_{1,1}(x_1), \ldots, \hat{p}_{n,m_n}(x_n))$$

Example 2.1.7. We consider the additive groups of integers mod 5. We let $P = \{\text{equals}_n : n \in \mathbb{Z}[5]\}$, and define

$$\widehat{\text{equals}_n}(x) = \textbf{true} \iff x \equiv n \pmod 5$$

Here it is not necessary to define a $\sigma(\text{equals}_n)$ for each n, since since we are not working with a many sorted Magma. We demonstrate that this family of predicates is F_{Group}-inductive, showing that equals_3 can be calculated inductively.

$$\widehat{\text{equals}_3}(e) = \widehat{\text{equals}_3}(0) = \textbf{false}$$

$$\widehat{\text{equals}_3}(-x) = \widehat{\text{equals}_2}(x)$$

$$\widehat{\text{equals}_3}(x+y) = \bigvee_{i=0}^{4} (\widehat{\text{equals}_i}(x) \wedge \widehat{\text{equals}_{3-i}}(y))$$

Similar constructions can be made for each equals_n, and hence P is a F_{Group}-inductive family of predicates.

2.2 Homomorphism and Automata

Much like we have Group homomorphisms and Ring homomorphisms, we can discuss homomorphisms between Magmas. We fix a set of sorts $\mathcal{S}$.

Definition 2.2.1 (Homomorphism)**.** Let $\mathbb{M}$ and $\mathbb{N}$ be F-Magmas. A homomorphism from $\mathbb{M}$ to $\mathbb{N}$ is a family of functions $(h_s)_{s \in \mathcal{S}}$ such that

- Each h_s is a function $\mathbb{M}_s \to \mathbb{N}_s$
- For each $f \in F$ with $\alpha(f) = (s_1, \ldots, s_n)$ and $\sigma(f) = s$, $x_i \in \mathbb{M}_{s_i}$, we have

$$h_s(f_{\mathbb{M}}(x_1, \ldots, x_n)) = f_{\mathbb{N}}(h_{s_1}(x_1), \ldots, h_{s_n}(x_n))$$

And we write $h : \mathbb{M} \to \mathbb{N}$. We omit the subscripts from the family of functions when it is unambiguous.

It is easy to see that Group homomorphisms and Vector Space homomorphisms fit within this definition. Homomorphisms also exemplify the idea of performing computations over expressions written in the algebra.

Definition 2.2.2 (Automaton)**.** Let $\mathbb{M}$ be an F-Magma, $s \in \mathcal{S}$, and $B \subseteq \mathbb{M}_s$. Then B is $\mathbb{M}$-*recognizable* if there is a locally finite F-Magma $\mathbb{A}$, a homomorphism $h : \mathbb{M} \to \mathbb{A}$, and a set $C \subseteq \mathbb{A}_s$ such that $h^{-1}(C) = B$. The triple $(h, \mathbb{A}, C)$ is called an *automaton*. Note that since $\mathbb{A}$ is locally finite, necessarily C is finite.

To apply some intuition from deterministic finite automata, we can consider $\mathbb{M}_s$ to be the inputs to the automata, h to be the transition function, $\mathbb{A}_s$ to be the states, C to be the final (accepting) states, and B to be the accepted inputs. From a computational viewpoint, the elements of B can be recognized in linear time, since h can be computed by traversing the tree structure of an expression and C is finite. With the next theorem, we demonstrate how you can show that some set is recognizable. We will see in the proof of Courcelle's theorem that this is a very important characterization of recognizable sets.

Theorem 2.2.3. *Let $\mathbb{M}$ be an F-Magma, $s \in \mathcal{S}$, and $B \subseteq \mathbb{M}_s$. Then B is recognizable if and only if there is an F-inductive, locally finite family of predicates P such that $B = L_p$ for some $p \in P$.*

Proof. ($\Leftarrow$) (Sketch) Let P be an F-inductive, locally finite family of predicates with $B = L_p$ for some $p \in P$. We construct an F-Magma $\mathbb{P}$ whose elements are functions $P \to \{\mathbf{true}, \mathbf{false}\}$. We define $h : \mathbb{M}_s \to \mathbb{P}_s$ such that $h(m) : P_s \to \{\mathbf{true}, \mathbf{false}\}$ is the map $h(m)(q) := \hat{q}(m)$. From this we determine $f_{\mathbb{P}}$ for each $f \in F$ such that h is a homomorphism (this is tedious but not exceedingly non-trivial). We define $Q := \{x \in \mathbb{P} : x(p) = \mathbf{true}\}$. It is clear then that $(h, \mathbb{P}, Q)$ is an automaton accepting B.

($\Rightarrow$) Let $(h, \mathbb{A}, C)$ be an automaton recognizing B. We define $P := \{c\} \cup \bigcup_{t \in \mathcal{S}} A_t$ for some c of sort s with $c \notin \mathbb{A}_s$. For each $a \in \mathbb{A}_t$, we define

$$\hat{a}(x) = \mathbf{true} \iff h(x) = a$$

For $f \in F$ with $\alpha(f) = (s_1, \ldots, s_n)$ and $\sigma(f) = s$, we define

$$\hat{c}(x) = \mathbf{true} \iff h(x) \in C$$

Since $\mathbb{A}$ is locally finite, P must also be locally finite. Moreover, it is clear that $B = L_c$. It remains to show that P is F-inductive. Let $f \in F$ with $\alpha(f) = (s_1, \ldots, s_n)$ and $\sigma(f) = t$. Let $a \in \mathbb{A}_t$.

$$\hat{a}(f_{\mathbb{M}}(x_1, \ldots, x_n)) = \bigvee_{(a_1, \ldots, a_n) \in f_{\mathbb{A}}^{-1}(a)} (\hat{a}_1(x_1) \wedge \ldots \wedge \hat{a}_n(x_n))$$

Since $\mathbb{A}$ is locally finite, each $f_{\mathbb{A}}^{-1}$ can be pre-calculated in finite time. Thus each $\hat{a}$ can be defined inductively. Let $f \in F$ with $\alpha(f) = (s_1, \ldots, s_n)$ and $\sigma(f) = s$.

$$\hat{c}(f_{\mathbb{M}}(x_1, \ldots, x_n)) = \bigvee_{a \in C} \hat{a}(f_{\mathbb{M}}(x_1, \ldots, x_n))$$

This latter equations conforms to the definition of F-inductive if we apply our equations for the $\hat{a}$s □

2.3 Graphs as Algebraic Objects

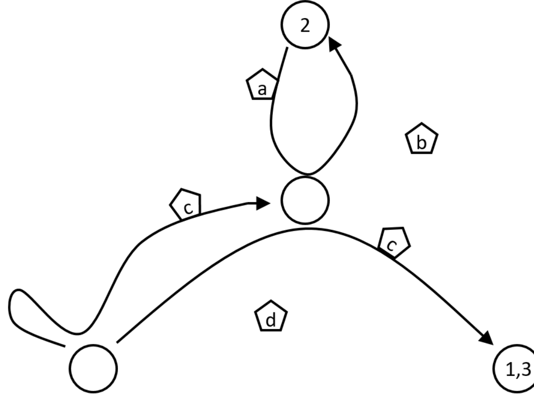
We consider finite, edge-labeled, directed hypergraphs with a finite sequence of sources. We will simply call these hypergraphs for sake of brevity. The labels are chosen from some finite alphabet A, where each $a \in A$ has an associated *type* $\tau(a) \in \mathbb{N}_0$. Note that even though the results included in this paper pertain to directed hypergraphs, they are also applicable to "regular" graphs.

Definition 2.3.1 (Concrete Hypergraph)**.** Let A be a finite alphabet (with associated types) and $n \in \mathbb{N}_0$. A *hypergraph of type n over A* is a quintuplet $G := \langle V_G, E_G, \mathbf{lab}_G, \mathbf{vert}_G, \mathbf{src}_G \rangle$ such that

- V_G is the set of vertices
- E_G is the set of hyperedges
- $\mathbf{lab}_G : E_G \to A$ is assigns a label to each edge of G
- $\mathbf{vert}\, E_G \to V_G^*$ assigns a sequence of vertices to each hyperedge. We impose that for any $e \in E_G$, the length of $\mathbf{vert}_G(e)$ is the same as the type of the label of e, $\tau(\mathbf{lab}_G(e))$.
- $\mathbf{src}_G \in V_G^n$ is the sequence of sources of G. We allow for repeated sources.

For an edge $e \in E_G$, we often write $\tau(e) := \tau(\mathbf{lab}_G(e))$. To get the ith source of G, we write $\mathbf{src}_G(i)$ (If $n = 0$, then the source sequence of G is empty). We refer to these graphs as *concrete* graphs to distinguish them from expressions written over the algebra we will define in this section.

Example 2.3.2. Let $A := \{a, b, c, d\}$ be an alphabet, with $\tau(a) = 2$, $\tau(b) = 0$, $\tau(c) = 3$, and $\tau(d) = 0$. The following depicts a hypergraph of type 3 over A:



The notion of a tree decomposition (and thus treewidth) can easily be extended to allow for hypergraphs. One instead uses a rooted tree, adds the condition that the bag associated with the root of the tree must contain every source of the hypergraph, and modifies the property describing each edges inclusion in the decomposition to account for hyperedges.

We consider three different operations on graphs, which we will use to form our algebra. We fix an alphabet A, and let G and H be hypergraphs of type n and m, respectively. Without loss of generality we assume that G and H's edge and vertex sets are disjoint. Then we define $G \oplus H$ to be the disjoint union of G and H, with source sequence $(\mathbf{src}_G(1), \ldots, \mathbf{src}_G(n), \mathbf{src}_H(1), \ldots, \mathbf{src}_H(m))$. Given the definition of the source sequence, we note that this operation is not commutative. The output of this operation has type $n + m$.

Define $[n] := \{1, \ldots, n\}$, in particular letting $[0] := \emptyset$. Let G be a hypergraph of type n, and let R be an equivalence relation on $[n]$. Then we let $\theta_R(G)$ be the graph obtained by identifying $\mathbf{src}_G(i)$ and $\mathbf{src}_G(j)$ whenever iRj. Often, we wish to only combine the ith and j^{th} sources of G, in which case we write $\theta_{i,j}(G)$. The output of this operation has type n.

Let G be a hypergraph of type n, let $p \in \mathbb{N}_0$, and let $c : [p] \to [n]$. Then we let $\sigma_c(G)$ be the graph obtained by setting the source sequence to $(\mathbf{src}_G(c(1)), \ldots, \mathbf{src}_G(c(p)))$. Often, if we want to choose some sources $i_1, \ldots, i_p$, we write $\sigma_{i_1,\ldots,i_p}(G)$ as shorthand for $\sigma_c(G)$ where $c(k) = i_k$. The output of this operation has type p.

Note that for θ and σ we write $\sigma_c(G)$ as opposed to $\sigma(c, G)$. We consider each σ_c (likewise each θ_R) to be a separate function, rather than considering c to be a parameter of the function. Additionally, we introduce a constant $\mathbf{0}$ representing the empty graph, a constant $\mathbf{1}$ representing a graph of type 1 with a single vertex, and a constant $\mathbf{a}$ for each $a \in A$ representing a graph of type $\tau(a)$ with a single hyperedge with label a. We will use these operations and constants to construct a graph algebra.

Let $: = \mathbb{N}_0$ be our set of sorts. Then we define the set of signatures

$$\begin{aligned} H_A : = \{&\mathbf{0}, \mathbf{1}\} \\ &\cup \{\mathbf{a} : a \in A\} \\ &\cup \{\oplus_{n,m} : n, m \in \mathbb{N}_0\} \\ &\cup \{\theta_{R,n} : n \in \mathbb{N}_0, R \in \text{Equiv}([n])\} \\ &\cup \{\sigma_{c,n,p} : n, p \in \mathbb{N}_0, c : [p] \to [n]\} \end{aligned}$$

where Equiv(S) is the set of all equivalence relations for a set S, and each of these symbols has the associated signature

- $\mathbf{0} : 0$
- $\mathbf{1} : 1$
- $\mathbf{a} : \tau(a)$
- $\oplus_{n,m} : (n \times m) \to (n + m)$
- $\theta_{R,n} : n \to n$
- $\sigma_{c,n,p} : n \to p$

We call the elements of $\mathbb{M}(H_A)$ graph expressions. For some graph expression t, we let $\mathbf{val}(t)$ be the unique concrete graph described by t. We note that $\mathbf{val}$ is a homomorphism, and that according to the following proposition,

Proposition 2.3.3. *Let G be a hypergraph of type n. Then there is some graph expression $t \in \mathbb{M}(H_a)_n$ such that $\mathbf{val}(t) = G$.*

it is a surjective homomorphism. A proof of this can be found in Bauderon and Courcelle [1]. Note that it is not injective, since any graph can be described by a number of graph expressions.

2.4 Tree Decompositions and Graph Expressions

Most importantly, describing a graph with a graph expression provides a second way to relate a graph to a tree-like structure. We show how the tree decompositions of a graph and the graph expressions of a graph are related. First, much like a tree decomposition, a graph expression has an associated width.

Definition 2.4.1 (Width)**.** The *width* of a graph expression is the largest type of graph represented within the expression. We define it inductively.

$$\begin{aligned}
\mathbf{wd}(\mathbf{0}) &= 0 \\
\mathbf{wd}(\mathbf{1}) &= 1 \\
\mathbf{wd}(\mathbf{A}) &= \tau(a), a \in A \\
\mathbf{wd}(\oplus_{n,m}(t_1, t_2)) &= \max\{n + m, \mathbf{wd}(t_1), \mathbf{wd}(t_2)\} \\
\mathbf{wd}(\theta_{R,n}(t)) &= \max\{n, \mathbf{wd}(t)\} \\
\mathbf{wd}(\sigma_{c,n,p}(t)) &= \max\{n, p, \mathbf{wd}(t)\}
\end{aligned}$$

Most importantly, the size of the tree decompositions of a graph and the size of the graph expressions of a graph are linearly related.

Theorem 2.4.2. *1. Let G be a graph. Let $\mathcal{T}$ be a tree decomposition of G. Then there is a graph expression t describing G such that*

$$|\mathcal{T}| = O(|t|), \mathbf{twd}(\mathcal{T}) = O(\mathbf{wd}(t))$$

2. Let t be a graph expression describing G. Then there is a tree decomposition $\mathcal{T}$ of G such that

$$|t| = O(|T|), \mathbf{wd}(t) = O(\mathbf{twd}(T))$$

Proof. The constructions required to prove each of these statements can be found in Courcelle [3]. □

Chapter 3

Logic

3.1 Graphs as Logical Objects

Describing a graph with a logical structure enables us to formulate graph properties with logical formulas, which is necessary for proving Courcelle's theorem. Fixing an alphabet A and $n \in \mathbb{N}_0$, we introduce the following symbols, which will be used to construct logical formulas:

- $\mathbf{edg}_a$ for each $a \in A$
- $\mathbf{s}_i$ for $1 \leq i \leq n$

Let G be a hypergraph of type n over A. Using these symbols, we associate with G the following logical structure:

$$|G| := \langle V_G, E_G, (\mathbf{edg}_{aG})_{a \in A}, (\mathbf{s}_i)_{i=1}^n \rangle$$

We take V_G and E_G to be the domain of discourse for formulas written with this structure. Moreover, we define the propositions $\mathbf{edg}_a$ and constants $\mathbf{s}_i$ thusly:

$$\mathbf{edg}_{aG}(e, x_1, \ldots, x_{\tau(a)}) \iff e \in E_G,\ \mathbf{lab}_G(e) = a, \text{ and } \mathbf{vert}_G(e) = (x_1, \ldots, x_{\tau(a)})$$

$$\mathbf{s}_{iG} := \mathbf{src}_G(i)$$

In the next section we will go over how, using various logical languages, we can use this structure to define various graph properties. In the last section of this book, we will primarily be describing graph properties of simple graphs without edge labels. When describing properties of such graphs, we introduce two more logical structures:

$$|G|_1 := \langle V_G, \mathbf{adj}_G \rangle$$
$$|G|_2 := \langle V_G, E_G, \mathbf{edg}_G \rangle$$

In the first structure, we restrict the domain of discourse to the vertices of the graph, with the proposition $\mathbf{adj}_G(x, y)$ defined by

$$\mathbf{adj}_G(x, y) \iff \text{there is an edge in } G \text{ from } x \text{ to } y$$

In the second structure, our domain of discourse is once again the vertices and edges of the graph, with the proposition $\mathbf{edg}_G(e, x, y)$ defined by

$$\mathbf{edg}_G(e, x, y) \iff e \text{ is an edge in } G \text{ from } x \text{ to } y$$

Differentiating between these two structures will prove useful later on. Moreover, in this chapter we will stick to writing formulas with these two structures for simplicities sake.

3.2 Logical Languages

3.2.1 First Order Logic

First order logic allows for quantification over variables. As an example, here is a formula in first order logic which expresses that a graph has no isolated vertex:

$$\forall x \exists y \exists e \left[\mathbf{edg}(e, x, y) \vee \mathbf{edg}(e, y, x)\right]$$

Any formula which can be expressed in first order logic can be decided in polynomial time. If a formula has q quantifications, then a naive algorithm (i.e a for-loop for each quantification) can determine whether the formula holds in the graph in time $O(n^q)$. This convenience, however, limits its expressive power. Properties in first order logic are constrained to defining local properties of vertices. More formally, a theorem found in Gaifman [6] states

Theorem 3.2.1. *Let P be a property of graphs. Then P is expressible in first order logic if and only if P can be expressed as a boolean combination of formulas of the following form:*

$$\exists v_1, \ldots, v_s \left[\bigwedge_{i=1}^{s} Q(N(v_i, r)) \wedge \bigwedge_{i \neq j} d(v_i, v_j) > 2r\right]$$

where

- $r, s \geq 0$
- *Q is a first-order expressible property of graphs*
- *$N(v, r)$ is the induced subgraph of vertices at most r steps away from v*
- *$d(v_i, v_j) > 2r$ denotes that the shortest path between v_i and v_j has more than $2r$ vertices.*

More "global" properties of graphs, such as k-colorability and connectivity, evade the bounds of this theorem. Thus, an extension of first order logic is necessary to describe more complex properties of graphs.

3.2.2 Second Order Logic

Second order logic is an extension of first order logic which allows for quantification over arbitrary predicates, not just variables. This added expressiveness allows for the ability to express properties you couldn't express in first order logic. For example, the following formula

$$\forall P \left[(\varphi \wedge \psi) \rightarrow \rho\right]$$

$$\varphi := \forall x \exists y \left[P(x, y)\right] \wedge \forall x \forall y \forall z \left[P(x, y) \wedge P(x, z) \rightarrow y = z\right]$$

$$\psi := \forall x \exists y \left[P(y, x)\right]$$

$$\rho := \forall x \forall y \forall z \left[P(y, x) \wedge P(z, x) \implies y = z\right]$$

expresses that the domain of discourse is finite. The formula φ expresses that P is a function, ψ expresses that P is surjective, and ρ expresses that P is injective. So the top formula says "if a function from a set to itself is surjective, then it is injective", which implies that the domain is finite. Notably, this cannot be expressed in first order logic (this is easy to see from Theorem 3.2.1). The added expressiveness of second order logic comes at a cost, however, in regards to decidability. A naive algorithm for the above formula would run in time $O\left(2^{n^2} n^3\right)$, and in general finding efficient algorithms for second order formulas is difficult.

3.2.3 Monadic Second Order Logic

To maintain some of the expressiveness of second order logic, while retaining the ease of decidability of first order logic, we introduce monadic second order logic. In monadic second order logic, we allow for quantification over variables and sets of variables. *Monadic* here refers to the ability to quantify over predicates of a single variable (monadic predicates), which is exactly equivalent to quantifying over sets. This added expressiveness allows us to express properties we couldn't in first order logic. For instance, k-colorability can be stated in monadic second order logic (here we show 2-colorability):

$$\exists X \forall x \forall y \, [\mathbf{adj}(x, y) \implies (x \in X \wedge y \notin X) \vee (x \notin X \wedge y \in X)]$$

In the next chapter, we will demonstrate how a monadic second order formula of graphs can be efficiently decided. In the last section of this paper, where we list properties which are expressible in monadic second order logic, it will be necessary to distinguish between formulas written in the structure $|G|_1$ and $|G|_2$. We will refer to the former as MS_1 and the latter as MS_2. It is easy to see that MS_2 is strictly more powerful than MS_1. The domain of discourse for MS_1 is a strict subset of the domain for MS_2, and moreover **adj** can be expressed in terms of **edg**:

$$\mathbf{adj}_G(x, y) \iff \exists e \, [\mathbf{edg}_G(e, x, y)]$$

Moreover, it is should also be clear that the structures $|G|_1$ and $|G|_2$ are strictly weaker than $|G|$, and therefore any results pertaining to add are applicable to the former two logical structures. In his paper, Courcelle adds to his monadic second order logic another predicate $\mathbf{Card}_{p,n}$, given by

$$\mathbf{Card}_{p,n}(X) \iff |X| \equiv p \pmod{n}$$

This further adds to the increased expressiveness, but still permits the main result in the next chapter, where we show how combining these ideas in logic and algebra allow for a beautiful complexity result about graphs.

Chapter 4

Courcelle's Theorem

4.1 Overview of Proof

4.1.1 Preliminaries

Courcelle's theorem states

Theorem 4.1.1. *Let φ be a statement about hypergraphs written in monadic second order logic. Then there is an algorithm which, for any hypergraph G with n vertices of treewidth k, decides whether φ holds in G in time $O(f(|\varphi|, k) \cdot n)$, for some computable function f.*

Using theorems and ideas we've gone over in earlier chapters, we will provide a sketch proof of this theorem. We fix a formula φ in monadic second order logic for hypergraphs of type n over a finite alphabet A which uses h quantifications. Our first course of action is to construct a tree automaton from φ which operates over expressions written with the set of signatures H_A. We will use Theorem 2.2.3 to accomplish this. Thus, we need a locally finite family of predicates P with $\varphi \in P$ which is H_A-inductive. We let $\mathscr{L}^h$ be the set of all monadic second order formulas which use *at most* h quantifications, and take this to be our family of predicates, where for each formula $\psi \in \mathscr{L}^h$

- the sort $\sigma(\psi) \in \mathbb{N}$ is the type of hypergraph over which ψ operates, and
- the associated predicate $\hat{\psi}$ is true for a hypergraph of type n G if and only if ψ is true in G.

It should not be immediately clear whether $\mathscr{L}^h$ satisfies the hypothesis of Theorem 2.2.3. We will first argue that it is locally finite. Let $\mathscr{L}^h_q$ be the set of formulas for hypergraphs of type q which use at most h quantifications — it suffices to convince ourselves that this set is (more or less) finite. Taken at face value, this set isn't necessarily finite. For example, we can generate formulas like

$$\exists x\, [x = x \wedge x = x \wedge \ldots \wedge x = x]$$

ad infinitum. However, any sentence of such form is tautologically equivalent to $\exists x\, [x = x]$. If we consider the elements of $\mathscr{L}^h_q$ up to tautological equivalence, however, then the set is in fact finite, and this can be proved by induction on h.

Lemma 4.1.2. *The set $\mathscr{L}^h_q$ is finite, up to tautological equivalence*

Proof. (Sketch) As a base case, we consider formulas with no quantification. If we only have access to a finite number of variables with which to write our formulas, then it is not difficult to see that we can only write a finite number of tautologically distinct formulas.

Now consider formulas which use $h + 1$ quantifications. Specifically, we considers formulas of the form $\forall x[\psi]$ (similarly $\exists x[\psi]$). Since ψ can only use h quantifications, by the inductive hypothesis there are only finitely many tautologically distinct choices for ψ. Then, quantifying over ψ, we can only construct finitely many distinct formulas. □

It remains to show that $\mathscr{L}^h$ is H_A-inductive. This is shown in Lemmas 4.5 - 4.7 in Courcelle [2], and is done by case analysis on expressions written over H_A and formulas in monadic second order logic. Here, we show how a monadic second order formula can be computed inductively over the θ operator.

First we must go over a bit of terminology. For a logical formula, a free variable is a variable which is not bound by some quantifier. For example, the variable x is free in $\forall y\,[y = x]$, while y is not free. We can consider the set of free variables for a formula, for example the set of free variables for $\exists x\,[x = y \vee x \in X]$ is $\{y, X\}$. If we were to evaluate this formula over some hypergraph G, we would need to assign these variables accordingly to elements of G. For some set of free variables $\mathcal{W}$, an assignment ν of $\mathcal{W}$ in G assigns each variable of $\mathcal{W}$ to a vertex or edge in G, and assigns each set variable of $\mathcal{W}$ to a set of vertices or edges in G. For a monadic second order formula ψ with free variable set $\mathcal{W}$ and some assignment ν of $\mathcal{W}$ in a hypergraph G, we write $(G, \nu) \models \varphi$ to mean "φ is true in G with respect to ν".

Let H be a hypergraph of type n, $\mathcal{W}$ a set of free variables, ν an assignment of $\mathcal{W}$ in H, and $i, j \in [n]$. Then we define $\theta_{i,j}(\nu)$, an assignment of $\mathcal{W}$ in $\theta_{i,j}(H)$, as follows:

$$\theta_{i,j}(\nu)(e) := \nu(e) \qquad \text{where } e \text{ is an edge variable}$$

$$\theta_{i,j}(\nu)(U) = \nu(U) \qquad \text{where } U \text{ is an edge set variable}$$

$$\theta_{i,j}(\nu)(x) = \begin{cases} \mathbf{src}_i(H) & \nu(x) = \mathbf{src}_j(H) \\ \nu(x) & \text{otherwise} \end{cases} \qquad \text{where } x \text{ is a vertex variable}$$

$$\theta_{i,j}(\nu)(X) = \begin{cases} (\nu(X) \setminus \{\mathbf{src}_j(H)\}) \cup \{\mathbf{src}_i(H)\} & \mathbf{src}_i(H) \in \nu(X) \\ \nu(X) & \text{otherwise} \end{cases} \qquad \text{where } X \text{ is a vertex set variable}$$

Lemma 4.1.3. *Let φ be a formula in monadic second order logic for hypergraphs of type n with free variable set $\mathcal{W}$. Then for any $i, j \in [n]$, there is a monadic second order formula φ' such that for any hypergraph G and assignment ν of $\mathcal{W}$ in G,*

$$(G, \nu) \models \varphi' \iff (\theta_{i,j}(G), \theta_{i,j}(\nu)) \models \varphi$$

Proof. We perform this by structural induction on φ. We first examine the cases where φ has no quantifications:

(1) φ is the formula $x = y$. When writing φ', we need to account for when x or y are the ith or j^{th} source of G. With this in mind, we define φ':

$$x = y \vee (x = \mathbf{s}_i \wedge y = \mathbf{s}_j) \vee (x = \mathbf{s}_j \wedge y = \mathbf{s}_i)$$

In the case where x and y are edge variables, no change is needed.

(2) φ is the formula $x \in X$. Similar to above, we write

$$x \in X \vee (x = \mathbf{s}_i \wedge \mathbf{s}_j \in X) \vee (x = \mathbf{s}_j \wedge \mathbf{s}_i \in X)$$

And in the case where x is an edge variable and X is an edge set variable, no change is needed.

(3) φ is the formula $\mathbf{edg}_a(e, x, y)$. Here we assume that $\tau(a) = 2$, but the construction is analogous for other edge types. Noting that θ does not change the type of any edges, we simply account for when e passes through the sources of concern. We define φ' to be the disjunction of the following formulas:

$$\mathbf{edg}_a(e, x, y)$$

$$x = \mathbf{s}_i \wedge \mathbf{edg}_a(e, \mathbf{s}_j, y)$$

$$x = \mathbf{s}_j \wedge \mathbf{edg}_a(e, \mathbf{s}_i, y)$$

$$y = \mathbf{s}_i \wedge \mathbf{edg}_a(e, x, \mathbf{s}_j)$$

$$y = \mathbf{s}_j \wedge \mathbf{edg}_a(e, x, \mathbf{s}_i)$$

$$x = \mathbf{s}_i \wedge y = \mathbf{s}_j \wedge \mathbf{edg}_a(e, \mathbf{s}_j, \mathbf{s}_i)$$

$$x = \mathbf{s}_j \wedge y = \mathbf{s}_i \wedge \mathbf{edg}_a(e, \mathbf{s}_i, \mathbf{s}_j)$$

(4) φ is the formula $\mathbf{Card}_{n,p}(U)$. If U is an edge set, then the formula is unchanged (since θ neither adds nor removes edges). If U is a vertex set, then we need to account for when the ith and j^{th} sources are both in U *and* these two sources are distinct vertices, and define φ':

$$(\mathbf{Card}_{n,p}(U) \wedge \psi) \vee (\mathbf{Card}_{n+1,p}(U) \wedge \neg\psi)$$

where ψ is the formula

$$\mathbf{s}_i \in U \wedge \mathbf{s}_j \in U \wedge \mathbf{s}_i \neq \mathbf{s}_j$$

(5-8) φ is one of $\psi_1 \wedge \psi_2$, $\psi_1 \vee \psi_2$, or $\neg\psi$. Inductively, we construct formulas ψ_1' and ψ_2' or ψ' (whichever applies). Then we let φ' be either $\psi_1' \wedge \psi_2'$, $\psi_1' \vee \psi_2'$, or $\neg\psi'$ (respectively). It is easy to see that the constructed φ' is correct.

(7-10) φ is one of $\exists x[\psi], \forall x[\psi], \exists X[\psi]$, or $\forall X[\psi]$. By the inductive hypothesis, we construct formula ψ'. If x is an edge variable or X is an edge set variable, then it is clear that replacing ψ with ψ' suffices. We argue that this same construction works for vertex variables and vertex set variables (we only show for existential quantification over set variables, and the argument for the other cases is analogous). We define ν_V to be a $\mathcal{W} \cup \{X\}$ assignment in G, extended to assign X to some vertex set V.

$$\exists X[\psi] \text{ is true for } G \text{ with respect to } \nu$$
$$\Longleftrightarrow$$
$$\text{There is some } V \subseteq V(G) \text{ such that } \psi \text{ is true for } G \text{ with respect to } \nu_V$$
$$\Longleftrightarrow$$
$$\text{There is some } V \subseteq V(\theta_{i,j}(G)) \text{ such that } \psi' \text{ is true for } \theta_{i,j}(G) \text{ with respect to } \theta_{i,j}(v)_V$$
$$\Longleftrightarrow$$
$$\exists X[\psi'] \text{ is true for } \theta_{i,j}(G) \text{ with respect to } \theta_{i,j}(\nu)$$

Therefore, we define φ' to be $\exists X[\varphi']$ (an analogously for other quantifications). □

As mentioned above, the proofs for the other operators in H_A, as well as the constants in H_A, can be found in Courcelle [2]. Note that, if you wish to read Courcelle's proof, that he proves it for an equivalent variant of monadic second order logic which only has set variables.

Thus, $\mathscr{L}^h$ satisfies the hypothesis of Theorem 2.2.3, and therefore L_φ (the set of hypergraphs which satisfy φ) is recognizable. Therefore, from φ, we can construct an automata which recognizes hypergraphs which satisfy φ. This automata is a tree automata which operates over hypergraph expressions written in H_A. This automata, however, would be infinitely large, and this is precisely because H_A has infinitely many symbols. To accommodate for this, we restrict the algebra to $H_A^{(k)}$, the set of symbols in H_A with sort less than or equal to k. This is a finite set of symbols, and thus the generated automata will also be finite. In particular, the size of the automata depends only on $|\varphi|$ and k.

4.1.2 The Algorithm

We present the following algorithm which proves Courcelle's Theorem. Let G be a hypergraph with n vertices with a tree decomposition $\mathcal{T} := \langle T, (B_t)_{t \in V(T)} \rangle$ of width k, and let φ be a formula in monadic second order logic. First, using Theorem 1.1.4 we construct (in linear time) a decomposition $\mathcal{T}'$ of width k whose tree has at most n vertices. Using Theorem 2.4.2, from $\mathcal{T}'$ we construct (in linear time) a hypergraph expression t defining G, whose size and with are linearly bounded by the size and width of $\mathcal{T}'$. Let $k' := \mathbf{wd}(t)$. From φ, we construct an automata $\mathcal{A}$ which recognizes elements of L_φ which have a hypergraph expression of width at most k'. The size of this automata, and the time it takes to construct it, depend only on $|\varphi|$ and k'. We then run this automata on the expression t in time $|\mathcal{A}| \cdot |t|$.

We now convince ourselves that the running time of this algorithm is that which is stated in Courcelle's Theorem. There are functions f, g such that $\mathcal{A}$ can be constructed in time $f(|\varphi|, k')$, and $|\mathcal{A}| = g(|\varphi|, k')$. Therefore, including construction time, the algorithm runs in

$$\begin{aligned} f(|\varphi|, k') + g(|\varphi, k') \cdot |t| &\leq f(|\varphi|, k') + g(|\varphi|, k') \cdot n \\ &\leq (f(|\varphi|, k') + g(|\varphi| k')) \cdot n \\ &\leq (f(|\varphi|, O(k)) + g(|\varphi|, O(k))) \cdot n \\ &= O((f(|\varphi|, k) + g(|\varphi|, k)) \cdot n) \end{aligned}$$

as desired.

4.2 The Conclusion, or Maybe, the Punchline

As an immediate corollary, we have that any graph property which can be expressed in monadic second order logic is fixed parameter tractable when parameterized by treewidth. One may be curious, understandably, about the running time of this algorithm. What is the behavior of the function f in the statement of the theorem? If you've been thinking that this theorem is too good to be true, then this is where your suspicions may be confirmed. The unfortunate truth of Courcelle's theorem is that the generated algorithm is much too inefficient to be used in practice. A result which can be found in Frick and Grohe [5] states that this function f cannot be bounded by an elementary function. In particular, f cannot be bounded by an iterated exponential whose height is bounded by $|\varphi|$ and k. Which is a lot of words to say that, even for small sentences such like $\forall X \exists x \, [x \in X]$, the algorithm that Courcelle's theorem generates, while linear, would likely have a constant so large as to be rendered useless. Which is not to say all hope is lost. For instance, we can use Courcelle's theorem to show that Independent Set is fixed parameter tractable when parameterized by treewidth and the desired size of the independent set:

$$\exists x_1, \ldots, x_M \left[\bigwedge_{i \neq j} (x_i \neq x_j \wedge \neg \, \mathbf{adj}(x_i, x_j)) \right]$$

The parameterization by the size of the independent set occurs because $|\varphi| = O(M^2)$, where M is the size of the desired set. However, there also exists a dynamic programming algorithm which, given a tree decomposition of width k, solves the maximum independent set problem in time $O(2^k \cdot n)$. Though Courcelle's theorem may have a poor lower bound, it still provides a strict upper bound on any fixed parameter tractable algorithm. The existence of a monadic second order formula describing a graph property is reason to believe that a better, more efficient algorithm may exist.

Part II

Expressing Graph Properties in Monadic Second Order Logic

Chapter 5

Basic Properties

We begin our exploration of which graph properties are expressible in monadic second order logic by first demonstrating properties which are easy to express. We present most of these without comment, but will expand further if there something needs further explaining. Also note that any part of a formula written in quotation marks is not an actual MS formula, but can easily easily be expressed as one. We will be working with undirected, unlabeled graphs without any source vertices. It is easy to see that Courcelle's theorem applies here, if we consider graphs of type 0 with alphabet $A = \{a\}$ with $\tau(a) = 2$. Moreover, while Courcelle's theorem is proved for directed graphs, to get around this we can imagine replacing every instance of $\mathbf{adj}(x, y)$ or $\mathbf{edg}(e, x, y)$ with $\mathbf{adj}(x, y) \vee \mathbf{adj}(y, x)$ and $\mathbf{edg}(e, x, y) \vee \mathbf{edg}(e, y, x)$, respectively. Complete definitions for most of these problems can be found in Garey and Johnson [7].

By convention, we use uppercase letters for set variables and lowercase letters for set variables. Moreover, we will generally use X for vertex sets, U for edge sets, x for vertices, and e for edges, though what each variable represents should be clear from context.

Due to the poor running time of any algorithm generated by some logical formula, we do not take care to minimize the length of expressions, and instead put more care to make the formulas easily understood.

5.1 Vertex Cover of Size K

$$\exists x_1, \ldots, x_K \forall e \left[\bigvee_{i=1}^{k} \mathbf{inc}(e, x_i) \right]$$

We define $\mathbf{inc}(e, x) := \exists y[\mathbf{edg}(e, y, x_i)]$, expressing that an edge e is incident to some vertex x. Notice that the length of the expression depends on K, we will see this with graph properties which are parameterized by some positive integer.

5.2 Dominating Set of size K

$$\exists x_1, \ldots, x_K \forall y \left[\bigvee_{i=1}^{K} (y = x_i \vee \mathbf{adj}(y, x_i)) \right]$$

5.3 Domatic Number K

$$\exists X_1, \ldots, X_K \left[\mathbf{partition}(X_1, \ldots, X_K) \wedge \bigwedge_{i=1}^{K} \text{“}X_i \text{ is a dominating set”} \right]$$

The formula $\mathbf{partition}(X_1, \ldots, X_K)$ states that the vertex sets $X_1, \ldots, X_K$ form a partition of the vertices,

and is defined as

$$\mathbf{partition}(X_1, \dots, X_K) := \forall x \left[\bigvee_{i=1}^{K} x \in X_i \right] \wedge \neg \exists x \left[\bigvee_{i \neq j}^{K} (x \in X_i \wedge x \in X_j) \right]$$

The sentence "X_i is a dominating set" can be easily constructed by modifying the formula for dominating set shown above. Its also worth noting that this is the first formula where we've used set quantifications (the previous formulas were all first order formulas). Here it is necessary, since we don't know the size of the dominating sets that form the partition of the graph.

5.4 K-colorability

$$\exists X_1, \dots, X_K \left[\mathbf{partition}(X_1, \dots, X_K) \wedge \forall x, y \left[\mathbf{adj}(x, y) \rightarrow \bigwedge_{i=1}^{K} \neg(x \in X_i \wedge y \in X_i) \right] \right]$$

5.5 Achromatic Number K

$$\exists X_1, \dots, X_K \left[\text{“}X_1, \dots, X_K \text{ is a } K\text{-coloring”} \wedge \bigwedge_{i \neq j}^{K} \text{“}X_i \cup X_j \text{ is not an independent set”} \right]$$

$$\text{“}X \cup Y \text{ is not an independent set”} := \exists x, y \, [x \in X \wedge y \in Y \wedge \mathbf{adj}(x, y)]$$

5.6 K-edge-colorability

$$\exists U_1, \dots, U_K \left[\mathbf{partition}(U_1, \dots, U_K) \wedge \forall e_1, e_2 \left[\exists x \, [\mathbf{inc}(e_1, x) \wedge \mathbf{inc}(e_2, x)] \rightarrow \bigwedge_{i=1}^{K} \neg(e_1 \in U_i \wedge e_2 \in U_i) \right] \right]$$

5.7 Minimum Maximal Matching

$$\exists U \, [\text{“}|U| \leq K\text{''} \wedge \text{“}U \text{ is a matching”} \wedge \forall U' \, [\text{“}U' \supseteq U\text{''} \rightarrow \neg\text{“}U' \text{ is a matching”}]]$$

There are several sentences that need defining here, but none are too complicated.

$$\text{“}|U| \leq K\text{''} := \forall e_1, \dots, e_{K+1} \left[\bigwedge_{i=1}^{K+1} e_i \in U \rightarrow \bigvee_{i \neq j}^{K+1} e_i = e_j \right]$$

$$\text{“}U \text{ is a matching”} := \forall e_1, e_2 \, [e_1, e_2 \in U \wedge e_1 \neq e_2 \rightarrow \neg \exists x \, [\mathbf{inc}(e_1, x) \wedge \mathbf{inc}(d_2, x)]]$$

$$\text{“}U' \supseteq U\text{''} := \forall e \, [e \in U \rightarrow e \in U']$$

Note how we denoted that the edge set U was a maximal matching — this is a useful pattern that will be used elsewhere. We also show how to lower bound the size of the set, and check equality on the size of a set:

$$\text{“}|U| \geq K\text{''} := \exists e_1, \dots, e_K \left[\bigwedge_{i \neq j}^{K} e_i \neq e_j \wedge \bigwedge_{i=1}^{K} e_i \in U \right]$$

$$\text{“}|U| = K\text{''} := \text{“}|U| \leq K\text{''} \wedge \text{“}|U| \geq K\text{''}$$

We cannot in general check that two sets have the same size — there is no monadic second order formula for "$|X| = |U|$", second order logic is necessary to describe bijections between sets.

5.8 Supergraph of H

We fix some graph H, with $V(H) = \{v_1, \ldots, v_n\}$ and $E(H) = \{e_1, \ldots, e_m\}$.

$$\exists x_1, \ldots, x_n, u_1, \ldots, u_m \left[\bigwedge_{i \neq j}^{n} x_i \neq x_j \wedge \bigwedge_{i \neq j}^{m} u_i \neq u_j \bigwedge_{i=1}^{n} \bigwedge_{j=1}^{n} \bigwedge_{k=1}^{m} \text{``}\mathbf{edg}(u_k, x_i, x_j) \iff \mathbf{edg}(e_k, v_i, v_j)''\right]$$

$$\text{``}\mathbf{edg}(u_k, x_i, x_j) \iff \mathbf{edg}(e_k, v_i, v_j)'' := \begin{cases} \mathbf{edg}(u_k, x_i, x_j) & \text{if } e_k \text{ is an edge in } H \text{ from } v_i \text{ to } v_j \\ \neg\, \mathbf{edg}(e_k, x_i, x_j) & \text{otherwise} \end{cases}$$

In this way, we find a group of distinct vertices and edges in our graph which, when taken as a subgraph, are isomorphic to H. Notice that the length of the formula depends on the size of H.

Chapter 6

Advanced Techniques

6.1 Reflexive, Transitive Closure

An important fact about monadic second order logic is that if we can express some binary relation of variables in monadic second order logic, then we can also express the reflexive, transitive closure of that relation. We will see that this is made possible by the ability to quantify over sets, and thus cannot be expressed in first order logic.

Theorem 6.1.1. *Let R be a binary relation on a set D. Then if R is expressible in monadic second order logic then so is the transitive, reflexive closure of R.*

Proof. Let R^+ denote the transitive reflexive closure of R. We say a set $X \subseteq D$ is *R-closed* if for any $x, y \in D$ such that $x \in X$ and xRy, we have $y \in X$.

Claim: Let $x \in D$ and let $X \subseteq$ be an R-closed set such that $x \in X$. Further suppose that X is minimal with respect to these properties. Then $y \in X \iff (x, y) \in \mathbb{R}^+$.
($\Rightarrow$) Suppose not. Then the set $X \setminus \{z \in D : yR^+z\}$ would also be R closed and contain x. But X was assumed to be minimal with respect to these properties, and so this is a contradiction.
($\Leftarrow$) If xR^+y, either $x = y$, xRy, or there is some chain $z_1, \ldots, z_n$ such that xRz_1, z_iRz_{i+1}, and z_nRy. In the first case, trivially $y \in X$. In the second case $y \in X$ since X is R-closed. In the third case, inductively we have $z_i \in X \implies y \in X$ since X is R-closed.

Let $\varphi(\cdot, \cdot)$ be a monadic second order logic formula which defines R. First, we write a formula which expresses that that a set X is R-closed.

$$\psi(X) := \forall x \, [x \in X \rightarrow \forall y \, [\varphi(x, y) \rightarrow y \in X]]$$

Using the above claim, we write a formula defining the transitive, reflexive closure of R:

$$\varphi^+(x, y) := \forall X \, [(\psi(X) \wedge x \in X \wedge \forall Y \, [a \in Y \wedge \psi(Y) \rightarrow \text{“}X \subseteq Y''\text{]}) \rightarrow y \in X]$$

□

As far as we are concerned, the main application of this theorem is to describe connectivity in graphs. Two vertices are connected if there is some chain of vertices between the two which are all adjacent. And any vertex is trivially connected to itself. In other words, connectedness is the transitive, reflexive closure of the adjacency relation.

Corollary 6.1.2. *The following properties are expressible in monadic second order logic:*

1. *Two vertices x and y are connected by a path*
2. *A graph is connected*

3. *An edge set U forms a path between vertices x and y*

4. *An edge set C forms a cycle*

Proof. 1. This follows immediately from our discussion above.

2. This is easily described with the formula $\forall x, y$[“x and y are connected”]

3. We observe that a set U forms a path between x and y if you can get from x to y using only edges in U, and that U is minimal with respect to this property. That x and y are connected with edges in U is the transitive, reflexive closure of the following formula:

$$\psi_J(a,b) := \exists e\,[e \in J \wedge \mathbf{edg}(e,a,b)]$$

Then the path property can be written as

$$\psi_U^+(x,y) \wedge \forall J\left[\left(\text{“}J \subseteq U'' \wedge \psi_J^+(x,y)\right) \rightarrow \text{“}J = U''\right]$$

where ψ_U^+ denotes the transitive, reflexive closure. The second half of the formula asserts that U doesn't contain any extraneous edges not in the path and that the path described doesn't contain any loops.

4. We first note that a graph is a cycle if and only if it is 2-regular and connected. It suffices then to express these properties when restricted to the edges in a set C. First, we need to ignore vertices which are not incident to any edge in C. We define $\varphi(x) := \exists e[e \in C \wedge \mathbf{inc}(e,x)]$, expressing that a vertex x is incident to an edge in C. Then, to describe that C induces a 2-regular graph, we write

$$\text{“}d_C(x) \leq 2'' := \forall e_1, e_2, e_3\left[\left(\bigwedge_{i=1}^{3} e_i \in C \wedge \bigwedge_{i=1}^{3} \mathbf{inc}(x, e_i)\right) \rightarrow \bigvee_{i \neq j}^{3} e_i = e_j\right]$$

$$\text{“}d_C(x) \geq 2'' := \exists e_1, e_2\,[e_1 \neq e_2 \wedge e_1, e_2 \in C \wedge \mathbf{inc}(e_1, x) \wedge \mathbf{inc}(e_2, x)]$$

$$\theta(C) := \forall x\,[\varphi(x) \rightarrow (\text{“}d_C(x) \leq 2'' \wedge \text{“}d_C(x) \geq 2'')]$$

That the edge set C induces a connected graph is easier to express, using the formula ψ defined above:

$$\eta(C) := \forall x, y\left[\,\neg\varphi(x) \vee \neg\varphi(y) \vee \psi_C^+(x,y)\right]$$

Then the formula $\theta(C) \wedge \eta(C)$ describes that the edge set C forms a cycle.

□

For ease of notation, we introduce new operators **cycle** and **path**, given by

$$\mathbf{cycle}(U) := \text{“The edge set } U \text{ forms a cycle”}$$

$$\mathbf{path}(U,x,y) := \text{“The edge set } U \text{ forms a path from } x \text{ to } y\text{”}$$

6.2 Transductions

One of the most powerful tools in writing logical formulas is that of transductions. Broadly, transductions allow you to describe a graph within (N disjoint copies of) a different graph. Transductions can be used to describe properties of subgraphs, supergraphs, and various graph transformations.

Definition 6.2.1 (Transduction Definition Scheme)**.** Let $N > 0$ and $\mathcal{W}$ be a set of variables. A *transduction definition scheme* is a triple $\Delta = \left\langle \varphi, (\psi_i)_{i=1}^3, (\mathbf{edg}_{i,j,k})_{i,j,k=1}^3 \right\rangle$, where

- An monadic second order formula φ with free variables in $\mathcal{W}$
- monadic second order formulas ψ_i with free variables in $\mathcal{W} \cup \{x\}$

- monadic second order formulas $\mathbf{edg}_{i,j,k}$ with free variables in $\mathcal{W} \cup \{x_1, x_2, x_3\}$

Definition 6.2.2. Let Δ be a transduction definition scheme, G be a graph, and γ be a definition of $\mathcal{W}$ in G, i.e a function $\mathcal{W} \to V(G) \cup E(G)$. We say the domain of the transduction is

$$D := \{(x,i) : x \in V(G) \cup E(G), 1 \leq i \leq N, (G, \gamma, x) \models \psi_i\}$$

Moreover, we say a graph H is defined by G, γ, and Δ if

- $(G, \gamma) \models \varphi$
- There is a bijective map $\delta : D \to (V(H) \cup E(H))$
- For any $(x_1, i), (x_2, j), (x_3, k) \in D$, we have that $(G, \gamma, x_1, x_2, x_3) \models \mathbf{edg}_{i,j,k} \iff \delta(x_1, i)$ is an edge in H going from $\delta(x_2, j)$ to $\delta(x_3, k)$.

Intuitively, φ checks that your variable assignment is valid for the purpose of the transduction, each ψ_i defines the domain of the ith copy of the graph, and $\mathbf{edg}_{i,j,k}$ defines the edge relation between the ith, j^{th}, and k^{th} copy of the graph.

Example 6.2.3. We demonstrate how we can use a transduction to describe the supergraphs of a graph obtained by adding $K \leq |V|$ edges. We define this transduction with 2 copies of the original graph thusly:

- $\mathcal{W} := \{u_1, \ldots, u_K, v_1, \ldots, v_K, t_1, \ldots, t_K\}$
- $\varphi := \bigwedge_{i=1}^{K} (u_i \neq v_i) \wedge \bigwedge_{i \neq j}^{K} (t_i \neq t_j)$
- $\psi_1(x) := \mathbf{true}$
- $\psi_2(x) := \bigvee_{i=1}^{K} (x = t_i)$
  - $\mathbf{edg}_{1,1,1}(e, x, y) := \mathbf{edg}(e, x, y)$
  - $\mathbf{edg}_{2,1,1}(e, x, y) := \bigvee_{i=1}^{K} (e = t_i \wedge x = u_i \wedge y = v_i)$
  - $\mathbf{edg}_{i,j,k}(e, x, y) := \mathbf{false}$

In this way, the first copy of the graph behaves normally with regard to logical formulas. With the second copy, we choose K distinct vertices and have them act as the new edges we are adding.

More specifically, what we have defined here is an MS_2 transduction. We can similarly define an MS_1 transduction, where instead of defining formulas $\mathbf{edg}_{i,j,k}$ for $1 \leq i, j, k \leq N$, we define formulas $\mathbf{adj}_{i,j}$ for $1 \leq i, j \leq N$. These will be useful when its difficult to bound the size of the edge set of the graph we're transducing to.

In order for a transduction scheme to be useful, we need to be able to write formulas for the graph we are transducing to. The fundamental property of transductions states that this is possible.

Theorem 6.2.4. *Let Δ be a transduction definition scheme with free variable set $\mathcal{W}$. Then for any monadic second order formula φ, you can construct a monadic second order formula φ' with free variables in $\mathcal{W}$ such that, for any graph H which is defined by G, γ, Δ (where G is a graph and γ is an assignment of $\mathcal{W}$ in G), we have*

$$H \models \varphi \iff (G, \gamma) \models \varphi'$$

Proof. The proof is done by induction on the structure of φ, and can be found in Courcelle [4]. $\square$

Chapter 7

Advanced Properties

7.1 Subgraph with property P

Suppose that P is some property which can be expressed in monadic second order logic. We can describe arbitrary subgraphs with the following transduction definition scheme:

- $\mathcal{W} := \{V, E\}$
- $\varphi := \forall e\,[e \in E \to \exists x, y\,[x, y \in V \wedge \mathbf{edg}(e, x, y)]]$
- $\psi_1(x) := x \in V \vee x \in E$
- $\mathbf{edg}_{1,1,1}(e, x, y) := \mathbf{edg}(e, x, y)$

With this, we can quantify over all subgraphs of a graph and determine whether any have property P

$$\exists X, U\,[\text{“(X, U) is a subgraph with property } P\text{”}]$$

Examples of properties which can be described in monadic second order logic and whose corresponding subgraph problem is NP-complete include:

- Bipartite
- Maximum degree less than d
- Planar: Planar graphs can be characterized by forbidden minors according to Kuratowski's theorem, and graph minors can be expressed in monadic second order logic (see Courcelle [3] for how to do this)
- Edge graph (edge graphs can be characterized by forbidden subgraphs, as demonstrated in Harary [8])
- Transative: $\forall x, y, z[\mathbf{adj}(x, y) \wedge \mathbf{adj}(y, z) \to \mathbf{adj}(x, z)]$
- 1-connected

Many of these problems require some extra constraint on the subgraph in order to be NP-complete, like constraining some minimum size for the subgraph. More detail can be found in Garey and Johnson [7], but in every case this extra constraint is easy to add in the monadic second order formula.

7.2 Partition into K subgraphs with property P

Suppose that P is some monadic second order expressible property. We can describe an induced subgraph with the following transduction definition scheme:

- $\mathcal{W} := \{V\}$

- $\varphi := \textbf{true}$
- $\psi_i(x) := x \in V \vee \exists y, z\,[y, z \in V \wedge \textbf{edg}(x, y, z)]$
- $\textbf{edg}_{1,1,1}(e, x, y) := \textbf{edg}(e, x, y)$

With this, we can quantify over all partitions of a graph into K subgraphs and determine whether any are such that each of the partitions have property P.

$$\exists X_1, \ldots, X_K \left[\textbf{partition}(X_1, \ldots X_K) \wedge \bigwedge_{i=1}^{K} \text{“}X_i \text{ induces a graph with property } P\text{”}\right]$$

Examples of P which are expressible in monadic second order logic, and whose corresponding partition problem is NP-complete, include:

- Hamiltonicity
- Acyclic (see Feedback Vertex Set)
- Complete: $\forall x, y\,[\textbf{adj}(x, y)]$
- Perfect Matching. A graph is a perfect matching if it is 1-regular, which we have shown can be described in monadic second order logic.

7.3 Supergraph with K extra edges with property P

In the previous section we showed how to describe the supergraph of a graph obtained by adding K edges. We then need to quantify over all possible edges to add and check if any such supergraph has property P. Note that by the construction of the transduction we require that the graph has at least K vertices.

7.4 Hamiltonian Cycle

$$\exists C\,[\textbf{cycle}(C) \wedge \forall x \exists e\,[e \in C \wedge \textbf{inc}(e, x)]]$$

7.5 K-connected

We use the characterization that in any K-connected graphs, there are at least K disjoint paths between any two vertices.

$$\forall x, y \exists P_1, \ldots, P_K \left[\bigwedge_{i=1}^{K} \textbf{path}(P_i, x, y) \wedge \bigwedge_{i \neq j}^{K} \text{“}P_i \cap P_j = \emptyset''\right]$$

7.6 Monochromatic Triangle

We first write a monadic second order formula which describes a triangle in a graph:

$$\exists x, y, z\,[\textbf{adj}(x, y) \wedge \textbf{adj}(y, z) \wedge \textbf{adj}(z, x)]$$

Next, we define a transduction for the graph induced by an edge set:

- $\mathcal{W} := \{U\}$
- $\varphi := \textbf{true}$
- $\psi_i(x) := x \in U \vee \exists e\,[e \in U \wedge \textbf{inc}(e, x)]$

- $\mathbf{edg}_{1,1,1}(x, y, z) := \mathbf{edg}(x, y, z)$

Then we can quantify over all partitions of the edge set of a graph, and check whether any such partition does not contain a triangle.

$$\exists U_1, U_2 \left[\mathbf{partition}(U_1, U_2) \wedge \bigwedge_{i=1}^{2} \text{“}U_i \text{ induces a graph without a triangle”} \right]$$

7.7 Perfect Graph

First, it should be noted that this result isn't terribly important from a complexity point of view, since a polynomial time algorithm exists to determine whether a graph is perfect. However, I felt it interesting to include it since it makes use of much of the machinery we have at hand for writing properties in monadic second order logic. Also, as far as I know, no one else has shown that determining whether a graph is perfect is FPT when parameterized by treewidth.

A graph G is perfect if, for any set $X \subseteq V(G)$, the chromatic number and max clique size of the induced subgraph $G[X]$ are equal. This characterization will not help us in writing a formula, since monadic second order logic is unable to do arithmetic (specifically equality of size of sets). There is another characterization of perfect graphs, however, which will prove useful to us. We recall that the complement of a graph $\overline{G}$ is the graph whose edge set consists of edges not in G.

Theorem 7.7.1 (Strong Perfect Graph Theorem)**.** *Let G be a graph. Then G is perfect if and only if G contains neither C_{2k+1} nor $\overline{C_{2k+1}}$ as an induced subgraph for $k \geq 2$.*

We first write a formula which describes graphs which are odd cycles of length at least 5:

$$\text{“connected”} \wedge \text{“2-regular”} \wedge \forall X \left[(\forall x \left[x \in X \right]) \rightarrow \mathbf{Card}_{1,2}(X) \wedge \text{“}|X| \geq 5'' \right]$$

We've previously established that all the expressions written in quotes above can be expressed in monadic second order logic. Then using the transduction used in section 7.2, we can write the formula

$$\forall X \left[\text{“}X \text{ does not induce } C_{2k+1} \text{ for } k \geq 2\text{”} \right]$$

It remains to show that we can express that a graph does not contain $\overline{C_{2k+1}}$ as an induced subgraph. If we can define a transduction definition scheme for the complement of a graph, then we are done. We have to be careful when defining this transduction. How many copies are necessary to describe the edges we are adding? Consider a totally disconnected graph with n vertices — the complement would have $\frac{n^2-n}{2}$ edges. In the worse case scenario, we would need n copies of the graph to perform the transduction. In other words, the size of the formula would depend on the size of the graph, rendering moot any potential complexity results.

Instead we use an MS_1 transduction. Since the complement of the graph has the same number of vertices as the original graph, we only require one copy of the graph to perform the transduction. We define the following transduction scheme to describe the complement of a graph:

- $\mathcal{W} := \emptyset$
- $\varphi := \mathbf{true}$
- $\psi_1(x) := \mathbf{true}$
- $\mathbf{adj}_{1,1}(x, y) := \neg \exists e [\mathbf{edg}(e, x, y)]$

Lastly, we need to confirm that we can apply this transduction to the above formula; that the above formula does not require quantification over edges or edge sets. Connectedness is the transitive closure of the **adj** operator, so it MS_1 expressible. That a graph is 2-regular can be written without edge quantifications thusly:

$$\text{“}d(x) \leq 2'' := \forall y_1, y_2, y_3 \left[\bigwedge_{i=1}^{3} \mathbf{adj}(x, y_i) \rightarrow \bigvee_{i \neq j}^{3} y_i = y_j \right]$$

$$\text{“}d(x) \geq 2'' := \exists y_1, y_2\, [\mathbf{adj}(x, y_1) \wedge \mathbf{adj}(x, y_2) \wedge y_1 \neq y_2]$$

$$\text{“2-regular”} := \forall x\, [\text{“}d(x) \leq 2'' \wedge \text{“}d(x) \geq 2'']$$

The rest of the formula trivially does not require edge quantification. Therefore, that a graph induces $\overline{C_{2k+1}}$ for $k \geq 2$ can be described in monadic second order logic. Then with the strong perfect graph theorem we can write a formula which describes that a graph is perfect:

$$\forall X \left[\text{“}X \text{ does not induce } C_{2k+1},\ k \geq 2\text{”} \wedge \text{“}X \text{ does not induce } \overline{C_{2k+1}},\ k \geq 2\text{”}\right]$$

7.8 Complement has property P

Suppose that P is some monadic second order expressible property which can be written in MS_1, i.e only using the **adj** operator and without edge quantifications. As we discussed above, we can then use a transduction to describe if the complement of a graph has property P. An important example of a property which requires edge quantifications is hamiltonicity.

7.9 Feedback Vertex Set of size K

We first define a transduction scheme for a graph resulting from removing K vertices.

- $\mathcal{W} := \{v_1, \ldots, v_K\}$
- $\varphi := \mathbf{true}$
- $\psi_i(x) := \bigwedge_{i=1}^{K} x \neq v_i$
- $\mathbf{edg}_{1,1,1}(x, y, z) := \mathbf{edg}(x, y, z)$

Using our formula for whether a set of edges describes a cycle we defined in the previous chapter, we can write a formula describing that a graph is acyclic:

$$\text{“acyclic”} := \forall U\, [\neg\, \mathbf{cycle}(U)]$$

Then using the above transduction, we can quantify over all groups of K edges and determine whether the removal of some group of K vertices makes the graph acyclic. In the same fashion, we can also describe that a set has a feedback edge set of size K in monadic second order logic.

Bibliography

[1] Michel Bauderon and Bruno Courcelle. "Graph Expressions and Graph Rewritings". In: *Mathematical Systems Theory* 20 (1987), pp. 83–127.

[2] Bruno Courcelle. "The Monadic Second-order Logic of Graphs I: Recognizable Sets of Finite Graphs". In: *Inf. Comput.* 85.1 (Mar. 1990), pp. 12–75. ISSN: 0890-5401.

[3] Bruno Courcelle. "The monadic second-order logic of graphs III: tree-decompositions, minors and complexity issues". In: *RAIRO - Theoretical Informatics and Applications - Informatique Théorique et Applications* 26.3 (1992), pp. 257–286.

[4] Bruno Courcelle. "The monadic second-order logic of graphs V: on closing the gap between definability and recognizability". In: *Theor. Comput. Sci.* 80 (1991), pp. 153–202.

[5] Markus Frick and Martin Grohe. "The complexity of first-order and monadic second-order logic revisited". In: *Annals of Pure and Applied Logic* 130 (2004), pp. 3–31.

[6] Haim Gaifman. "On local and non local properties". In: *Proceedings of the herbrand symposium, logic colloquium* 81 (1982), pp. 105–135.

[7] Michael R. Garey and David S. Johnson. *Computers and Intractability; A Guide to the Theory of NP-Completeness*. USA: W. H. Freeman & Co., 1990. ISBN: 0716710455.

[8] Frank Harary. *Graph Theory*. CRC Press, 1994. ISBN: 0201410338.