

Exploiting Parallelism in Decomposition Methods for Constraint Satisfaction



Dmitri Akatov
Merton College
University of Oxford

A thesis submitted for the degree of
Doctor of Philosophy
Hilary 2010

Abstract

Constraint Satisfaction Problems (CSPs) are NP-complete in general, however, there are many tractable subclasses that rely on the restriction of the structure of their underlying hypergraphs. It is a well-known fact, for instance, that CSPs whose underlying hypergraph is acyclic are tractable. Trying to define “nearly acyclic” hypergraphs led to the definition of various hypergraph decomposition methods. An important member in this class is the hypertree decomposition method, introduced by Gottlob et al. It possesses the property that CSPs falling into this class can be solved efficiently, and that hypergraphs in this class can be recognized efficiently as well. Apart from polynomial tractability, complexity analysis has shown, that both afore-mentioned problems lie in the low complexity class LOGCFL and are thus moreover efficiently parallelizable. A parallel algorithm has been proposed for the “evaluation problem”, however all algorithms for the “recognition problem” presented to date are sequential.

The main contribution of this dissertation is the creation of an object oriented programming library including a task scheduler which allows the parallelization of a whole range of computational problems, fulfilling certain complexity-theoretic restrictions. This library merely requires the programmer to provide the implementation of several classes and methods, representing a general alternating algorithm, while the mechanics of the task scheduler remain hidden. In particular, we use this library to create an efficient parallel algorithm, which computes hypertree decompositions of a fixed width.

Another result of a more theoretical nature is the definition of a new type of decomposition method, called Balanced Decompositions. Solving CSPs of bounded balanced width and recognizing such hypergraphs is only quasi-polynomial, however still parallelizable to a certain extent. A complexity-theoretic analysis leads to the definition of a new complexity class hierarchy, called the DC-hierarchy, with the first class in this hierarchy, DC^1 , precisely capturing the complexity of solving CSPs of bounded balanced width.

Acknowledgements

This thesis would not have been possible without the help and support of a number of individuals and institutions. I would like to thank

- ☕ Prof. Georg Gottlob, my supervisor, for his support, encouragement, advice, intuition and his ability to recognize potential areas of interest in Computer Science for his students,
- ☕ Prof. Michael Benedikt and Prof. Franz Wotawa, my examiners, for their constructive criticism on parts of this dissertation,
- ☕ Prof. Luke Ong, my college advisor and undergraduate tutor, for his advice, motivation and belief,
- ☕ Prof. Luigi Palopoli, for being a great host during my visit to the University of Calabria in Rende, Italy,
- ☕ Greg Tarr, for laying the cornerstone to the parallelization of hyper-tree computations and passing it onto me,
- ☕ Joachim Gehweiler, for helping with the PUBWCL Java library and the first prototype implementation in Java,
- ☕ Joshua Leahy, for helping with the transition from Java to C++,
- ☕ Evgenij Thorstensen, for implementing several ideas linked to “intermediate separators” and for proof-reading parts of my thesis,
- ☕ Bruno Marnette, my friend and colleague, for introducing me to Georg, for making life at the office more fun, for many fruitful discussions during coffee breaks, and for proof-reading parts of my thesis,
- ☕ Clint Sieunarine, for proof-reading parts of my thesis,
- ☕ Oxford University Ice Hockey Club, its members and alumni, for making my stay in Oxford ever so enjoyable,
- ☕ Merton College, for their supportive environment and for providing me with food and accommodation over the many years,
- ☕ my comrades, for supporting me in good times and bad times, in particular Vidya Kumar for keeping me on track at a crucial time,
- ☕ my grandparents and my siblings, for their love, support, belief in me and never-ending motivation,
- ☕ My mother Alyona Akatova-Rösler, for her love and confidence, and her culinary support during the writing of my transfer dissertation,
- ☕ My father Leonid Makharinksy, for his love and financial support, and for being a role-model.

Contents

1	Introduction	1
1.1	Motivation	1
1.2	Structure of Thesis	3
2	Preliminaries	7
2.1	Notation and Basic Definitions	7
2.2	Computational Models	12
2.2.1	Turing Machines	12
2.2.2	Alternating Turing Machines	15
2.2.3	Auxiliary Pushdown Automata	16
2.2.4	Auxiliary Stack Automata	18
2.3	Important Complexity Classes	19
2.4	The Boolean Conjunctive Query Problem	22
2.5	Hypergraph Decompositions	24
2.5.1	Tree Decompositions	26
2.5.2	Generalized Hypertree Decompositions	28
2.5.3	Hypertree Decompositions	30
2.6	Game-theoretic Characterizations of Decompositions	31
2.6.1	Robber and Cops	32
2.6.2	Robber and Marshals	33

3	Balanced Decompositions: Divide and Conquer	35
3.1	Introduction	35
3.2	Structure of Chapter	38
3.3	Balanced Decompositions	39
3.3.1	Normal Forms	42
3.4	Robber and Sergeants	43
3.5	Another look at Stack Automata	47
3.5.1	The DC Hierarchy	48
3.5.2	Regular Stack Automata	49
3.6	Evaluating Balanced Queries	58
3.6.1	Membership in DC^1	59
3.6.2	DC^1 -Completeness	61
3.7	Computing Balanced Decompositions	65
3.8	Determinization and Parallelization	68
3.9	Reflections	69
4	Computing Hypertree Decompositions	70
4.1	Motivation	70
4.2	Structure of Chapter	71
4.3	A Sequential Polynomial-Time Algorithm	71
4.3.1	The Alternating Algorithm k -decomp	77
4.3.2	The Deterministic Algorithm $\det$ - k -decomp	81
4.3.3	Improvements to $\det$ - k -decomp	91
4.3.4	Adding Output and Decreasing Space Requirements	97
4.3.5	Issues with Parallelization	99
4.4	A Divide and Conquer Algorithm	105
4.4.1	The Alternating Algorithm k -divide-decomp	107
4.4.2	A Deterministic $O(\log^2 n)$ -space algorithm	114

4.4.3	Making the Algorithm Polynomial and Parallel	116
4.5	The Best of Both Worlds	119
4.5.1	Combining Both Approaches	120
4.5.2	Designing par-det- k -decomp	122
4.5.3	Heuristic for par-det- k -decomp	122
4.6	Reflections	126
5	Parallelizing Alternation Efficiently	127
5.1	Motivation	127
5.2	Structure of Chapter	128
5.3	Another look at ATMs	129
5.4	Search and Backtracking	134
5.5	Parallelizing the search	142
5.6	The Shared-Memory Approach	146
5.6.1	Working on Tasks	150
5.6.2	Acquiring new Tasks	151
5.6.3	Updating the Configuration Graph	155
5.6.3.1	Cancelling and Suspending Configurations	158
5.6.3.2	Resuming Configurations and Changing Priorities	160
5.6.3.3	Failing and Succeeding Configurations	164
5.6.3.4	Putting Things Together	164
5.6.4	Starting and Stopping the Scheduler	168
5.7	Reflections	168
6	Implementation and Experimental Results	171
6.1	Roadmap	171
6.2	The Scheduler	172
6.2.1	Languages and Libraries	172

6.2.2	Abandoned Approaches	173
6.2.3	Successful Approach	176
6.2.4	Performance Issues	177
6.3	Hypertree Decompositions	180
6.3.1	Data Structures	180
6.3.2	Performance	183
6.4	Experimental Results	184
6.5	Reflections	188
7	Conclusion and Future Work	189
7.1	Conclusion	189
7.2	Future Work	190
	Bibliography	i

List of Figures

2.1	Acyclic and Cyclic Hypergraphs	10
2.2	Hypergraph H_2	27
2.3	Tree Decomposition for H_2	27
2.4	Generalized Hypertree Decomposition for H_2	29
2.5	Hypertree Decomposition for H_2	30
3.1	Hypergraph H_3 with $\text{GHW}(H) = 3$ and $\text{BW}(H) = 2$	36
3.2	Generalized Hypertree Decomposition of H_3	36
3.3	Balanced Decomposition of H_3	37
3.4	Hypergraph G_3 with $\text{BW}(G_3) = 2$ and $\text{SW}(G_3) = 1$	41
3.5	Shallow Decomposition of G_3 of width 1	42
3.6	DC^1 in the Complexity Hierarchy	49
4.1	Component C	102
4.2	Non-balanced Separator	111
5.1	Reducing the Computation Tree	131
5.2	<i>EState.state</i> State Transition Diagram	157
6.1	Memory Levels in an x86 Computer	179
6.2	CPU and Memory Usage	185

List of Algorithms

1	<i>k</i> -hd-bcq	59
2	<i>k</i> -robber-sergeants	66
3	<i>k</i> -decomp	79
4	<i>k</i> -decomp with output	81
5	The class <i>Hypergraph</i>	82
6	Enumerating Components	83
7	det- <i>k</i> -decomp — first attempt	84
8	det- <i>k</i> -decomp	86
9	Enumerating Separators as in [32] — Part I	89
10	Enumerating Separators as in [32] — Part II	90
11	Enumerating Separators (improved)	92
12	<i>Hypergraph.coverVertex(Component)</i> — Part I	93
13	<i>Hypergraph.coverVertex(Component)</i> — Part II	94
14	<i>Hypergraph.coverVertex(Component)</i> — Part III	95
15	<i>Hypergraph.removeEdge(Component)</i>	96
16	det- <i>k</i> -decomp — Output	98
17	<i>decomposable</i> separated into <i>decomposable1</i> and <i>decomposable2</i> . . .	101
18	<i>k</i> -divide-decomp	112
19	Sorting Edges — Heuristic	125
20	<i>ATM</i> , <i>EState</i> , <i>UState</i>	141
21	<i>Scheduler</i> and <i>Thread</i>	149

22	<i>Thread.run()</i>	150
23	<i>Thread.doWork()</i>	151
24	<i>Scheduler.getTask()</i>	152
25	<i>Scheduler.firstFreeDescendant()</i>	153
26	<i>Scheduler.cancel(EState)</i> and <i>Scheduler.cancel(UState)</i>	159
27	<i>Scheduler.suspend(EState)</i> and <i>Scheduler.suspend(UState)</i>	160
28	<i>Scheduler.resume(EState)</i> and <i>Scheduler.resume(UState)</i>	161
29	<i>Scheduler.fail(EState/UState)</i> and <i>Scheduler.succeed(EState, UState)</i>	163
30	<i>Scheduler.update(EState)</i> and <i>Scheduler.update(UState)</i>	165
31	<i>Thread.update()</i> — Part I	166
32	<i>Thread.update()</i> — Part II	167
33	<i>Scheduler.start()</i> and <i>Scheduler.finish()</i>	168

Chapter 1

Introduction

1.1 Motivation

Evaluating *Boolean Conjunctive Queries* over a relational database (the BCQ evaluation problem) is one of the most important topics in database theory. Similarly, the *Constraint Satisfaction Problem* (CSP) is a central problem in artificial intelligence. Both of these problems are NP-complete in general as was shown by Chandra and Merlin [14], and by Mackworth [39], respectively. Moreover, these problems are very easily reducible to each other.

Since these problems are so important, both the database and the constraint communities developed methods to deal with NP-completeness by finding tractable subclasses of these problems. In particular, the *decomposition* approach proved to be very successful, in which the problem is divided into smaller sub-problems, such that each of these can be solved in polynomial time, and the “sub-solutions” can be combined into a full solution also in polynomial time.

Many of these decompositions are based on the decomposition of the *underlying hypergraph* of the conjunctive query, or, respectively, of the set of constraints. Such *hypergraph decompositions* usually have an associated *width*, which represents the size of each sub-problem.

Even though the BCQ evaluation problem becomes tractable for classes of queries with underlying hypergraphs of bounded width, we are also interested in recognizing

hypergraphs admitting such decompositions, and computing them, if they do. Thus the *recognition problem* for a certain type of hypergraph decomposition becomes very important.

For many decomposition methods this problem turns out to be NP-hard as well, rendering the use of such decompositions questionable in practice. However, for certain kinds of decompositions it was shown that this problem is tractable. In particular, for the method of *hypertree decompositions* [29], it was shown, that the recognition problem for fixed width decompositions is not only tractable, but is contained in the low complexity class LOGCFL.

LOGCFL is equivalent to the class SAC^1 of semi-unbounded boolean circuits of polynomial size and logarithmic depth. It is thus contained in AC^1 and NC^2 , which lie very low in the NC-AC-hierarchy. This in turn implies, at least in theory, the existence of a *parallel algorithm* using a polynomial number of processors and running in time $O(\log^2 n)$, where n is the size of the instance.

A recent trend developed in the hardware industry, substituting the design of processors with ever increasing clock speed, by that of processors with multiple cores and lower power consumption. It is to be expected, that in the near future most workstations will be operating on processors with up to 8 cores. Supercomputers have supported a large number of computational cores for some time now, however, the processors were still physically separated — sometimes on a single motherboard, sometimes on different machines — thus incurring a penalty on the “communication time”. The increase of cores on a single chip will also be highly beneficial to the *High Performance Computing* (HPC) community, as it will reduce the memory throughput bottleneck in many instances.

Thus, *parallel algorithms* will become ever more important for the consumer market as well as for the enterprise market. Considering the possible applications of hypertree decompositions, it will be important to have efficient parallel algorithms

not only to evaluate conjunctive queries, but also to find such decompositions.

An efficient parallel algorithm for the BCQ evaluation problem has already been devised in [28]. The parallelization of the recognition problem has been pioneered in Tarr’s Masters dissertation [54]. His implementation used a distributed-memory approach and did not make use of *memoization*, a crucial tool for ensuring polynomial running time. Its performance was much worse than that of any other mature implementations of sequential algorithms [32, 40, 38, 49], however some ideas were novel.

The aim of this dissertation is to design and implement an *efficient, scalable, parallel* algorithm, which constructs hypertree decompositions, and to exhibit its superior performance over existing methods.

1.2 Structure of Thesis

In Chapter 2 we provide the necessary background to the various topics covered in this dissertation. We motivate the study of hypergraphs and hypergraph decompositions through the *Boolean Conjunctive Query* (BCQ) evaluation problem. We then present a number of hypergraph decompositions already studied in the literature — most prominently *Hypertree Decompositions* — and illustrate how some types of these decompositions can be visualized as two-player games. Furthermore, some results from complexity theory are presented, which are directly relevant to the problems studied in this thesis. In particular, the Alternating Turing Machine, the auxiliary Pushdown Automaton and the auxiliary Stack Automaton models are described, which are very important to understand the results in following chapters. Finally, a brief overview of some relevant complexity classes is presented.

In Chapter 3 we present two new types of decompositions, called *Balanced Decompositions* and *Shallow Decompositions*. Originally introduced as a heuristic for

computing hypertree decompositions, bounded *Balanced Width* and bounded *Shallow Width* capture much wider classes of hypergraphs than bounded *Hypertree Width*. These decompositions impose a restriction on the shape of the underlying hypertree *a priori*. An important motivation to study these decompositions is the existence of a quasi-polynomial algorithm for the BCQ evaluation problem, which is moreover easily parallelizable on a linear number of processors. The analysis of the exact complexity of the BCQ evaluation problem for the class of queries with bounded *Balanced Width* leads to the definition of a new complexity class hierarchy, defined by restricting computational resources on an auxiliary Stack Automaton. We show that recognizing hypergraphs of bounded balanced width is feasible in the lowest member of this hierarchy, DC^1 , and that the BCQ evaluation problem for the class of queries of bounded balanced width is complete for this class. We also relate DC^1 to existing complexity classes by placing it between LOGCFL and $NTiSp(n^O(1), \log^2 n)$.

In Chapter 4 we deal with the computation of *Hypertree Decompositions*. Hypertree decompositions are important because of the polynomial tractability of both the recognition problem as well as the BCQ evaluation problem on classes of queries with bounded hypertree width. The computation of hypertree decompositions is a well studied topic, and there are a number of algorithms and heuristics developed for it already. However, all the good algorithms created so far are sequential. We examine the best known¹ algorithm to date, *det- k -decomp* [32], which, for an input parameter k , checks whether a hypertree decomposition of width k exists and computes it, if it does. This algorithm is the *determinization* of the alternating algorithm *k-decomp* [29]. It is based on *search and backtracking*, and guarantees a polynomial running time through the use of a dynamic table, thus limiting the total number of recursive calls. We examine some obvious possibilities for parallelization of this algorithm

¹ Here, we take the *minimal width* to be the deciding factor, i.e. if a hypergraph H has hypertree width k , then *det- k -decomp* is faster than other algorithms for widths close to k , whereas other algorithms may be faster for higher widths.

and exhibit their shortcomings. Being a problem in LOGCFL, the computation of hypertree decompositions is also in the class of deterministic $\log^2$ -SPACE. By using ideas from the previous chapter, we exhibit such an algorithm and show that it is easily parallelizable on a linear number of processors. This algorithm constructs a decomposition by making *balanced hypercuts* and thus guessing future nodes in the hypertree, rather than computing it node by node, top-down. Simply reducing the space-bound brings with it a super-polynomial running time requirement (even with a polynomial number of processors). However, by maintaining a dynamic table we can still guarantee polynomial running time and optimal parallelism. We call this algorithm *par- k -decomp*. Unfortunately, this algorithm is much worse asymptotically than *det- k -decomp*, simply because there are a lot more (even if still polynomially many) possible table entries. We examine how to combine both approaches to get better performance: We can use a few steps of *par- k -decomp* to divide the hypergraph into several regions of approximately the same size. For each of these regions we can then use *det- k -decomp* to find a partial decomposition. Whenever two of these partial decompositions fit together, we have a decomposition for the union of these two regions, and we can proceed recursively to reconstruct a decomposition for the whole hypergraph. Finally, we show how to simulate a step of *par- k -decomp* using *det- k -decomp* itself, thus eliminating the need to compute separators from scratch altogether, and how to deal with regions “not fitting together”. We thus derive an efficient parallelization of *det- k -decomp*, which we call *par-det- k -decomp*. As an addendum, we improve the original *det- k -decomp* algorithm by providing an alternative way to compute “adjacent separators”. We also show how to ensure a good distribution of “initial starting points” for *par-det- k -decomp* taking inspiration from electrostatics.

In Chapter 5 we construct a *task scheduler* and an *API* which provides the necessary features to implement general alternating algorithms in the complexity class

LOGCFL on a *shared-memory* machine with multiple cores, in particular also k -decomp. We show how the original det- k -decomp algorithm corresponds to a search in the *configuration graph* of an ATM. We formalize the semantics of this algorithm as finding the *fixed-point* of a *monotone map* on a *complete partial order*. We then use these semantics to extend the search to a generic parallel algorithm, and to show its correctness. We proceed to analyze the specifics of the configuration graph of a problem in the class LOGCFL, and show how to get good heuristics for the parallelization. We then present an API generic enough to capture any alternating algorithm satisfying these conditions. The programmer thus simply has to implement the functions representing transitions from a configuration to its children, while the specifics of the scheduler remain hidden. Finally, we present the design of the scheduler.

In Chapter 6 we describe the implementation of the scheduler using C++ and Pthreads. We briefly argue how to improve its performance on the x86 architecture. We then present the specific data structures we used to implement the hypertree computation algorithm from chapter 4, together with an analysis of their space requirements. Finally, we present results from test runs on several example hypergraphs and compare the performance of our algorithm to that of the original det- k -decomp implementation.

We conclude the dissertation in Chapter 7 and outline directions for future research.

Chapter 2

Preliminaries

Unless otherwise stated, we assume all sets in this thesis to be *finite*. We write $\mathcal{P}(S)$ to denote the *powerset* of S , i.e. the set of all subsets of S . We write $*$ to denote the *Kleene operator*, i.e. S^* is the set of all finite sequences of elements of S , including the empty set. We write $|S|$ for the cardinality of S , i.e. the number of elements contained in S .

2.1 Notation and Basic Definitions

Definition 2.1. A *hypergraph* is a tuple $H = (V(H), E(H))$ where $V(H)$ is a set called the *vertices* of H , and $E(H) \subseteq \mathcal{P}(V(H)) \setminus \{\emptyset\}$ is a set called the *hyperedges* or *edges* of H .

A *graph* is a hypergraph G such that for all $e \in E(G)$, $|e| = 2$.

A hyperedge is a non-empty set of any number of vertices. We do not allow repetitions (“parallel edges”). A graph, according to the above definition, is, strictly speaking, an *undirected graph*. We will only use the term hyperedge to explicitly emphasize the fact that we are dealing with a hypergraph, rather than a graph, when the context is ambiguous.

We will usually use $u, v, w, \dots, U, V, W, X, Y, \dots$ to denote vertices of a hypergraph, and $e, f, g, \dots$ to denote its edges.

Here are some standard notions from graph theory that are also applicable to hypergraphs:

- a vertex v and an edge e are *incident* iff $v \in e$;
- two vertices u and v are *adjacent* iff there exists $g \in E(H)$ such that $u, v \in g$;
- two edges e and f are *adjacent* iff $e \cap f \neq \emptyset$.
- a *vertex-path* is an indexed set $\{v_i\}_{i=0}^t$ of vertices such that for all $i, 0 \leq i < t$, $v_i \neq v_{i+1}$ and v_i and v_{i+1} are adjacent; the *length* of the path is t .
- an *edge-path* is an indexed set $\{e_i\}_{i=0}^t$ of edges such that for all $i, 0 \leq i < t$, $e_i \neq e_{i+1}$ and e_i and e_{i+1} are adjacent; the *length* of the path is t .
- two vertices u and w are *connected* iff there exists a vertex-path $\{v_i\}_{i=0}^t$ such that $u = v_0$ and $w = v_t$;
- two edges f and g are *connected* iff there exists an edge-path $\{e_i\}_{i=0}^t$ such that $f = e_0$ and $g = e_t$.

Definition 2.2. Let H be a hypergraph and let $C \subseteq V(H)$, $R, M \subseteq E(H)$.

- Two vertices u and v are $[C]$ -*adjacent* iff $u \notin C$, $v \notin C$ and u and v are adjacent. (A vertex in C is not $[C]$ -adjacent to any vertex, including itself.)
- A $[C]$ -*vertex-path* is a vertex-path $\{v_i\}_{i=0}^t$ such that for all $i, 0 \leq i \leq t$, $v_i \notin C$.
- Two vertices u and w are $[C]$ -*connected* iff there exists a $[C]$ -vertex-path $\{v_i\}_{i=0}^t$ such that $u = v_0$ and $w = v_t$.
- A $[C]$ -*vertex-component* is a maximal $[C]$ -connected subset D of $V(H)$: for all $u, v \in D, w \in V(H)$, u and v are $[C]$ -connected and if w is $[C]$ -connected to u , then $w \in D$.

- Two edges e, f are $[C]$ -adjacent iff they are equal or share a vertex outside C , i.e. if there exists $v \in V(H) \setminus C$ such that $v \in e$ and $v \in f$.
- A $[C]$ -edge-path in R is an edge path $\{e_i\}_{i=0}^t$ such that for all $i, 0 \leq i \leq t$, $e_i \in R$ and for all $i, 0 \leq i < t$, e_i is $[C]$ -adjacent to e_{i+1} . If $R = E(H)$, we simply speak of a $[C]$ -edge path.
- Two edges $f, g \in R$ are $[C]$ -connected in R iff there exists an edge-path $\{e_i\}_{i=0}^t$ in R such that $f = e_0$, $e_t = g$. If $R = E(H)$, we simply say that f and g are $[C]$ -connected.
- A $[C]$ -edge-component of R is a maximal $[C]$ -connected subset S of R : for all $e, f \in S, g \in R$, e and f are $[C]$ -connected, and if g is $[C]$ -connected to e in R , then $g \in S$. If $R = E(H)$, we simply call S a $[C]$ -edge-component.
- Two vertices are $[M]$ -adjacent iff they are $[\bigcup M]$ -adjacent.
- Two vertices are $[M]$ -connected iff they are $[\bigcup M]$ -connected.
- An $[M]$ -vertex-component is simply a $[\bigcup M]$ -vertex-component.
- Two edges e, f are $[M]$ -adjacent if $e, f \notin M$ and they are $[\bigcup M]$ -adjacent. (An edge in M is not $[M]$ -adjacent to any edge, including itself.)
- A $[M]$ -edge-path in R is a $[\bigcup M]$ -edge-path $\{e_i\}_{i=0}^t$ in R such that for all $i, 0 \leq i \leq t$, $e_i \notin M$. If $R = E(H)$, we simply speak of a $[M]$ -edge-path.
- Two edges $f, g \in R$ are $[M]$ -connected in R iff there exists a $[M]$ -edge-path $\{e_i\}_{i=0}^t$ in R such that $f = e_0$ and $e_t = g$. If $R = E(H)$, we simply say that f and g are $[M]$ -connected.
- An $[M]$ -edge-component of R is a maximally $[M]$ -connected subset S of R : for all $e, f \in S, g \in R$, e and f are $[M]$ -connected in R and if g is $[M]$ -connected to e in R , then $g \in S$. If $R = E(H)$, we simply call S a $[M]$ -edge-component.

In this dissertation we assume that all hypergraphs are *connected*. That is $V(H)$ is an $[\emptyset]$ -vertex-component or, alternatively, $E(H)$ is an $[\emptyset]$ -edge-component and there are no *isolated vertices* (vertices not contained in any edge).

An important notion in graph theory is that of *acyclicity*. A graph is acyclic, if it has no cycles. A *cycle* is a vertex-path with the same starting and finishing vertex otherwise containing no repetition of vertices. Unfortunately this definition is too restrictive for the setting of hypergraphs: Any hypergraph containing an edge of cardinality at least 3 would be deemed to contain a cycle.

Definition 2.3. [6] A hypergraph H is *acyclic* iff the GYO reduct of H is the empty hypergraph (a hypergraph containing no vertices or hyperedges).

The *GYO reduct* of H is a hypergraph obtained from H by repeatedly applying one of the following two reduction rules in any order, as long as at least one of them is applicable:

- If there are hyperedges e, f such that $e \subseteq f$, remove e from $E(H)$.
- If there is a vertex v , which is only contained in one hyperedge e , then remove v from $V(H)$ and from e . If e becomes empty, remove it from $E(H)$, also.

Figure 2.1 illustrates some cyclic and acyclic hypergraphs.

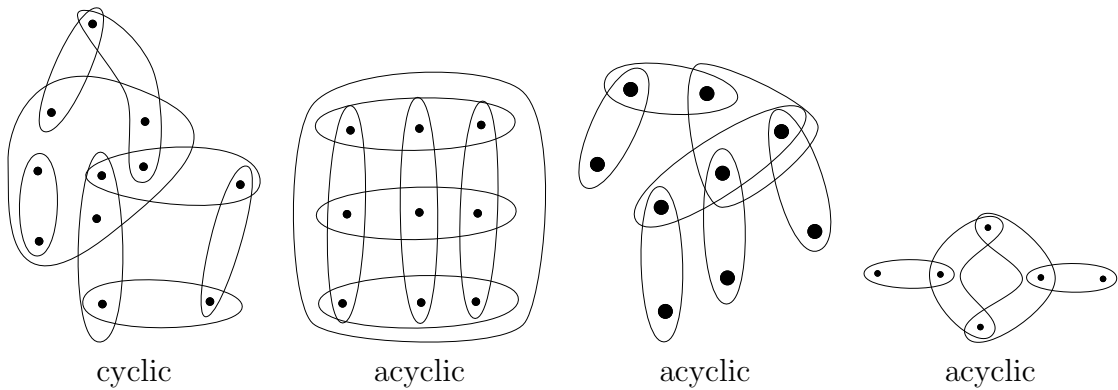


Figure 2.1: Acyclic and Cyclic Hypergraphs

A *tree* is simply a connected acyclic graph. In this dissertation we will however require two special kinds of trees, and will use some non-standard notation for their usage:

Definition 2.4. A *rooted tree* is a connected undirected acyclic graph T with a distinguished node $O(T)$, the *root* of T . We write T synonymously for the set of nodes of T . We define $\text{size}(T) = |T|$.

Given $v \in T$, we define $\text{depth}_T(v)$ to be the length of the (unique) shortest path between v and $O(T)$. $O(T)$ itself has depth 0. We define $\text{depth}(T) = \max_{v \in T} \text{depth}_T(v)$.

A node v is the *parent* of a node u (u is a *child* of v) iff u and v are adjacent in T and u has greater depth.

The definition of a rooted tree induces a “child function” $T : T \rightarrow \mathcal{P}(T)$, where we also write T for the function, which can be defined for every $v \in T$ as $T(v) = \{u \in T \mid u \text{ is a child of } v\}$.

We also get an “ancestor relation” $\sqsubseteq_T$ which is the reflexive transitive closure of the child function T : For nodes $u, v \in T$, $u \sqsubseteq_T v$ iff $u = v$ or u is an ancestor of v . This relation induces a (complete) partial order on the nodes of T with $O(T)$ being the (unique) minimal element, and we can thus write $u \sqcap_T v$ for the “deepest common ancestor” of both u and v .

Given $p \in T$ we define T_p to be the sub-tree of T rooted at p : $T_p = \{v \in T \mid p \sqsubseteq_T v\}$, $O(T_p) = p$ and $T_p(v) = T(v)$ for all $v \in T_p$.

An immediate consequence of this definition is

Proposition 2.1. *For any rooted tree T , $\text{depth}(T) \leq \text{size}(T) - 1$.*

Definition 2.5. An *ordered tree* T is a rooted tree additionally imposing an order on the children of every node. Hence the “child function” becomes $T : T \rightarrow T^*$.

For two vertices $u, v \in T$ we write $u <_T v$ iff there exists a node x and sequences of nodes $a, b, c \in T^*$ such that $T(x) = aubvc$ (i.e. if u is an “older sibling” of v).

We can extend the “ancestor” and “older sibling” relations to a total ordering $\leq_T$ on T in the following manner: $u \leq_T v$ iff there exist nodes w, x such that $w \sqsubseteq_T u$, $x \sqsubseteq_T v$ and either $w = x$ or $w <_T x$. We call $\leq_T$ the *tree traversal order of T* .¹

2.2 Computational Models

In this section we present several relevant computational models. In particular, starting with the standard definition of the Turing Machine, we then proceed to consider Alternating Turing Machines, Auxiliary Pushdown Automata and finally Auxiliary Stack Automata.

2.2.1 Turing Machines

There are numerous definitions of the Turing Machine in the literature, e.g. [51], [43]. We adopt the one with a separate read-only *input tape* which contains left and right hand *end markers*, to make sure the input head does not move past them, and a single *work tape*, which contains a left hand end marker and is otherwise initially blank.

Definition 2.6. [51] A *Turing Machine* (TM) is a tuple $(Q, \Sigma, \Gamma, \delta, q_0, q_A, q_R)$ where Q, Σ, Γ are all finite sets and

- Q is the set of *states*,
- Σ is the *input alphabet* not containing the special *blank* symbol $\sqcup$ and containing the *end marker symbols* $\triangleright$ and $\triangleleft$.
- Γ is the *tape alphabet*, where $\sqcup \in \Gamma$ and $\Sigma \subseteq \Gamma$,
- $\delta \subseteq (Q \times \Sigma \times \Gamma) \times (\{L, R, N\} \times \Gamma \times \{L, R, N\} \times Q)$ is the *transition relation*,
- $q_0 \in Q$ is the *initial state*,

¹ This is the order a depth-first search would traverse the nodes of T .

- $q_A \subseteq Q$ are the *accepting states*,
- $q_R \subseteq Q$ are the *rejecting states* where $q_A \cap q_R = \emptyset$.

If δ is a (partial) function, we say that the Turing Machine is *deterministic*. Otherwise, we say that the Turing Machine is *nondeterministic*.

More work tapes can be added when required, with the appropriate modifications to the transition relation. Also, we may introduce a fixed right hand end marker on the work tape(s) to limit space usage.

Definition 2.7. Let $M = (Q, \Sigma, \Gamma, \delta, q_0, q_A, q_R)$ be a Turing Machine. A *configuration* of M is a tuple (q, h_i, h_w, W) , where

- $q \in Q$ is the *current state*,
- $h_i, h_w \in \mathbb{N}$ are the *input* and *work tape head positions*, respectively,
- $W \in \Gamma^*$ is the contents of the worktape (we always omit trailing blank symbols).

A configuration is *accepting* iff $q \in q_A$. A configuration is *rejecting* iff $q \in q_R$. The *initial configuration* is $(q_0, 0, 0, \sqcup)$. For $s \in \mathbb{N}$, we say a configuration uses space s iff $|W| = s$.

Given a Turing Machine M , an input word $w \in \Sigma^*$, and configurations c_1, c_2 , we say that (c_1, c_2) is a *valid transition*, written $c_1 \xrightarrow{M, w} c_2$ iff the transition obeys the rules of the transition relation δ of M .² We write $\xrightarrow{M, w}^*$ for the reflexive transitive closure of $\xrightarrow{M, w}$.

Definition 2.8. Let M be a TM, w an input string and let $\mathcal{C}$ be the set of configurations of M . The *computation tree of M on w* is a rooted labelled tree (T, c) where $c : T \rightarrow \mathcal{C}$ such that for all non-leaf nodes $t \in T$, if $u \in T(t)$, then $c(t) \xrightarrow{M, w} c(u)$. The computation tree is *accepting* iff $c(O(T))$ is the initial configuration and there is

² We omit the details of this, as they are standard.

a leaf $l \in T$ such that $c(l)$ is accepting. The computation tree is *rejecting* iff $c(O(T))$ is the initial configuration and if $l \in T$ is a leaf, then $c(l)$ is rejecting.

For deterministic TMs the computation tree is simply a chain, since any one configuration can have at most one successor.

Definition 2.9. Let $L \subseteq \Sigma^*$ be a *language*. Let M be TM. We say that M *accepts* L iff for all $w \in L$ there is an accepting computation tree of M on w . We say that M *decides* L iff for all $w \in L$ there is either an accepting or rejecting tree of M on w .

Definition 2.10. Let M be a TM and let $T(n), S(n) : \mathbb{N} \rightarrow \mathbb{N}$. We say that M operates in *time* $T(n)$ and *space* $S(n)$ if for all accepted inputs of length n there is an accepting computation tree of height at most $T(n)$ each of whose nodes is labelled by a configuration using space at most $S(n)$.

Definition 2.11. Given constructible functions $T(n)$ and $S(n)$, $\text{DTiSp}(T(n), S(n))$ is the class of all languages decided by Deterministic Turing Machines operating (simultaneously) in time $O(T(n))$ and space $O(S(n))$. $\text{NTiSp}(T(n), S(n))$ is the corresponding class decided by Nondeterministic Turing Machines.

Complexity classes derived from both time **and** space limited Turing Machines were studied e.g. in [11], [17] and [42].

Of course we can limit only one of the resources (time or space). We write:

$$\text{DTime}(T(n)) = \text{DTiSp}(T(n), \infty)$$

$$\text{DSpace}(S(n)) = \text{DTiSp}(\infty, S(n))$$

$$\text{NTime}(T(n)) = \text{NTiSp}(T(n), \infty)$$

$$\text{NSpace}(S(n)) = \text{NTiSp}(\infty, S(n))$$

2.2.2 Alternating Turing Machines

A standard definition of Alternating Turing Machines can be found in e.g. [13]:

Definition 2.12. An *Alternating Turing Machine* is a tuple $(Q, \Sigma, \Gamma, \delta, q_0, Q_A, Q_R, Q_U, Q_E)$ such that $(Q, \Sigma, \Gamma, \delta, q_0, Q_A, Q_R)$ is a TM and additionally

- $Q_E \subseteq Q$ are the *existential states*,
- $Q_U \subseteq Q$ are the *universal states*,
- Q_A, Q_R, Q_E, Q_U form a partition of Q .

Thus, every state (including the initial state) is either accepting, rejecting, existential or universal. Given a configuration c (defined exactly as for TMs), we also speak of accepting, rejecting, existential or universal configurations.

For an ATM to accept a word, we will require an existential configuration to have at least one successor which leads to acceptance, whereas all successors of a universal configuration will be required to lead to acceptance.

Definition 2.13. [47] Let M be an ATM, w a string and $\mathcal{C}$ the set of configurations of M . The *computation tree of M on w* is a rooted labelled tree (T, c) where $c : T \rightarrow \mathcal{C}$ such that the following hold:

- For all non-leaf nodes $t \in T$, if $c(t)$ is existential, then there exists $u \in T(t)$, such that $c(t) \xrightarrow{M, w} c(u)$.
- For all non-leaf nodes $t \in T$, if $c(t)$ is universal and if $c(t) \xrightarrow{M, w} d$, then there exists $u \in T(t)$, such that $c(u) = d$.

A computation tree is *accepting*, if the following hold:

- $c(O(T))$ is the initial configuration of M on w .
- For all leaf nodes $t \in T$, $c(t)$ is accepting.

Definition 2.14. [47] An ATM uses *time* $T(n)$ if for all accepted inputs of length n there is an accepting computation tree of height at most $T(n)$.

An ATM uses *space* $S(n)$ if for all accepted inputs of length n there is an accepting computation tree each of whose nodes is labelled by a configuration using space at most $S(n)$.

An ATM uses *tree-size* $Z(n)$ if for all accepted inputs of length n there is an accepting computation tree of size (number of nodes) at most $Z(n)$.

An ATM uses $A(n)$ *alternations* if for all accepted inputs of length n there is an accepting computation tree such that on any path from the root of the tree the number of changes between universal and existential states is at most $A(n)$.

Definition 2.15. Given constructible functions $T(n)$, $S(n)$ and $Z(n)$, we define $\text{ATiSpSz}(T(n), S(n), Z(n))$ to be the class of languages decided by an Alternating Turing Machine simultaneously operating in time $O(T(n))$ and space $O(S(n))$, having tree-size at most $O(Z(n))$.

We write

$$\text{ATiSp}(T(n), S(n)) = \text{ATiSpSz}(T(n), S(n), \infty)$$

$$\text{ASpSz}(S(n), Z(n)) = \text{ATiSpSz}(\infty, S(n), Z(n))$$

$$\text{ATiSz}(T(n), Z(n)) = \text{ATiSpSz}(T(n), \infty, Z(n))$$

$$\text{ATime}(T(n)) = \text{ATiSp}(T(n), \infty)$$

$$\text{ASpace}(S(n)) = \text{ATiSp}(\infty, S(n))$$

2.2.3 Auxiliary Pushdown Automata

The study of context-free languages introduced the fairly common model of the *pushdown automaton* (PDA). This is a machine with a finite control, an input tape and a pushdown. The pushdown is a LIFO (last in first out) container. A symbol can be pushed onto the top of the (contents of the) pushdown, and the top symbol of

the pushdown can be read or removed from it (popped). A PDA can be described as a one-tape TM with a restricted transition relation, which allows read and write access only to the last written element on the work tape and makes sure that the head of the work tape points to that last element or to the blank symbol immediately after it. Just like TM's, these automata can be deterministic or nondeterministic. They are weaker than Turing Machines (TMs), since a tape is more powerful than a pushdown. In fact nondeterministic PDAs (NPDAs) only recognize precisely the *context-free languages*.

An *auxiliary pushdown automaton* (AuxPDA) is a generalization of both the TM and the PDA. It possesses both a tape and a pushdown. Obviously, adding a pushdown to a TM does not give it more power than adding an extra tape (which, of course, can be simulated on the old tape). However, if we consider resource-bounded subclasses of such machines, it turns out that an AuxPDA (without any or with only “mild” restrictions on the usage of the pushdown tape) is more powerful than a Turing Machine, with equal restrictions on the running time and worktape space usage.

One can use any one of several equivalent definitions of AuxPDAs, e.g. [16]. Here, we adopt the view that an AuxPDA is a two-worktape TM whose second worktape is the pushdown, i.e. the operations accessing this worktape can only be done at its right-hand end. A configuration of an AuxPDA M now also contains the contents of the pushdown. The computation tree is defined analogously to the computation tree of a Turing Machine. We distinguish between the usage of space on the tape and on the pushdown, limiting them independently. We call the maximal amount of space used by the pushdown tape the *maximal pushdown height*.

Definition 2.16. An AuxPDA uses *time* $T(n)$ if for all accepted inputs of length n there is an accepting computation tree of height at most $T(n)$.

An AuxPDA uses *space* $S(n)$ if for all accepted inputs of length n there is an accepting computation tree each of whose nodes is labelled by a configuration with worktape

space usage at most $S(n)$.

An AuxPDA uses *maximal pushdown height* $H(n)$ if for all accepted inputs of length n there is an accepting computation tree each of whose nodes is labelled by a configuration with pushdown height at most $H(n)$.

Definition 2.17. Given constructible functions $T(n), S(n)$ and $H(n)$, we define $\text{DTiSpPh}(T(n), S(n), H(n))$ to be the class of languages decided by a deterministic AuxPDA simultaneously operating in time $O(T(n))$, space $O(S(n))$ and using maximal pushdown height $O(H(n))$. $\text{NTiSpPh}(T(n), S(n), H(n))$ is the corresponding class for nondeterministic AuxPDAs.

Note that this definition only makes sense if the stack is larger than the worktape, i.e. if $H(n)$ is asymptotically larger than $S(n)$. AuxPDAs allow to run “recursive algorithms” by pushing “temporary variables” before a recursive call, and popping them back onto the worktape, after the call returns.

2.2.4 Auxiliary Stack Automata

Ginsburg et al. introduced the *stack automaton* (SA) model originally for the study of compilers [24]. Instead of a pushdown, an SA has a *stack*, which acts like a pushdown for writing (pushing and popping), but like a tape for reading. That is, when adding or deleting a new symbol to or from the stack, this has to be done at the right-hand end of the stack tape (the top of the stack), but the head can move freely over the stack tape and read its contents. Again, an SA can be deterministic or nondeterministic. SAs are more powerful than PDAs, but still less powerful than TMs.

Analogously to extending TMs with a pushdown (or PDAs with a worktape), Ibarra proposed to do the same with SAs [36]. This gives the model of the *auxiliary stack automaton* (AuxSA). We interpret it again as a two-tape TM, (one being the stack tape), such that the transition relation forbids the head of the stack to

write anywhere but on the right-most symbol of the stack or the blank immediately following it.

Again, we can limit the familiar resources time and space, as well as the maximal stack height.

Definition 2.18. An AuxSA uses *time* $T(n)$ if for all accepted inputs of length n there is an accepting computation tree of height at most $T(n)$.

An AuxSA uses *space* $S(n)$ if for all accepted inputs of length n there is an accepting computation tree each of whose nodes is labelled by a configuration using space at most $S(n)$.

An AuxSA uses *maximal stack height* $H(n)$ if for all accepted inputs of length n there is an accepting computation tree each of whose nodes is labelled by a configuration with stack height at most $H(n)$.

Definition 2.19. Given constructible functions $T(n), S(n)$ and $H(n)$, we define $\text{DTiSpSh}(T(n), S(n), H(n))$ to be the class of languages decided by a deterministic AuxSA simultaneously operating in time $O(T(n))$, space $(O(S(n)))$ and using maximal stack height $O(H(n))$. $\text{NTiSpSh}(T(n), S(n), H(n))$ is the corresponding class for nondeterministic AuxSAs.

Auxiliary Stack Automata also allow “recursive” programs, which additionally can read previously pushed variables.

2.3 Important Complexity Classes

We assume familiarity with *log-space reductions* and *complete problems* (see e.g. [43]). Some standard results relating complexity classes are the following:

- $\text{DSpace}(S(n)) \subseteq \text{NSpace}(S(n))$
- $\text{NSpace}(S(n)) \subseteq \text{DSpace}((S(n))^2)$

- $\text{DTime}(T(n)) \subseteq \text{NTime}(T(n))$
- $\text{NTime}(T(n)) \subseteq \text{DSpace}(T(n))$

We assume the reader is also familiar with the standard complexity classes defined using Turing Machines (see e.g. [51]):

- $L = \text{DSpace}(\log n)$
- $NL = \text{NSpace}(\log n)$
- $P = \text{DTime}(n^{O(1)})$
- $NP = \text{NTime}(n^{O(1)})$
- $PSPACE = \text{DSpace}(n^{O(1)}) = \text{NPSPACE} = \text{NSpace}(n^{O(1)})$

The famous relationship between these classes is

$$L \subseteq NL \subseteq P \subseteq NP \subseteq PSPACE$$

with at least one of these inclusions being strict.³

Using the model of *Boolean Circuits* (see e.g. [55]) one can define the NC *hierarchy*, the AC *hierarchy* and the SAC *hierarchy*: For all $k \in \mathbb{N}$

- NC^k is the class of languages decided by uniform boolean circuits with bounded fan-in using $O(n^{O(1)})$ gates and having depth $O(\log^k n)$
- AC^k is the class of languages decided by uniform boolean circuits with unbounded fan-in using $O(n^{O(1)})$ gates and having depth $O(\log^k n)$
- SAC^k is the class of languages decided by uniform boolean circuits with semi-unbounded fan-in (all OR-gates have unbounded fan-in, all AND-gates have bounded fan-in) using $O(n^{O(1)})$ gates and having depth $O(\log^k n)$

³ It is widely believed that all of the above inclusions are strict.

We define $\text{NC} = \bigcup_{k \in \mathbb{N}} \text{NC}^k$, similarly SAC and AC . It turns out that $\text{NC} = \text{SAC} = \text{AC} \subseteq \text{P}$. Otherwise, for all $k \in \mathbb{N}$, we have $\text{NC}^k \subseteq \text{SAC}^k \subseteq \text{AC}^k \subseteq \text{NC}^{k+1}$.

Interestingly enough, elements in the NC hierarchy admit a characterization in terms of NauxPDAs [47]: For all $k \geq 2$, $\text{NC}^k = \text{NSpPh}(\log n, \log^k n)$. Their union can also be described similarly: $\text{NC} = \text{NSpPh}(\log n, 2^{\log^{O(1)} n})$.

There is even a characterization of P in terms of NauxPDAs [47]:

$$\text{P} = \text{NTiSpPh}(2^{n^{O(1)}}, \log n, \infty)$$

As was shown in [47], there is a very close connection between NauxPDAs and ATMs: under similar space restrictions, time on a NauxPDA is reflected in the proof-tree size of the ATM, whereas time on the ATM is reflected in the maximal pushdown height of the NauxPDA: For $S(n) = \Omega(\log n)$, $T(n), Z(n) = \Omega(n^{O(1)})$

$$\text{ATiSpSz}(T(n), S(n), Z(n)) = \text{NTiSpPh}(Z(n), S(n), T(n))$$

This gives different perspectives on some important complexity classes, and allows a better understanding of the algorithms for problems in these classes. In particular, ATMs are commonly used as a model for parallel computation.

A very central complexity class in this dissertation is the class LOGCFL . It was originally defined as the class of languages that are log-space reducible to context-free languages. Besides this original definition, it also admits many other equivalent definitions [47]:

- $\text{LOGCFL} = \text{SAC}^1$
- $\text{LOGCFL} = \text{ASpSz}(\log n, n^{O(1)})$
- $\text{LOGCFL} = \text{NTiSpPh}(n^{O(1)}, \log n, \infty)$

Another very important result from [47] is

$$\text{LOGCFL} = \text{NTiSpPh}(n^{O(1)}, \log n, \log^2 n)$$

and hence also

$$\text{LOGCFL} \subseteq \text{ATiSp}(\log^2 n, \log n)$$

which also implies the existence of algorithms using only $O(\log n)$ alternations.

We can summarize this chapter with the following series of inclusions:

$$\text{AC}^0 \subset \text{NC}^1 \subseteq \text{L} \subseteq \text{NL} \subseteq \text{LOGCFL} \subseteq \text{AC}^1 \subseteq \text{NC}^2 \subseteq \text{NC} \subseteq \text{P}$$

2.4 The Boolean Conjunctive Query Problem

We adopt the usual logical representation of a relational database [6], where data tuples are identified with logical ground atoms and conjunctive queries are represented as datalog rules.

Consider for instance a relational schema with three relation names *plays* (*Person*, *Team*, *Position*); *coaches* (*Person*, *Team*); *father* (*Person*, *Person*), and the following two queries:

$$Q_1 : \text{ans}() \leftarrow \text{plays}(S, T, P), \text{coaches}(F, T), \text{father}(F, S).$$

$$Q_2 : \text{ans}() \leftarrow \text{plays}(S, T, P), \text{coaches}(F, T'), \text{father}(F, S).$$

The query Q_1 asks: “Is there a player playing for a team coached by his father?” The query Q_2 asks: “Is there a player whose father coaches some team?”

Consider the following database:

plays			coaches		father	
Person	Team	Pos.	Person	Team	Person	Person
Brandon Sutter	Hurricanes	C	Brent Sutter	Flames	Brent Sutter	Brandon Sutter
Brett Sutter	Flames	C	Paul Maurice	Hurricanes	Darryl Sutter	Brett Sutter
Alex Ovechkin	Capitals	LW	Bruce Boudreau	Capitals	Louis Sutter	Brent Sutter
Ilya Kovalchuk	Devils	RW	Dan Bylsma	Penguins	Louis Sutter	Darryl Sutter
Sergei Gonchar	Penguins	D	Jacques Lemaire	Devils	Gordie Howe	Mark Howe
Evgeni Nabokov	Sharks	G	Todd McLellan	Sharks	Gordie Howe	Marty Howe

In this database the query Q_2 has a satisfying assignment: $S = \text{Brandon Sutter}$, $T = \text{Hurricanes}$, $P = \text{Center}$, $F = \text{Brent Sutter}$, $T' = \text{Flames}$. That is Brandon Sutter, who plays center for the Carolina Hurricanes, is the son of Brent Sutter, who

coaches the Calgary Flames. The query Q_1 , on the other hand, has no satisfying assignment for this database. (Even though Brent Sutter coaches the Flames and Brett Sutter plays center also for the Flames, they are not father and son, but are rather uncle and nephew.)

We call a conjunctive query a *Boolean Conjunctive Query* (BCQ), if the head of the datalog rule contains no variables, just like in the two queries above. The *Boolean Conjunctive Query Evaluation Problem* is then the following decision problem:

Input:	$\langle DB, Q \rangle$, (relational database, boolean conjunctive query)
Question:	Does Q have a satisfying assignment in DB ?

We write $DB \models Q$, if Q has a satisfying assignment in DB . Unfortunately, just like many such important and interesting computational problems, the BCQ evaluation problem is believed to be intractable:

Theorem 2.1. [14] *The BCQ evaluation problem is NP-complete.*

Proof. Reduction from 3-colourability. Let G be a graph. Construct a database consisting of a single table with the schema $edge(Colour, Colour)$. The contents of the table is presented on the right.

Construct the query as follows: For every edge $e = \{X, Y\} \in E(G)$ create the atom $edge(X, Y)$ and take the conjunction over all edges in G . It is easy to see that the query is satisfiable iff the graph is 3-colourable. $\square$

edge	
Colour	Colour
Red	Green
Red	Blue
Green	Red
Green	Blue
Blue	Red
Blue	Green

The BCQ evaluation problem is equivalent to the *Constraint Satisfaction Problem* (CSP). Formally, a CSP is a triple (X, D, C) , where X is a set of *variables*, D is a *domain of values*, and C is a *set of constraints*. Every element of C is a pair (t, R) , where t is a tuple of variables from X , and $R \subseteq D^{|t|}$. An *evaluation* is a function $v : X \rightarrow D$. An evaluation satisfies a constraint $((x_1, \dots, x_n), R)$ iff $(v(x_1), \dots, v(x_n)) \in R$. A *solution* is an evaluation which satisfies **all** constraints. A CSP is *satisfiable* iff it has a solution.

Given a CSP (X, D, C) , we can construct a database, having a table for each constraint (t, R) with the table having a tuple for each element of R . We can then construct a BCQ, having an atom for the tuple t of variables in every constraint, with the same variables. Conversely, given a database D and a BCQ Q , we can construct a CSP by having a constraint for every atom in Q . The variables are the same as in the atom, and the relation consists of all the tuples in the table of D corresponding to the atom. It is obvious that either way $D \models Q$ iff (X, D, C) is satisfiable.

Apart from CSPs, the BCQ evaluation problem is equivalent to a whole range of computational problems, including the following:

- Conjunctive Query Containment
- Conjunctive Query Equivalence
- Query of Tuple Problem
- Homomorphism of Relational Structures Problem
- Clause Subsumption in Theorem Proving

With the BCQ evaluation problem being such an important computational problem, it is very important to find *tractable subclasses*.

2.5 Hypergraph Decompositions

It is straight-forward to associate a hypergraph to a BCQ:

Definition 2.20. Given a BCQ Q with variables V and atoms A , its *underlying hypergraph* is the hypergraph $H(Q)$ where

- $V(H(Q)) = V$
- $E(H(Q)) = \{\text{var}(a) | a \in A\}$

If $H(Q)$ is acyclic we also say that the BCQ Q is acyclic. Unlike BCQs in general the class of acyclic BCQs is tractable:

Theorem 2.2. [28] *The BCQ evaluation problem for the class of acyclic BCQs is LOGCFL-complete.*

The idea of the algorithm is first to evaluate each atom of the query independently and then to take (semi-)joins of the relations along the *join tree*. This operation is cheap and we only need to use it a linear number of times. Moreover, this algorithm can be easily parallelized by “merging” two adjacent atoms into new relations using the join simultaneously in different places, and then proceeding to merge these bigger relations into bigger ones again. The acyclicity of the query guarantees that a single join preserves the validity of each new relation, since new atoms do not “interfere” with previously computed tuples, because all variables are “new”.

Unfortunately, acyclic hypergraphs comprise only a small subclass of all hypergraphs. We would like to be able to identify “nearly acyclic” hypergraphs and exploit this near acyclicity to still be able to get tractable algorithms for the BCQ problem.

A *hypergraph decomposition* thus usually transforms a hypergraph into an acyclic structure (a labelled tree) thus reducing the complexity of the BCQ problem (and with it many equivalent NP-hard problems whose underlying structure can be represented as the corresponding hypergraph). The tractability result relies on the acyclicity of the tree on the one hand and also on certain properties of its labels on the other. An important notion usually linked to the labels of a decomposition is the *width* of the decomposition. For tractability we usually require that the width is independent from the particular hypergraph under consideration.

Thus the two main complexity-theoretic problems considered in the context of hypergraph decompositions are the following:

- **Problem 1.** What is the complexity of recognizing hypergraphs admitting a decomposition of a fixed width? (The *recognition problem*.)

- **Problem 2.** What is the complexity of the BCQ evaluation problem for queries with a decomposition of fixed width? (The *evaluation problem*.)

Arguably the most prominent decomposition method is the *tree decomposition* of [46], originally developed for graphs, but also applicable to hypergraphs. There exist a number of hypergraph decomposition methods in the literature [27, 15, 33, 35, 41]. Some of these methods are *(generalized) hypertree decompositions*, *spread cut decompositions* and *fractional hypertree decompositions*. We will not discuss any of the other methods apart from tree and (generalized) hypertree decompositions. Hypertree decompositions are particularly important since they are the *most general* decompositions (capturing the widest class of hypergraphs) for which both the BCQ evaluation problem as well as the recognition problem is known to be tractable.

Definition 2.21. [22] Given a hypergraph H , a *tree for H* is a tuple (T, χ) , where T is a rooted tree, and $\chi : T \rightarrow \mathcal{P}(V(H))$ is a labelling function associating to each node of the tree a subset of vertices of H . Given $p \in T$ we define $\chi(T_p) = \bigcup \{\chi(q) \mid q \in T_p\}$. The *width* of a tree is $\max_{p \in T} |\chi(p)| - 1$.

In the above definition, remember that T_p denotes the subtree of T rooted at p (Definition 2.4).

Definition 2.22. [29] Given a hypergraph H , a *hypertree for H* is a triple (T, χ, λ) , where (T, χ) is a tree for H and $\lambda : T \rightarrow \mathcal{P}(E(H))$ is a labelling function which associates to each node of T a subset of edges of H . Given $p \in T$ we define $\lambda(T_p) = \bigcup \{\lambda(q) \mid q \in T_p\}$.

The *width* of a hypertree is $\max_{p \in T} |\lambda(p)|$.

2.5.1 Tree Decompositions

Definition 2.23. [46] Let H be a hypergraph. A tree (T, χ) for H is a *tree decomposition for H* iff it satisfies the following conditions:

- For all $e \in E(H)$, there exists $p \in T$, such that $e \subseteq \chi(p)$,
- For all $v \in V(H)$, the subset $\{p \in T \mid v \in \chi(p)\}$ is connected in T (considered as a graph).

The width of a tree decomposition is the width of its tree. The *tree width* of H , $\text{tw}(H)$, is the minimal width over all its tree decompositions.

Consider the hypergraph H_2 presented in Figure 2.2.

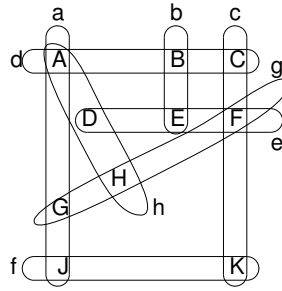


Figure 2.2: Hypergraph H_2

In Figure 2.3 we present a tree decomposition for H_2 of width 5. This is in fact a tree decomposition of minimal width for this particular hypergraph, and we thus have $\text{tw}(H_2) = 5$. We shall adopt the convention of placing the root of the tree at the top of a figure.

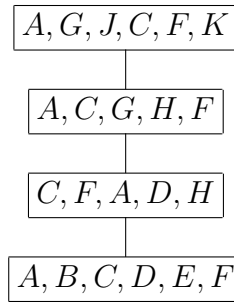


Figure 2.3: Tree Decomposition for H_2

Remark 2.1. In the original definition of tree decompositions there was a further requirement:

- For all $v \in V(H)$, there exists $p \in T$, such that $v \in \chi(p)$.

Since we only consider connected hypergraphs which thus do not have isolated vertices, this condition is subsumed by the edge covering condition.

Strictly speaking the tree of a tree decomposition does not have to be rooted: It is easy to see, that a tree decomposition remains valid if the root of the tree is moved to any other node. Both the BCQ and the recognition problems for the class of queries with a bounded tree width are tractable.

2.5.2 Generalized Hypertree Decompositions

Definition 2.24. [29] A *generalized hypertree decomposition* of a hypergraph H is a hypertree (T, χ, λ) for H satisfying the following conditions:

- (T, χ) is a tree decomposition of H ;
- For each $p \in T$, $\chi(p) \subseteq \bigcup \lambda(p)$.

The width of a (generalized) hypertree decomposition is the width of its hypertree. The *generalized hypertree width* of a hypergraph H , $\text{GHW}(H)$, is the minimal width over all its (generalized) hypertree decompositions.

Thus, every generalized hypertree decomposition is also a tree decomposition. However, the width is measured differently and, the generalized hypertree width of any hypergraph is at most as big as its tree width, but generally much smaller. An extreme example is a hypergraph with a hyperedge containing all vertices. Its generalized hypertree width is 1, whereas its tree width is $n - 1$, where n is the number of vertices in the hypergraph.

In Figure 2.4 we present a Generalized Hypertree Decomposition of the hypergraph H_2 from Figure 2.2 of minimal width 2. Here, we represent the λ - and χ -labels in the following way: For instance, for the child node p of the root, the “combined label”

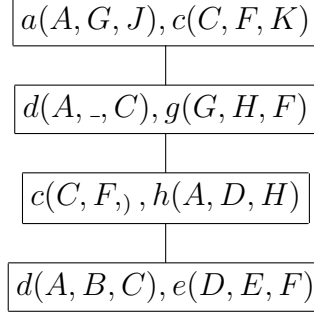


Figure 2.4: Generalized Hypertree Decomposition for H_2

$d(A, -, C), g(G, H, F)$ means that $\lambda(p) = \{d, g\}$ and $\chi(p) = \{A, C, G, H, F\}$, whereas B , which is a vertex in the edge d , is not in the χ -label of p .

We can even use the generalized hypertree width to check a hypergraph’s acyclicity:

Proposition 2.2. [29] *Let H be a hypergraph. H is acyclic iff $\text{GHW}(H) = 1$.*

Just like with tree decompositions, the tree of a GHD does not have to be rooted. Using the acyclicity of the generalized hypertree decomposition and closely following the ideas of [28], we get tractability for the BCQ problem:

Theorem 2.3. *Let k be a fixed integer. The BCQ evaluation problem for the class of queries with a generalized hypertree width at most k is LOGCFL-complete.*

In particular, the algorithm considers each “bag” of atoms as a separate query, evaluates it (in time $O(n^k)$), and finally performs the (semi-)joins along the path of the tree. Unfortunately, the tractability does not apply to the recognition problem even for small widths:

Theorem 2.4. [31] *Recognizing hypergraphs with generalized hypertree width at least 3 is NP-complete.*

2.5.3 Hypertree Decompositions

Definition 2.25. [29] A generalized hypertree decomposition (T, χ, λ) of a hypergraph H is a *hypertree decomposition* iff it satisfies the *special condition*:

- For each $p \in T$, $\bigcup \lambda(p) \cap \chi(T_p) \subseteq \chi(p)$.

The width of a hypertree decomposition is the width of its hypertree. The *hypertree width* of a hypergraph H , $\text{HW}(H)$, is the minimal width over all its hypertree decompositions.

The special condition requires the tree to be rooted. Since every HD is also a GHD, the tractability of the BCQ evaluation problem follows immediately. The special condition now also guarantees tractability of the recognition problem:

Theorem 2.5. [29] *Let k be a fixed positive integer. Recognizing hypergraphs of hypertree width at most k is feasible in LOGCFL.*

It is unknown whether this problem is LOGCFL-complete or whether it lies in a lower complexity class.

Since every hypertree decomposition is also a generalized hypertree decomposition, it is obvious that for any hypergraph H we have $\text{GHW}(H) \leq \text{HW}(H)$. On the other hand, there exist hypergraphs H for which $\text{GHW}(H) < \text{HW}(H)$. For instance, the hypergraph H_2 from Figure 2.2 has hypertree width 3. In Figure 2.5 we present a hypertree decomposition of minimal width.

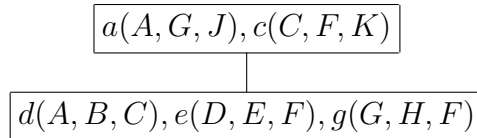


Figure 2.5: Hypertree Decomposition for H_2

Adler et al. [8] analyze the relationships between various hypergraph invariants, showing that they are all related by a constant factor. In particular, for HW and

GHW we have the following relationship:

$$\text{GHW}(H) \leq \text{HW}(H) \leq 3\text{GHW}(H) + 1$$

It is not yet known, whether the second inequality is tight.

2.6 Game-theoretic Characterizations of Decompositions

Given a hypergraph decomposition, to construct an algorithm for the BCQ evaluation problem is usually straightforward. However, to devise an algorithm to find such decompositions is not, particularly since it is NP-hard for several of them.

To better understand and visualize tree decompositions, Seymour and Thomas proposed the *robber and cops game* played on the hypergraph by two players, one moving the robber and the other moving the cops [50]. Both cops and robbers occupy vertices on the hypergraph and move from vertex to vertex along hyperedges, with the cops trying to capture the robber, while the robber tries to escape them. This game turns out to be in 1-1 correspondence with tree decompositions, i.e. $k + 1$ cops have a winning strategy on a hypergraph H iff there is a tree decomposition of width k of H . Moreover, a strategy in this game can be used directly to construct a tree decomposition. Thus, the problem of finding tree decompositions is reduced to finding winning strategies in the robber and cops game.

Analogously to the robber and cops game, Gottlob et al. proposed the *robber and marshals game*, also played by two players on a hypergraph [30]. Marshals are more powerful than cops, accounting for the fact that hypertree decompositions measure their width in terms of the number of hyperedges in each label, rather than the number of vertices; the marshals thus occupy whole hyperedges, rather than a single vertex. To get 1-1 correspondence between hypertree decompositions and strategies in the robber and marshals game, one additional condition is required, however: the

game must be *monotone*, that is at every step the marshals take, the *escape space* of the robber must decrease strictly. For the monotone case, k marshals have a winning strategy iff there exists a hypertree decomposition of width k .

Adler analyzed the non-monotone variant of the robber and marshals game [7] concluding that the non-monotone *marshal width* is smaller or equal than the GHW for any hypergraph.

2.6.1 Robber and Cops

Let H be a hypergraph.

A *position* in the *robber and cops game* is a tuple (C, r) where $C \subseteq V(H)$ is the position of the cops and $r \in V(H)$ is the position of the robber. The game is played by two players: player $\mathcal{R}$ and player $\mathcal{C}$.

The game is played in the following way: First, player $\mathcal{R}$ chooses her starting point, $r \in V(H)$. The *initial position* is then $(\emptyset, r)$.

Given a position (C, r) , a move in the game consists of the following actions:

1. Player $\mathcal{C}$ announces his next position, $D \subseteq V(H)$.
2. Player $\mathcal{R}$ chooses her next position $s \in V(H)$, such that s is $[C \cap D]$ -connected to r . (The robber runs along hyperedges to her new position, but is not allowed to step on a vertex which is still blocked by a cop.)
3. The new position is (D, s) .

Player $\mathcal{C}$ wins once a position (C, r) is reached in which $r \in C$. Player $\mathcal{R}$ wins if she can maintain play forever.

We say that player $\mathcal{C}$ uses k cops iff at any position (C, r) in the game, we have $|C| \leq k$. We then also talk about the robber and k cops game.

Theorem 2.6. [50] *Let H be a hypergraph. H has a tree decomposition of width k iff player $\mathcal{C}$ has a winning strategy in the robber and $k + 1$ cops game.*

Given a position (C, r) , the *escape space* of the robber is the $[C]$ -vertex-component containing r . An important point in designing strategies for player $\mathcal{C}$ is the realization that the exact position of the robber does not matter; the only deciding factor is her escape space. Another important realization is the fact that player $\mathcal{C}$ has a winning strategy in the robber and cops game iff he has a winning strategy in the *monotone robber and cops game*: A robber and cops game is monotone iff for all successive positions (C, r) , (D, s) , we have $S \subset R$, where R is the $[C]$ -vertex-component containing r and S is the $[D]$ -vertex-component containing s , that is, the cops force the capture of the robber in a monotone way, gradually decreasing her escape space.

2.6.2 Robber and Marshals

Let H be a hypergraph.

A *position* in the *robber and marshals game* is a tuple (M, r) where $M \subseteq E(H)$ is the position of the marshals and $r \in V(H)$ is the position of the robber. The game is played by two players: player $\mathcal{R}$ and player $\mathcal{M}$.

The game is played in the following way: First, player $\mathcal{R}$ chooses her starting point, $r \in V(H)$. The *initial position* is then $(\emptyset, r)$.

Given a position (M, r) , a move in the game consists of the following actions:

1. Player $\mathcal{M}$ announces his next position, $N \subseteq E(H)$.
2. Player $\mathcal{R}$ chooses her next position $s \in V(H)$, such that s is $[(\bigcup M) \cap (\bigcup N)]$ -connected to r . (The robber runs along hyperedges to her new position, but is not allowed to step on a vertex which was covered before the marshals' and remains covered after their move.)
3. The new position is (N, s) .

Player $\mathcal{C}$ wins once a position (M, r) is reached in which $r \in \bigcup M$. Player $\mathcal{R}$ wins if she can maintain play forever.

Again, if for all positions (M, r) in the game we have $|M| \leq k$, for some positive integer k , then we speak of the robber and k marshals game.

Given a position (M, r) in the game, the *escape space* of the robber is the $[M]$ -vertex-component containing r . Again, the exact position of the robber does not matter, as far as strategies are concerned — it is enough to know her escape space. The *monotone robber and marshals game* has additionally the following condition:

- For any successive positions (M, r) , (N, s) , we have $S \subset R$, where R is the $[M]$ -vertex-component containing r and S is the $[N]$ -vertex-component containing s .

Theorem 2.7. [30] *Let H be a hypergraph. H has a hypertree decomposition of width k iff player $\mathcal{M}$ has a winning strategy in the monotone robber and k marshals game.*

Chapter 3

Balanced Decompositions: Divide and Conquer

3.1 Introduction

For most hypergraphs, the trees of all valid (generalized) hypertree decompositions of minimal width do not possess any special structure and are often deep (linear in the size of the hypergraph) and narrow (branching factor of one for most nodes). This has negative effects on the parallelization of finding such a decomposition: Parallelization works best if a problem splits into smaller sub-problems of approximately the same size, in the style of many well-known *divide and conquer* type algorithms. For the “computation tree” this means a branching factor of two or more at most internal nodes, a relatively shallow depth, and approximately equal depth of all leaves. In other words, we would like the computation tree to be *balanced*. However, a top-down algorithm usually follows the path of the decomposition tree itself, which in general does not have these properties. By results from [47], the membership of the hypertree computation problem in LOGCFL implies the existence of an algorithm with a shallow computation tree (in theory), which then in turn is more likely to be balanced and thus can be better used for parallelization. In the search for such an algorithm (or at least a good heuristic) we decided to change the rules of the Robber and Marshals game, requiring all strategy trees to be balanced *a priori*.

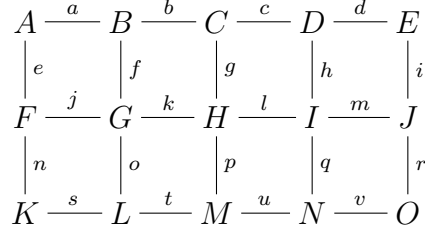


Figure 3.1: Hypergraph H_3 with $\text{GHW}(H) = 3$ and $\text{BW}(H) = 2$

Consider the hypergraph H_3 presented in Figure 3.1, which is in fact a graph. The generalized hypertree width of this hypergraph is 3 and Figure 3.2 presents a generalized hypertree decomposition of it (which is in fact a hypertree decomposition) which corresponds to a strategy in the Robber and Marshals game.

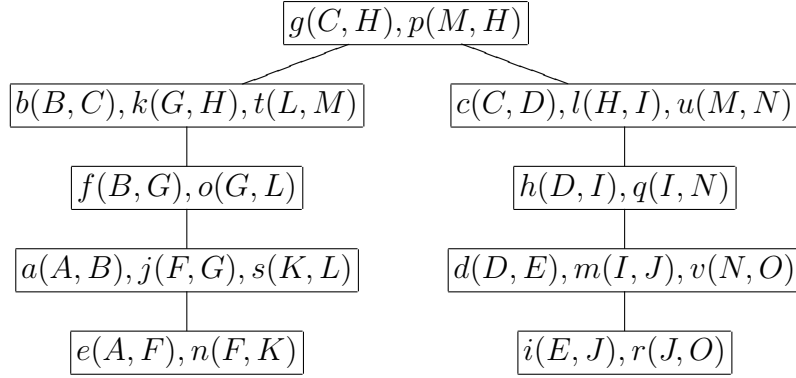


Figure 3.2: Generalized Hypertree Decomposition of H_3

Note that the tree is not balanced: we have just two long chains, the length of each of which is linear in the size of the hypergraph. In fact, this is the shallowest possible tree over all hypertree decompositions.

Now consider the hypertree for H_3 presented in Figure 3.3:

Note that this is not a valid hypertree decomposition, since the “connectedness” condition is violated in quite a few places. We will need to make our “marshals” more powerful, to account for such deficiencies. On the plus side, the hypertree is perfectly balanced: The new marshals *divide* the escape space at *every* step to *conquer* it recursively (to catch the robber). Also note, that we reduced the width from 3 to 2.

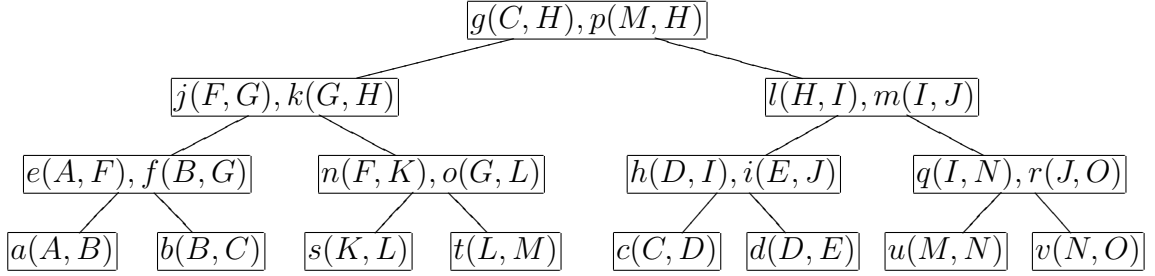


Figure 3.3: Balanced Decomposition of H_3

Even though we originally analyzed it as a heuristic for hypertree decompositions, this new method is a hypergraph decomposition method in its own right, and hence deserves further analysis, for a number of reasons, including the following:

- It possesses properties that are beneficial for parallelization.
- It captures wider classes of hypergraphs than other known decomposition methods (like HD and GHD).
- It can provide more insight into the structure of NP-complete problems.

We call these new decompositions *Balanced Decompositions* (BD) and in this chapter we address the following questions:

- How do we formally define balanced decompositions and balanced width?
- Can these decompositions be characterized game theoretically?
- How do these decompositions relate to other known types of decompositions?
- What is the complexity of recognizing hypergraphs with a bounded balanced width?
- What is the complexity of the BCQ evaluation problem for bounded width *balanced queries* (the class of queries whose underlying hypergraphs have bounded balanced width)?

- How can the balancedness of the trees be used for parallelization of the above tasks?

3.2 Structure of Chapter

In Section 3.3 we concentrate on the decompositions themselves. We propose a formal definition by only slightly changing the definition of generalized hypertree decompositions and imposing restrictions on the shape of the trees. We also compare balanced decompositions to generalized hypertree decompositions and establish several bounds on the various widths of hypergraphs.

In Section 3.4 we provide a game-theoretic characterization, called the *Robber and Sergeants Game* for hypergraphs.

Initial analysis of the complexity of the computational problems exhibited the upper bound $\text{NTiSp}(n^{O(1)}, \log^2 n)$ (the space-bounded subclass of NP only allowing square-logarithmic usage of the worktape [42]) for both problems, and the lower bound LOGCFL for the BCQ evaluation problem. However, the restricted usage of the worktape suggested analyzing our problem on the Nondeterministic Auxiliary Stack Automaton (NauxSA) model of [36].

In Section 3.5 we review the NauxSA computational model and proceed to define a new complexity class hierarchy, called the DC *Hierarchy*, by limiting the resources of a NauxSA. This hierarchy captures a wide range of *divide and conquer* type algorithms that can be more powerful than those on the Auxiliary Pushdown Automaton (AuxPDA) model [16] with similar resource bounds. The classes are separated by their maximal “recursion depth”. In particular we shall be interested in the class DC^1 at the bottom of the hierarchy. We relate this class to several known complexity classes.

In Section 3.6 and Section 3.7 we proceed to analyze the complexity of the BCQ and recognition problems for balanced queries. In particular, we show that that rec-

ognizing hypergraphs of bounded BW is feasible in DC^1 , whereas the BCQ evaluation problem for queries of bounded BW is complete for this class.

In Section 3.8 we discuss how the algorithms from the previous sections can be made deterministic and parallel.

Finally, Section 3.9 reflects on the results of this chapter and outlines directions for future research.

3.3 Balanced Decompositions

Definition 3.1. Let H be a hypergraph. A *hypercut decomposition* of H is a hypertree (T, χ, λ) for H which satisfies the following conditions:

- For each $e \in E(H)$ there exists $p \in T$ such that $e \in \lambda(p)$;
- For each $Y \in V(H)$, write $T^Y = \{p \in T \mid Y \in \chi(p)\}$; We require $\sqcap T^Y \in T^Y$, i.e. that the set T^Y contains its deepest common ancestor (its meet under the ancestor relation $\sqsubseteq_T$ — see also Definition 2.4);
- For each vertex $p \in T$, $\chi(p) = \bigcup \lambda(p)$.

A hypercut decomposition is a *shallow decomposition* iff additionally the following condition holds:

- $\text{depth}(T) \leq \log |E(H)|$.

A hypercut decomposition is a *balanced decomposition* iff additionally the following condition holds:

- For each $p \in T, q \in T(p)$, $|\lambda(T_q)| \leq |\lambda(T_p)|/2$

The *width* of a shallow decomposition or the *width* of a balanced decomposition is $\max_{p \in T} |\lambda(p)|$. The *shallow width*, respectively *balanced width*, of H is the minimum width over all its shallow, respectively balanced, decompositions. We write $SW(H)$ for the shallow width and $BW(H)$ for the balanced width of H .

We have already presented an example of a balanced decomposition of width 2 in Figure 3.3 for the hypergraph H_3 (Figure 3.2).

Remark 3.1. A hypercut decomposition is uniquely defined by T and λ alone. The χ -labels are second-class citizens only required for the second condition.

We did not define a hypercut width, since every hypergraph would then have width one: One can simply construct T as a chain of length $|E(H)|$ where each node is labelled by precisely one edge in $E(H)$. However the balanced and shallow widths will be usually greater than one, and will differ substantially for different hypergraphs.

Remark 3.2. Note that in the definition of shallow decompositions we could have taken $c \log |E(H)|$ as the bound for the depth of T , where $c \geq 1$ is a constant real number not depending on H . Similarly, in the definition of balanced decompositions we could have picked $d|\lambda(T_p)|$ as the bound for the size of $|\lambda(T_q)|$, where $1/2 \leq d < 1$ is a constant real number not depending on H . We could thus associate two values to a shallow decomposition (k and c), and two values to a balanced decomposition (k and d).¹ For complexity-theoretic purposes, however, the exact values of c and d are irrelevant, as long as they are fixed and do not depend on the hypergraphs considered. We thus choose $c = 1$ and $d = 1/2$ for this chapter.

We briefly look at the relationship between balanced decompositions, shallow decompositions and (generalized) hypertree decompositions:

Proposition 3.1. *Let H be a hypergraph. The following holds:*

$$\text{SW}(H) \leq \text{BW}(H) \leq \text{GHW}(H) \leq \text{SW}(H) \log |E(H)|$$

¹ Our suspicion is that for both kinds of decompositions these values strongly depend on each other, e.g. we suspect that k and c are inversely proportional for the same classes of hypergraphs. The analysis of these properties is left for future work.

Proof. (Sketch.) It is easy to see that every balanced decomposition is also shallow, since at every new level, the number of edges in the sub-tree is halved, and hence the tree of the balanced decomposition can be at most of logarithmic depth.

Given a generalized hypertree decomposition we can move any node of it to the root creating a balanced cut and use this as the label of the root of the balanced cut decomposition. For every sub-tree we repeat the same procedure, in turn placing the label of the node creating the balanced cut at the root of the sub-tree. It is easy to see that the resulting hypertree is in fact a balanced cut decomposition.

Finally, we can build a generalized hypertree decomposition from a shallow cut decomposition by using the same tree, but labelling every node with the union of the corresponding label of the shallow cut decomposition and the labels of all its ancestors. It is easy to see that this is a valid generalized hypertree decomposition and that the width is increased by at most a factor of $\log |E(H)|$. $\square$

A tree with n nodes which is shallow (depth $\leq \log n$) will at some point have good branching, however this branching can occur only at very few nodes, as opposed to (perfectly) balanced trees, in which the branching of the nodes is evenly distributed throughout all internal nodes. Hence also the distinction between balanced and shallow decompositions. In particular they emphasize different characteristics of the tree and hence different properties of (parallel) computation. However, as it turns out they also capture slightly different classes of hypergraphs.

For example, let G_3 be the (hyper-)graph presented in Figure 3.4.

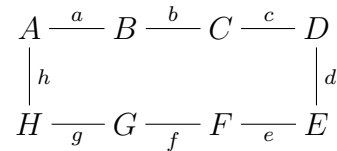


Figure 3.4: Hypergraph G_3 with $\text{BW}(G_3) = 2$ and $\text{SW}(G_3) = 1$

Then there exists a shallow decomposition of G_3 of width 1, presented in Figure 3.5

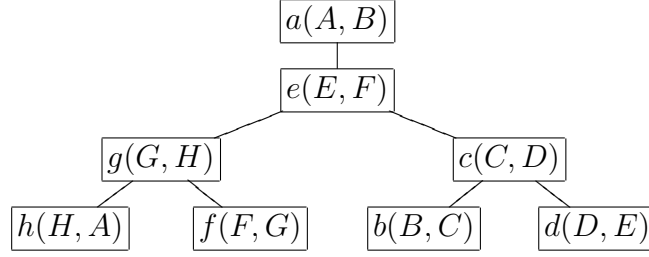


Figure 3.5: Shallow Decomposition of G_3 of width 1

The balanced width of G_3 is 2. The reason for this is, as we shall see later in the Robber and Sergeants game, that the Sergeants' moves (the λ -labels) must separate the escape space into approximately equal-sized components. However, a cyclic graph cannot be separated by the removal of any single edge. In fact, the balanced width of a hypergraph is 1 iff the hypergraph is acyclic, which is not true for the shallow width.

However, for all complexity-theoretic purposes, the distinction between balanced and shallow decompositions is not relevant, since our results apply to both of them.

3.3.1 Normal Forms

There is a notion of a *normal form* for (generalized) hypertree decompositions, which plays a crucial part in many proofs, in particular in the correspondence between hypertree decompositions and strategies in the monotone Robber and Marshals game [30]. Similarly, we can define a normal form for hypercut decompositions.

Definition 3.2. A hypercut decomposition (T, χ, λ) for H is in *normal form* if for every $e \in E(H)$ there is at most one $p \in T$, such that $e \in \lambda(p)$.

Lemma 3.1. *Let H be a hypergraph. There exists a k -width shallow/balanced decomposition of H iff there exists a k -width shallow/balanced decomposition of H in normal form.*

Proof. The “if” part is obvious.

For the “only if” part, suppose (T, χ, λ) is a shallow/balanced decomposition which

is not in normal form. Then there exist an edge $e \in E(H)$ and nodes $q, r \in T$ such that $e \in \lambda(q)$, $e \in \lambda(r)$ and $q \neq r$. By the first condition of hypercut decompositions, there exists a node $p \in T$ such that $e \subseteq \chi(p)$. Without loss of generality we can assume that p is the shallowest such node. Then $p \sqsubseteq_T q$, for $\chi(p) \cap \chi(q) \neq \emptyset$ and by the second condition one of the two has to be an ancestor of the other, but p is shallowest amongst all nodes v with $e \subseteq \chi(v)$. Similarly $p \sqsubseteq_T r$. If one of q, r is equal to p , let us say $q = p$, we simply remove e from $\lambda(r)$ and update $\chi(r)$ to be the union of the other edges left in $\lambda(r)$. If both q and r are descendants of p , then do this procedure for both these nodes. If any of the λ -labels (and thus χ -labels) becomes empty, just delete the node and attach all its children as children to that node's parent. It is easy to see that all conditions of hypercut decomposition are still satisfied, also, T does not increase in depth, and $\lambda(T_q)$ and $\lambda(T_r)$ do not increase in size, thus also preserving the conditions for shallow/balanced decompositions. We can repeat this procedure a finite number of times, each time removing at least one edge from the λ -labels (counting multiples), until the decomposition is eventually in normal form. $\square$

From now on we always assume that balanced/shallow decompositions are in normal form.

3.4 Robber and Sergeants

As with many other decomposition methods for hypergraphs it helps to visualize a decomposition in terms of a two player game between a Robber and some Law Enforcement Entity. We shall define the *Robber & k Sergeants Game* on a hypergraph H ($R \& S^k$). It is quite similar to the Robber and Marshals game with two important differences:

- The robber is positioned on edges rather than vertices (an escape space hence becomes an edge-component rather than a vertex-component).

- The sergeants only have to cover any edge once, since it remains covered for the rest of the game (the robber can never go to that edge again).

Hence the game is by definition monotone (the escape space can never increase).

Let H be a hypergraph, let k be a positive integer, and let $A \subseteq E(H)$ such that A is connected, be the *initial escape space*. The Robber and k Sergeants Game from A ($\text{R\&S}^k(A)$) is played by two players - $\mathcal{R}$ (the robber) and $\mathcal{S}$ (the sergeants). Player $\mathcal{S}$ announces moves by choosing a set S of up to k edges of A . If S covers the whole of A , player $\mathcal{S}$ wins. Otherwise, player $\mathcal{R}$ chooses an $[S]$ -connected component of A , say B . They then proceed to play the game $\text{R\&S}^k(B)$.

If the game is shallow, then player $\mathcal{R}$ wins $\text{R\&S}^k(A)$ if he can sustain play for more than $\log |A|$ moves. If the game is balanced, then player $\mathcal{R}$ wins if from any escape space A and a sergeants' move S he can select an $[S]$ -component B of A such that $|B| > |A|/2$.

The game $\text{R\&S}^k(H)$ is the game $\text{R\&S}^k(E(H))$ (on the full hypergraph).

Player $\mathcal{S}$ has a winning strategy, if for any possible move of player $\mathcal{R}$, he can still win the game.

We can now formally define winning strategies:

Definition 3.3. Let k be a positive integer, let H be a connected hypergraph. A *winning strategy* for $\text{R\&S}^k(H)$ is a tuple (T, ρ, λ) , where T is a rooted tree and $\rho, \lambda : T \rightarrow \mathcal{P}(A)$ are labelling functions (escape space and sergeants' moves, respectively) such that the following conditions hold:

- *Initial Condition:* $\rho(O(T)) = E(H)$.
- *Boundedness:* For all $t \in T$, $1 \leq |\lambda(t)| \leq k$.
- *Completeness:* For all $s \in T$, $\rho(s) = \mu(s) \cup \bigcup_{t \in T(s)} \rho(t)$.
- *Separation:* For all $s \in T, t \neq u \in T(s)$, $\rho(t) \cap \rho(u) = \emptyset$.

- *Connectedness*: For all $s \in T, t \in T(s), e \in \rho(s), f \in \rho(t)$, e is $[\lambda(s)]$ -connected to f in $\rho(s)$ iff $e \in \rho(t)$.

A winning strategy in the shallow $R\&S^k$ game additionally satisfies

- $\text{depth}(T) \leq \log |E(H)|$

A winning strategy in the balanced $R\&S^k$ game additionally satisfies

- For all $s \in T, t \in T(s)$, $|\rho(t)| \leq |\rho(s)|/2$.

The separation and connectedness conditions say that escape space labels of the children of a node s are distinct $[\lambda(s)]$ -components of $\rho(s)$. The completeness condition says that we include all such components, and also that for each node s we have $\lambda(s) \subseteq \rho(s)$.

Proposition 3.2. *Let H be a hypergraph and let (T, ρ, λ) be a winning strategy in the game $R\&S^k(H)$. For all $s \neq t \in T$ such that neither is a descendant of the other, and for all $e \in \rho(s), f \in \rho(t)$, there exists $r \in T$ such that $r \sqsubseteq_T s$, $r \sqsubseteq_T t$ and $e \cap f \subseteq \bigcup \lambda(r)$.*

Proof. If $e \cap f = \emptyset$ we can just pick any common ancestor as r . Otherwise, let r be the shallowest common ancestor of s and t , and let $s', t' \in T(r)$ such that $s' \sqsubseteq s$ and $t' \sqsubseteq t$. Then $e, f \in \rho(r)$, $e \in \rho(s')$ and $f \in \rho(t')$. By the connectedness and separation conditions, e and f must be in different $[\lambda(r)]$ -components of $\rho(r)$, so e is not $[\lambda(r)]$ -connected to f in $\rho(r)$. But if there was some vertex $v \in e \cap f \setminus \bigcup \lambda(r)$ then they would be $[\lambda(r)]$ -connected in $\rho(r)$. Hence we must have $e \cap f \subseteq \bigcup \lambda(r)$. $\square$

Lemma 3.2. *Let H be a hypergraph. There exists a k -width shallow/balanced decomposition of H iff there exists a winning strategy in the shallow/balanced $R\&S^k$ game on H .*

Proof. We first show that a k -width hypercut decomposition of H (in normal form) exactly corresponds to the game $\text{R\&S}^k(H)$. Let (T, ρ, λ) be a winning strategy in the $\text{R\&S}^k(H)$ game. Define $\chi : T \rightarrow \mathcal{P}(E(H))$ by $\chi(p) = \bigcup \lambda(p)$. Then (T, χ, λ) is a hypercut decomposition. By the completeness and initial conditions of the game, it is easy to see that for every $e \in E(H)$ there is some $p \in T$ such that $e \in \lambda(p)$. The third condition is trivially satisfied by our choice of χ . For the second condition, choose a vertex Y . Suppose there are $q \neq r \in T$ such that neither is a descendant of the other and $Y \in \chi(q)$ and $Y \in \chi(r)$. Then there are edges $f \neq g$ such that $f \in \lambda(q)$, $g \in \lambda(r)$, and $Y \in f$ and $Y \in g$. By the previous proposition there exists $p \in T$ such that $e \cap f \subseteq \bigcup \lambda(p)$, so in particular $Y \in \bigcup \lambda(p) = \chi(p)$.

Conversely, let (T, χ, λ) be a hypercut decomposition. Define $\rho : T \rightarrow \mathcal{P}(E(H))$ by $\rho(p) = \lambda(T_p)$. The initial and boundedness conditions are obviously satisfied. Completeness and separation are easily seen by the definition of ρ and the normal form of the decomposition. For connectedness, we can work inductively from the top down. $\rho(O(T))$ is connected by definition. Now let $s \in T$, suppose $\rho(s)$ is connected and let $e, f \in \rho(s)$. Every edge on any path from e to f uniquely corresponds to a descendant node of s in T . By starting from the node corresponding to e and applying the shallowest common ancestor operation for every node corresponding to every edge on the path to f , we eventually get a node t . If this t is a proper descendant of s , then the path did not pass through any vertex covered by $\lambda(s)$, and indeed e and f will both be in $\rho(t)$ and in $\rho(t')$, where t' is an ancestor of t and a child of s , and they are in the same $[\lambda(s)]$ -component of $\rho(s)$. If for all paths from e to f the node t is always equal to s , this means that every path passes through a node covered by $\lambda(s)$. Hence e and f are in different $[\lambda(s)]$ -components of $\rho(s)$, and since we always took the shallowest common ancestor, e and f must have been in the labels of different sub-trees of s to start off with (otherwise we would have found a shallowest common ancestor of the path which is not s), and will be in different escape spaces.

The equivalence of the balancedness and shallowness conditions for the game and the decomposition is obvious. $\square$

As with the monotone Robber and Marshals Game for hypertree decompositions, the Robber and Sergeants game will turn out to be a useful tool in the computation of hypercut decompositions.

3.5 Another look at Stack Automata

Rather than presenting algorithms for the problems outlined in the introduction straight away, we will provide the necessary theoretical background to analyze these problems in as much detail as possible. To this purpose we first look at a certain computational model which generalizes Turing Machines and allows a better complexity analysis. In the later sections of the paper we will perform such an analysis of our problems on this model, in particular also giving (nondeterministic) algorithms. In the conclusion we shall briefly discuss how such algorithms can be implemented deterministically and on parallel machines.

We can limit the resources available to an AuxSA, in particular the running time, the space and the maximal stack height. In particular for nondeterministic AuxSAs (NauxSAs) we write $\text{NTiSpSh}(T(n), S(n), H(n))$ for the class of problems solvable by a NauxSA which is simultaneously bounded by time $O(T(n))$, space $O(S(n))$ and maximal stack height $O(H(n))$, where $T(n)$, $S(n)$ and $H(n)$ are constructible functions. When restricting the height of the stack, there is no point of having the stack smaller than or equal to the worktape, as it does not give any additional computational power (anything that can be done with a stack). So the definition only makes sense if $H(n)$ is asymptotically strictly larger than $S(n)$.

Proposition 3.3. *For constructible functions $T(n)$, $S(n)$ and $H(n)$*

$$\begin{aligned} & \text{NTiSpPh}(T(n), S(n), H(n)) \\ & \subseteq \text{NTiSpSh}(T(n), S(n), H(n)) \\ & \subseteq \text{NTiSp}(T(n), H(n)) \end{aligned}$$

Proof. A pushdown can obviously be simulated by a stack, just as a stack can be simulated by a worktape. $\square$

AuxSAs allow recursive algorithms, just like AuxPDAs, but these algorithms additionally have access to all previously computed temporary variables (the accumulated temporary variables) and are thus more powerful.

3.5.1 The DC Hierarchy

We now use AuxSAs to define the following hierarchy of complexity classes:

Definition 3.4. For all non-negative integers k define:

$$\text{DC}^k = \text{NTiSpSh}(n^{O(1)}, \log n, \log^{k+1} n)$$

and

$$\text{DC} = \bigcup_{k \geq 0} \text{DC}^k$$

The name suggests on the one hand the way an algorithm in such a particular class might work (Divide and Conquer through recursion), and on the other hand the maximal depth of recursive calls ($O(\log^k n)$ for DC^k , each time storing $O(\log n)$ cells on the stack). A single function call then is an NL algorithm which additionally can access previously computed temporary variables. This is not unlike the Guess-and-Check model of [12] in which the stack is used as a nondeterministically guessed “advice string”. In particular, we have $\text{GC}(\log^{k+1} n, \text{NL}) \subseteq \text{DC}^k$, where the former class is the class of languages for which an advice string of length $O(\log^{k+1} n)$ can be guessed such that the original input plus the advice string can be decided in NL.

In this chapter we will only consider the class DC^1 , allowing a logarithmic number of recursive calls. In particular, $\log n$ recursive calls exactly allow at each call to divide an input of length n into at least two parts each at most half as big until the parts have constant size.

Corollary 3.1.

$$\text{LOGCFL} \subseteq DC^1 \subseteq \text{NTiSp}(n^{O(1)}, \log^2 n)$$

Figure 3.6 presents the position of DC^1 amongst the other complexity classes presented in Section 2.3.

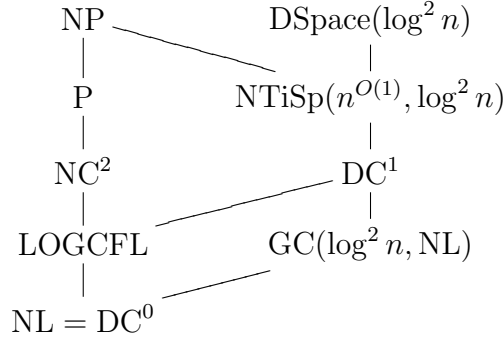


Figure 3.6: DC^1 in the Complexity Hierarchy

3.5.2 Regular Stack Automata

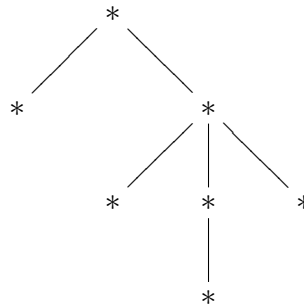
To simplify the reasoning about AuxSAs, we can assume that the stack is always written to, read from and popped in *chunks* proportional to the size of the worktape (usually of size $O(\log n)$). This can be realized by adding a symbol to the alphabet which acts as a delimiter between the chunks. It does not make sense to read more contents of the stack at any one time, as the information would not fit onto the worktape. We can assume, that the machine has an extra tape just for reading the stack, and whenever a chunk is read, it is copied to that tape. Also, as long as the number of chunks on the stack does not exceed $O(2^{S(n)})$, where $S(n)$ is the space available on the worktapes, we can simulate random access to any chunk: We

introduce an extra tape to which we write the index (number) of the chunk we would like to read. Then we scan the stack from the bottom (the left-end marker) decreasing the index by one every time we encounter a delimiter, until the index reaches zero. Then we actually copy the next chunk to the special tape. By introducing one more tape we can also keep track of the number of chunks on the stack at any one time. All these simulations can be performed with only a polynomial overhead in time and no overhead in space (for space bounds in $O(\log n)$).

For any AuxSA, AuxPDA, PDA or SA we define the *push-pop* tree to be an ordered tree which is constructed in the following manner:

1. Create the root. Set x to the root.
2. If machine pops, set x to its parent.
3. If machine pushes, create a new child of x which becomes the youngest of its siblings, if it has any, and set x to this new child.
4. Repeat steps 2-3 until machine halts.

For instance, if the push-pop sequence for some particular machine is *Push, Pop, Push, Push, Pop, Push, Push, Pop, Pop, Push, Pop, Pop*, then its push-pop tree looks like this:



Definition 3.5. We call a NauxSA M *regular* if it has the following properties:

- The push-pop tree of M is a full binary tree ($2^k - 1$ nodes, for some integer k)

- After every push the entire contents of the stack is read.
- Whenever M does not push, it acts deterministically (the only nondeterministic steps are the pushes).

In the remainder of this section we will show how to transform any NauxSA into a regular NauxSA. First, we will look at the push-pop tree.

Definition 3.6. An *ordered labelled tree with dummies* is an ordered labelled tree in which some nodes can be labelled with the special DUMMY symbol $*$.

We say an ordered labelled tree T *embeds* into an ordered labelled tree with dummies U iff there exists an injection $e : T \rightarrow U$ such that the following conditions hold:

- All labels of $U \setminus e(T)$ are dummies.
- For each $u \in T$, the label of $(e(u))$ is the same as the label of u .
- For all $u, v \in T$, if $u \leq_T v$, then $e(u) \leq_U e(v)$.

In other words, we introduce some dummy nodes, such that the tree traversal order of the new tree preserves the tree traversal order of the original labels. We will now show how to simulate a shallow push-pop tree using a binary push-pop tree with dummies at most a constant times deeper.

Proposition 3.4. *Let k be a non-negative integer. Let $\{a_i\}_{i=1}^m$ be a sequence of integers, such that for all i , $1 \leq a_i \leq 2^k$ and $2^k < S = \sum_{i=1}^m a_i < 2^{k+1}$. Then precisely one of the following holds:*

1. *The sequence has precisely one element of size 2^k and can be split into three subsequences, one possibly empty, such that exactly one of them has sum 2^k and the other two each have sum less than 2^k .*
2. *The sequence can be split into three subsequences, one possibly empty, such that each subsequence has sum less than 2^k .*

Proof. Consider such a sequence. Call the subsequences $\{a_i\}_{i=1}^{i_1}$, $\{a_i\}_{i=i_1+1}^{i_2}$, $\{a_i\}_{i=i_2+1}^m$ and their respective sums s_1 , s_2 and s_3 .

Suppose there is exactly one element of size 2^k . If $a_1 = 2^k$ or $a_m = 2^k$, then setting the indices $i_1 = 1$ and $i_2 = m - 1$ is enough. If the large element is somewhere in the middle, then set i_1 such that $s_1 < 2^k$, but $s_1 + a_{i_1+1} > 2^k$ and $i_2 = i_1 + 1$. It is easy to see that a_{i_2} then is the large element, and hence $s_3 < 2^k$.

Now suppose there is no element of size 2^k . Set i_1 such that $s_1 < 2^k$, but $s_1 + a_{i_1+1} \geq 2^k$ and set $i_2 = i_1 + 1$. Then $s_2 < 2^k$ because $i_2 < 2^k$ and $s_3 < 2^k$ because $s_1 + s_2 \geq 2^k$. $\square$

Proposition 3.5. *Let K be a non-negative integer. Any ordered labelled tree T of size at most 2^K and depth at most $K + x$, where x is an integer such that $0 \leq x \leq 2^K - K - 1$, can be embedded into a binary ordered labelled tree with dummies of depth at most $4K + 2x$.*

Proof. By induction on K .

Base Case 0. For $K = 0$, $\text{size}(T) \leq 1$ so $0 \leq x \leq 2^0 - 0 - 1 = 0$, so $x = 0$ and $\text{depth}(T) = 0$, and T is already a binary tree of depth at most $0 \leq 4k + 2x$.

Base Case 1. For $K = 1$, $\text{size}(T) \leq 2$ so $0 \leq x \leq 2^1 - 1 - 1 = 0$, so $x = 0$ and $\text{depth}(T) = 1$, and T is already a binary tree of depth at most $1 \leq 4k + 2x$.

Inductive Hypothesis. Suppose that for every $k \leq K$, for every tree T of size at most 2^k and depth at most $k + x$, where $0 \leq x \leq 2^k - k - 1$, there is an equivalent binary tree $B(T)$ of depth at most $4k + 2x$.

Inductive Step. Consider an ordered tree T , such that $2^K < \text{size}(T) \leq 2^{K+1}$ and $\text{depth}(T) \leq K + 1 + X$, where $0 \leq X \leq 2^{K+1} - K - 2$. We need to show that there is an equivalent binary tree $B(T)$ of depth at most $4(K + 1) + 2X$.

T looks like this:

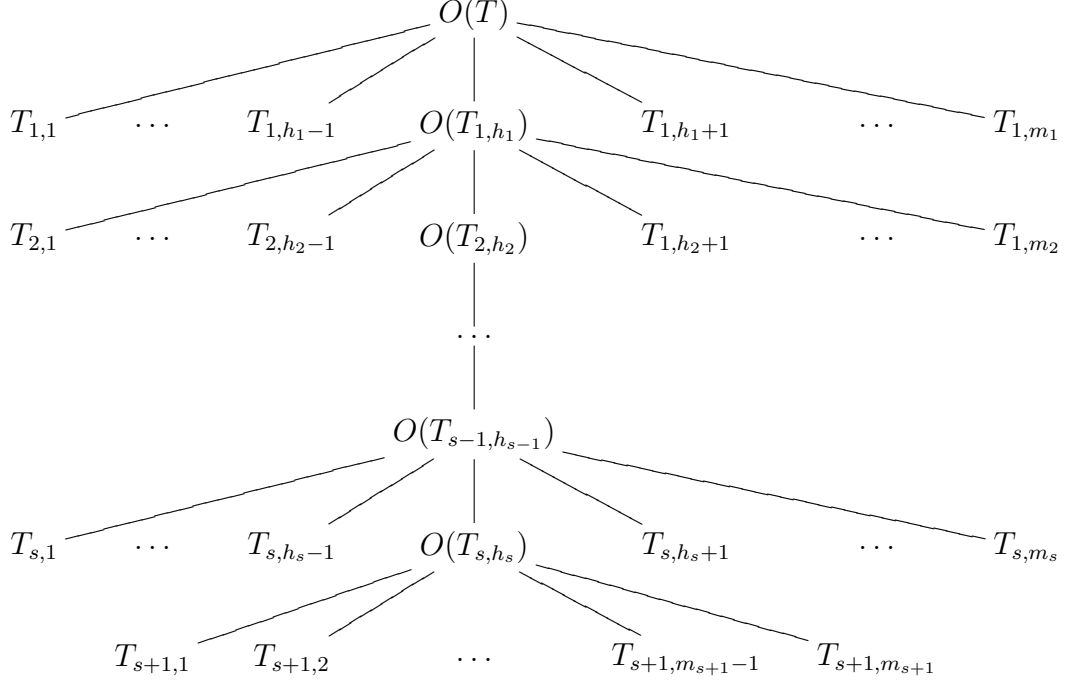


for some m_1 , $1 \leq m_1 \leq 2^{K+1} - 1$, where $O(T)$ is the root node of T and each $T_{1,i}$ is the sub-tree rooted at the i th child of $O(T)$. We have $\sum_i^{m_1} \text{size}(T_{1,i}) \leq 2^{K+1} - 1$ (since the root node counts for one), and $\text{depth}(T_{1,i}) \leq K + X$ for all i . Let h_1 be such that $\text{size}(T_{1,h_1}) \geq \text{size}(T_{1,i})$ for all $i \neq h_1$. We call T_{1,h_1} a heavy sub-tree of T (there might be several). T_{1,h_1} might have a heavy sub-tree of its own, T_{2,h_2} , which might have a heavy sub-tree of its own, etc. For convenience let us call $T = T_{0,h_0}$, where $h_0 = 1$, and refer to T_{0,h_0} as a heavy sub-tree, also.

It is easy to see, that there always exists an integer s ($0 \leq s \leq \min(2^K - 1, K + X)$) such that $\text{size}(T_{s,h_s}) > 2^K$ and $\text{size}(T_{s+1,h_{s+1}}) \leq 2^K$, i.e. T_{s,h_s} is the shallowest heavy sub-tree containing more than half the nodes of T . Also it is easy to see that such a sub-tree is unique.

Now, for each t , $1 \leq t \leq s$, let m_t be the number of children of $T_{t-1,h_{t-1}}$, and let's call the sub-tree of $T_{t-1,h_{t-1}}$ rooted at its i th child $T_{t,i}$.

We have the following picture:



For each t and $i \neq h_t$ we have $\text{size}(T_{t,i}) \leq 2^K - 1 - t$. The reason for this is that at least $2^K + 1$ nodes are taken up by T_{s,h_s} , and at least t nodes are taken up by $O(T_{u,h_u})$ for $0 \leq u < t$.)

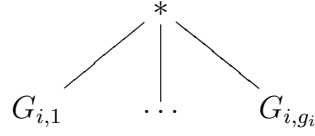
For each t let $X_t \leq \min(2^K - K - 1 - t, X - t)$. Then for $i \neq h_t$, we have $\text{depth}(T_{t,i}) \leq K + X_t$. (We limit the depth either by size of $T_{t,i}$ or by X but subtracting the t previous levels, whichever is smaller.)

For each t , $1 \leq t \leq s$ we create trees L_t and R_t by putting a dummy node at the root and attaching the sub-trees like so:



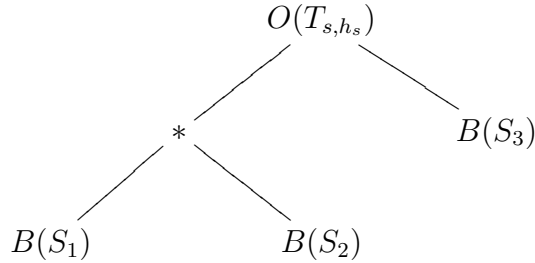
It is obvious that $\text{size}(L_t) \leq 2^K$ and $\text{depth}(L_t) \leq K + X_t + 1$. So the inductive hypothesis applies. Hence there are equivalent binary trees $B(L_t)$ of depth at most $4K + 2X_t + 2$. The same argument gives us binary trees $B(R_t)$ of the same maximal depth.

Now consider T_{s,h_s} . We have $2^k < \text{size}(T_{s,h_s}) \leq 2^{k+1}$ and $\text{depth}(T_{s,h_s}) \leq K + 1 + X - s$. Since no sub-tree of T_{s,h_s} exceeds 2^k in size, by the previous lemma we can group them into three sequences according to their sizes: G_1, G_2 and G_3 (from left to right, one might be empty). Now we create three trees S_1, S_2, S_3 like so: If G_i is a singleton, then S_i is the element of G_i . Otherwise, S_i has a dummy as its root, and all the elements of G_i as its children in sequence:



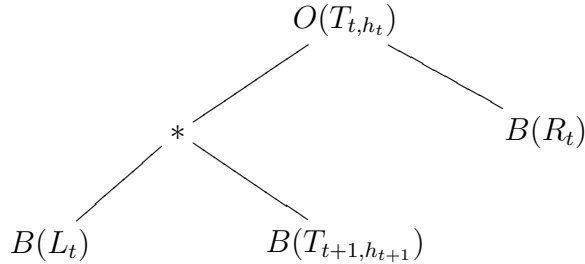
For each S_i we have $\text{size}(S_i) \leq 2^K$ and $\text{depth}(S_i) \leq K + 1 + X - s$, so the inductive hypothesis applies. We get equivalent binary trees $B(S_1), B(S_2)$ and $B(S_3)$ of depth at most $4K + 2 + 2X - 2s$.

Define the “binarization” $B(T_{s,h_s})$ as follows:

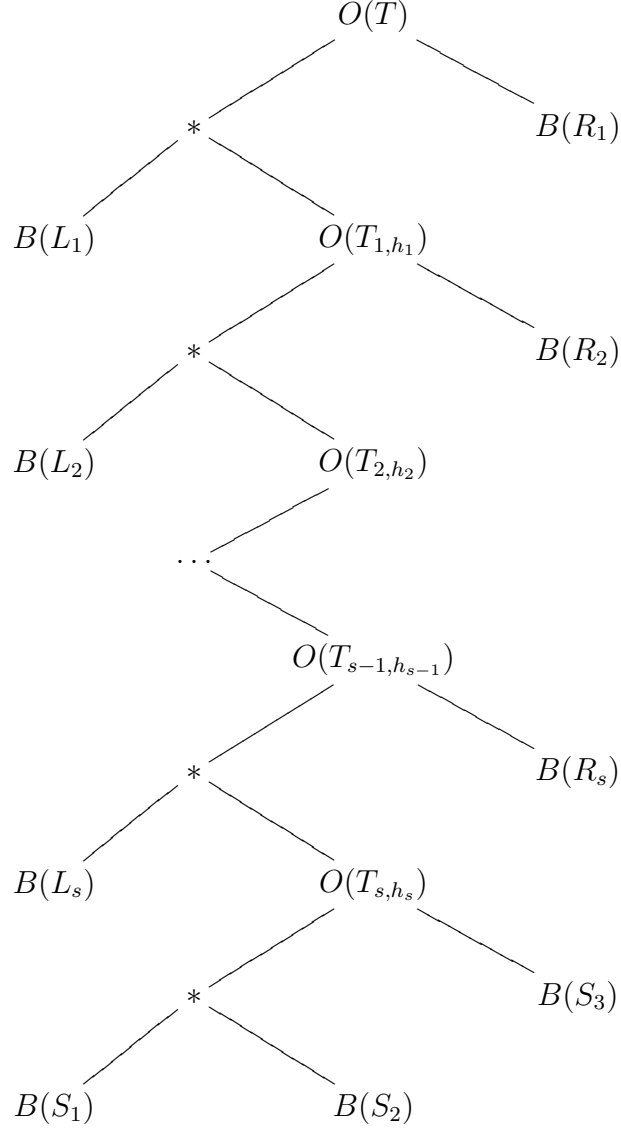


The maximal depth of $B(T_{s,h_s})$ is $2 + 4K + 2 + 2X - 2s = 4K + 4 + 2X - 2s$.

Now, for each $t < s$ we define $B(T_{t,h_t})$ like so:



so our final binary tree $B(T)$ looks like this:



where each of $B(L_t)$, $B(R_t)$ and $B(S_1)$, $B(S_2)$ and $B(S_3)$ is a binary tree.

Now we consider the maximal depth of any node in $B(L_t)$. For this matter we will need to consider X_t for $1 \leq t \leq s$, where $X_t = \min(X - t, 2^K - K - 1 - t)$, and $s \leq \min(2^K - 1, K + X)$.

If $X \geq 2^K - K$, then $X_t = 2^K - K - 1 - t$. The maximal depth of a node in $B(L_t)$ is then $2t + 4K + 2X_t + 2 = 2t + 4K + 2(2^K - K - 1 - t) + 2 = 4K + 2(2^K - K) \leq 4(K + 1) + 2X$ as required.

If $X < 2^K - K$, then $X_t = X - t$. The maximal depth of a node in $B(L_t)$ is then

$2t + 4K + 2X_t + 2 = 2t + 4K + 2X - 2t + 2 = 4K + 2 + 2X \leq 4(K + 1) + 2X$ as required.

A similar argument applies to nodes in $B(R_t)$.

Now consider the maximal depth of any node in $B(T_{s,h_s})$. It is $2s + 4K + 4 + 2X - 2s = 4K + 4 + 2X \leq 4(K + 1) + 2X$ as required.

Hence $\text{depth}(B(T)) \leq 4(K + 1) + 2X$ as required. By the construction of $B(T)$ it is also obvious that T embeds into it. $\square$

Remark 3.3. A lot of the lists of sub-trees presented here could be empty. But the proof still works by simply attaching a dummy node for an empty list.

Remark 3.4. This is probably not the optimal binarization. We suspect that the minimal depth of a binarization is something like $3k - 2 + x$ (we haven't examined any trees that require a larger bound, yet). As an intuition: the 0th level always only adds 1 layer overall (hence the base case(s) could be changed and hence the -2). Also, long chains are already binary trees (hence only x) and don't need two for one layers each as in above proof. To get an optimal arrangement one would have to consider the exact structure of every sub-tree, rather than just its size and depth.

Lemma 3.3. *An ordered tree of size n and depth at most $\log n$ embeds into a binary ordered tree with dummies of depth at most $4 \log n$.*

Proof. Immediately follows by above proposition. $\square$

We now have the means to transform a NauxSA into a regular NauxSA:

Theorem 3.1. *Let M be a NauxSA with stack size $O(\log^2 n)$, tape size $O(\log n)$ running in polynomial time and deciding the language L . Then there exists a regular NauxSA M' with same stack and tape sizes running in polynomial time deciding L . Moreover, there exists a log-space reduction from M to M' .*

Proof. (Sketch.) First, we can assume that M only acts nondeterministically when it pushes. For every nondeterministic choice, one could push a nondeterministic bit on the stack, read it and pop it, and make the corresponding choice deterministically based on the value of that bit. Since the total running time is polynomial, this only gives polynomially many extra pushes on the stack, and does not need any more stack depth.

Construct M' as follows: We add the dummy symbol to M' 's alphabet. M' behaves exactly as M does when it does not access the stack. Whenever M would push, M' can either push whatever M would push, or it can perform any number of pushes and pops of dummy's according to its regular behaviour, and then push whatever M would push, and then switch to the state M would switch to. Whenever M' reads, it ignores the dummy's and only cares about what M would care about. Whenever M pops, M' performs the necessary amount of dummy pushes/pops before performing the pop that corresponds to M 's pop. It then enters the same state as M would. If M' 's stack is full and M would push, then M' rejects. M' can keep track of the regularity of its cell pushes and pops with a binary string of logarithmic length.

Since M runs in polynomial time, there will be at most polynomially many pushes, say p . Let T be the push/pop tree of M . Let $k = \max\{\text{depth}(T), \log p\}$, so $\text{size}(T) \leq 2^k$. By Lemma 3.3, there exists an equivalent binary tree with dummies of depth at most $4k$. Hence there exists a valid binary push/pop tree $B(T)$ with dummies for M' of polynomial size. By the construction of M' it is obvious that M' accepts x iff M accepts x . It is also obvious that this reduction can be done in log-space. $\square$

3.6 Evaluating Balanced Queries

We will now consider the complexity of the computational problem of evaluating balanced queries.

3.6.1 Membership in DC¹

Consider the algorithm k -hd-bcq presented in Algorithm 1.

Algorithm 1 k -hd-bcq

```

1: fixed parameter  $k$ : Integer
2: input Database  $d$ 
3: input Query  $q$  with its  $k$ -width hypercut decomposition
4: satisfiable (root ( $q$ ))
5: accept
6:
7: procedure satisfiable( $u$ : Node) {
8:   for all (Atom  $a \in \lambda(u)$ ) {                                 $\triangleright$  at most  $k$  repetitions
9:     guess Tuple  $t \in \text{table}(d, \text{relname}(a))$ 
10:    for all ((Atom, Tuple) pair  $(b, s)$  on the stack) {
11:      if (not compatible( $(a, t), (b, s)$ )) {
12:        reject
13:      }
14:    }
15:    push ( $a, t$ )
16:  }
17:  for all (Node  $v \in \text{children}(u)$ ) {
18:    satisfiable( $v$ )
19:  }
20:  pop all  $(a, t)$  pairs which were pushed during current function call
21: }
```

We assume that the tree of the hypercut decomposition is an ordered tree, and that we can access its root using the function *root*, and that given a node u , we can iterate through *children*(u) using a single pointer. Given an atom a the function *relname*(a) returns the “schema” s of that atom, and we can use *table*(d, s) to access the appropriate table in d . When we call *satisfiable* recursively, we assume that the current temporary variables (in this case only u and v) are placed on the stack, and popped after the recursive call returns. We push and pop the atom a and tuple t explicitly at every iteration within one call, because we need them on the stack before *satisfiable* is called recursively. The function *compatible*($(a, t), (b, u)$) checks whether tuples t and u are compatible under the schemas of a and b , respectively. I.e., it

returns **false** iff there is a shared variable between a and b , such that that variable's values in t and u differ.

Theorem 3.2. *Fix a positive integer k . Given a database D , a boolean conjunctive query Q with associated hypergraph H and a hypercut decomposition (T, χ, λ) of H of width at most k , the algorithm k -hd-bcq accepts iff $D \models Q$. Moreover, the algorithm operates in $\text{NTiSpSh}(n^{O(1)}, \log n, d \log n)$, where n is the size of the whole input and $d = \text{depth}(T)$.*

Proof. Note that checking whether two tuples are compatible (have no contradicting assignments to their variables) in D can be done deterministically in space $O(\log |D|)$.

The algorithm only requires the pointers u and v (pointers to nodes in the decomposition of Q), a (a pointer to an atom in Q) and t (a pointer to a tuple in D). The space required to store each pointer is $\log n$. Hence space $O(\log n)$ is enough. Also for every call of the function at most $2k + 2$ pointers will be added to the stack taking up space $2(k + 1) \log n = O(\log n)$. The depth of the call stack is equal to the depth of the decomposition tree and hence the stack needs space $O(\text{depth}(T) \log n)$.

It is easy to see that the algorithm is correct: First note that the algorithm performs a tree traversal of T in its appropriate order. For every atom a of Q , precisely one tuple t in D is guessed. Let (a, t) and (b, s) be such atom-tuple pairs, and let $p, q \in T$ such that $a \in \lambda(p)$ and $b \in \lambda(q)$. If a and b share a variable Y , then by definition of a hypercut decomposition there exists $r \in T$ such that $Y \in \chi(r)$ and $p, q \in T_r$. Also there will be an atom $c \in \lambda(r)$ such that $Y \in c$. But then the tuple for c will have been guessed before both t and s , and will be sitting on the stack, once the algorithm gets to b and a . Hence t and s will be both required to have the same assignment to Y as the tuple of c . If Q cannot be satisfied, then for any assignment of tuples to atoms, there will be at least one inconsistent variable Y , and the algorithm will eventually expose it and reject. Since this is a non-deterministic algorithm, it will find a satisfying assignment and accept iff one exists and will reject otherwise. $\square$

Corollary 3.2. *For queries of fixed (bounded) shallow or balanced width the BCQ evaluation problem is in DC^1 .*

3.6.2 DC^1 -Completeness

By Theorem 3.1, we can assume that we can always first transform a NauxSA into a regular NauxSA. We now construct a reduction from the description of any regular NauxSA in $NTiSpSh(n^{O(1)}, \log n, \log^2 n)$ to a bounded balanced width BCQ evaluation problem.

Theorem 3.3. *Let M be a regular NauxSA running in polynomial time, logarithmic space and with maximal stack depth k and x a string. Then there exists a database B and a Boolean Conjunctive Query Q with a Balanced Cut Decomposition of width 8 such that $B \models Q$ iff M accepts x . Moreover there exists a log-space reduction from (M, x) to (B, Q) .*

Proof. The database B will have the tables $I(C)$, $A(C)$, $D(C_1, C_2)$ and for each i , $1 \leq i \leq k$ the tables $U_i(C_1, S, C_2)$, $R_i(C_1, S, C_2)$ and $O_i(C_1, C_2)$. The table I will contain the initial configuration of M as the only tuple. The table A will contain all accepting configurations of M . (These tables can be both set up in constant space and time). A tuple (c, d) is in D iff M starting in configuration c (deterministically) reaches configuration d without performing any stack operations, and the next operation of M would be a stack operation. To create the tuples of D the transducer simply emulates M for all possible configurations. Obviously this can be done in log-space, since M acts deterministically. A tuple (c, d) is in O_i iff M starting in configuration c would next perform a pop and end up in configuration d . The tables O_i can be filled up by the transducer simply by checking for all configurations c whether M would perform a pop of the i th cell in this configuration. Obviously this can be done in log-space, since the tape of M is $O(\log n)$ and M has a constant number of states. A tuple (c, s, d) is in U_i iff M starting in configuration c would next push s into the i th

cell and end up in configuration d . Similarly a tuple (c, s, d) is in R_i iff M starting in configuration c would next read s from the i th cell and end up in configuration d . To create the tuples of U_i and R_i the transducer would check for every configuration c whether M would push / read next, and if it does, then for every string s check which configuration it would enter next. Again this can be done in log-space, since c and s are at most $O(\log n)$ long.

Now we build the query Q which corresponds to the run of M , contracting multiple consecutive deterministic steps into one atom. Let T be a full binary tree of depth k (containing $2^{k+1} - 1$ nodes). For a node N let $p(N)$, $l(N)$ and $r(N)$ denote the parent, left child and right child of N , respectively. We also write $d(N) = \text{depth}_T(N)$ for short, and $p^j(N)$ denotes the j -th ancestor of N (e.g. $p^3(N)$ would be the great grandparent of N).

For each $N \in T \setminus O(T)$, define the query Q_N^R in the following way: If $\text{depth}_T(N) = 1$, then $Q_N^R = R_1(X_{N,2}, S_N, X_{N,3})$. Otherwise

$$\begin{aligned}
Q_N^R = & R_{d(N)}(X_{N,2}, S_N, Y_{N,d(N)}) \wedge D(Y_{N,d(N)}, Z_{N,d(N)}) \\
& \wedge R_{d(N)-1}(Z_{N,d(N)}, S_{p(N)}, Y_{N,d(N)-1}) \\
& \wedge D(Y_{N,d(N)-1}, Z_{N,d(N)-1}) \\
& \wedge R_{d(N)-2}(Z_{N,d(N)-1}, S_{p^2(N)}, Y_{N,d(N)-2}) \\
& \dots \\
& \wedge D(Y_{N,2}, Z_{N,2}) \wedge R_1(Z_{N,2}, S_{p^{d(N)-1}(N)}, X_{N,3})
\end{aligned}$$

Q_N^R now encodes the actions of the machine which read and process the contents of the stack, after it reached the node N in the push-pop tree. The variables S_N represent the strings already pushed to the stack, and the variables $Y_{N,i}$ represent the intermediate configurations between successive reads. Note how this query will be

“attached” into the “simulation” of the machine through its first and last variables (corresponding to the starting and finishing configurations of the “stack processing”).

For each $N \in T$, define a query Q_N in the following way.

If N is the root:

$$\begin{aligned} Q_N = & D(X_{N,1}, X_{N,2}) \wedge U_1(X_{N,2}, S_{l(N)}, X_{l(N),1}) \wedge \\ & D(X_{l(N),7}, X_{N,3}) \wedge U_1(X_{N,3}, S_{r(N)}, X_{r(N),1}) \wedge \\ & D(X_{r(N),7}, X_{N,4}) \end{aligned}$$

If N is a leaf:

$$\begin{aligned} Q_N = & D(X_{N,1}, X_{N,2}) \wedge Q_N^R \wedge \\ & D(X_{N,3}, X_{N,4}) \wedge O_{d(N)}(X_{N,4}, X_{N,7}) \end{aligned}$$

Otherwise:

$$\begin{aligned} Q_N = & D(X_{N,1}, X_{N,2}) \wedge Q_N^R \wedge \\ & D(X_{N,3}, X_{N,4}) \wedge U_{d(N)+1}(X_{N,4}, S_{l(N)}, X_{l(N),1}) \wedge \\ & D(X_{l(N),7}, X_{N,5}) \wedge U_{d(N)+1}(X_{N,5}, S_{r(N)}, X_{r(N),1}) \wedge \\ & D(X_{r(N),7}, X_{N,6}) \wedge O_{d(N)}(X_{N,6}, X_{N,7}) \end{aligned}$$

Q_N now encodes all the actions of the machine after it reached the node N in the push-pop tree. The variables $X_{N,i}$ represent the configurations of M while it has S_N pushed in the $d(N)$ th stack cell. First it reads and processes the stack (unless N is the root and the stack is empty), then it pushes to the stack and “passes control” to the first child, then it pushes to the stack again and passes control to the second child (unless N is a leaf and it does not have children), and finally pops the stack and

passes control to its parent (unless N is the root). This passing of control is achieved through sharing of variables, which correspond to the according configurations of the machine at any such point in the computation.

Between all steps accessing the stack we also introduce a deterministic step. In case the machine does not need any deterministic steps, this can be encoded in the table D by having a tuple with two equal elements.

Finally define

$$Q = I(X_{O(T),1}) \wedge (\bigwedge_{N \in T} Q_N) \wedge A(X_{O(T),4})$$

Here we glue all partial queries together, and we also require the first configuration to be the initial configuration of M , and the last configuration to be an accepting configuration. A valid instantiation of the variables in Q corresponds to a successful run of M . Hence $B \models Q$ iff M accepts x .

The hypergraph corresponding to Q will have a hyperedge for every atom in the query, since they are all different. In particular, every sub-query Q_N^R will correspond to $2\text{depth}_T(N) - 1$ hyperedges. Additionally we have 5 hyperedges from the root, 3 hyperedges from each leaf, 7 hyperedges from every internal node, and 2 more hyperedges for the overall query. Altogether we get $4(k+1)2^k - 1$ hyperedges. We can build a balanced cut decomposition in the following way: For every Q_N^R it is easy to build an incomplete binary tree such that each node is labelled with exactly one atom (D or R) and that the tree is a balanced cut decomposition of Q_N^R of width 1, since Q_N^R is acyclic (ignoring the variables S_N which will be present in the final tree already). Call each of these trees R_N . Now let T' be a tree which is like T , and label every $N \in T$ with Q_N without Q_N^R . Now “merge” each R_N with the corresponding node N of T' by adding the label of the root of R_N to the label of N and attaching the rest of the sub-tree to N . Also, add two more nodes as children of the root, one labelled with the I -atom and the other with the A -atom, to produce the final (T', λ) . There will be at most 8 atoms in every label.

It is quite obvious that (T', λ) is balanced, since every node has two (for leaves) or four (for all other nodes) children and is built absolutely symmetrically. $\square$

This results in the following

Corollary 3.3. *The problem of answering Boolean Conjunctive Queries of bounded balanced width is complete for DC^1 .*

Proof. By Theorem 3.1, any algorithm in DC^1 can first be transformed into a regular NauxSA, and by Theorem 3.3 reduced to the problem of answering a Boolean Conjunctive Query with a balanced decomposition of width 8. Hence this problem is hard for DC^1 . Membership in DC^1 was shown in Corollary 3.2. Hence this problem is complete for DC^1 . $\square$

3.7 Computing Balanced Decompositions

Winning strategies in the $R\&S^k$ game give us an easy way to find balanced decompositions. Hence, we can now look at the complexity of the problem of recognizing hypergraphs with bounded balanced width. Consider the algorithm k -robber-sergeants presented in Algorithm 2.

Here the function $connected(x, y)$ checks whether hyperedge x is $[S]$ -connected to hyperedge y , where S is the set of all sergeant hyperedges already on the stack. This can be done, by guessing the path one hyperedge at a time and checking that it is $[S]$ -adjacent to the last hyperedge, which is feasible in NL. An alternative approach would consist of removing all vertices covered by sergeants from the hypergraph, transforming it into its dual hypergraph, then taking the primal graph of that hypergraph (this can all be done in LOGSPACE) and finally using undirected st -connectivity algorithm of [45] which is also in LOGSPACE.)

Algorithm 2 k -robber-sergeants

```
1: input Hypergraph  $H$ 
2: fixed parameter  $k$ : Integer
3: check-win(firstEdge( $H$ ),  $|V(H)|$ )
4: accept
5:
6: function connected(Edge  $x$ , Edge  $y$ ): Boolean {
7:   ▷ checks if  $x$  is  $[S]$ -connected to  $y$ ,
8:   ▷ where  $S$  is set of edges on the stack
9: }
10:
11: function count-connected-edges(Edge  $e$ , Edge[1... $k$ ]  $sergs$ ): Integer {
12:   push  $sergs$ 
13:   Integer  $n := 0$ 
14:   for all (Edge  $f \in E(H)$ ) {
15:     if (connected( $e$ ,  $f$ )) {
16:        $n := n + 1$ 
17:     }
18:   }
19:   pop  $sergs$ 
20:   return  $n$ 
21: }
22:
23: procedure check-win(Edge  $r$ , Integer  $size$ ) {
24:   guess Edge[1... $k$ ]  $sergs$ 
25:   for all (Edge  $f \in E(H)$ ) {
26:     if (connected( $r$ ,  $f$ )) {
27:       Integer  $n :=$  count-connected-edges( $f$ ,  $sergs$ )
28:       if ( $n > size/2$ ) {
29:         reject
30:       } else if ( $n > 0$ ) {
31:         check-win( $e$ ,  $n$ )
32:       }
33:     }
34:   }
35: }
```

Theorem 3.4. *Let k be a positive integer, and H a hypergraph. The algorithm k -robber-sergeants accepts iff a balanced decomposition of H of width at most k exists. Moreover this algorithm is in DC^1 .*

Proof. Correctness is easily seen by the nondeterministic nature of the algorithm. All possible robber escape routes are always considered, and the algorithm accepts iff the robber is captured on all such routes and the escape space always shrinks by at least a factor of 2. If no decomposition exists, then there will always be two consecutive robber positions r_1 and r_2 in $\text{R\&S}^k$, such that the escape space at r_2 is more than half as big as at r_1 , and the algorithm will necessarily reject.

Since at every recursive call of *check-win* the size of the new escape space has to be at most half as big as the previous one, the depth of the call stack is at most $\log |E(H)|$. Also, worktape needs only to store a fixed amount of integers, hyperedges and an array of at most k hyperedges, hence requiring $O(\log n)$ space. Also every chunk pushed to the stack has the same size, and hence the stack uses at most $O(\log^2 n)$ memory. The algorithm is polynomial: At the first recursion level, the function *check-win* is called n times. At the second recursion level, the function *check-win* is also called at most n times - every component of size s calls it s times, and the sum of all the components is at most n . This continues to the lowest level. Altogether, there will be at most $O(n \log n)$ calls to the function *check-win*, every run of which takes polynomial time. Hence the overall algorithm is polynomial. $\square$

Note, that this algorithm repeats a lot of work, since rather than checking the existence of a winning strategy for a whole component, it checks it for every edge in that component. However, this does not affect the correctness of the algorithm and preserves polynomial running time, while at the same time ensuring logarithmic worktape space, and square-logarithmic space on the stack. In practice, a deterministic version of this algorithm would of course trade space for time, avoiding any unnecessary redundant checks.

Remark 3.5. We can adapt k -robber-sergeants to check for shallow decompositions: Instead of checking the size of each component and rejecting if it is larger than half the size of the previous component, we can simply keep track of the depth of the recursive call stack, and reject once it becomes greater than $\log |E(H)|$.

3.8 Determinization and Parallelization

A standard technique to make a nondeterministic algorithm deterministic is a brute-force search of the computation tree, trying out all nondeterministic choices and back-tracking whenever a choice leads to failure. In many cases this increases the time requirement, however not always the space requirement. For the class DC^1 this is also the case, in particular since it is a subclass of $DSPACE(\log^2 n)$. Once we have a nondeterministic choice (once we reach a new node r in the push-pop tree), we simply enumerate all possibilities for this choice. For each such choice, the push-pop tree will have two children s and t (for the case of regular stack automata). Moreover, the result of each of these children does not depend on the result of the other, since their only shared information is the one that was pushed onto the stack up until r was reached. Hence, we can “split” the current thread of computation into two threads — one of them works on s and the other one on t . This procedure can be repeated all the way down to the deepest level of the push-pop tree, until n threads operate in parallel. Every time a thread encounters a failure of a node in the push-pop tree, it needs to stop the thread working on the sibling of that node and then resume working on their parent. Even though the number of nondeterministic choices at each node in the push-pop tree is polynomial, the total amount of work that potentially needs to be done is super-polynomial: There are $O(n^{O(1) \log n})$ possibilities for the contents of the stack (at the deepest level in the push-pop tree). Hence the algorithm, even with full parallelization, remains super-polynomial. It is however still *quasi-polynomial*.²

²i.e. in $O(n^{O(1) \log n})$

Moreover, if a positive answer exists, the algorithm could still find it quickly, and the parallel speed-up would be quite noticeable. Of course, we can force the algorithm to avoid computing the same data twice by maintaining a table of results. Each entry in the table would have the contents of the stack as its key and would indicate whether the computation from that point onwards led to success or failure. Unfortunately, by above argument, the space requirement for the table would be only quasi-polynomial.

3.9 Reflections

We have defined new hypergraph decomposition methods, called balanced and shallow decompositions. We defined a new hierarchy of complexity classes (the DC hierarchy) in terms of resource bounded nondeterministic auxiliary stack automata. We analyzed the computational complexity of the problem of recognizing hypergraphs of bounded balanced width and bounded shallow width and showed its membership in DC^1 . We analyzed the computational complexity of the problem of answering BCQs of bounded balanced width (and hence also bounded shallow width) and showed that it is complete for DC^1 .

Future Work includes a better analysis of relations between resource-bounded (N)AuxSAs and other models of computation, in order to relate the complexity classes in the DC hierarchy to other known complexity classes, in particular those presented in [47], [25] and [9]. Another direction of work is to establish whether the problem of recognizing hypergraphs of bounded BW is complete for DC^1 or whether it belongs to a lower complexity class.

In this dissertation we decided not to implement any of the algorithms described in this chapter, in favour of tractable algorithms. Thus, from now on we shall focus on hypertree decompositions alone. However, we will still use some inspiration from the ideas presented in this chapter to optimize parallel work in hypertree computations, in particular in Section 4.3.

Chapter 4

Computing Hypertree Decompositions

4.1 Motivation

Unfortunately, balanced cut decompositions only give quasi-polynomial running time for the BCQ evaluation problem. Also, we do not know if computing balanced cut decompositions of a fixed width is tractable.

The best known decompositions that fulfil both of these requirements are *hypertree decompositions*. Hence it is important to find good algorithms for computing them.

Also, as the hardware market slowly shifts from producing processors with greater frequency to producing processors with more computational cores, lower power consumption and higher memory throughput, new parallel algorithms become ever so important. Since the problem of computing hypertree decompositions lies in the low complexity class LOGCFL, good parallel algorithms must exist. Finding such algorithms, however, is not trivial.

Ruzzo [47] showed that any problem in LOGCFL has an alternating algorithm in $\text{ATiSp}(\log^2 n, \log n)$. Moreover, such an algorithm uses only $O(\log n)$ alternations, and an accepting computation tree is still only polynomial. In particular, the bound on its height implies good branching at universal nodes, which in turn facilitates the parallelization of a deterministic algorithm derived from an alternating one.

Unfortunately, the best alternating algorithm *k-decomp*, which forms the basis of the best deterministic algorithm *det-k-decomp* does not possess the above property.

4.2 Structure of Chapter

In Section 4.3 we analyze the best known sequential algorithm, *det-k-decomp*, which is the deterministic version of the alternating algorithm *k-decomp*. In particular, we propose several improvements to the heuristics used in [32], and finally argue why the algorithm is not trivially suitable for parallelization.

In Section 4.4 we look at several properties of LOGCFL and derive the alternating algorithm *k-divide-decomp* which only uses $O(\log n)$ alternations and very much resembles the algorithm for finding balanced decompositions. We show how to make it into a deterministic $O(\log^2 n)$ -space algorithm and finally derive the polynomial-time and fully parallelizable algorithm *par-k-divide-decomp*.

In Section 4.5 we examine the problems of *par-k-divide-decomp* and discuss two approaches to combine the ideas from the previous sections. We then finally discuss how to create the algorithm *par-det-k-decomp*, which has all the good properties of *det-k-decomp*, but also takes full advantage of parallelization. Its two ingredients are the same basic data structures and algorithms as in *det-k-decomp*, and a system to schedule the individual computations, which will be presented in detail in Chapter 5. We also discuss an important heuristic for *par-det-k-decomp*.

Finally, Section 4.6 concludes the chapter by reflecting upon its results and outlining the remaining issues.

4.3 A Sequential Polynomial-Time Algorithm

Before turning our attention to computing hypertree decompositions, we shall take another look at the robber and marshals game and formally define strategies.

Definition 4.1. Let H be a hypergraph and let k be a positive integer. A *winning strategy in the robber and k marshals game on H* ($\text{R\&M}^k(H)$) is a tuple (T, ρ, λ) , where T is a rooted tree, and $\rho, \lambda : T \rightarrow \mathcal{P}(E(H))$ are labelling functions such that

- *Boundedness*: for all $r \in T$, $|\lambda(r)| \leq k$ (k marshals can cover at most k edges),
- *Initial Condition*: $\rho(O(T)) = E(H)$ (the initial escape space consists of all vertices),
- *Completeness*: for all $r \in T, e \in \rho(r)$, either $e \subseteq \bigcup \lambda(r)$ or there exists $s \in T(r)$ such that $e \in \rho(s)$ (after every move, every edge of the escape space is either covered by a marshal or is part of a new escape space),
- *Connectedness*: for all $r \in T, s \in T(r)$, $\rho(s)$ is a $[\lambda(r)]$ -edge-component,
- *Separation*: for all $r \in T, s, t \in T(r)$, if $s \neq t$, then $\rho(s) \cap \rho(t) = \emptyset$ (the new escape spaces are pairwise disjoint).

We call $\rho(r)$ the *edge escape space at r* and $\lambda(r)$ the *marshal position at r* . For brevity, we will also call such a labelled tree simply a *strategy*.

Definition 4.2. Let H be a hypergraph, k a positive integer and let (T, ρ, λ) be a winning strategy in $\text{R\&M}^k(H)$. (T, ρ, λ) is a winning strategy in the *monotone $\text{R\&M}^k(H)$* iff it satisfies

- *Monotonicity*: for all $r \in T, s \in T(r)$, $\rho(s) \subset \rho(r)$ (the escape space of the robber must be properly decreased every move).

For brevity, we will also call such a strategy a *monotone strategy*.

Note that in these definitions we use sets of edges rather than sets of vertices as escape spaces. The intuition behind this is that the edges of the escape space are the ones that the robber can move along without being captured by the marshals. Also,

this definition makes the conditions easier, since, to find the new escape spaces, we now only have to look at the old escape space and at the marshal position rather than two subsequent marshal positions. In fact, it is quite easy to derive the classic *vertex escape space* from ρ : Define $\rho_V : T \rightarrow \mathcal{P}(V(H))$ by

- $\rho_V(O(T)) = \bigcup \rho(O(T))$,
- for all $r \in T, s \in T(r)$, $\rho_V(s) = \bigcup(\rho(s)) \setminus (\bigcup \lambda(r))$.

We call $\rho_V(s)$ the *vertex escape space at s* .

Since the classic game and its strategies are defined using vertex-components, we shall now show that these definitions are equivalent.

Proposition 4.1. *Let H be a hypergraph, k a positive integer and (T, ρ, λ) a winning strategy. Then $\rho_V(O(T)) = V(H)$.*

Proof. Trivial, since we assume that H has no isolated vertices. □

Proposition 4.2. *Let H be a hypergraph, k a positive integer and (T, ρ, λ) a winning strategy. For all $r \in T, s \in T(r)$, $\rho_V(s)$ is a $[\lambda(r)]$ -vertex-component.*

Proof. Let $u, w \in \rho_V(s)$. Then there are edges $f, g \in \rho(s)$ such that $u \in f, w \in g$. Since $\rho(s)$ is a $[\lambda(r)]$ -edge-component, there exists a $[\lambda(r)]$ -edge-path $\{e_i\}_{i=0}^t$ between f and g . Let $\{v_i\}_{i=1}^t$ be the corresponding $[\lambda(r)]$ -vertex-path. Then, for each $i, 1 \leq i \leq t$ we have $v_i \notin \bigcup \lambda(r)$. Hence, $u, v_1, \dots, v_t, w$ is a $[\lambda(r)]$ -vertex-path between u and w , and hence u and w are $[\lambda(r)]$ -connected.

Now pick an $x \in V(H)$ such that u and x are $[\lambda(r)]$ -connected. Then there exists a $[\lambda(r)]$ -vertex-path $\{y_i\}_{i=0}^n$ between u and x . Let $\{h_i\}_{i=1}^n$ be the corresponding $[\lambda(r)]$ -edge-path, and since $h_1 \in \rho(s)$ we also have $h_n \in \rho(s)$. And since $x \notin \bigcup \lambda(r)$, we have $x \in \rho_V(s)$. □

Proposition 4.3. *Let H be a hypergraph, k a positive integer and (T, ρ, λ) a winning strategy. Then for all $r \in T, s, t \in T(r)$, if $s \neq t$, then $\rho_V(s) \cap \rho_V(t) = \emptyset$.*

Proof. Suppose there was a $v \in \rho_V(s) \cap \rho_V(t)$. Then there exists an edge e such that $v \in e$ and we have $e \in \rho(s) \cap \rho(t)$ contradicting separation. $\square$

Definition 4.3. [7] Let H be a hypergraph, $r \in V(H)$, $X, Y \subseteq E(H)$, and let $R \subseteq V(H)$ be the $[X]$ -vertex-component containing r . The *set of possible escape spaces with respect to Y* is

$$\mathcal{R}(Y; X, R) = \{C \subseteq V(H) | C \text{ is a } [Y]\text{-vertex-component such that there is a } [(\bigcup X) \cap (\bigcup Y)]\text{-vertex-path from } R \text{ to } C\}$$

Proposition 4.4. Let H be a hypergraph, let k be a positive integer and let (T, ρ, λ) be a strategy. Let $s \in T$. Then for each $C \in \mathcal{R}(\lambda(s); M, \rho_V(s))$, there exists precisely one $t \in T(s)$ such that $C = \rho_V(t)$, where

$$M = \begin{cases} \emptyset & \text{if } s = O(T) \\ \lambda(r) & \text{if there exists } r \in T \text{ such that } s \in T(r) \end{cases}$$

Proof. The two above propositions say that for any $s \in T, t \in T(s)$, every $\rho_V(t)$ is a unique $[\lambda(s)]$ -vertex-component. Hence we just need to show “completeness”, i.e. that all the components in $\mathcal{R}(\lambda(s); M, \rho_V(s))$ are covered.

If $s = O(T)$, then $\mathcal{R}(\lambda(s); \emptyset, V(H)) = \{C \subseteq V(H) | C \text{ is a } [\lambda(s)]\text{-vertex-component such that there is a } [\bigcup Y]\text{-vertex-path from } V(H) \text{ to } C\}$, which is simply $\{C \subseteq V(H) | C \text{ is a } [\lambda(s)]\text{-vertex-component}\}$. Let $v \in V(H) \setminus \bigcup \lambda(s)$, and let $e \in E(H)$ be any edge such that $v \in e$. Since e is not covered by $\bigcup \lambda(s)$, by completeness, there exists $t \in T(s)$ such that $e \in \rho(t)$. But then we must have $v \in \rho_V(t)$, since $v \in \bigcup \rho(t) \setminus \bigcup \lambda(s)$.

Now suppose that $r \in T, s \in T(r)$. Let $C \in \mathcal{R}(\lambda(s); \lambda(r), \rho_V(s))$ and $u \in C$. Then u is $[(\bigcup \lambda(r)) \cap (\bigcup \lambda(s))]$ -connected to $\rho_V(s)$. If $u \in \rho_V(s)$, then there is an edge $f \in \rho(s)$ such that $u \in f$, and since $u \notin \bigcup \lambda(s)$ and hence f is not covered by $\bigcup \lambda(s)$, there will be a $t \in T(s)$ such that $f \in \rho(t)$ and hence also $u \in \rho_V(t)$. Suppose $u \notin \rho_V(s)$. There is a vertex $w \in \rho_V(s)$ such that there is a $[(\bigcup \lambda(r)) \cap (\bigcup \lambda(s))]$ -vertex-path $\{v_i\}_{i=0}^n$

from w to u . Let $\{e_i\}_{i=1}^n$ be the corresponding $[(\bigcup \lambda(r)) \cap (\bigcup \lambda(s))]$ -edge-path, i.e. $v_{i-1}, v_i \in e_i$, for all $i, 1 \leq i \leq t$. Let j be maximal such that $v_{j-1} \in \rho_V(s)$ and $v_j \notin \rho_V(s)$, i.e. $v_i \notin \rho_V(s)$ for all $i \geq j$. Since $\rho_V(s)$ is a $[\lambda(r)]$ -vertex-component, we must have $v_j \in \bigcup \lambda(r)$, and since $\{v_i\}_{i=j}^n$ is a $[(\bigcup \lambda(r)) \cap (\bigcup \lambda(s))]$ -vertex-path, we must also have $v_j \notin \bigcup \lambda(s)$. However, we have $e_j \in \rho(s)$, and since e_j is not covered by $\bigcup \lambda(s)$, there exists $t \in T(s)$ such that $e_j \in \rho(t)$. Since $v_j \in e_j$ and $v_j \notin \bigcup \lambda(s)$, we must have $v_j \in \rho_V(t)$. But then $\{v_i\}_{i=j}^n$ is a $[\lambda(s)]$ -vertex-path from v_j to u , and since $\rho_V(t)$ is a $[\lambda(s)]$ -component, we must also have $u \in \rho_V(t)$. $\square$

Corollary 4.1. *Our strategies are equivalent to traditional strategies using vertex-components such as those in [7].*

Proposition 4.5. *Let H be a hypergraph, k a positive integer and (T, ρ, λ) a monotone strategy. Then for all $r \in T, s \in T(r)$, $\rho_V(s) \subset \rho_V(r)$.*

Proof. Let $v \in \rho_V(s)$. If $r = O(T)$, then obviously $v \in \rho_V(r)$. Suppose that $r \in T(q)$. Since $v \notin \bigcup \lambda(r)$, for every edge e such that $v \in e$ we must have $e \in \rho(s)$. By monotonicity, for every such edge we also have $e \in \rho(r)$, and hence none of these edges is in $\lambda(q)$. Hence, $v \notin \bigcup \lambda(q)$ and also $v \in \rho_V(r)$.

Now, since $\rho(s) \subset \rho(r)$, there exists an edge $e \in \rho(r)$ such that $e \notin \rho(s)$. If there exists $t \in T(r)$ such that $e \in \rho(t)$, then there also exists $v \in \rho_V(t)$ such that $v \in e$, and we certainly have $v \notin \rho_V(s)$ by separation. Otherwise, we must have $e \subseteq \lambda(r)$, in which case again, there is some $v \in e$ such that $v \in \rho_V(r)$, $v \in \bigcup \lambda(r)$ and $v \notin \rho_V(s)$. $\square$

Corollary 4.2. *Our monotone strategies are also equivalent to traditional monotone strategies with vertex-components.*

Also, note that when comparing these strategy trees to standard game trees (see e.g. [30]), where every node is labelled by a position in the game, we “move up” the marshal labels by one level in the tree, “reversing” the position labels. That is we fix

a label after every marshal move, rather than after every robber move. On the one hand, this emphasizes the fact that this is a strategy for the marshals — for every move the robber takes (for every escape space he chooses), there is an appropriate response by the marshals. On the other hand, this representation is closer to the actual hypertrees that correspond to strategies.

Definition 4.4. Let H be a hypergraph, k a positive integer and (T, ρ, λ) a strategy. Let us define the auxiliary labelling function $\chi : T \rightarrow \mathcal{P}(V(H))$ by

$$\chi(s) = \left(\bigcup \rho_E(s) \right) \cap \left(\bigcup \lambda(s) \right)$$

We call $\chi(s)$ the *covered vertices at s* .

The auxiliary function χ gives a direct way to construct a hypertree decomposition from a strategy. To show the other half of Theorem 2.7 we also need to define *normal forms* for hypertree decompositions [29]. Without going into any detail of the definitions, we state that the results in [30] and [7] imply that normal form hypertree decompositions always exist, and that monotone strategies precisely capture all such decompositions.

To find a hypertree decomposition we now simply have to compute a winning strategy in the monotone R&M^k game.

Remark 4.1. As a final remark, we state that edge-components are stronger than vertex-components in the sense that there might be edge-components where there are no vertex-components — this happens when there is an edge whose vertices are completely covered by the marshals, but which itself is not. We could thus replace the completeness condition by the following *strong completeness condition*:

- For all $r \in T, e \in \rho_E(r)$, either $e \in \lambda(r)$ or there exists $s \in T(r)$ such that $e \in \rho_E(s)$.

We would thus include edge-components whose corresponding vertex-components are empty. Each such component would consist of a single edge.

In fact, such strategies correspond to what Gottlob et al. call *complete hypertree decompositions*, i.e. hypertree decompositions in which every edge is *strongly covered* [29]. Complete decompositions exist iff decompositions exist, therefore both variants of the definition of strategies are more or less equivalent. We will speak of *complete strategies* whenever we assume that the strong completeness condition is used.

4.3.1 The Alternating Algorithm k -decomp

Definition 4.5. Let H be a hypergraph, k a positive integer and (T, ρ, λ) a strategy. Let us define the auxiliary labelling function $\gamma : T \rightarrow \mathcal{P}(V(H))$ by

$$\gamma(s) = (\bigcup \rho(s)) \cap (\bigcup (E(H) \setminus \rho(s)))$$

We call $\gamma(s)$ the *connecting set at s* .

Proposition 4.6. *If (T, ρ, λ) is a monotone strategy, then for each $r \in T$, $\lambda(r) \cap \rho(r) \neq \emptyset$.*

Proof. If $T(r) = \emptyset$, then, by completeness, for each $e \in \rho(r)$, $e \in \lambda(r)$. Now suppose $T(r) \neq \emptyset$ and $\rho(r) \cap \lambda(r) = \emptyset$. Then for every edge $e \in \rho(r)$ we have $e \notin \lambda(r)$. So by completeness and connectedness, r has a child s with $\rho(s) = \rho(r)$, which contradicts monotonicity. $\square$

Proposition 4.7. *If (T, ρ, λ) is a monotone strategy, then for each $r \in T$, $\gamma(r) \subseteq \bigcup \lambda(r)$.*

Proof. Let $v \in \gamma(r)$. Then there are edges $e \in \rho(r)$, $f \notin \rho(r)$ such that $v \in e \cap f$. Suppose $v \notin \bigcup \lambda(r)$. Then $e \notin \lambda(r)$ and hence there exists $s \in T(r)$ such that $e \in \rho(s)$. Also, we must have $f \notin \lambda(r)$, and hence e and f are $[\lambda(r)]$ -adjacent. By connectedness we must have $f \in \rho(s)$, contradicting the monotonicity condition. $\square$

In fact, from the above two conditions we can derive the monotonicity condition.

Proposition 4.8. *Let (T, ρ, λ) be a strategy fulfilling the following conditions:*

- *for each $r \in T$, $\lambda(r) \cap \rho(r) \neq \emptyset$,*
- *for each $r \in T$, $\gamma(r) \subseteq \bigcup \lambda(r)$.*

Then (T, ρ, λ) is a monotone strategy.

Proof. Let $r \in T, s \in T(r)$. Let $f \in \rho(s)$. Then f is not covered by $\bigcup \lambda(r)$. By completeness and connectedness, there exists $g \in \rho(r)$ such that $g \in \rho(r)$, g is not covered by $\bigcup \lambda(r)$ and f and g are $[\lambda(r)]$ -connected. Let $\{e_i\}_{i=0}^t$ be the $[\lambda(r)]$ -edge-path from g to f . Let $\{v_i\}_{i=1}^t$ be the corresponding $[\lambda(r)]$ -vertex-path. Then for all $i, 1 \leq i \leq t$, $v_i \notin \bigcup \lambda(r)$. If $f \notin \rho(r)$, then any edge-path connecting f and g must pass through a vertex in $\gamma(r)$, so there must be some $j, 1 \leq j \leq t$ such that $v_j \in \gamma(r)$. But then $v_j \in \bigcup \lambda(r)$, and $\{v_i\}_{i=1}^t$ cannot be a $[\lambda(r)]$ -vertex-path. Hence we must have $f \in \rho(r)$.

Now since $\lambda(r) \cap \rho(r) \neq \emptyset$, pick any $h \in \lambda(r) \cap \rho(r)$. Then $h \notin \rho(s)$, since $\rho(s)$ is a $[\lambda(r)]$ -edge-component.

Hence $\rho(s) \subset \rho(r)$, and (T, ρ, λ) is a monotone strategy. $\square$

From now on we will also call an escape space a *component*, and a valid marshal position covering the γ -label a *separator*.

The completeness, connectedness and separation conditions describe how to compute new escape spaces (new components) given an old escape space (old component) and a valid marshal position (separator). The two new conditions tell us which marshal positions (separators) are valid.

There is another way to define the connecting set, provided the strategy is monotone, which we include here for the sake of completeness:

Proposition 4.9. *Let (T, ρ, λ) be a monotone strategy. Then for all $r \in T, s \in T(r)$, $\gamma(s) = (\bigcup \rho(s)) \cap (\bigcup \lambda(r))$.*

Proof. It is obvious that $\lambda(r) \subseteq E(H) \setminus \rho(s)$ by the connectedness condition. Hence $(\bigcup \rho(s)) \cap (\bigcup \lambda(r)) \subseteq \gamma(r)$.

To show the converse, let $e \in E(H) \setminus \rho(s)$ such that $e \cap (\bigcup \rho(s)) \neq \emptyset$. Then we must have $e \subseteq \bigcup \lambda(r)$ for otherwise we would have $e \in \rho(s)$. Hence $\gamma(r) \subseteq (\bigcup \lambda(r))$ and also $\gamma(r) \subseteq (\bigcup \lambda(r)) \cap (\bigcup \rho(s))$. $\square$

We can thus construct the alternating algorithm k -decomp [32], presented in Algorithm 3.

Algorithm 3 k -decomp

```

1: type Vertex = Integer
2: type Edge = Set⟨Vertex⟩
3: type Hypergraph = Set⟨Edge⟩
4: var  $h$ : Hypergraph
5: var  $k$ : Integer
6: input  $h$ 
7: input  $k$ 
8: decomposable( $h, \emptyset$ )
9: accept
10:
11: procedure decomposable( $comp$ : Set⟨Edge⟩,  $oldSep$ : Set⟨Edge⟩) {
12:   var  $separator$ : Set⟨Edge⟩
13:   guess  $separator$ 
14:   if ( $|separator| > k$  or  $(\bigcup comp) \cap (\bigcup oldSep) \not\subseteq \bigcup separator$  or  $comp \cap separator = \emptyset$ ) {
15:     reject
16:   }
17:   var  $components$ : Set⟨Set⟨Edge⟩⟩ := separate( $comp, separator$ )
18:   for all ( $c \in components$ ) {
19:     decomposable( $c, separator$ )
20:   }
21: }
22:
23: function separate( $es$ : Set⟨Edge⟩,  $sep$ : Set⟨Edge⟩): Set⟨Set⟨Edge⟩⟩ {
24:    $\triangleright$  separates  $es$  using  $sep$  and returns a set of components
25: }
```

We omit the details of the function *separate* for now. Depending on the implementation, it can return components whose vertex escape space is empty (single hyperedges whose vertices are completely covered by the separator), or not.

Remark 4.2. Note, that k -decomp can be implemented using only a constant number of pointers — in the procedure *decomposable*, every separator needs at most k pointers. Also, rather than representing a component using all its hyperedges, we can represent it by a single hyperedge, say the hyperedge with the lowest number, together with the k hyperedges of the separator that was used to create this component. It is easy to see that such a representation is unique.

Moreover, all the set operations can be done directly on the input,¹ also using only a constant number of pointers. Checking whether two edges are in the same component is equivalent to *undirected s - t -connectivity*, which is feasible in L [45], so the function *separate* also needs at most $O(\log n)$ space, where $n = |E(H)|$.

The total number of distinct components that can be created using a separator of size at most k is bounded by n^{k+1} . Hence, there can be at most polynomially many configurations in an accepting computation tree. Therefore, k -decomp is feasible in LOGCFL.

Notice that this algorithm merely checks whether a hypertree decomposition exists, rather than computing one. It is very easy to transform this algorithm into one with an output, by adding a few lines to the procedure *decomposable*. The changes are presented in Algorithm 4.

The output is now a hypertree representation using the grammar

$$\text{Hypertree} := (\chi\text{-label}, \lambda\text{-label}, \text{Hypertree}^*)$$

i.e. a hypertree first shows its χ - and λ -labels, and then lists all its sub-trees, if it has any.

¹ We can assume that the input is formatted according to our needs, since all representations of a hypergraph are equivalent, up to a LOGSPACE-reduction.

Algorithm 4 k -decomp with output

```
1: procedure decomposable(comp: Set⟨Edge⟩, oldSep: Set⟨Edge⟩) {  
2:   ...  
3:   components := separate(comp, separator)      ▷ changes start after this line  
4:   var chi: Set⟨Vertex⟩ := ( $\bigcup$  separator)  $\cap$  ( $\bigcup$  comp)  
5:   output "(" + chi + "," + separator + ","  
6:   for all ( $c \in$  components) {  
7:     decomposable(c, separator)  
8:   }  
9:   output ")"  
10: }
```

4.3.2 The Deterministic Algorithm det- k -decomp

This section is mostly a summary of [32].

To make the algorithm in the previous section deterministic, we must be able to enumerate all possibilities for every single nondeterministic guess step. In our case, we have to be able to enumerate all admissible separators for any given component. An important point here is that we would like to return new separators **as they are computed**, rather than computing the set of all possible separators, and then returning them one by one.² This would take up lots of memory and would also be much slower (since in many instances we won't have to compute all possible separators).

We adopt an object-oriented style of pseudo-code and we assume that we have classes *Hypergraph*, *Component* and *Separator*. As we will see later, the latter two will be required to store more than just the component and separator edges. Also, a hypergraph object will now contain the width k of the decomposition that we want to compute, and we assume that that parameter is passed appropriately from the input. We present how a *Hypergraph* object could look like in Algorithm 5.

For the enumeration of all possible separators, we will have a method *Hypergraph.nextSeparator* taking a *Component* as an argument and returning either a *Separator* object or **nil** if all admissible separators were already returned for this

² This technique is well-known as *lazy evaluation* in functional programming.

Algorithm 5 The class *Hypergraph*

```
1: class Hypergraph {  
2:   field edges: Set<Edge>  
3:   field width: Integer  
4:   method nextComponent(Separator): Component  
5:   method nextSeparator(Component): Separator  
6:   constructor Hypergraph(File, Integer)  
7: }
```

component. A *Component* will thus have to have some internal information about which separators were already tested. To adhere to the same style, we will also have a method *Hypergraph.nextComponent* taking a *Separator* (which will also store the information about the component that this separator is used on) and returning *Components*, one by one, until it eventually returns **nil**, when all components were already returned.

Computing components given an old component and a separator is quite easy using a depth-first search. We present a possible implementation together with the details of the *Separator* class in Algorithm 6.

Computing separators for a given component is more tricky. We will leave the details of enumerating all possible separators until the end of this section. For now we just assume that the method *Hypergraph.nextSeparator* does in fact list all admissible separators that could lead to a valid hypertree decomposition.

We can now transform the alternating algorithm *k-decomp* into a deterministic algorithm, as shown in Algorithm 7.

This algorithm, as it stands, is quite bad, since it is very easy for work to be repeated. For example, if two different separators s_1, s_2 produce the same subcomponents for some component C , and C is not decomposable, then the algorithm would try to decompose all those subcomponents twice. Furthermore, even if different components with different separators eventually lead to the same subcomponent C , which is not decomposable, the algorithm might try to decompose C again.

Algorithm 6 Enumerating Components

```
1: class Separator {
2:   field sepEs: Set⟨Edge⟩           ▷ the set of edges of the separator
3:   field sepVs: Set⟨Vertex⟩         ▷ the set of vertices covered by them
4:   field compEs: Set⟨Edge⟩         ▷ the set of edges left in the component
5:   field compVs: Set⟨Vertex⟩       ▷ the set of vertices left in the component
6:   constructor (ses: Set⟨Edge⟩, ces: Set⟨Edge⟩, cvs: Set⟨Vertex⟩) {
7:     sepEs := ses
8:     sepVs :=  $\bigcup$  sepEs
9:     compEs := ces \ sepEs
10:    compVs := cvs \ sepVs
11:  }
12: }
13: ▷ parameter s is passed by reference
14: method Hypergraph.nextComponent(var s: Separator): Component {
15:   if (s.compVs =  $\emptyset$ ) {
16:     return nil
17:   }
18:   var newCompEs: Set⟨Edge⟩ :=  $\emptyset$ 
19:   var newCompVs: Set⟨Vertex⟩ :=  $\emptyset$ 
20:   var connVs: Set⟨Vertex⟩ :=  $\emptyset$ 
21:   dfs(s.compVs.first(), s, newCompEs, newCompVs, connVs)
22:   return new Component(newCompEs, newCompVs, connVs)
23: }
24: ▷ parameters s, compEs, compVs and connVs are passed by reference
25: procedure dfs(v: Vertex, var s: Separator, var compEs: Set⟨Edge⟩, var
   compVs: Set⟨Vertex⟩, var connVs: Set⟨Vertex⟩) {
26:   s.compVs := s.compVs \ {v}
27:   if (v ∈ s.sepVs) {
28:     connVs := connVs ∪ {v}
29:   } else {
30:     compVs := compVs ∪ {v}
31:     for all (var e: Edge ∈ s.compEs) {
32:       if (v ∈ e) {
33:         compEs := compEs ∪ {e}
34:         for all (var v': Vertex ∈ e) {
35:           if (v' ∉ compVs and v' ∉ connVs) {
36:             dfs(v', s, compEs, compVs, connVs)
37:           }
38:         }
39:       }
40:     }
41:   }
42: }
```

Algorithm 7 det- k -decomp — first attempt

```
1: var  $h$ : Hypergraph
2: input  $h$ 
3: if (decomposable( $h$ ,  $h$ .initialComponent())) {
4:   accept
5: } else {
6:   reject
7: }
8:
9: function decomposable( $h$ : Hypergraph,  $comp$ : Component): Boolean {
10:   while ((var  $sep$ : Separator :=  $h$ .nextSeparator( $comp$ ))  $\neq$  nil) {
11:     var  $allSuccessful$ : Boolean := true
12:     while ((var  $newComp$ : Component :=  $h$ .nextComponent( $sep$ ))  $\neq$  nil) {
13:       if (not decomposable ( $h$ ,  $newComp$ )) {
14:          $allSuccessful$  := false
15:         break
16:       }
17:     }
18:     if ( $allSuccessful$ ) {
19:       return true
20:     }
21:   }
22:   return false
23: }
```

Luckily, we have seen that the total number of components is bounded by the polynomial $|E(H)|^{k+1}$. Thus, it is in principle possible to store all these components and to avoid repeating work.

We introduce a dynamic table that stores the results of already computed components. Once a component C is known to be decomposable, we put it in the table, and whenever the algorithm tries to decompose the same component C again, it simply returns **true**. Conversely, once a component C is known not to be decomposable, we also put that result into the table, and the algorithm returns **false** once it tries to decompose C again. We implement this table as a map (e.g. a hash-map) with the class *Component* as the key type and *Boolean* as the value type.

We assume that we have a class of type *Map* with methods *contains*, *value* and *store*. The class is clever enough to recognize when two *Component* objects represent the same component (i.e. have the same component edges, connecting set etc.), even though some of their internal fields (like the ones used to store information about separators that were already tried) might differ.

The required changes are only minor and we can present the deterministic algorithm *det- k -decomp* [32] in Algorithm 8.

The usage of the table makes sure that *decomposable* is executed at most once for every possible component. Since the total number of possible components is limited by a polynomial, and, assuming that every execution of *decomposable* is also polynomial, *det- k -decomp* runs in polynomial time overall. To estimate the exact running time, however, we need to know how many times the outer **while**-loop is executed for each component — but for that we need to look at the method *Hypergraph.nextSeparator*.

An obvious way to enumerate admissible separators for a given component is simply to enumerate all possible subsets of the edges of the hypergraph, returning only those subsets which cover the connecting set. However, as we shall see later, many of such separators will be unnecessary and this procedure would just take up

Algorithm 8 det- k -decomp

```
1: var  $h$ : Hypergraph
2: var  $table$ : Map(Component, Boolean)
3: input  $h$ 
4: if (decomposable( $h$ ,  $h.initialComponent()$ )) {
5:   accept
6: } else {
7:   reject
8: }
9:
10: function decomposable( $h$ : Hypergraph,  $comp$ : Component): Boolean {
11:   if ( $table.contains(comp)$ ) {
12:     return  $table.value(comp)$ 
13:   }
14:   while ((var  $sep$ : Separator :=  $h.nextSeparator(comp)$ )  $\neq$  nil) {
15:     var  $allSuccessful$ : Boolean := true
16:     while ((var  $newComp$ : Component :=  $h.nextComponent(sep)$ )  $\neq$  nil) {
17:       if (not decomposable( $h$ ,  $newComp$ )) {
18:          $allSuccessful$  := false
19:         break
20:       }
21:     }
22:     if ( $allSuccessful$ ) {
23:        $table.store(comp, \mathbf{true})$ 
24:       return true
25:     }
26:   }
27:    $table.store(comp, \mathbf{false})$ 
28:   return false
29: }
```

more time than needed.

Gottlob and Samer define *strong normal forms* for hypertree decompositions. Translated into our language of strategies this amounts to the following:

Definition 4.6. [32] A strategy (T, ρ, λ) is in *strong normal form* iff every node of T is *minimally labelled*. The root $O(T)$ is minimally labelled iff $|\lambda(O(T))| = 1$. For all $r \in T, s \in T(r)$, s is minimally labelled iff for all $e \in \lambda(s)$, either

- *Connectivity violation*: $\gamma(s) \not\subseteq \bigcup(\lambda(s) \setminus \{e\})$, or
- *Monotonicity violation*: $(\lambda(s) \setminus \{e\}) \cap \rho(s) = \emptyset$.

Lemma 4.1. [32] *For every strategy there exists a strategy in strong normal form.*

This characterization helps in the computation of hypertree decompositions, since it cuts down the search space for new separators. When looking for separators for a component, it is enough only to consider edges which are incident on its connecting set, and we only need to add one extra “component edge” when the edges covering the connecting set are all outside the component. Moreover, it is enough only to consider *minimal covers*, i.e. covers in which the removal of any edge would uncover a vertex in the connecting set.

Gottlob and Samer propose the following procedure to enumerate separators [32]: Given a component $comp$ with connecting set $comp.connVs$ and component edges $comp.compEs$,

1. compute the set of bounding edges

$$comp.boundEs := \{e \in E(H) \mid e \cap comp.connVs \neq \emptyset\};$$

2. remove any redundant edges from this set, i.e. all edges $e \in comp.boundEs \setminus comp.compEs$ for which there exists another edge $f \in comp.boundEs$ such that $e \cap comp.connVs \subseteq f$;

3. order the remaining (non-redundant) edges by the number of vertices an edge covers in *comp.connVs*;
4. compute all subsets of *comp.boundEs* of size at most k , preferring edges at the beginning of the list;
5. when adding a new edge to the subset, only allow it if it covers some yet uncovered vertex of the connecting set;
6. whenever we compute a subset of *comp.boundEs* that covers *comp.connVs* completely, which also includes an element of *comp.compEs*, we return it;
7. whenever we compute a subset of *comp.boundEs* that covers *comp.connVs* completely, but which does not include an element of *comp.compEs*, we successively add one edge from *comp.compEs* to the subset, and return these new separators one by one, if the size of the subset permits it (if it is at most $k - 1$); otherwise we just compute the next subset of *comp.compEs*;
8. whenever there are more uncovered vertices of *comp.connVs* than we could possibly cover with the remaining edges (i.e. because we know that they are all too small), we can stop the search of that particular branch early and look for a different subset.

This procedure has the advantage that we are more likely to cover the connecting set quickly by first trying the edges that cover more vertices in the connecting set.

Thus, the implementation of *Hypergraph.nextSeparator* used in [32] is presented in Algorithm 9 and Algorithm 10.

In particular, the method *Hypergraph.selectEdge(Component)* selects a new edge for the separator if a valid edge is available and returns false otherwise. The method

Algorithm 9 Enumerating Separators as in [32] — Part I

```

1: class Component {
2:   field compEs: List⟨Edge⟩
3:   field compVs: Set⟨Vertex⟩
4:   field connVs: Set⟨Vertex⟩
5:   field boundEs: List⟨Edge⟩
6:   field index: Integer := 1           ▷ this index goes through boundEs
7:   field compIndex: Integer := 1      ▷ this index goes through compEs
8:   field selectedEs: List⟨Edge⟩ := []
9:   constructor Component (es: Set⟨Edge⟩, vs: Set⟨Vertex⟩, cs: Set⟨Vertex⟩) {
10:    compEs := es
11:    compVs := vs
12:    connVs := cs
13:    boundEs := {e ∈ E(H) | e ∩ connVs ≠ ∅}
14:    while (there exist e, f ∈ boundEs such that e ∉ compEs and e ∩ connVs ⊆
15:      f ∩ connVs) {
16:      boundEs := boundEs \ {e}
17:    }
18:    sort boundEs by size of e ∩ connVs, for each e ∈ boundEs, decreasing
19:    coveredVs := []
20:  }
21:
22: method Hypergraph.nextSeparator(c: Component): Separator {
23:   if (|c.compEs| ≤ width) {           ▷ heuristic for very small components
24:    return new Separator(c.compEs, c.compEs, c.compVs)
25:   } else if (c.connVs = ∅ and |c.compEs| ≤ 2width) {   ▷ another heuristic
26:    set c.selectedEs to contain first k edges of c.compEs
27:    return new Separator(c.selectedEs, c.compEs, c.compVs)
28:   } else {
29:    repeat {
30:     if (not selectEdge(c)) {
31:      if (not removeEdge(c)) {
32:       return nil
33:      }
34:     }
35:   } until (c.connVs ⊆ ⋃ c.selectedEs and c.selectedEs ∩ c.compEs ≠ ∅)
36:   return new Separator(c.selectedEs, c.compEs, c.compVs)
37: }
38: }

```

Algorithm 10 Enumerating Separators as in [32] — Part II

```
39: method Hypergraph.selectEdge(c: Component): Boolean {
40:   if ( $|c.selectedEs| = width$  or ( $c.connVs \subseteq \bigcup c.selectedEs$  and  $c.selectedEs \cap c.compEs \neq \emptyset$ )) {
41:     return false
42:   }
43:   if ( $c.connVs \subseteq \bigcup c.selectedEs$ ) {
44:     while ( $c.compIndex \leq |c.compEs|$ ) {
45:       var e: Edge := c.compEs.elementAt(c.compIndex)
46:       c.compIndex := c.compIndex + 1
47:       c.selectedEs.append(e)
48:       return true
49:     }
50:     c.compIndex := 1 ▷ prepare for next time
51:     return false
52:   }
53:   while ( $c.index \leq |c.boundEs|$ ) {
54:     var e: Edge := c.boundEs.elementAt(c.index)
55:     c.index := c.index + 1
56:     if ( $|c.connVs \setminus \bigcup c.selectedEs| > |e \cap c.connVs| \times (width - |c.selectedEs|)$ ) ▷ heuristic to stop early
57:     {
58:       return false
59:     }
60:     if ( $(e \cap c.connVs) \not\subseteq (\bigcup c.selectedEs)$ ) {
61:       c.selectedEs.append(e)
62:       return true
63:     }
64:   }
65:   return false
66: }
67: method Hypergraph.removeEdge(c: Component): Boolean {
68:   if (c.selectedEs = []) {
69:     return false
70:   }
71:   var e: Edge := c.selectedEs.lastElement()
72:   index := (index of e in c.boundEs) + 1
73:   remove last element in c.selectedEs
74: }
```

Hypergraph.removeEdge(Component) removes the last selected edge from the separator such that we have more edges available to select from for the next candidate. The *index* and *compIndex* fields keep track of which edge was tried last.

One final remark is perhaps at place here: Gottlob and Samer propose to randomize the order of the edges and vertices in the hypergraph, such that the algorithm does not depend on a particular representation of the hypergraph. As a result, the running time of the algorithm will differ for every run on any particular hypergraph, and can in fact differ quite dramatically.

4.3.3 Improvements to det- k -decomp

There are few substantially different possibilities to implement the method *Hypergraph.nextComponent*.³ However, enumerating the separators for a component admits a variation to the algorithm proposed in [32], which we present in Algorithm 11, Algorithm 12, Algorithm 13, Algorithm 14 and Algorithm 15.

We avoid the need of computing the set of bounding edges altogether, which also eliminates the need of sorting that set. In fact, assuming that a *Hypergraph* object incorporates an “incidence list” for all vertices, i.e. for each vertex v a list of all edges e such that $v \in e$, we can sort each of these lists by the size of the edges in it in decreasing order. Thus we compute the heuristic of “big edges first” only once, rather than every time a new component is created. We assume that a *Hypergraph* object has a field called *vToEs*, which is an array of lists of edges (*vToEs*[v] is the list of edges containing vertex v). Of course, the reader might object that a big edge will not always be an edge that covers the most vertices in the connecting set for a particular component. However, sorting of bounding edges for each component was also only a heuristic, and in the worst case we will have to perform an exhaustive search anyway.

³ One possibility to improve performance marginally, depending on the language of implementation, is to transform the depth-first search into loop form together with a stack data-structure (unwinding the recursion).

Algorithm 11 Enumerating Separators (improved)

```
1: class Component {
2:   field compEs: List⟨Edge⟩
3:   field compVs: Set⟨Vertex⟩
4:   field connVs: Set⟨Vertex⟩
5:   field forbiddenEs: Set⟨Edge⟩ := ∅
6:   field index: Integer := 0      ▷ this index goes through vToEs[v] and also
   through compEs
7:   field oldIndex: Integer      ▷ used to restore index, when we remove an edge
8:   field selectedEs: List⟨Edge⟩ := []
9:   constructor Component (es: Set⟨Edge⟩, vs: Set⟨Vertex⟩, cs: Set⟨Vertex⟩) {
10:    compEs := es
11:    compVs := vs
12:    connVs := cs
13:    coveredVs := []
14:  }
15: }
16:
17: method Hypergraph.nextSeparator(c: Component): Separator {
18:   if ( $|c.compEs| \leq width$ ) {      ▷ heuristic for very small components
19:     return new Separator(c.compEs, c.compEs, c.compVs)
20:   } else if (c.connVs = ∅ and  $|c.compEs| \leq 2width$ ) {    ▷ another heuristic
21:     set c.selectedEs to contain first k edges of c.compEs
22:     return new Separator(c.selectedEs, c.compEs, c.compVs)
23:   } else {
24:     repeat {
25:       if (not coverVertex(c)) {
26:         if (not removeEdge(c)) {
27:           return nil
28:         }
29:       }
30:     } until (c.connVs ⊆ ⋃ c.selectedEs and c.selectedEs ∩ c.compEs ≠ ∅)
31:     return new Separator(c.selectedEs, c.compEs, c.compVs)
32:   }
33: }
```

Algorithm 12 *Hypergraph.coverVertex(Component)* — Part I

```
1: method Hypergraph.coverVertex(c: Component): Boolean {
2:
3:   ▷ if we cannot add more edges or already have a valid separator
4:   if ( $|c.selectedEs| = width$  or ( $c.connVs \subseteq \bigcup c.selectedEs$  and  $c.selectedEs \cap c.compEs \neq \emptyset$ )) {
5:     return false
6:   }
7:
8:   var e: Edge
9:
10:  ▷ if we are looking for additional component edges
11:  if ( $c.connVs \subseteq \bigcup c.selectedEs$ ) {
12:    repeat {
13:       $c.index := c.index + 1$ 
14:      if ( $c.index \leq |c.compEs|$ ) {
15:         $e := c.compEs.elementAt(c.index)$ 
16:      }
17:    } until ( $c.index > |c.compEs|$  or ( $e \notin c.forbiddenEs$ ))
18:    if ( $c.index > |c.compEs|$ ) {
19:      return false
20:    } else {
21:       $c.selectedEs.append(e)$ 
22:       $c.oldIndex := c.index$ 
23:      return true
24:    }
25:  }
26:
27:  ▷ continued on next page
28:  ...
29: }
```

Algorithm 13 *Hypergraph.coverVertex(Component)* — Part II

```
30: method Hypergraph.coverVertex(c: Component): Boolean {
31:   ...
32:   ▷ continuing from previous page
33:
34:   var v: Vertex := first vertex in c.connVs \ ( $\bigcup c.selectedEs$ )
35:
36:   ▷ if we are looking for a component edge covering v
37:   if (c.index = 0 or vToEs[v].elementAt(c.index) ∈ c.compEs) {
38:     for all (var ix: Integer ∈ [1 ... c.index] such that vToEs[v].elementAt(ix)
      ∈ c.compEs) {
39:       c.forbiddenEs := c.forbiddenEs ∪ {vToEs[v].elementAt(ix)}
40:     }
41:     repeat {
42:       c.index := c.index + 1
43:       if (c.index ≤ |vToEs[v]|) {
44:         e := vToEs[v].elementAt(c.index)
45:       }
46:     } until (c.index > |vToEs[v]| or (e ∈ c.compEs and e ∉ c.forbiddenEs))
47:     if (c.index ≤ |vToEs[v]|) {
48:       c.selectedEs.append(e)
49:       c.forbiddenEs := c.forbiddenEs ∪ {e}
50:       c.oldIndex := c.index
51:       return true
52:     } else {
53:       c.index := 1
54:     }
55:   }
56:
57:   ▷ continued on next page
58:   ...
59: }
```

Algorithm 14 *Hypergraph.coverVertex(Component)* — Part III

```
60: method Hypergraph.coverVertex(c: Component): Boolean {
61:   ...
62:   ▷ continuing from previous page
63:
64:   ▷ if we are looking for a non-component edge covering v
65:   for all (var ix: Integer ∈ [1...|vToEs[v]|] such that vToEs[v].elementAt(ix)
      ∈ c.compEs) {                                     ▷ forbid previously used component edges
66:     c.forbiddenEs := c.forbiddenEs ∪ {vToEs[v].elementAt(ix)}
67:   }
68:   for all (var ix: Integer ∈ [1...c.index] such that vToEs[v].elementAt(ix)
      ∉ c.compEs) {                                     ▷ forbid previously used non-component edges
69:     c.forbiddenEs := c.forbiddenEs ∪ {vToEs[v].elementAt(ix)}
70:   }
71:   repeat {
72:     c.index := c.index + 1
73:     if (c.index ≤ |vToEs[v]|) {
74:       e := vToEs[v].elementAt(c.index)
75:     }
76:   } until (c.index > |vToEs[v]| or (e ∉ c.compEs and e ∉ c.forbiddenEs))
77:   if (c.index ≤ |vToEs[v]|) {
78:     c.selectedEs.append(e)
79:     c.forbiddenEs := c.forbiddenEs ∪ {e}
80:     c.oldIndex := c.index
81:     return true
82:   }
83:   return false
84: }
```

Algorithm 15 *Hypergraph.removeEdge(Component)*

```
1: method Hypergraph.removeEdge(c: Component): Boolean {
2:   if (c.selectedEs = []) {
3:     return false
4:   }
5:   remove last element in c.selectedEs
6:   c.index := c.oldIndex
7:   c.forbiddenEs :=  $\emptyset$ 
8:   for all (var ix: Integer  $\in [1 \dots |c.selectedEs|]$ ) {
9:     var e: Edge := c.selectedEs.elementAt(ix)
10:    var v: Vertex := first vertex in c.connVs  $\setminus \bigcup c.selectedEs[1 \dots (ix - 1)]$ 
11:    c.oldIndex := index of e in vToEs[v]
12:    if (e  $\in c.compEs$ ) {
13:      for all ( $1 \leq ixx \leq c.oldIndex$  such that vToEs[v].elementAt(ixx)
```

$\in c.compEs$) {

```
14:        c.forbiddenEs := c.forbiddenEs  $\cup \{vToEs[v].elementAt(ixx)\}$ 
15:      }
16:    } else {
17:      for all ( $1 \leq ixx \leq |vToEs[v]|$  such that vToEs[v].elementAt(ixx)
```

$\in c.compEs$) {

```
18:        c.forbiddenEs := c.forbiddenEs  $\cup \{vToEs[v].elementAt(ixx)\}$ 
19:      }
20:      for all ( $1 \leq ixx \leq c.oldIndex$  such that vToEs[v].elementAt(ixx)
```

$\notin c.compEs$) {

```
21:        c.forbiddenEs := c.forbiddenEs  $\cup \{vToEs[v].elementAt(ixx)\}$ 
22:      }
23:    }
24:  }
25: }
```

Instead of trying to select an edge from the set *comp.boundEs* and then testing whether it covers any new vertices in *comp.connVs*, we instead look for a vertex in *comp.connVs* which is uncovered, and then successively select edges incident on that vertex for our set *comp.selectedEs*. Moreover, as an additional heuristic, we prefer edges which are also contained in *comp.compEs*. We thus first construct those separators that cover as much of the component as possible.

To avoid any repetitions we require an additional field, *comp.forbiddenEs*, and two index variables instead of one, which both index a list in *Hypergraph.vToEs*. An edge *e* is forbidden if we are considering a configuration in which we covered a vertex *v* with an edge *f*, and we have already tried to cover *v* with *e* earlier. Also, an index will now point to the position of the last edge selected (in *vToEs[v]* or in *comp.compEs*), rather than the position after it. By using the *comp.forbiddenEs*, we can now avoid any possible repetition of separators, since additional component edges can now only be selected if they are not forbidden.

After removing an edge from *comp.selectedEs* we have to update *comp.forbiddenEs* and *comp.oldIndex*. To do that, we have to run through every edge still left in *comp.selectedEs* and repeat the steps that we originally performed while selecting those edges. Unfortunately, there is no way to derive that information from the removed edge alone, unless we use additional data-structures to store that information. We decided not to do this out of consideration for space usage.

4.3.4 Adding Output and Decreasing Space Requirements

As the reader might have noticed, we omitted any output in *det-k-decomp*. The algorithm merely accepts or rejects, depending on the existence of a hypertree decomposition of specified width.

However, by modifying the dynamic table only slightly, it is very easy to store sufficient information for a hypertree decomposition to be recovered. Rather than

simply storing SUCCESS / FAILURE as the values for a component in the table, we can also store the separator that was actually used to reach a successful decomposition (in the case of SUCCESS). Thus, after the algorithm terminates successfully, we will have enough information in the table to reconstruct the hypertree decomposition: Starting from the first component (which includes all edges), we simply look up the separator that leads to a successful decomposition, use that separator to divide the component into subcomponents, and then apply the same procedure for each subcomponent.

To decrease the space required for the table, we need only to store enough information to uniquely identify the components and separators used. Thus, for a component it is enough to store e.g. the connecting set and a vertex inside the component (the vertex with the lowest (integer) label).⁴ For a separator it is enough to store the separator edges. We will use objects of type *ComponentID* and *SeparatorID* to denote such “reduced” representations, and we assume that we have methods *Component.getID()* and *Separator.getID()* that return the corresponding representation objects.

Assuming that *det-k-decomp* terminated successfully, we can use the procedure *OutputHypertree* with the initial component as parameter to output the hypertree decomposition. This function is presented in Algorithm 16.

Algorithm 16 *det-k-decomp* — Output

```

1: procedure OutputHypertree(comp: Component) {
2:   var sid: SeparatorID := table.value(comp.getID())
3:   var s: Separator := new Separator(sid.edges, comp.compEs, comp.compVs)
4:   output “(“
5:   while ((var c: Component := h.nextComponent(s)) ≠ nil) {
6:     OutputHypertree(c)
7:   }
8:   var chi: Set⟨Vertex⟩ := ( $\bigcup$  comp.compEs) ∩ ( $\bigcup$  s.sepEs)
9:   output “, “ + chi + “, “ + s.sepEs + “)”
10: }
```

⁴ Alternatively, one could store e.g. the “old separator” and an edge inside the component.

In the remainder of this dissertation we will omit any further discussion of output, since it can be easily added to all the deterministic algorithms using a dynamic table.

4.3.5 Issues with Parallelization

It is difficult to conceive any good parallel algorithm from scratch. Thus, parallel algorithms are usually derived from sequential algorithms, and good parallel algorithms are usually derived from good sequential algorithms.

There are algorithms, like McMahan’s heuristic approach based on *Bucket Elimination* [40], that perform better than $\text{det-}k\text{-decomp}$ when looking for decompositions of non-minimal width. However, when looking for decompositions of *minimal* or close to minimal width, $\text{det-}k\text{-decomp}$ outperforms it. Another algorithm computing decompositions of minimal width is $\text{opt-}k\text{-decomp}$ [26, 38, 49], however, it uses a bottom-up approach requiring a lot of memory. It is also much slower than $\text{det-}k\text{-decomp}$. Thus, if the width of a hypertree decomposition is the deciding factor, $\text{det-}k\text{-decomp}$ is the best available algorithm to date. Our parallel algorithm will therefore be based on $\text{det-}k\text{-decomp}$.

When deriving parallel programs from sequential ones, there are generally many choices for which parts to parallelize. For example, if a program has two nested loops, it might be possible to parallelize the inner or the outer loop. The right choice usually depends on the program itself, the data structures it uses and sometimes also on the target architecture (the machine the program will run on). As a general rule of thumb, we can say, that parallelization works best when the data structures affected by each thread of execution are disjoint, and there are as few *synchronization points* as possible. Thus, a loop iterating over an array works best when the result of one element does not depend on the other elements, and also when the array is very large or when each repetition of the loop takes a long time.

In any case, parallelization should affect the *critical* portions of the program, i.e. those portions which take up most of the running time.

Profiling *det-k-decomp* revealed that the program spends most of its running time executing the methods *Hypertree.coverVertex* and *Hypertree.removeEdge*, i.e. computing new separators for a component.⁵ Thus, one immediate thought would be to parallelize a single execution of either of these methods (parallelization on a *small scale*). This is a bad idea for two reasons:

1. These methods are executed very frequently. Thus, the overhead linked to setting up and deleting threads before and after each execution would be rather large.
2. Each of these methods iterates over relatively small arrays, each repetition is very quick, and the results may depend on previously computed results. Thus it will be hard to get any significant increase in speed, and the overhead for possible synchronization issues will be very large.

Luckily, we have another obvious approach to parallelization, due to the fact that *det-k-decomp* is derived from an alternating algorithm. Each *Component* object has the existential choice of a *Separator*, while every *Separator* object is required to decompose all of its resulting *Components*. We could, in fact, separate the function *decomposable* into two different functions, one for components and one for separators. This is suggested in Algorithm 17.

For convenience, we omit the use of the table for now (it can easily be added to *decomposable1*).

⁵ If a decomposition exists, *Hypertree.nextSeparator* takes up about 95% of the total execution time, while *Hypertree.coverVertex* takes up about 70% – 80% and *Hypertree.removeEdge* about 15% – 25% of the total execution time, depending on the hypergraph. If no decomposition is found, however, a significant portion of total execution time (up to 40%) is spent on *Hypergraph.nextComponent*, performing the depth-first search.

Algorithm 17 *decomposable* separated into *decomposable1* and *decomposable2*

```
1: function decomposable1(h: Hypergraph, comp: Component): Boolean {
2:   while ((var sep: Separator := h.nextSeparator(comp)) ≠ nil) {
3:     if (decomposable2 (h, sep)) {
4:       return true
5:     }
6:   }
7:   return false
8: }
9:
10: function decomposable2(h: Hypergraph, sep: Separator): Boolean {
11:   while ((var newComp: Component := h.nextComponent(sep)) ≠ nil) {
12:     if (not decomposable1 (h, newComp)) {
13:       return false
14:     }
15:   }
16:   return true
17: }
```

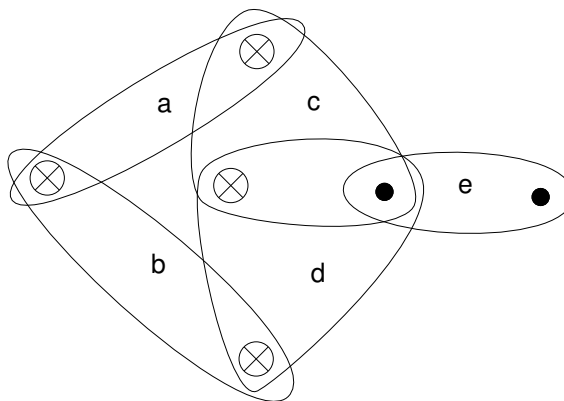
Now, each of the functions *decomposable1*, *decomposable2* contains a loop. This loop cannot be parallelized directly, since the object it works on is mutable, and every new loop execution relies on the new state of that object after the completion of the previous loop execution. However, remember that for a *decomposable1* to succeed, it is enough for **one** of the separators to lead to a decomposition, while for *decomposable2* to fail, it is enough for one of the components to fail. Thus, in principle, we don't have to wait for one separator before trying the next one (similarly for components).

Even though we cannot parallelize the while-loops themselves, we can parallelize the “recursive calls” of the functions. As soon as the loop in *decomposable1* computes a new separator, we can launch a thread executing *decomposable2* on that separator, while the loop carries on computing more separators. Similarly, as soon as the loop in *decomposable2* computes a new component, we can launch a thread executing *decomposable1* on that component, while the loop carries on computing more components. Of course, we would have to keep track of which threads are waiting on which threads' results.

Let us examine this sort of parallelization for each function call independently. Parallelizing recursive calls to *decomposable1* within *decomposable2* is very attractive in deed: Since a separator divides the old component into (possibly several) new components, which are all disjoint, the work carried out from here by each thread will be completely independent from the other threads. Unfortunately, for most hypergraphs a new separator will very often simply reduce the old component in size, rather than properly dividing it into several components.⁶ Hence, this sort of parallelization is very attractive in principle, but would help only in very few instances, without changing the original algorithm dramatically.

Now, parallelizing the recursive calls to *decomposable2* within *decomposable1* does not have the problem of low branching: For any component, there will generally be many different ways to select a separator covering the connecting set. The problem here is of a different nature.

Consider for instance the component C in Figure 4.1



Vertices marked $\otimes$ are the connecting set.

Figure 4.1: Component C

There are two possibilities to select a separator of size 2 covering the connecting set: either $\{a, d\}$ or $\{b, c\}$. Both separators, however, lead to the same new component $\{e\}$.

⁶ This is also the reason that most hypertrees are long and deep with relatively low branching.

Let us say we start a new thread for each of these separators, and both threads eventually proceed to this component. Even if we were using the dynamic table to store known results, the result for that component will not be known until either of the threads finishes decomposing it. Therefore neither thread will know that another thread is repeating its work.

The above example is not a solitary instance. In fact, the same components will very often be reached from some “parent component” via different separators, and usually also via several intermediate components and separators.⁷ Thus, it makes more sense to wait for a recursive call of *decomposable2* to return, before calling it again on a different separator.

We can thus summarize:

- Parallelization at Separators (which correspond to *universal states* of the ATM) is *good*, such states will generally have few children
- Parallelization at Components (which correspond to *existential states* of the ATM) is *bad*, even though these states will generally have many children.

Thus, even if we only parallelize at separators, the parallel speed-up will be minimal (generally less than factor 2), and will not scale to a large number of processors. If we only parallelize at components, and proceed to operate sequentially once all available processors are busy, the threads will generally be working completely independently of one another sharing little useful information, possibly repeating work twice or even more times, and a solution will be returned once the first thread finds a solution (independently from the other threads). Even parallelizing at both separators **and** components will yield little improvement, since the threads won’t be “cooperating” in a coordinated manner.⁸

⁷ That is precisely the reason why we use a dynamic table.

⁸ These statements rely on actual testing of early attempts at parallelization of det-*k*-decomp. We do not include the description of these programs or their test results due to space limitations in this dissertation.

Thus, in principle this would be little more than simply starting the same program with possibly different random seeds on different machines and stopping when the first machine stops. This approach is used in practice for several computational problems, for instance there is a SAT-solver using this approach and extending it with communication between the machines at certain time intervals. However, these problems are NP-complete and thus P-hard, and hence generally not easily parallelizable. This is somewhat of a desperate last-resort approach at parallelizing randomized programs of problems which are believed not to admit efficient parallelization.

However, we know that the problem of finding hypertree decompositions is in LOGCFL, which lies within AC^1 and NC^2 and is thus “highly parallelizable”. In principle, if we had enough electric gates and wires available, we could create an electric circuit solving our problem for hypergraphs of size at most n and decomposition widths at most k in time $O(\log^2 n)$, even if this means reducing the ATM representation of k -decomp to a *Boolean Circuit*. In fact, we could use threads to emulate the electric gates and create a generic algorithm for a hypergraph of any size and a decompositions of any width.

However, this approach is also impractical, since the boolean circuit itself will be very large (even though still polynomial), even for small n and k , requiring a large number of threads to emulate it — a lot more than we will have processors available in practice (a constant number for most universities).

Thus, a logical conclusion is to look for a different alternating algorithm which has better branching at universal states, while still ensuring that the results of the children of a single universal state are completely independent of one another. Ideally, we would like to have something resembling the algorithm k -robber-sergeants of Section 3.7.

4.4 A Divide and Conquer Algorithm

Taking inspiration from Chapter 3, we want to build a decomposition in a “balanced” manner. A question arises immediately: Is there a way to derive hypertree decompositions from balanced or shallow decompositions? Unfortunately, as was demonstrated in Chapter 3, the balanced width is generally smaller than the generalized hypertree width and hence the hypertree width of a hypergraph. Also, in a hypercut decomposition, the separator only needs to separate the remaining edge-component, without having to worry about connectivity to any “outside edges”. Our goal in this section will be to find separators which are (almost) balanced for a component, and which do not allow the new subcomponents to “reconnect” with any “outside edges”.

When examined carefully, this corresponds to guessing a node of a decomposition in advance, or guessing a future position in the R&M^k game that can be reached from a current position, rather than computing a decomposition node by node, i.e. playing R&M^k step by step. Moreover, given a current position and some future position which can be reached from the current position, we will try to guess an intermediate position *en route* to that future position. For this, we will need several results.

First, we assume that the strategies we use from now on are all *complete*, (see Remark 4.1).

Definition 4.7. Let H be a hypergraph, and let (T, ρ, λ) be a monotone strategy. We define the auxiliary labelling function $\beta : T \rightarrow \mathcal{P}(V(H))$ by

$$\beta(r) = \bigcup_{s \in T(r)} \gamma(s)$$

We call $\beta(r)$ the *lower connecting set at r* .

Definition 4.8. Let H be a hypergraph. Let $C, D \subseteq V(H)$. A *separator for (C, D)* is $T \subseteq E(H)$ such that for all $c \in C, d \in D$, c is not $[T]$ -connected to d .

Lemma 4.2. *Let H be a hypergraph, k a positive integer, and let (T, ρ, λ) be a complete monotone strategy. Let $r \in T, t \in T_r$. Then for each $s \in T_r$ such that $t \in T_s$, $\lambda(s)$ is a separator for $(\gamma(r), \beta(t))$.*

Proof. First, if $s = r$, then $\gamma(r) \subseteq \bigcup \lambda(s)$ and $\lambda(s)$ is obviously a separator for $(\gamma(r), \beta(t))$. Similarly, if $s = t$, then $\beta(t) \subseteq \bigcup \lambda(s)$ and again, $\lambda(s)$ is a separator for $(\gamma(r), \beta(t))$.

Second, if $r = O(T)$ and hence $\gamma(r) = \emptyset$, then any set of edges, in particular $\lambda(s)$ is a separator for $(\gamma(r), \beta(t))$. Similarly, if $T(t) = \emptyset$ and hence $\beta(t) = \emptyset$, then also any set of edges, in particular $\lambda(s)$ is a separator for $(\gamma(r), \beta(t))$.

Now suppose $r \neq s \neq t$, $r \neq O(T)$ and $T(t) \neq \emptyset$. Suppose $\lambda(s)$ is not a separator. Then there exist vertices $u \in \gamma(r), w \in \beta(t)$ which are $[\lambda(s)]$ -connected. Since $u \in \gamma(r)$, $u \subseteq \bigcup \lambda(r')$, where $r' \in T(r')$, and hence $u \notin \rho_V(r)$, and by vertex monotonicity $u \notin \rho_V(x)$ for all $x \in T_r$.

Let $e \in \rho(t)$ such that $w \in e$. Such an e must exist, since $w \in \beta(t)$ and $\beta(t)$ is a lower connecting set. By monotonicity, $e \in \rho(s)$, and since $w \notin \bigcup \lambda(s)$, and hence also $e \notin \lambda(s)$, by strong completeness, there exists $s' \in T(s)$ such that $e \in \rho(s')$. Now, since $w \notin \bigcup \lambda(s)$ and $w \in e$, we must have $w \in \rho_V(s')$.

But since u and w are $[\lambda(s)]$ -connected and $w \in \rho_V(s')$, we must also have $u \in \rho_V(s')$ by vertex connectedness, contradicting the fact that u is not in the vertex escape space of any descendant of r . $\square$

Corollary 4.3. *Let H be a hypergraph, k a positive integer and $C, D \in V(H)$. If there is no edge-separator for (C, D) of size at most k , then there is no monotone strategy (T, ρ, λ) with nodes $r \in T, t \in T_r$ such that $\gamma(r) = C$ and $\beta(t) = D$.*

This gives us a way to test whether two nodes r and t , or rather their γ - and β -labels, respectively, are feasible in a hypertree decomposition, by checking whether there exists a separator of size k for them.

Note, that if t is a *capture node*, i.e. if $T(t) = \emptyset$ and hence $\beta(t) = \emptyset$, or if $r = O(T)$ and $\gamma(r) = \emptyset$, then any set of edges (including the empty set) is a separator for $(\gamma(r), \beta(t))$.

4.4.1 The Alternating Algorithm k -divide-decomp

The results of the previous section give us a way to construct an alternating algorithm that constructs a hypertree by guessing future nodes in the strategy tree, rather than just guessing the immediate descendants of any one node. For every two nodes, with one a descendant of the other, it then has to guess an intermediate node, and proceed in a recursive manner. Once we guessed an intermediate node s between r and t , we know $\lambda(s)$ and can also derive $\beta(s)$. We cannot be certain what $\rho(s)$ or $\gamma(s)$ is until we finish building the decomposition between r and s . Also, there is a possibility that s will have several children (there will be several $[\lambda(s)]$ -components), with only one of them having t as a descendant. We also need to account for the cases when we only have an “ancestor node” (r) and no “descendant node” (t). (We eliminate the converse possibility with a “descendant” but no “ancestor node” by requiring the algorithm to always guess the root node first).

To make the reasoning easier, we introduce some additional notation.

Definition 4.9. Let H be a hypergraph and (T, ρ, λ) a monotone strategy. We define the auxiliary labelling function $\tau : T \rightarrow \mathcal{P}(E(H))$ by

$$\tau(r) = E(H) \setminus \left(\bigcup_{s \in T(s)} \rho(s) \right)$$

Remark 4.3. Note that $\tau(r)$ will in many cases be a $\beta(r)$ -edge-component, but it is possible that $\tau(r)$ will consist of several such disconnected edge-components.

Definition 4.10. Let H be a hypergraph and (T, ρ, λ) a monotone strategy. For two nodes $r \in T, s \in T_r$, we define

$$[r, s] = \rho(r) \cap \tau(s)$$

The idea of this definition is that $[r, s]$ is the “intermediate component” with $\gamma(r)$ as the upper connecting set and $\beta(s)$ as the lower connecting set. Of course, $[r, s]$ will often be a $[\gamma(r) \cup \beta(s)]$ -edge-component, however it can also consist of several such disconnected edge-components.

Proposition 4.10. *Let H be a hypergraph and (T, ρ, λ) a monotone strategy. Then for any node $r \in T$, $[r, r] \subseteq \lambda(r)$.*

Proof. This is obvious by strong completeness. □

Proposition 4.11. *Let H be a hypergraph and (T, ρ, λ) a monotone strategy. Let $r \in T, s \in T_r, t \in T_s, u \in T_t$. Then $[r, t] \cap [s, u] = [s, t]$.*

Proof. $[r, t] \cap [s, u] = \rho(r) \cap \tau(t) \cap \rho(s) \cap \tau(u)$. But $\rho(s) \subseteq \rho(r)$ and $\tau(t) \subseteq \tau(u)$. So

$$[r, t] \cap [s, u] = \rho(s) \cap \tau(t) = [s, t]$$

□

An intermediate component $[r, t]$ can be uniquely represented by $\lambda(q)$ and $\lambda(t)$, where $r \in T(q)$. At least one edge of $\lambda(t)$ will have to be in $\rho(r)$, say e , which can be marked as a special edge in $\lambda(t)$. Given e and $\lambda(q)$, we can check whether any given edge is in $\rho(r)$. Furthermore, given an edge $f \in \rho(r)$, we can test whether f is in $[r, t]$ by testing whether it is in $\lambda(t) \cap \rho(r)$, or if not, by testing whether there exists an edge-path from f reaching some edge in $\lambda(q)$ without having to pass through any vertices covered by $\lambda(t)$. These procedures all rely on undirected st -connectivity and hence are feasible in deterministic LOGSPACE [45].

For convenience, we will include $\gamma(r), \beta(t)$ and $[r, t]$ when specifying such an intermediate component. Given such a component, Lemma 4.2 tells us how we can guess an intermediate node s by guessing a separator $\lambda(s)$ for $(\gamma(r), \beta(t))$, and having guessed such a node, we will have at least two new components $[r, s]$ and $[s', t]$,

where $s' \in T(s)$ such that $t \in T_{s'}$, and possibly several other components $\rho(u)$ where $u \in T(s)$. Provided each of these new components admits a decomposition, then $[r, t]$ also admits a decomposition with s as an intermediate node. If at some point we can guess a separator $\lambda(s)$ such that $[r, t] \subseteq \lambda(s)$, then we can set $r = s = t$, and we have a valid decomposition of $[r, t]$.

There is just one small problem with guessing intermediate nodes, which we still need to address: Just guessing descendant nodes rather than successor nodes does not cut down the total amount of work, since the same number of nodes have to be guessed. In fact, since every successor node is also a descendant, if we guess badly, we would just repeat what k -decomp does, and there is no gain. On the other hand, if we guess well, then we divide the “branch under consideration” rather evenly.

Let us formalize this. There are two cases we need to consider.

- We are decomposing a component $\rho(r)$ with just a single upper connecting set $\gamma(r)$.
- We are decomposing a component $[r, t]$ with the upper connecting set $\gamma(r)$ and the lower connecting set $\beta(t)$.

We do not consider the possibility of only a lower connecting set without an upper connecting set, since our algorithm will always proceed in a top-down manner, always requiring for an upper connecting set to exist.

Let us look at the first case.

Proposition 4.12. *Let H be a hypergraph and (T, ρ, λ) be a monotone strategy. Let $r \in T$. Then there exists $s \in T_r$, such that $||[r; s]|| \leq |\rho(r)|/2$ and for any $t \in T(s)$, $|\rho(t)| \leq |\rho(r)|/2 + k$.*

Proof. Let $s \in T_r \in T(s)$, such that $||[r; s]|| \leq |\rho(r)|/2$ and for all $s' \in T(s)$, $||[r; s']|| > |\rho(r)|/2$. It is easy to see that such an s always exists, by monotonicity. Since

$[r; s'] \cap \rho(s') = [s'; s']$, and $[s'; s'] \subseteq \lambda(s')$ and since $||[r; s']|| > |\rho(r)|/2$, we have $||[r; s'] \setminus \rho(s')|| > |\rho(r)|/2 - k$, and hence also $|\rho(s')| \leq |\rho(r)|/2 + k$. $\square$

So for the case that we do not have a lower connecting set to worry about, we can always find “balanced separators”, and we are left with at least two components that we need to decompose, each about half as big as the original component. It is possible that we will get more than two components (which is a good thing), in which case each of them again will have the same size restriction. On the other hand, if no such balanced separator of size k exists, then we are certain, that there is no strategy for the marshals to cover the escape space $\rho(r)$.

Now let us look at the second case.

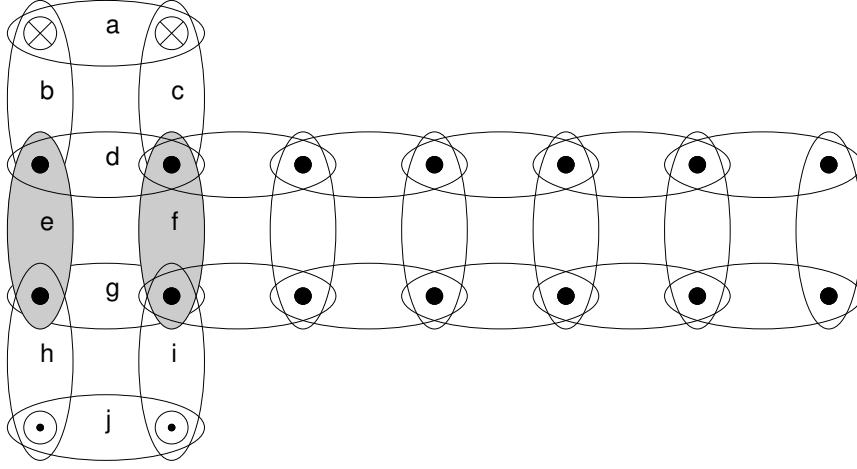
Proposition 4.13. *Let H be a hypergraph and let (T, ρ, λ) be a monotone strategy. Let $r \in T, t \in T_r, t \neq r$. Then there exists $s \in T_r$ such that $t \in T_s$ and $||[r; s]|| \leq ||[r; t]||/2$ and, moreover, for $s' \in T(s)$ such that $t \in T_{s'}$ we have $||[s'; t]|| \leq ||[r; t]||/2 + k$.*

Proof. (Sketch) Let $r = s_0, s_1, \dots, s_n = t$ be the descendants of r in the decomposition starting at r and finishing at t . We have for every i , that $[r; s_i] \subset [r; s_{i+1}]$. There has to be a least j such that $||[r; s_j]|| > ||[r; t]||/2$. Set s to s_{j-1} and $s' = s_j$. Then $||[r; s]|| \leq ||[r; t]||/2$.

Since $[r; s'] \cap [s'; t] \subseteq \lambda(s')$ and $||[r; s']|| > ||[r; t]||/2$, we have $||[r; s'] \setminus [s'; t]|| > ||[r; t]||/2 - k$, and hence also $||[s'; t]|| \leq ||[r; t]||/2 + k$. $\square$

So if we do have a lower connecting set and are trying to decompose the component $[r, t]$, then there will be a separator $\lambda(s)$ such that $[r, s]$ and $[s', t]$ are each at most about half as big as $[r, t]$. $\lambda(s)$ may not be balanced in a proper sense, though, as Figure 4.2 illustrates.

After using $\lambda(s)$ as a separator, even though both $[r, s]$ and $[s', t]$ end up being quite small as compared to $[r, t]$, we get another large component, $\rho(u)$ “dangling” on the side. Still, it is obvious that $|\rho(u)| < ||[r; t]||$, and $\rho(u)$ does not have any lower



Vertices marked $\otimes$ are in $\gamma(r)$, vertices marked $\odot$ are in $\beta(t)$, $\lambda(s) = \{e, f\}$ (shaded edges). “Upper component” is $[r, s] = \{b, c, d, e, f\}$. “Lower component” is $[s', t] = \{h, i, j\}$. One “side component” is $\{g\}$, the other consists of all unnamed edges.

Figure 4.2: Non-balanced Separator

connecting set, so at the next step it will be decomposed into at least two components at most about half as big $||[r; t]||$.

We can now present the alternating algorithm k -divide-decomp, as Algorithm 18. We will come back to the function *separate* in the next section. For now, it is enough for us to know that it produces the desired components together with their respective upper and lower connecting sets. Moreover, if the separator happens to cover the whole component, i.e. if we are separating $[r, r]$ or $\rho(r)$ that can be covered as a whole, then it returns the empty set.

As mentioned before, every component can be uniquely identified by only $2k$ edges. Hence the algorithm can be implemented on an ATM using space $O(\log n)$, where $n = |E(H)|$. Moreover, there are at most n^{2k} distinct components possible ($2k$ being the total number of edges in the lower and upper λ -labels). Hence, any accepting computation tree will have polynomial size, so k -divide-decomp is feasible in LOGCFL.

This is nothing new, since we already had a LOGCFL alternating algorithm before. But, and this was the point of this whole section, we can now state the following:

Algorithm 18 k -divide-decomp

```
1: type Component = (up: Set⟨Vertex⟩, edges: Set⟨Edge⟩, low: Set⟨Vertex⟩,)
2: var h: Hypergraph
3: var k: Integer
4: input h
5: input k
6: decomposable(( $\emptyset$ , h,  $\emptyset$ ))
7: accept
8:
9: procedure decomposable(comp) {
10:   var separator: Set⟨Edge⟩
11:   guess separator
12:   if ( $|separator| > k$  or not separates(comp.up, comp.low, separator)) {
13:     reject
14:   }
15:   var components: Set⟨Component⟩
16:   components := separate(comp, separator)
17:   for all (c ∈ components) {
18:     if (comp.low =  $\emptyset$ ) {
19:       if ( $|c.edges| > |comp.edges|/2 + k$ ) {
20:         reject
21:       }
22:     } else {
23:       if (c.low ≠  $\emptyset$  and  $|c.edges| > |comp.edges|/2 + k$ ) {
24:         reject
25:       }
26:     }
27:     decomposable(c)
28:   }
29: }
30:
31: function separate(C: Set⟨Vertex⟩, D: Set⟨Vertex⟩, S: Set⟨Edges⟩): Boolean {
32:   ▷ checks whether S separates (C, D)
33: }
34: function separate(comp: Component, sep: Set⟨Edge⟩): Set⟨Component⟩ {
35:   ▷ separates a component comp using separator sep
36:   ▷ returns the set of separated components (with their connecting sets)
37: }
```

Theorem 4.1. *In any accepting computation tree of k -divide-decomp there are at most $O(\log n)$ alternations.*

Proof. A computation branch terminates whenever we reach a component *comp* that is fully covered by the separator *separator*. Since we are dealing with complete strategies, this happens only when $|comp.edges| \leq k$. It is therefore enough to analyze how many alternations (which is twice the number of recursive calls to *decomposable*) it takes for a component to reach this size.

Let $e \in E(H)$. Let c_i be the component containing e after i recursive calls to *decomposable*. Then $|c_0.edges| = n$. If $c_j.low = \emptyset$, then $|c_{j+1}.edges| \leq |c_j.edges|/2 + k$, so certainly $|c_{j+2}.edges| < |c_j.edges|/2 + k$. If $c_j.low \neq \emptyset$, then either $c_{j+1}.low \neq \emptyset$ and $|c_{j+1}.edges| \leq |c_j.edges|/2 + k$, and hence $|c_{j+2}.edges| < |c_j.edges|/2 + k$, or $c_{j+1}.low = \emptyset$, in which case $|c_{j+1}.edges| < |c_j.edges|$, and hence $|c_{j+2}.edges| < |c_j.edges|/2 + k$. So in either case, after two steps the component shrinks from size x to at most $x/2 + k$. It is a basic exercise to check that after $s = 2 \log n$ steps, we must have $|c_s.edges| \leq 2k$. A component this small can be decomposed in $k = O(1)$ steps.

Hence, the maximal depth of recursion in k -divide-decomp and hence the maximal number of alternations is $O(\log n)$. $\square$

As the reader may have noticed, we omitted any output in k -divide-decomp. Adding output is slightly trickier in this instance to create proper hypertrees, since if we simply output the separators according to the order of the recursive calls, we will actually get something closer to a shallow or balanced decomposition. However, it is not very hard to see that such output could be transformed in LOGSPACE into a proper hypertree.⁹

A perhaps non-trivial corollary to Theorem 4.1 is the existence of shallow hypertree decompositions:

⁹ We omit further discussion of this matter, due in part to space considerations, and in part to the fact that we do not actually use this algorithm in any real-life implementation.

Theorem 4.2. *Let H be a hypergraph and k a positive integer. If there is a hypertree decomposition of width k , then there is a hypertree decomposition of width $3k$ and depth at most $O(\log |E(H)|)$.*

Proof. (Sketch.) The basic idea relies on the fact that given a strategy (T, ρ, λ) , we can construct a “shallow” hypertree decomposition by following the “path” of the algorithm k -divide-decomp, and using the λ -label $\lambda(r) \cup \lambda(t) \cup \lambda(s)$ to separate the component $[r; t]$. The new components will be $[r; s]$, $[s', t]$, where $s' \in T(s)$ and possibly some other components $\rho(u)$, where $u \in T(s)$. Checking the hypertree conditions for the resulting hypertree is a mere technicality.

The fact that $[r; t]$ might not be a connected edge-component is not a problem: The derived hypertree decomposition might not be in normal form, but it will still be a valid hypertree decomposition. Alternatively, we can use the same separator for each connected part of $[r; t]$. This will give hypertree decompositions that are in normal form, but possibly not in strong normal form. $\square$

4.4.2 A Deterministic $O(\log^2 n)$ -space algorithm

The algorithm k -divide-decomp can be made deterministic with a few adjustments. In particular, instead of guessing a separator, a deterministic algorithm would enumerate all possible subsets of edges to be tried as separators. Now, instead of rejecting whenever a separator fails to either separate the upper and lower connecting sets, or whenever one of the subcomponents is too big, it would simply backtrack and try out the next separator.

We mentioned in the previous subsection that we only require space $O(\log n)$ to represent any single component or separator. Moreover, the functions *separate* and *separates* could all be implemented in deterministic LOGSPACE. Even if they were implemented in NL, we could still run them on a $O(\log^2 n)$ -space deterministic machine, by Savitch’s Theorem [48].

Hence, a single execution of the deterministic version of *decomposable* in k -divide-decomp would need space $O(\log n)$ on the “stack”, plus $O(\log^2 n)$ or $O(\log n)$ space for the auxiliary functions. By Theorem 4.1, there are at most $O(\log n)$ alternations in any accepting computation tree of k -divide-decomp, which corresponds to at most $O(\log n)$ recursive calls of *decomposable*. Thus, the total space required by the algorithm is $O(\log^2 n)$ for the “stack” plus $O(\log^2 n)$ or $O(\log n)$ for the auxiliary functions. Hence, the deterministic equivalent of k -divide-decomp is in $\text{DSpace}(\log^2 n)$.

If we removed the space restriction, we could store all edges of a component, and thus cut down the search space for a separator. In particular, when decomposing a component $[r, t]$ it is easy to see that for any intermediate node s we have $\lambda(s) \subseteq (\rho(r) \cap \tau(t)) \cup B$, where $B = \{e \in E(H) \mid e \cap \gamma(r) \neq \emptyset\}$ is the set of bounding edges for r . This is of course assuming that our strategies are in strong normal form and do not include any “useless” separator edges.

We can also briefly outline the workings of the function *separate*.

Since we are now dealing with complete decompositions, every edge of a component will become part of a new subcomponent, unless the separator covers **all** the edges of the component. In the latter case *separate* returns the empty set, and *decomposable* terminates successfully for that component.

Suppose now that the component is represented by upper connecting set C , lower connecting set D and edge set E , and that the separator consists of the edges S such that $E \not\subseteq S$.

If for some edge $e \in E$ we have $e \subseteq \bigcup S$ and $e \notin S$, then e becomes a new component with upper connecting set e and lower connecting set $\emptyset$. We update E by removing these new edges.

To build the subcomponent between C and $\bigcup S$ we do the following, for every vertex in $v \in C$:

1. add all edges $e \in E$ incident on v ;

2. perform a depth-first search from each such e , at every step adding all still available $f \in E$ which are $[S]$ -adjacent to e ;
3. whenever we add an edge e , we remove it from E .

Finally, we also add all edges $e \in E \cap S$.

To build the subcomponent between $\bigcup S$ and D we run a depth-first search starting from every vertex $v \in D$, adding $[S]$ -adjacent edges at every step from the remainder of E .

Finally, if there are still edges remaining in E , we can build one or several more $[S]$ -components by running a depth-first search from each such remaining edge in E , adding $[S]$ -adjacent edges at every step.

Thus, every edge $e \in E$ will now be part of some new subcomponent.

4.4.3 Making the Algorithm Polynomial and Parallel

Again, the deterministic version of k -divide-decomp is due to repeat work. It is easy to see that even starting with two different components, we can eventually arrive at trying to decompose the same component, provided that the original components were overlapping. Thus, a lot of work could be repeated, and the algorithm would be super-polynomial.¹⁰

However, we can use the same idea as for $\text{det-}k\text{-decomp}$ to bound the total amount of work that can be done, by having a dynamic table that stores any partial results. This table will have to be bigger than for $\text{det-}k\text{-decomp}$, since we can have $O(n^{2k})$ entries rather than just $O(n^{k+1})$, but this number is still only polynomial. Also, it would make sense to update the table entries not just for the results that were computed directly, but also for all entries that can be derived from such a result, to have the best available information at all times. E.g. let's say we showed that $[r, s]$ and $[t, u]$ are decomposable, where t is the only child of s . Then we know not only

¹⁰ Since it is deterministic and in space $O(\log^2 n)$ it will however still be *quasi-polynomial*.

that $[r, u]$ is decomposable, but also $[r, t]$ and $[s, u]$. In any case, the number of such updates will still be only polynomial, because there are at most as many as there are components.

As far as parallelization is concerned, we are in luck: Since every universal configuration has at least two successors, all independent of one another, we can simply start a new thread of computation for each of them as they cannot possibly interfere. Once one of them completes its computation successfully, it simply waits for the others. If one thread, however, completes its computation returning failure, it can stop its sibling threads and start looking for a different separator.

At any one time, there can be at most n threads working independently in parallel, since there are n edges and each edge can be in at most one component that is worked on at a time, so for ideal parallelization we would need only a linear number of processors.

Having said all this, we still have two major issues. The deterministic version of k -divide-decomp, just like $\text{det-}k\text{-decomp}$, relies on two factors to ensure polynomial running time:

1. There are at most polynomially many distinct components.
2. There are at most polynomially many separators for any component.

For $\text{det-}k\text{-decomp}$ both these polynomials were bounded by n^{k+1} . For $\text{det-}k\text{-divide-decomp}$, however, the first is only bounded by n^{2k} .

So the overall running time of the deterministic algorithm is only in $O(n^{3k+1})$ as compared to $O(n^{2k+2})$ for $\text{det-}k\text{-decomp}$.¹¹ Even the parallel version with n processors working “full time”, we still only get an upper bound $O(n^{3k})$. Of course, heuristics for finding “good” separators can decrease the running time for the positive instances,

¹¹ The extra factor of n for both algorithms comes from the loop going through all subcomponents of a separated component, which then calls *decomposable* for each. Note, that we are taking very conservative upper bounds here. For more details please see [32].

in which those separators lead to a valid decomposition quickly. However, the upper bound still stands.

This brings us to the other issue. For $\text{det-}k\text{-decomp}$ we have good heuristics to cut down the search space for separators and to increase the probability of finding a successful separator quickly. The heuristic for finding balanced separators is nowhere as good for two reasons:

1. It leaves too much room to include “redundant edges”, i.e. edges that are not essential in separating the upper and lower connecting sets. These edges can be identified, but only after we actually separate the component.
2. We have to actually try to separate a component before we know that a separator in deed separates the upper connecting set from the lower connecting set.

There is a possible improvement that neither cuts down the search space for separators nor eliminates the need to test separation, but it improves the chances of finding a balanced separator quickly: We can try to form the separator out of those edges first which are approximately “in the middle” between the upper and lower connecting sets. Let us formalize this.

Definition 4.11. Let H be a hypergraph. We define the *distance function* $d_H : E(H) \times E(H) \rightarrow \mathbb{N}$ by $d_H(e, f) = t$, where t is the length of the shortest path between e and f , for all $e, f \in E(H)$.

It is easy to check the d_H is a metric on $E(H)$.

We can extend this distance function to vertex-edge pairs, and also to a pair consisting of an edge and a set of vertices:

$$d_H(v, e) = \min_{f \in e} d_H(v, f)$$

$$d_H(V, e) = \min_{v \in V} d_H(v, e)$$

Now, given a component $[r, t]$ we can define a “centrality function” $c : [r, t] \rightarrow \mathbb{N}$ by $c(e) = |d_H(\gamma(r), e) - d_H(\beta(t), e)|$. The idea is that an edge in the middle between $\gamma(r)$ and $\beta(t)$ will have a low centrality value, whereas an edge closer to one of the connecting sets will be further away from the other, and will hence have a higher centrality value.

The distance metric can be computed at the beginning, using, e.g. the *Floyd-Warshall algorithm* [21]. Every time a new component is created through the function *separate*, we compute the centrality values for all its edges and sort them in increasing order.

Floyd-Warshall is feasible in time $O(n^3)$. Computing the centrality values of the edges in a component can be done in time $O(n^2)$ and sorting them can be done in time at most $O(n^2)$, or even $O(n \log n)$, using any of the standard sorting algorithms [18]. So the total running time remains polynomial, and for positive instances, we might find a decomposition quicker. However, it remains to be seen whether this extra work is worth in practice.

Another approach to finding balanced separators would be to first “grow” the components from the upper and lower connecting sets, before coming to a point where a brute-force exhaustive search becomes necessary.

4.5 The Best of Both Worlds

As was outlined at the end of last section, even though the deterministic version of k -divide-decomp can be parallelized easily, its overall time and space requirements are too high. We are thus left with the following question:

Can we find an algorithm which is parallel, but whose upper bound on time and space is no worse than that of det- k -decomp?

4.5.1 Combining Both Approaches

Remember, when we had a component $[r, t]$, and we guessed the λ -label of an intermediate node, it was well within the possible that we actually guessed the node r itself. But guessing $\lambda(r)$ is the same as finding a separator that covers $\gamma(r)$ and includes an edge from $\rho(r)$, just like in the *Hypergraph.nextSeparator* method of *det- k -decomp*. Actually, we will only be allowed to use edges in $[r, t]$ for that special edge (not necessarily the whole of $\rho(r)$), but that's OK — if there is no valid separator using an edge in $[r, t]$, then there is no decomposition of $[r, t]$, but there might still be a decomposition of $\rho(r)$ — we will just need a different “ t ”.

This brings us to the following idea: Assuming we only have a limited number of processors, say p , available, where $p < n$, we could stop decomposing components “in the middle” once we have reached p distinct components. After this, every processor will try to decompose its own component using *det- k -decomp*. The termination rules are exactly the same as in the divide and conquer algorithm, since a separator “adjacent” to the upper connecting set is still a separator separating the upper and lower connecting sets.

If *det- k -decomp* terminates successfully for a particular component, then we know that this component is decomposable. We mark it in the table together with all other results derivable from that decomposition. The free processor can be used on some other component now — just pick the one with the most edges in it, separate it down the middle and let the free processor work on one of the new parts. If *det- k -decomp* terminates with a failure, then we know that this particular component, $[r, t]$ say, is not decomposable, so we can halt the threads working on the “siblings” (in the call hierarchy of *decomposable*) of $[r, t]$, and look for a different “intermediate” separator, i.e. a different “ t ” or a different parent of r . In any case, we will have computed many valid decompositions of $[r, s]$, where s is a descendant node of r reached using *det- k -decomp*.

This approach removes the need of finding balanced separators for many components, which improves the running time of the separator enumeration function. However, it does not cut down the total number of components, since we still always have an upper and a lower connecting set. Moreover, many partially valid decompositions using adjacent separators will have to stop with failure only because the original balanced separator was chosen badly. Suppose we divided $[r, u]$ using $\lambda(s)$ into $[r, s]$ and $[t, u]$. If we are not restricted to select only edges “between” $\gamma(r)$ and $\beta(s)$ for “adjacent separators” in the search starting from r , then we may find a decomposition that goes beyond the lower connecting set of s , and eventually reaches a subcomponent of $[t, u]$ that was already decomposed properly using $\text{det-}k\text{-decomp}$ on $[t, u]$. We would thus have a valid decomposition of $[r, u]$, even though there might be no decomposition with s as an intermediate node.

So, instead of decomposing $[r, u]$ by dividing it into $[r, s]$ and $[t, u]$, we could try to decompose $\rho(r)$ using $\text{det-}k\text{-decomp}$ directly, by first guessing a possible $s \in T_r$, then using $\text{det-}k\text{-decomp}$ starting at s , trying to decompose $\rho(s)$, and simultaneously using $\text{det-}k\text{-decomp}$ starting at r in the hope that $\rho(s)$ will be decomposed first, and the search from r will reach some node in that decomposition. This way we go back to components with only an upper connecting set, of which there are only at most n^{k+1} .

The obvious problem here is the following: If the search from s does not terminate before the search from r reaches a node that is shared in both searches, the threads won’t know about each other doing the same work until one of them eventually terminates and updates the table. In the meantime, a lot of work could have been repeated.

The other problem is that we are still left with the task of finding balanced separators. We only have to do it if there is an idle processor, since otherwise every processor will be running $\text{det-}k\text{-decomp}$, however, every time we enumerate one such

separator, we still need to test the separation condition, before we can use it.

4.5.2 Designing par-det- k -decomp

The solution to the first problem lies at hand — rather than updating the table once we know whether a component can be decomposed or not, we can also update it once a thread starts working on that component. Another thread that tries to decompose the same component can now simply wait for a result to become available. This way, we can be sure that no work is repeated twice. Ever.

The problem of enumerating balanced separators can be tackled from two sides. First, we do not have to require the separator to be balanced. If we have enough processors available, we can start the search at enough distinct points in the hypergraph, going into all possible directions. Also, a separator that actually separates the hypergraph into at least two components can be found using det- k -decomp — assuming we haven't used up all k edges yet, the *Hypergraph.nextSeparator* method will eventually enumerate a separator that consists of exactly the old edges (which cover the connecting set completely), plus another edge from the component (another edge from the hypergraph). Thus, eventually all proper separators will be enumerated.

We thus reduce our algorithm to a parallelization of det- k -decomp with a cleverer table and threads that can wait on results from other threads.

This parallelization however is not trivial — especially since we want to implement it in practice — and deserves a closer analysis in its own right, which will be presented in Chapter 5.

4.5.3 Heuristic for par-det- k -decomp

Since in practice we will only have a limited (constant) number of processors available, and since using too many threads will reduce performance due to overhead, we will eventually look for a way to cut down the number of active threads at any one time. To decrease the interference between the active threads it will be important that they

work evenly distributed throughout the hypergraph. For this to happen, it is crucial that the very first separators considered (the ones consisting of a single edge) are also evenly distributed throughout the hypergraph, since we will usually grow separators adjacent to their old edges.

Since our algorithm will enumerate these one-edge separators in the order that the edges appear in the hypergraph, it is crucial to find a good ordering of these edges.

Definition 4.12. Let H be a hypergraph. Define $e_H : E(H) \rightarrow \mathbb{N}$ by $e_H(e) = \max_{f \in E(H)} d_H(e, f)$. We call $e_H(e)$ the *eccentricity* of e .

Eccentricity is a standard notion in graph theory, usually defined on vertices of a graph. It is directly transferable to edges of a hypergraph: the eccentricity of an edge is equal to the eccentricity of its corresponding vertex in the primal graph of the dual hypergraph.

The eccentricity of an edge tells us approximately how central it is. We define the *diameter* $d(H)$ of a hypergraph to be the maximal eccentricity over all its edges, and the *radius* $r(H)$ to be the minimal such value. An edge whose eccentricity is equal to the diameter is called a *peripheral edge*. An edge whose eccentricity is equal to the radius is called a *central edge*.

We want to have all peripheral edges at the end of our list, since a search from a peripheral edge will not be likely to find a balanced separator.

We could start our list with a central edge, i.e. an edge whose eccentricity is equal to the radius of the hypergraph — a balanced separator will then likely be found quicker. The next edges we would want to start from should be about in the middle between that first central edge and the peripheral edges, the following edges should be in the middle between all those edges, etc. Every next edge in our list should thus be as far away as possible from all other edges that preceded it, and from the peripheral edges.

For our heuristic, we thus decided to take inspiration from an electric field — a charged particle which is released into a potential field will settle down in a location with the least potential.

You cannot break the laws of physics, laws of physics, laws of physics, but we can bend them to our requirements for simplicity's sake. Suppose that the hypergraph corresponds to a space which we call the *Q-continuum*. Every edge corresponds to a discrete location in the *Q-continuum* which we call a *quadrant*. Each quadrant can hold precisely one object. The distance between two quadrants is equal to the distance between the two corresponding edges. We will be gradually filling the *Q-continuum* with objects called *dilithium crystals*. We assume that a dilithium crystal in a peripheral quadrant does not affect diametrically opposed quadrants, the potential difference being 0. We also assume that every dilithium crystal has a charge of value 1, and that the amount of potential difference decreases quadratically with distance. Thus, the amount by which a dilithium crystal in quadrant e increases the potential of a quadrant f is

$$\left(\frac{d(H) - d_H(e, f)}{d(H)} \right)^2.$$

Finally, we assume that once a dilithium crystal settles down in a quadrant it is immovable.

We first fill all peripheral quadrants with dilithium crystals. We then compute the new potential for all remaining quadrants. If we were to release a new dilithium crystal into the field, it would settle down in the quadrant with the least overall potential. After it settles down, we compute the new overall potential and continue the process until the whole *Q-continuum* is filled.

We can summarize our results in Algorithm 19 (we assume that every edge has a unique integer associated with it).

Algorithm 19 Sorting Edges — Heuristic

1: **var** h : Hypergraph $\triangleright n$ is number of edges in h
2: **var** es : Array[1... n](Integer)
3: **var** d_H : Array[1... n , 1... n](Integer)
4: **var** $diameter$: Integer := 0
5: **var** $radius$: Integer := ∞
6: **var** e_H : Array[1... n](Integer) := 0, for all indices
7: **var** $potential$: Array[1... n](Real) := 0.0, for all indices
8: **var** $pdif$: Array[1... n , 1... n](Real)
9: **var** $marked$: Array[1... n](Boolean)
10: Compute “distance function” d_H using Floyd-Warshall
11: **for** ($e \in [1 \dots n]$, $f \in [1 \dots n]$) {
12: $e_H[e] := \max(e_H[e], d_H[e, f])$
13: $diameter := \max(diameter, e_H[e])$
14: $radius := \min(radius, e_H[e])$
15: }
16: **for** ($e \in [1 \dots n]$, $f \in [1 \dots n]$) {
17: $pdif[e, f] := ((diameter - d_H[e, f])/diameter)^2$
18: }
19: Put all edges e with $e_H[e] = diameter$ at end of es
20: Set $marked[e] = \mathbf{true}$ for all such edges
21: **for all** (such peripheral $e \in [1 \dots n]$) {
22: $potential := potential + pdif[e]$ $\triangleright$ Addition of arrays is done component-wise
23: }
24: **var** $newEdge$: Integer
25: **var** pos : Integer := 1
26: Set $newEdge$ such that $e_H(newEdge) = radius$
27: **while** ($marked$ is not all $\mathbf{true}$) {
28: **var** $nextEdge$: Integer
29: $es[pos] := newEdge$
30: $pos := pos + 1$
31: $marked[newEdge] := \mathbf{true}$
32: **var** $minPotential$: Real := n $\triangleright$ the potential cannot be higher than n
33: **for** ($e \in [1 \dots n]$) {
34: $potential[e] := potential[e] + pdif[newEdge][e]$
35: **if** (**not** $marked[e]$ **and** $potential[e] < minPotential$) {
36: $nextEdge := e$
37: $minPotential := potential[e]$
38: }
39: }
40: $newEdge = nextEdge$
41: }

4.6 Reflections

We have described $\text{det-}k\text{-decomp}$ and proposed an improvement to the function enumerating all possible separators for a given component. We have described an easy way to make it parallel, and also discussed the issue of repeated work linked to this approach.

We have presented a divide and conquer type algorithm by defining “intermediate components” in a hypertree decomposition and analyzing the size of these components. This algorithm is in LOGCFL, but needs only $O(\log n)$ alternations and hence also only space $O(\log^2 n)$, when made deterministic. We showed how to make this algorithm polynomial and fully parallel, on up to n processors. Unfortunately, that algorithm’s upper bound on the running time and space requirements is unacceptable.

After some further analysis we showed that a cleverer parallelization of $\text{det-}k\text{-decomp}$ could achieve good work-load distribution over the different cores of a processor without repeating work. We also proposed an edge sorting heuristic which reduces the interference of threads in such a computation.

How this parallelization can be achieved in practice will be extensively discussed in Chapter 5.

Chapter 5

Parallelizing Alternation Efficiently

5.1 Motivation

Checking the existence of hypertree decompositions is only one of many important decision problems in Database Theory that happen to lie in the low complexity class LOGCFL. Thus, rather than considering the parallelization of this one particular problem we want to recognize its “good parallelizability properties” and see how these properties can be recognized in other problems in this complexity class. We can then work on a method to efficiently parallelize any such problem, thus abstracting away the specifics of the hypertree decomposition problem.

LOGCFL was originally defined as the class of languages that are LOGSPACE-reducible to *Context-Free Languages* [53]. However, it turns out that this class admits a whole range of alternative definitions, as was outlined in Section 2.3. In particular we use the following models of computation:

- Nondeterministic Auxiliary Pushdown Automata (NauxPDAs)
- Semi-unbounded Boolean Circuits
- Alternating Turing Machines (ATMs)

Each model of computation admits a different perspective at a problem in this complexity class. The advantage of the ATM approach is the simplicity of the algorithms.

Once one understands how existential and universal configurations interoperate, it is usually quite easy to write a “recursive program”.

For the example of computing hypertree decompositions, an existential configuration corresponds to a component. The selection of the right separator for that component corresponds to the existential choice. A component together with a separator correspond to a universal state. The requirement that every subcomponent be decomposable corresponds to the universal requirement of the eventual acceptance by all children of that universal configuration.

We propose an Object Oriented API to create programs modelled on the operation of an ATM by deriving abstract classes and implementing virtual methods representing the configurations and configuration transitions of the ATM. We also propose a scheduler which provides the necessary facilities to execute the above implemented methods asynchronously on multiple cores in a shared-memory system, while still preserving the soundness of the program, as long as the problem lies in the complexity class LOGCFL and the functions using the API interface satisfy certain conditions.

5.2 Structure of Chapter

In Section 5.3 we take a closer look at ATMs, define the *configuration graph* of an ATM and also its *reduced configuration graph*. We identify a “nice” property of such graphs for several problems in LOGCFL, including the hypertree decomposition problem. In Section 5.4 we study how the execution of a deterministic simulation of the ATM corresponds to a search of the reduced configuration graph. In Section 5.5 we investigate how we can make this search parallel. In Section 5.6 we devise a deterministic algorithm, together with the interfaces to execute such a parallel search. We conclude the chapter in Section 5.7.

5.3 Another look at ATMs

Recall that the set of states of an ATM is partitioned into accepting, rejecting, existential and universal states. We can modify the definition slightly to partition the set of states into just two sets: existential and universal. We say that a state is accepting iff it is universal and there are no possible transitions from it. Similarly a state is rejecting iff it is existential and there are no possible transitions from it. The intuition behind this is, that the unit of the operation $\vee$ (corresponding to an existential quantifier over some domain) is **false**, whereas the unit of the operation $\wedge$ (corresponding to a universal quantifier over some domain) is **true**. Thus, if we use the codomain of the transition relation as the domain for our quantifiers, we get exactly our desired result.

Recall the definition of acceptance for ATMs from Subsection 2.2.2. An ATM accepts iff it has an accepting computation tree. A computation tree is a rooted tree, in which every node is labelled with a configuration of the ATM, and there exists a transition from the label of a node to the label of its child. A computation tree is accepting, if all nodes labelled with an existential configuration have at least one child, all nodes labelled with a universal configuration have all possible children, and all leaf nodes are labelled with an accepting configuration.

Thus, with our new definition, we require all leaf nodes to be labelled with a universal configuration, such that there are no more transitions from that configuration.

An accepting computation tree represents only a part of the possible behaviour of the ATM, in particular it is the behaviour that leads to eventual acceptance of the input, by making all the right existential choices. However, in general, there might be more than one possibility for a transition from an existential configuration appearing as a label in the computation tree.

We can construct a tree which includes all such successors to existential nodes.

Definition 5.1. Let M be an ATM, w a string and $\mathcal{C}$ the set of configurations of M on w . The *complete computation tree of M on w* is a computation tree of M on w which additionally satisfies:

- For all non-leaf nodes $t \in T$, if $c(t) \rightarrow^{M,w} d$, then there exists $u \in T(t)$, such that $c(u) = d$.

This tree is still a valid computation tree, however it will almost definitely not be an accepting one. Any accepting computation tree is a sub-tree (considered as a sub-graph) of the complete computation tree.

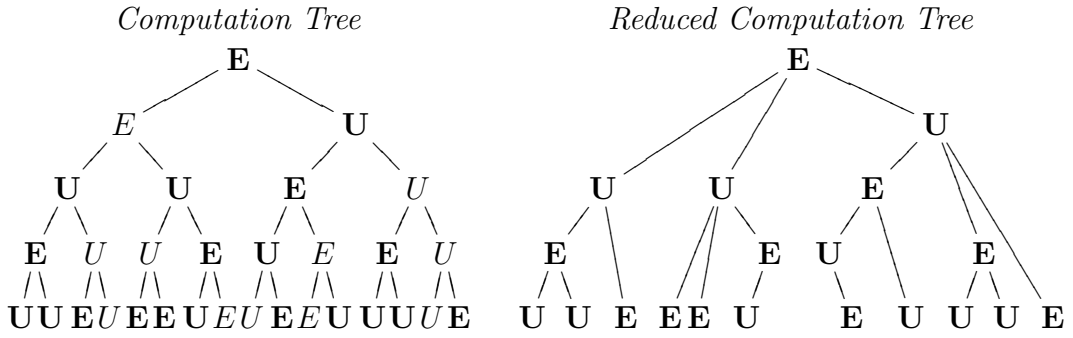
Generally, on a computation path we will have many configurations marked with the same quantifier following each other with little or no branching. The reason for this is, that a computation tree represents the full computation of the ATM, step by step, and thus includes deterministic computations (transitions with a single successor). The “interesting stuff” happens when a change of quantification occurs, that is when an existential configuration non-deterministically branches out to many universal configurations, or when a universal configuration branches out to many existential ones, requiring all outgoing paths to accept eventually. When a configuration branches out into several configurations, such that some of these successors have the same quantification, we could “merge” the computations of these children into the “deterministic computation part” of their parent and attach their successors as children of this parent instead, by making the parent branch out into more configurations. Thus, in the end, the really important configurations are the ones that immediately follow a configuration of a different quantifier, since they indicate a new existential choice or universal requirement (according to the quantifier of their predecessor).

We can formalize this as follows:

Definition 5.2. Let M be an ATM, w a string and $\mathcal{C}$ the set of configurations of M . Let (T, c) be a computation tree of M on w . A *reduced computation tree of*

M on w is a rooted labelled tree (T', c) where $T' \subseteq T$ and c is the same labelling function, such that $t \in T'$ iff $t = O(T)$ (if t is the root) or there exists $s \in T$ such that $t \in T(s)$ and $c(s)$ and $c(t)$ are oppositely quantified (precisely one is universal and the other is existential), and for all $s, t \in T'$, $t \in T'(s)$ iff for all nodes r on the unique path between s and t in T , $c(s)$ and $c(r)$ have the same quantification (either both existential, or both universal). A reduced computation tree is *accepting*, iff all leaf nodes are universal.

That is, we proceed deleting all “intermediate” nodes in the computation tree whose parent has the same quantification, and joining any children nodes of the deleted node to the deleted node’s parent. Figure 5.1 illustrates this process.



The nodes that make it into the reduced computation tree are marked in boldface.

Figure 5.1: Reducing the Computation Tree

The idea behind this construct is that the reduced computation tree induces an alternation of the quantifiers of its nodes at every level (thus the number of alternations of the ATM is equal to the depth of the tree), while still preserving the soundness of the ATM:

Proposition 5.1. *If a computation tree is accepting, then its reduced computation tree is also accepting.*

Proof. It is easy to see that the deletion of any “intermediate node” and the attachment of all its descendants to its parent does not change any of the conditions in

Definition 2.13. □

A configuration in the reduced computation tree can be viewed as representing a deterministic computation followed by an existential or universal branching into its successor configurations.

For an ATM it is conceivable, and in fact happens very often, that a single configuration can be reached via different computation paths. In this case, the complete computation tree would have several distinct nodes labelled with the same configuration. For example, if the ATM M operates in $\text{ASpace}(\log n)$, there are only polynomially many distinct configurations, however, the complete computation tree of M on some input w can in principle be exponentially large, or even unbounded, if there are “loops” in the computation. Even for the low complexity class LOGCFL, in which any accepting computation tree must have polynomial size, the complete computation tree — and even the complete reduced computation tree — does not have to be bounded by a polynomial. Conversely, if the complete (reduced) computation tree of such a machine is exponential, then it must contain exponentially many nodes which share their label with another node.

Furthermore, there is no rule forbidding the repetition of labels for accepting computation trees.

We can define a structure representing all possible runs of an ATM without repetition of labels in the following way:

Definition 5.3. Let M be an ATM, w a string and $\mathcal{C}$ be the set of configurations of M on w . The *configuration graph of M on w* is a directed graph $G = (V(G), E(G))$. We define $V(G) \subseteq \mathcal{C}$ such that the initial configuration $c_0 \in V(G)$ and $c \in V(G)$ iff $c = c_0$ or $c_0 \xrightarrow{M, w}^* c$ (i.e. iff c is in the transitive closure of the transition relation on c_0). For $c, d \in V(G)$, $(c, d) \in E(G)$ iff $c \xrightarrow{M, w} d$.

A computation tree is a *witness* of a run of the ATM, passing through a subset of all possible configurations according to the transition relation. Thus, any computation tree considered as a directed graph can be mapped homomorphically (but not necessarily embedded) into the configuration graph.

Just as with computation trees, we can reduce configuration graphs. In this case, the definition is more cumbersome. Given a configuration graph G and configurations $c, d \in V(G)$, we write $c \approx d$, iff c and d have the same quantification, and there exists an **undirected** path from c to d in G , such that all configurations on this path have the same quantification as c and d . It is easy to see, that $\approx$ is an equivalence relation. Write $\bar{c}$ for the equivalence class of a configuration c .

Definition 5.4. Let M be an ATM and w a string, and let G be the configuration graph of M on w . The *reduced configuration graph of M on w* is a directed graph G' , where $V(G') = \{\bar{c} | c \in V(G)\}$; for all $c, d \in V(G')$ such that $c \neq d$ we have $(c, d) \in E(G')$ iff there exist $c' \in \bar{c}, d' \in \bar{d}$, such that $(c', d') \in E(G)$; and $(c, c) \in E(G')$ iff there exists a cycle in G entirely contained in $\bar{c}$.

In general, the reduced configuration graph of an ATM can have any form. However, for several alternating algorithms in LOGCFL, these graphs has a certain regularity.

Property 5.1. *Let M be an ATM deciding a language L in LOGCFL. We say M is nice, iff for all $w \in \Sigma^*$ the reduced configuration graph G of M on w has the following properties:*

- G is acyclic,
- any universal node has precisely one predecessor,
- for any two distinct existential nodes with a common parent the sets of their descendants are disjoint.

It is fairly straightforward to check this property for k -decomp (as was already mentioned in Chapter 4), as well as for the algorithm evaluating acyclic BCQs [28], which is complete for LOGCFL.

5.4 Search and Backtracking

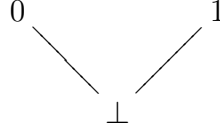
Since a reduced computation tree is also a witness to a run of the ATM, any reduced computation tree can be mapped to the reduced configuration graph. Looking for a successful run of M on w now corresponds to finding a special sub-graph of the (reduced) configuration graph which itself corresponds to an accepting (reduced) computation tree. In particular we are looking for paths starting at the initial configuration (the source) of the graph, and ending at nodes with no outgoing edges (sinks of the graph) which are labelled with a universal configuration, such that whenever a path passes through a node labelled with a universal configuration, all the successors of this node are covered by at least one of the paths.

We can now reformulate the decision problem of whether M accepts w as follows:

Given an ATM M , a string w and the reduced configuration graph G of M on w , do there exist paths $(P_i)_{i=1}^n$ such that for all i , the first node of P_i is c_0 , the initial configuration, and the last node of P_i is a universally labelled sink, and such that if c is universal and in some P_i , then for all successors d of c in G , there exists a j , such that d is in P_j .

Instead of looking for paths in the graph from the source to the sinks, there is a simpler procedure working its way backwards from the sinks to the source. This procedure is formally presented in [13]: A *labelling of configurations* is a map $l : \mathcal{C} \rightarrow \{\perp, 0, 1\}$. Here $\{\perp, 0, 1\}$ represent “unknown”, “reject” and “accept”, respectively.

Additionally we consider this set as a partial order with $\perp \sqsubset 0$ and $\perp \sqsubset 1$:



We start with the *minimal labelling* Ω , where $\Omega(c) = \perp$ for all $c \in \mathcal{C}$, and keep applying the operator τ mapping labellings to labellings, adding information to a labelling at each step, such that any node labelled $\perp$ becomes labelled with one of the following:

- 1, if it is existential and one of its children was labelled 1,
- 0, if it is existential and all of its children were labelled 0, or if it has no children,
- 1, if it is universal and all of its children were labelled 1, or if it has no children,
- 0, if it is universal and one of its children was labelled 0,
- $\perp$, otherwise

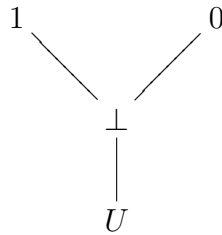
Since τ is monotone on the partial order of labellings (the order extended from the above partial order coordinate-wise to $\mathcal{C}$), it has a least fixed point. M accepts w iff after we reach a fixed point, the source is marked 1. M rejects w iff after we reach a fixed point, the source is marked 0. There is of course the possibility that the fixed point is not reached after any finite number of repetitions of the above procedure, which can happen when the set of configurations is infinite. This corresponds to non-termination of the ATM due to diversion. However, if the set of reachable configurations, i.e. the set of nodes in the configuration graph, is finite, we are guaranteed to reach a fixed point on the configuration graph after finitely many steps. There is also a possibility that even when a fixed point is reached in finite time, the source is still marked as $\perp$, due to cycles in the reduced computation graph. This situation corresponds to a non-terminating run of the ATM (due to looping).

The important thing is that it is enough to consider τ on the configuration graph, rather than on the full set of configurations. Also, the above procedure works for the configuration graph as well as for the reduced configuration graph.

In practice, there is usually no way to know which configurations of M on w are reachable without actually simulating all possible runs of the ATM. Hence we do not know in advance what the reduced configuration graph looks like, and we cannot apply the above technique directly. Computing the entire configuration graph would be impractical, since we are only interested in finding any one accepting run as quickly as possible, preferably without doing any work that can be avoided. We do however know that each node in the graph depends only on its parent (according to the transition function of the ATM), so once we have a procedure to compute all successor nodes for any one node, or to test whether any such successor nodes exist at all, we can compute only the portions of the graph we are interested in.

So now to find a solution we can perform a search on the graph starting at the source simply “expanding” all the children of a node with successors on the fly. But rather than finding only one “goal” node, we will need to find several, according to the rules of the ATM.

We can use the technique from [13] and adapt it to a top-down procedure. For this we extend the range of labellings. Let G be the (reduced) configuration graph of M on w . A labelling is a function $l : V(G) \rightarrow \{U, \perp, 0, 1\}$, where the range is a partial order with $U \sqsubset \perp$, $\perp \sqsubset 0$ and $\perp \sqsubset 1$:



The label U indicates that a node (a configuration) was not yet visited in our search. We also assume that the list of children of any node is ordered. Begin with

the minimal labelling Ω' which labels every configuration with U and the root with $\perp$.

We will now have two functions, σ, τ from labellings to labellings.

σ operates in the following manner:

1. Pick a set of nodes C , such that for each $u \in C$, u is labelled $\perp$, and either u has no children, or u has at least one child v labelled U . For each $u \in C$ do the following:
2. If u has no children and is existential, update the labelling of u to 0.
3. If u has no children and is universal, update the labelling of u to 1.
4. Otherwise, let v be the first child of u labelled U . Update the labelling of v to $\perp$.
5. If no such set C exists, then σ is the identity.

It is easy to see that σ is monotone.

The function τ now operates somewhat differently than it did in [13]. For every node u we update the labelling in the following manner:

1. If u is labelled U , 0 or 1, do not change its labelling.
2. Otherwise, if u has no children, do not change its labelling,
3. If u is existential, labelled and one of its children is labelled 1, change the label of u to 1,
4. If u is existential and all of its children are labelled 0, change its label to 0,
5. If u is universal and all of its children are labelled 1, change its label to 1,
6. If u is universal and one of its children is labelled 0, change its label to 0.

Again, it is easy to see that τ is monotone.

We define a new operand $\phi = \mu(\tau) \circ \sigma$, where μ is the least fixed point operator and $\circ$ denotes composition of functions. This operand roughly says: check if some nodes marked with $\perp$ are terminating configurations, and if so mark them with their according results, or otherwise “expand” some unvisited children; then keep propagating the new results to the ancestors as far as possible. It is also monotone and has a least fixed point.

We did not specify how σ picks the set C , yet, as this will correspond to our *search strategy*, and there may be several different strategies available. However, in any case σ is monotone, and becomes the identity on a node only once it is labelled $\perp$ (or 0 or 1 for a leaf node). Also, τ is monotone and also becomes the identity on a node, once it becomes labelled 0 or 1. Of course, a node may become labelled 0 or 1, while some of its children are still labelled $\perp$ or even U .

From now on, **we assume that the ATM has Property 5.1**. In particular, the reduced configuration graph is acyclic. Thus, it is easy to see, that after any sequence of applications of σ and τ to Ω' in any order, if the root is not labelled 0 or 1, then either τ or σ will not be the identity. Hence, any sequence of applications of σ and τ will eventually reach the label 0 or 1 for the root node in the (reduced) configuration graph. Moreover, the value of that label does not depend on the particular sequence.

Also, since applying $\mu(\sigma)$ to the initial labelling Ω' basically delivers the initial labelling Ω (plus 0's and 1's for all leaf nodes), the eventual label of any node in the (reduced) configuration graph will be the same in both cases. In particular, we have

$$\mu(\phi)(\Omega')(c_0) = \mu(\tau)(\Omega)(c_0)$$

So the continuous application of ϕ gives us a way to compute the outcome of running M on w , by searching the reduced configuration graph from its root.

We now return to the function σ and consider possibilities of selecting the set C of nodes to be “expanded”. Let us for now assume that C always consists of a single element u .

Remember, that we assume that the children of any one node are ordered. One possibility of selecting u is by a depth-first principle: Select u such that u is the deepest node among all nodes labelled $\perp$. It is not hard to see, that such a node is always unique.

Another possibility of selecting u is by a breadth-first principle: Pick the shallowest node u which can expand more children. If there are several such nodes, pick the first one in lexicographical order.

Recall the algorithm *det- k -decomp* from Subsection 4.3.2, and its splitting into two mutually recursive functions in Subsection 4.3.5. If we follow the order of which Components (reduced existential states) and Separators (reduced universal states) are considered in which order, we get exactly a depth-first search on the reduced configuration graph.

The advantage of doing a depth-first search rather than a breadth-first search is the total memory required at any one point. In a depth-first search, we only need to store one “line of descent” of nodes in the configuration graph. In a breadth-first search, we would have to store **all** currently expanded nodes [18, 37].

Another important point besides our desire to avoid unnecessary work, perhaps even more important, is our desire to avoid doing the same work twice. As mentioned before, different paths in the computation graph can pass through the same nodes, and we want to avoid computing the same paths from that node twice. Hence an implementation should recognize if a newly computed node has already been computed before. This can be accomplished through a dynamic programming technique, maintaining a table of previously visited nodes and their outcome. This table corresponds to the currently known labelling (the portion of the configuration not labelled U).

Once a new node is computed, we always check first if it is already in the table, and if it is, then we use that information instead of expanding the node.

Remember, that in the configuration graph of a LOGCFL ATM, all universal nodes have a unique parent. This eliminates the need to store the results for universal nodes, since a universal node can only be “expanded” once.

Also, the configuration graph does not have any cycles. Thus, as long as we expand nodes in a depth-first manner sequentially, we will never encounter a node marked $\perp$ in the table. This means we do not need to store nodes marked $\perp$ in the table before they are marked 0 or 1, if we do a depth-first search. This is exactly the approach taken in *det- k -decomp* for the particular instance of hypertree decomposition computation.

Let us define two abstract classes *UState* and *EState*, corresponding to universal and existential configurations of an ATM, respectively. Let us also define an abstract class *ATM* representing the ATM itself, with the abstract methods *ATM.nextChild(UState)*, returning an *EState*, and *ATM.nextChild(EState)*, returning a *UState*.

This is completely analogous to the methods *Hypergraph.nextChild(Component)* and *Hypergraph.nextChild(Separator)* from Subsection 4.3.2. In fact, we chose the method names in such a way, that *Hypergraph*, *Component* and *Separator* would extend the classes *ATM*, *EState* and *UState*, respectively. We present the class declarations in Algorithm 20. Note that we have several additional fields, which are irrelevant at the moment, but will become very important in Section 5.6.

Our table, will now be a *Map* from *EStates*, or rather *EStateIDs* (which we can obtain through *EState.getID()*), to *Booleans*, or rather *UStateIDs* (which we obtain through *UState.getID()*). *TableData* functions as a common superclass for both *EState* and *UStateID*. This is again irrelevant at this stage, but will become important in Section 5.6.

Algorithm 20 *ATM, EState, UState*

```
1: class ATM {
2:   method nextChild(e: EState): UState
3:   method nextChild(u: UState): EState
4:   method initialConfiguration(): EState
5: }
6:
7: class EState extends TableData {
8:   field priority: Integer := LOWEST_PRIORITY
9:   field state: {WAITING, WORKING, SUSPEND, CANCEL, INQUEUE} :=
    WAITING
10:  field parents: List⟨UState⟩ := []
11:  field children: List⟨UState⟩ := []
12:  field hasMoreChildren: Boolean := true
13:  method getID(): EStateID
14: }
15:
16: class UState {
17:   field priority: Integer := LOWEST_PRIORITY
18:   field parent: EState := nil
19:   field children: List⟨EState⟩ := []
20:   method getID(): UStateID
21: }
22:
23: class TableData {
24: }
25:
26: class EStateID {
27: }
28:
29: class UStateID extends TableData {
30:   method type(): {SUCCESS, FAILURE}
31: }
```

We can now write a generic sequential deterministic algorithm, by using *det- k -decomp* (Algorithm 8), while replacing every occurrence of *Hypergraph* with *ATM*, every occurrence of *Component* with *EState* and every occurrence of *Separator* with *UState*. For generality's sake, we can rename the function *decomposable* into *successful*. We can now run *det- k -decomp*, or any other alternating algorithm, whose configuration graph is acyclic, by simply providing the implementation of the above classes, and of the methods *ATM.nextChild*.

5.5 Parallelizing the search

Using a depth-first search in this manner removes the need to store relationships between the configurations — there will always be only a single line of descent expanded at any one time, and the parent-child relationship of the configurations is represented by the call stack on the function *successful*. However, if we wanted to adapt a different search strategy, i.e. a different function σ , we might have to be able to represent larger parts of the expanded configuration graph. An easy way of doing this, is to include a list of children and a list of parents for *EStates* and *UStates*. However, since we decided to concentrate on problems in LOGCFL, we can assume that in the configuration graph any *UState* will have exactly one parent.

If we update the *children* and *parents* lists properly every time we create a new *EState* or *UState*, we will have a complete representation of the currently expanded portion of the configuration graph. We can follow the pointers to parents and children freely around the graph and pick out any configuration, whose children we want to expand next. We could easily change the depth-first search of *det- k -decomp* into loop form, by keeping a pointer to the deepest expanded node, and backtracking whenever a result for it becomes known. Moreover, we can delete *EStates* and *UStates* for which we already have a result, since we don't need to expand any of their children

again. The memory required by a depth-first search algorithm will be linear in the maximal depth of the configuration graph (not counting the dynamic table).

Is there any point of using a different search algorithm?

If we used a different search algorithm, e.g. a breadth-first search, we might need more space. We can still delete configurations for which we know a result, but we might have more than a single “line of descent” stored in the expanded configuration graph at one time. Also, now it will be possible for an existential configuration to have several parents at once. With this level of difficulty, would it make any sense to use a breadth-first search at all?

Consider a node u in the configuration graph. Let us say, u was recently expanded by our algorithm, and we are now trying to compute the result of u . For simplicity, let us assume, that no descendant of u in the configuration graph is yet known. The only unit that we can measure the required work to compute the value of any node v by, is the number of times the method *nextChild* is called. Let W_v be the amount of work required to compute the value of v , for every child v of u .

By looking at the configuration graphs of several problems in LOGCFL, like hypertree decomposition computation, we can make the following observation:

- For an **existential** node u , each child v of u has about the same total number of descendants (within an order of magnitude). Also the total number of children may be large.
- For a **universal** node u , there may be children with very different number of descendants. Also, the total number of children is usually small.

This can be easily seen in the case of hypertree decomposition computation: A separator will separate component into one or more components, but the total number of edges in those components will be about the same for all possible separators, and there may be many separators possible. The number of edges in a single such new

component will, however, strongly depend on the particular separator chosen, but there will generally be few new components.

Let us assume for simplicity's sake, that W_v is proportional to the number of descendants of v in the configuration graph (again a reasonable assumption).

Suppose u is existential and succeeds. If we perform a depth-first search, in the worst case the total amount of work required to compute the value of u is $\sum_{v \in G(u)} W_v$. If we perform a breadth-first search, in the worst case the total amount of work will be the same, if W_v is about the same for all v . In the average case, we will perform $(\sum_{v \in G(u)} W_v)/2$ using a depth-first search. A breadth-first search would still give $\sum_{v \in G(u)} W_v$. In the best case, a depth-first search would give W_{v_1} work. A breadth-first search would still give $\sum_{v \in G(u)} W_v$.

Now suppose u is existential and fails. If we perform a depth-first search, the total amount of work required is always $\sum_{v \in G(u)} W_v$. This is the same for breadth-first search.

Now suppose u is universal and succeeds. Again, the total amount of work required is $\sum_{v \in G(u)} W_v$ for both depth-first and breadth-first search.

However, suppose u is universal and fails. Let v_j be the child requiring the least work. Now, in the worst case, a depth-first search will need work $\sum_{v \in G(u)} W_v$. In the best case, it will only need work W_{v_j} . A breadth-first search will always need nW_{v_j} , where n is the number of children of u . So it may be a lot faster, and in the worst case is slower than depth-first search by a factor of n , which, as we said, is usually small.

We can conclude, that depth-first search is better on existential nodes, whereas breadth-first search can be much better and is usually not much worse on universal nodes.

One way to implement an algorithm combining depth-first search on existential nodes and breadth-first search on universal nodes, is to maintain a FIFO queue of

EStates. We would pick out an *EState*, compute its next *UState* child, and then immediately expand all the *EState* children of that *UState*, while adding them all to the back of the queue. Instead of picking a newly expanded configuration as the next thing to work on, we will now always pick the first element from the queue. Of course, we need to be able to put *EStates* back into the queue, if more of their children are required to be expanded. Similarly, we would have to make sure that *EStates* currently in the queue can be deleted, in case they are no longer required themselves.

This additional data-structure becomes very important once we consider parallelization: As mentioned on many occasions, an important observation about universal states for the class LOGCFL is, that their children are entirely independent. This suggests, that we could pick several *EStates* from the queue at a time and give them to separate threads to process. This approach corresponds to picking σ in the following manner: Whenever a new universal configuration is encountered, we pick it, until all its children are expanded. After that we set C to contain all those children. Running such a “parallel” program with a single thread would, in fact, have the behaviour of a sequential program combining depth-first on *EStates* and breadth-first search on *UStates*. Of course, we would have to deal with precedence issues, i.e. which new *EStates* are put into the queue first.

The fact that the children of a *UState* are completely disjoint suggests the possibility of splitting the work not only between different processors, but between different machines (with separate memory). We could for example serialize an *EState*, send it to a different machine, which then checks whether that *EState* succeeds and returns the result and new information for the table back to the “master” machine.¹

However, this approach is not ideal for several reasons.

- Synchronizing information between different machines can take a lot of time.

¹ In fact, we tested this approach in practice.

(We would want to synchronized portions of the dynamic table, to make sure machines do not repeat work already done by another machine.)

- As discussed in Subsection 4.3.5, universal configurations will often have only a single successor. Thus, we would only have a few active machines at a time, unless we use different algorithms. In this case, the parallel speed-up would certainly not be worth the programming effort.
- Synchronization between machines would be required quite often (every time one of the children of the *UState* produces a result), producing communication overhead at a high pace.

Of course, one could design different algorithms, which reduce the need for continuous communication between the machines, if we are not too bothered, that machines might repeat work. However, using a *shared-memory* approach seemed like the plausible choice for our problem, in particular due to the following reasons:

- A global table which is accessible by **all** threads guarantees, that we can avoid repeating **any** work twice, if we design our algorithm carefully.
- The recent trend of producing processors with less power consumption and more cores provides ample support for shared-memory applications at the moment, and it is reasonable to assume that this trend will continue for some time, allowing programs with even more threads.

5.6 The Shared-Memory Approach

We can now assume that all configurations and the table are stored in a central location, accessible by all threads of execution. Thus, we do not have to bother about sending information from one thread to another. Of course, we still have to take care of synchronization issues, linked to updating the configuration graph

(inserting new *UStates* and *EStates* into it), the table of results, and possibly to further data-structures required by the parallel algorithm (like the queue mentioned in the previous section).

Moreover, rather than just parallelizing work only at *UStates*, we want to have something closer to par-det- k -decomp described in Subsection 4.5.2, instead. However, as we already mentioned in Section 4.6, we will not have nearly enough processors to run even a small portion of all possible threads described in that algorithm simultaneously. Thus, we would have a huge overhead of processors switching between threads.

A good parallelization approach to computational problems with many interdependent sub-problems, which have different size, might require different amounts of time to compute, and for which the control flow will generally be very different, depending on the sub-problem, is *task parallelism*. This approach is a big contrast to *data parallelism*, in which the **same code** (usually a sequence of primitive instructions with no, or very limited branching, which is the same for all threads) is executed on **different data** simultaneously, and in a synchronized manner. The latter approach is heavily used in scientific computing, however it does not fit our particular needs. Thus, task parallelism seems like the right way forward.

The main idea is to create a fixed number of threads, which acquire *tasks* to work on from some sort of *task pool*, and each such task may either return a result or spawn new tasks. This eliminates the overhead of creating and deleting threads; also, the number of threads can be set to the best number for the specific architecture the program is running on. Moreover, this model eliminates synchronization points in the code, at which any one thread has to wait for all the other threads to reach that point as well. The threads can thus work in an *asynchronous* manner, reducing the time at which any one processor is idle.

It is straightforward to consider *EStates* and *UStates* as tasks. Each thread of

execution would thus acquire an *EState* or a *UState* and either conclude that that particular configuration returns either SUCCESS or FAILURE and update the configuration graph accordingly, or it would spawn one or several children and put them into the pool. This corresponds to “evaluating” the function ϕ , described in Section 5.4, on the reduced configuration graph. Having picked an asynchronous approach, we can assume that a single thread “evaluates” this function once, and parallelization happens due to the fact that “evaluating” the function σ (creating new children) takes more time than evaluating $\mu(\tau)$ (updating the configuration graph). Threads thus spend most of their time creating new children, for different nodes in the configuration graph simultaneously. The only part which we have to take special care of is updating the configuration graph — for this we will have to introduce some sort of synchronization points, to make sure the graph remains consistent.

We introduce an object of type *Scheduler*, which will maintain the threads, the task pool, the relationships between the tasks and will also keep the *ATM* object, which is used to create new configurations from old ones. It will also maintain the table mapping *EStateIDs* to either *EStates*, if they were created but haven’t produced a result yet, or to *UStateIDs*, which represent already computed results (see also Subsection 4.3.4). We present the declaration of these classes and of their methods in Algorithm 21. We will explain what the particular fields or methods mean, as we go along.

Objects of type *Thread* encapsulate a physical thread. Calling *Thread.start()* initiates this thread, which then executes the method *Thread.run()*. This method performs the task of acquiring new tasks to work on, working on them, and finally processing the results. As mentioned before, a *Thread* will repeat the above procedure until the *Scheduler* tells it to stop. To wait for a thread to stop, we simply call *Thread.join()*.

We present the listing of the method *Thread.run()* in Algorithm 22.

Algorithm 21 *Scheduler and Thread*

```
1: class Scheduler {
2:   field threads: Array⟨Thread⟩
3:   field atm: ATM
4:   field root: EState
5:   field queue: PriorityQueue⟨EState⟩
6:   field table: Map⟨EStateID, TableData⟩
7:   method lock()
8:   method unlock()
9:   method wait()
10:  method notify()
11:  method addTask(e: EState):
12:  method getTask(): EState
13:  method firstFreeDescendant(): EState
14:  method suspend(e: EState)
15:  method suspend(u: UState)
16:  method resume(e: EState)
17:  method resume(u: UState)
18:  method cancel(e: EState)
19:  method cancel(u: UState)
20:  method update(e: EState)
21:  method update(u: UState)
22:  method fail(e: EState)
23:  method fail(u: UState)
24:  method succeed(e: EState, u: UState)
25:  method newLowestPriority(): Integer
26:  method sortQueue()
27: }
28:
29: class Thread {
30:   field task: EState
31:   field newUstate: UState
32:   field newEstates: List⟨EState⟩
33:   field scheduler: Scheduler
34:   method start()
35:   method join()
36:   method run()
37:   method doWork()
38:   method update()
39: }
```

Algorithm 22 *Thread.run()*

```
1: method Thread.run() {  
2:   scheduler.lock()  
3:   loop {  
4:     task := scheduler.getTask()  
5:     scheduler.unlock()  
6:     scheduler.notify()  
7:     if (task = nil) {  
8:       break  
9:     }  
10:    doWork()  
11:    scheduler.lock()  
12:    update()  
13:  }  
14: }
```

Notice that out of the three things a *Thread* does during every loop, *Scheduler.getTask()* and *Thread.update()* are considered *critical*. That is, these portions of code affect information which is shared between the threads, and to maintain consistency we must acquire an exclusive *lock* (mutex) on the *Scheduler* before changing that information. Only one thread at a time can have this lock, and it must *release the lock* before another thread can acquire it. Thus only one thread at a time will be ever executing *Thread.update()* or *Scheduler.getTask()*. The methods to acquire and release the lock are called *Scheduler.lock()* and *Scheduler.unlock()*, respectively.

We use a trick to inform a thread that it can stop working: Setting *Thread.task* to **nil** by the *Scheduler.getTask()* method is the *Scheduler*'s way to say that no more tasks are required to be computed.

5.6.1 Working on Tasks

As mentioned at the end of the previous section, we want to use a depth-first strategy on *EStates* and a breadth-first search strategy on *UStates* (especially now that we have multiple threads available). Thus, we can simplify our program by putting only *EStates* into the task pool, and during each “atomic computation step” of a thread

expanding a new *UState* and **all** its children directly.

Algorithm 23 *Thread.doWork()*

```

1: method Thread.doWork() {
2:   newEstates.clear()
3:   if ((newUstate := scheduler.atm.nextChild(task)) ≠ nil) {
4:     while ((var ch: EState := scheduler.atm.nextChild(newUstate)) ≠ nil)
5:     {
6:       newEstates.add(ch)
7:     }
8:   } else {
9:     task.hasMoreChildren := false
10:  }
```

We use local variables *newUstate* and *newEstates* to store these new configurations, before inserting them into the main configuration graph, which is centrally stored as the descendants of the *EState Scheduler.root*.

5.6.2 Acquiring new Tasks

As mentioned earlier, the *Scheduler* object will have a *queue*, in which it stores *EStates*, which are then acquired by the *Threads* to work on. We leave the specifics of *inserting* new elements into the queue until the next subsection. For now it suffices to know that the queue holds the *EStates* in the “best” order, that is an *EState* that is **most essential** in the computation of an answer will be at the front of the queue. To indicate that we do not require to expand any more children of any *EStates* (i.e. we found a solution), a special element will be put in the front of the queue — **nil**. The method used to return an *EState* to work on is called *Scheduler.getTask()*, presented in Algorithm 24.

It is very well conceivable that the queue will be empty after several threads acquired all the *EStates* from it, while there are still more threads looking for “work” and calling *Scheduler.getTask()*. Of course these threads could wait for the queue to fill up again, however, the way that *Thread.doWork()* works implies that we would

Algorithm 24 *Scheduler.getTask()*

```
1: method Scheduler.getTask(): EState {  
2:   loop {  
3:     if (queue.empty()) {  
4:       var toret: EState := firstFreeDescendant()  
5:       if (toret  $\neq$  nil) {  
6:         toret.state := WORKING  
7:         return toret  
8:       } else {  
9:         wait() ▷ wait to be notified by another thread  
10:        continue ▷ start the loop again  
11:      }  
12:    }  
13:    toret := queue.front()  
14:    if (toret  $\neq$  nil) {  
15:      queue.removeFront()  
16:      toret.state := WORKING  
17:    }  
18:    return toret  
19:  }  
20: }
```

only perform a breadth-first search on universal configurations, which in general gives only limited speed-up (as discussed before). Moreover, a *Thread* unlocks the *Scheduler* only after it gets a new task — so very often a working thread would compute a single new *EState*, put it in the queue during its call to *Thread.update()* (this will be discussed in the next subsection), and retrieve it immediately from the queue, before unlocking the scheduler. So the queue would remain empty, and many other threads would remain idle, until some *UState* produces several children.

Instead, we want to be able to acquire new *EStates* to work on directly from the configuration graph. But which *EStates* are safe to select? Since *EStates* store the information about which child they will expand next internally, it would be disastrous to allow two threads to call *ATM.nextChild(EState)* on the same *EState* simultaneously. As we outline in the next subsection, every *EState* will have a field called *state*, representing what is happening to the *EState* at any moment in the course of

Algorithm 25 *Scheduler.firstFreeDescendant()*

```
1: method Scheduler.firstFreeDescendant(): EState {
2:   for (var  $d$ : Integer  $\in [0, \dots, |threads| - 1]$ ) {
3:     var  $ret$ : EState :=  $root.firstFree(d)$ 
4:     if ( $ret \neq \mathbf{nil}$ ) {
5:       return  $ret$ 
6:     }
7:   }
8:   return  $\mathbf{nil}$ 
9: }
10:
11: method EState.firstFree( $d$ : Integer): EState {
12:   if ( $state = \mathbf{WAITING}$  and  $hasMoreChildren$ ) {
13:     return this
14:   } else if ( $d = 0$ ) {
15:     return  $\mathbf{nil}$ 
16:   }
17:   if ( $hasMoreChildren$ ) {
18:      $d := d - 1$ 
19:   }
20:   for all (var  $ch$ : UState  $\in children$ ) {
21:     var  $ret$ : EState :=  $ch.firstFree(d)$ 
22:     if ( $ret \neq \mathbf{nil}$ ) {
23:       return  $ret$ 
24:     }
25:   }
26:   return  $\mathbf{nil}$ 
27: }
28:
29: method UState.firstFree( $d$ : Integer): EState {
30:   for all (var  $ch$ : EState  $\in children$ ) {
31:     var  $ret$ : EState :=  $ch.firstFree(d)$ 
32:     if ( $ret \neq \mathbf{nil}$ ) {
33:       return  $ret$ 
34:     }
35:   }
36:   return  $\mathbf{nil}$ 
37: }
```

the program. In particular, this variable will take one of the following values: *IN-QUEUE*, *WAITING*, *WORKING*, *CANCEL* or *SUSPEND*. We will describe these in more detail in the next subsection. For now, it suffices to say, that if an *EState* is in state *WAITING*, and our thread has a lock on the scheduler, then we can guarantee that no other thread is accessing that *EState*. We could thus pick any *EState* in the configuration graph which is marked as *WAITING*, and which can still produce more children (for which the *hasMoreChildren* flag is set).

How should we choose such waiting *EStates*?

When expanding several children of an existential configuration in the “middle” of the configuration graph, there is a big chance that different universal configurations will lead to the same existential configuration after only a few steps. We would thus gain little from parallelizing work at such an *EState*. The children of the initial configuration in the configuration graph are likely to be “further apart” in the configuration graph in that respect: it will take longer for two “lines of descent” to “converge”. Furthermore, such a convergence will be advantageous: the “shared configuration” will be relatively deep within one line of descent, whereas it will be relatively shallow within the other, and thus many of its descendants will be already computed. This is exactly the principle the algorithm *par-det-k-decomp* relies on, to “distribute” its work evenly throughout the hypergraph, as was described in Chapter 4.

This principle suggests to always pick the root of the configuration graph whenever possible. If the root has already created all its possible children, or if some thread is already working on it, while more *EStates* are waiting in the configuration graph, we could pick one of them to avoid idle processor time. The reasoning of the previous paragraph, again, suggests that a breadth-first strategy will produce better results. This search is performed using the method *Scheduler.firstFreeDescendant()*, presented in Algorithm 25.

To avoid the space overhead incurred by a full breadth-first search we use an

iterative deepening search instead. We use the total number of threads minus one as the cut-off depth, where we count the depth of a *UState* to be the same as that of its parent *EState*, and where we ignore *EStates* which cannot produce any more children (with *hasMoreChildren* set to **false**): we will have at least t *EStates* capable of producing children in the portion of the graph up to depth $t - 1$, where t is the number of threads. Thus, there will be at least one *EState* that is not being worked on by a thread in that portion of the configuration graph.

As a final remark, we should add that the method *Scheduler.wait()* does the following actions:

1. It releases the lock on the scheduler, acquired using *Scheduler.lock()*;
2. It suspends the current thread, until some other thread calls *Scheduler.notify()*;
3. After *Scheduler.notify()* was called, the current thread starts acquiring the lock on the scheduler again;
4. After the current thread acquires the lock on the scheduler, it resumes its operation.

5.6.3 Updating the Configuration Graph

We mentioned before the field *state* of an *EState*. This variable is used to inform threads which might want to modify this *EState* in any way about its current situation.

Let e be an *EState*. Suppose some thread is creating new children for e . Another thread might be updating the configuration graph, and might conclude that e should be *deleted* (this would happen when some universal child of e returned SUCCESS, and we no longer need e or any of its children). We cannot delete e before the first thread finished processing it, and the second thread should know that another thread is working on e . Conversely, the thread which is creating the children of e should know that e should be deleted, once it finishes creating these new children.

Since we decided not to repeat **any** work twice, we might want to avoid deleting *EStates* for which we do not yet know an outcome, but which are not instrumental for the computation of the result at the moment. This could happen, for example, when a *UState* u is not needed any more (because we know the result of the parent of u), but some children of u have not returned a result as of yet. These children are not needed at the moment, but a different path from the root of the configuration graph might discover one of these children at some point in the future. We can keep the already expanded portion of the configuration graph in memory, without expanding it further, for some thread to pick it up later, should it become necessary again. Thus, a thread might want to *suspend* e and all its “orphaned” descendants (those descendants whose transitive closure of the parent “relation” does not include the root of the configuration graph), particularly when another thread is creating children for e . Thus, the “working thread” should know not to put the new *EState* descendants of e into the queue.

It is even conceivable that one thread decides to suspend e , and then another thread decides to *resume* it again, and all that could happen while a third thread is still expanding children of e .

Finally, we could have a situation in which a new *EState* e is created and added to the queue of the scheduler, but, before a thread could retrieve e to work on it, another thread decides that e should be suspended. This thread should know that e is in the queue and remove e from it.

We can summarize the states and the possible state transitions of an *EState* in Figure 5.2.

The dotted lines represent transitions which happen as part of the *Thread.update()* method, without being initiated by another method call. The meaning of the individual values is the following:

- WAITING — this is the “default” state, it represents an *EState* sitting in the

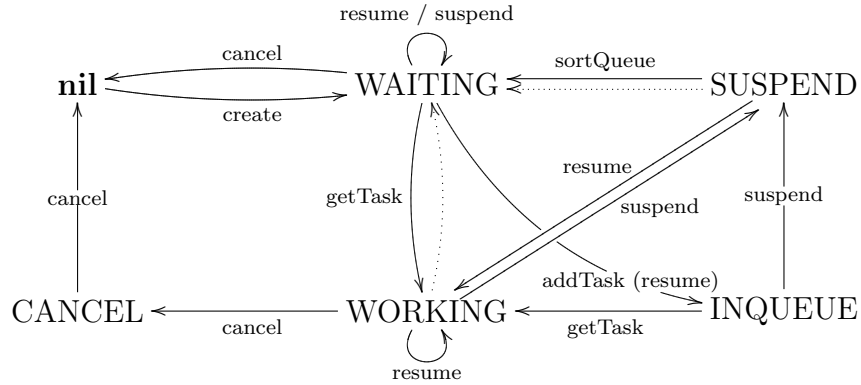


Figure 5.2: *EState.state* State Transition Diagram

configuration graph without a thread creating any of its children. A WAITING configuration could also have been suspended. In any case, no other thread is currently changing the *EState* and we can delete it safely.

- WORKING — this state indicates that a thread acquired this *EState* to work on. Other threads might change its relationship to its parents or children, but they are not allowed to delete or suspend a WORKING configuration directly. Instead, they change the state to either one of CANCEL or SUSPEND.
- CANCEL — indicates that a thread is still working on the *EState*, but another thread wants to delete it.
- SUSPEND — indicates that a thread is still working on the *EState*, but another thread wants to suspend it. If another thread decides to resume it again, the state is simply changed back to WORKING.
- INQUEUE — indicates that the *EState* is currently in the queue.

Before going into the details of the *Thread.update()* method, we will now talk about how the queue should prioritize between different *EStates* that placed in it.

To handle priority issues, we will have a field called *priority* for both *EStates* and *UStates*. *UStates* will need the *priority* field to propagate their priority to their

EState children. The scheduler’s queue will now be a *priority queue* putting *EStates* with high priority at the front of the queue. The method adding an *EState* to the queue is called *Scheduler.addTask(EState)*.

The scheduler will keep track of the lowest priority of any configuration in the configuration graph. This does not include suspended configurations, which are “orphaned” and thus are not technically part of the configuration graph. Whenever a new lower priority is needed, the method *Scheduler.newLowestPriority()* returns that priority and updates the scheduler’s internal counter.

When a configuration is created, its *priority* is initialized to a constant, *LOWEST_PRIORITY*, which represents that the configuration is suspended. When a configuration is inserted into the configuration graph, its *priority* should be updated according to the following rules: Suppose we have a *UState* u , which has several children. All children are equally important in the computation of the outcome of u , that is until one of them returns FAILURE, in which case we can suspend all the other active children, provided they don’t have another active parent. But as long as we do not know any of the children to fail yet, all children (and their descendants) should receive equal *priority*. On the other hand, suppose we have an *EState* e . If we create the first universal child u of e , surely it should receive the same priority as e . If e already has children $u_1, \dots, u_j$, and we create a new child u_{j+1} , it should receive lower priority than any of the existing children. This is due to the fact that we assume that *ATM.nextChild(EState)* creates new *UStates* using a heuristic which produces *UStates* likely to lead to a SUCCESS quickly first, before creating all other possible universal successors.

5.6.3.1 Cancelling and Suspending Configurations

Of course we will have to deal with priorities, as configurations are deleted and suspended from the configuration graph. The deletion of configurations, for which the answer has been computed, performed by the methods *Scheduler.cancel(EState)* and

Algorithm 26 *Scheduler.cancel(EState)* and *Scheduler.cancel(UState)*

```
1: method Scheduler.cancel(e: EState) {
2:   for all (var u: UState ∈ e.children) {
3:     cancel(u)
4:   }
5:   e.children.clear()
6:   if (e.state = WAITING) {
7:     delete e
8:   } else {
9:     e.state := CANCEL
10:  }
11: }
12:
13: method Scheduler.cancel(u: UState) {
14:   for all (var e: EState ∈ u.children) {
15:     e.parents.remove(u)
16:     suspend(e)
17:   }
18:   delete u
19: }
```

Scheduler.cancel(UState), is presented in Algorithm 26. Note, that the grandchildren of the cancelled *EState* are merely suspended, since they might be required in a different computation branch later.

We now look a bit closer at suspending configurations.

Suppose that we are suspending an *EState* *e* (because its parent *u* has been cancelled). If *e* does not have any other parents, this is not a problem. If it does, then we may have to change the priority of *e* (and its descendants) to that of its other active parent. If it has several active parents, its new priority should be the highest priority of any of its active parents.

If, during a suspension process, we need to suspend an *EState* which is in the queue, we should remove it from there. Also, if an *EState* in the queue changes priority, we should sort the queue, to account for the new priorities. Both these tasks are accomplished with the method *Scheduler.sortQueue()*, which also removes elements in the state SUSPEND from the queue.

The methods *Scheduler.suspend(EState)* and *Scheduler.suspend(UState)* are presented in Algorithm 27.

Algorithm 27 *Scheduler.suspend(EState)* and *Scheduler.suspend(UState)*

```

1: method Scheduler.suspend(e: EState) {
2:   if (e.state = CANCEL) {
3:     return
4:   }
5:   var newpriority: := LOWEST_PRIORITY
6:   for all (var u: UState ∈ e.parents) {
7:     newpriority := max(newpriority, u.priority)
8:   }
9:   if (newpriority = LOWEST_PRIORITY) {
10:    e.priority := newpriority
11:    if (e.state = INQUEUE) {
12:      e.state := SUSPEND;
13:      sortQueue()
14:    } else if (e.state ≠ WAITING) {
15:      e.state := SUSPEND
16:    }
17:    for all (var u: UState ∈ e.children) {
18:      suspend(u)
19:    }
20:  } else if (e.priority ≠ newpriority) {
21:    e.priority := newpriority
22:    resume(e)
23:  }
24: }
25:
26: method Scheduler.suspend(u: UState) {
27:   u.priority := LOWEST_PRIORITY
28:   for all (var e: EState ∈ u.children) {
29:     suspend(e)
30:   }
31: }
```

5.6.3.2 Resuming Configurations and Changing Priorities

The dual operation to suspending configurations is resuming configurations. It is performed by the methods *Scheduler.resume(EState)* and *Scheduler.resume(UState)*, presented in Algorithm 28. And since we represent suspended configurations using

their priority, we can also use the same functions to change the priorities of active configurations.

Algorithm 28 *Scheduler.resume(EState)* and *Scheduler.resume(UState)*

```

1: method Scheduler.resume(e: EState) {
2:   if (e.state = CANCEL) {
3:     return
4:   } else if (e.state = INQUEUE) {
5:     sortQueue()
6:     return
7:   } else if (e.state = SUSPEND) {
8:     e.state := WORKING
9:   }
10:  if (not e.children.empty()) {
11:    var u1: UState := e.children.first()
12:    u1.priority := e.priority
13:    resume(u1)
14:    for all (var u: UState ∈ e.children \ {u1}) {
15:      u.priority := newLowestPriority()
16:      resume(u)
17:    }
18:  } else {
19:    addTask(e)
20:  }
21: }
22:
23: method Scheduler.resume(u: UState) { u.state := WAITING
24:   for all (var e: EState ∈ u.children) {
25:     if (e.priority < u.priority) {
26:       e.priority := u.priority
27:       resume(e)
28:     }
29:   }
30: }
```

There are several further scenarios in which we may need to change the priority of a configuration other than the ones already mentioned: suppose that a universal configuration *u* computes a child *e* which is already present in the table. If we know the result (FAILURE or SUCCESS), we either immediately fail *u* or simply ignore the child *e*, respectively. If, on the other hand, *e* has not returned a result yet, we

would want to attach e to u in that part of the configuration graph. There are three possibilities for the situation of e here: If e is suspended, we can simply resume it with the same priority as u . If e has lower priority than u , we must set its new priority to be the same as that of u , and also update the priorities of its descendants. If e has a higher priority than u , then we can leave it and all its descendants as they are. Remember, that an existing *EState* cannot have the same priority as u , because that would contradict Property 5.1.

In the case that we do update the priority of e , we have to update the priority of its first child to be the same as e itself. Any possible younger children should receive new lower priorities.

Finally, the whole point of resuming a branch of computation is to make configurations in that branch appear in the queue again. We should thus add those resumed *EStates* without any children to the queue.

A non-case of changing priorities can happen, when a universal configuration returns FAILURE: suppose *EState* e has children $u_1, \dots, u_j$ and suppose that a child u_i returns FAILURE. If $i > 1$, we can simply cancel u_i . If, however, $i = 1$, then we are cancelling the first child of e which had the same priority as e . Do we need to update the priorities of $u_2, \dots, u_j$? Since the creation of additional universal children is done using a breadth-first approach, the priority of u_2 will only be “slightly” lower than that of u_1 and there should not be any other states with a priority in-between those two.² Hence, the program should behave in exactly the same way, whether the priorities are changed or not. So in this case, it is not necessary to update the priorities of the children of e .

Algorithm 29 *Scheduler.fail(EState/UState) and Scheduler.succeed(EState, UState)*

```

1: method Scheduler.fail(u: UState) {
2:   var p: EState := u.parent
3:   if (p ≠ nil) {
4:     p.children.remove(u)
5:     cancel(u)
6:     update(p)
7:   } else {
8:     cancel(p)
9:   }
10: }
11:
12: method Scheduler.fail(e: EState) {
13:   table.put (e.getID(), FAILURE)
14:   var ps: List⟨UState⟩ := e.parents
15:   for all (var u: UState ∈ ps) {
16:     u.children.remove(e)
17:   }
18:   if (e = root) {
19:     finish()
20:   }
21:   cancel(e)
22:   for all (var u: UState ∈ ps) {
23:     fail(u)
24:   }
25: }
26:
27: method Scheduler.succeed(e: EState, u: UState) {
28:   table.put (e.getID(), u.getID())
29:   var ps: List⟨UState⟩ := e.parents
30:   for all (var u: UState ∈ ps) {
31:     u.children.remove(e)
32:   }
33:   if (e = root) {
34:     finish()
35:   }
36:   cancel(e)
37:   for all (var u: UState ∈ ps) {
38:     update(u)
39:   }
40: }

```

5.6.3.3 Failing and Succeeding Configurations

Cancellation of a configuration happens whenever we have enough information to conclude that the configuration is either failing or successful. The suspension and possible change of priorities of other configurations are by-products of this process. To make the exact process of which configurations get cancelled more transparent we introduce the methods *Scheduler.fail(EState)*, *Scheduler.fail(UState)* and *Scheduler.succeed(EState, UState)*, which update the table and also act as initiators of *Scheduler.cancel(EState)* and *Scheduler.cancel(UState)*. They are presented in Algorithm 29.

Scheduler.fail(EState) should be called, whenever we are sure that an *EState* fails, i.e. when all its children failed and it can produce no more children. *Scheduler.fail(UState)* should be called, whenever we are sure that a *UState* fails, i.e. as soon as one of its children fails. *Scheduler.succeed(e, u)* should be called, as soon as *u* succeeds, i.e. as soon as it has no more children in the configuration graph (this will be explained later), and if *e* is the parent of *u*.

As soon as a configuration succeeds or fails, it might affect its parents. We make use of the auxiliary methods *Scheduler.update(EState)* and *Scheduler.update(UState)*, presented in Algorithm 30, to check whether the changes on the parent should result in a new call to *Scheduler.fail* or *Scheduler.succeed*, or if possibly an *EState* needs to be put into the queue.

5.6.3.4 Putting Things Together

We finally have all the parts to update the configuration graph after a thread has created a new *UState* and its children, or concluded that no more children can be

² It is possible that a different additional universal configuration is created after u_1 and before u_2 due to synchronization issues. However, this occurs very rarely, does not change the soundness of the program, and does not significantly affect its performance.

Algorithm 30 *Scheduler.update(EState)* and *Scheduler.update(UState)*

```
1: method Scheduler.update(e: EState) {
2:   if (e.state ≠ WAITING) {
3:     return
4:   }
5:   if (e.children.empty()) {
6:     if (e.hasMoreChildren and e.priority > LOWEST_PRIORITY) {
7:       addTask(e)
8:     } else {
9:       fail(e)
10:    }
11:  }
12: }
13:
14: method Scheduler.update(u: UState) {
15:   if (u.children.empty()) {
16:     succeed(u.parent, u)
17:   }
18: }
```

created. The code for the method *Thread.update()* is presented in Algorithm 31 and Algorithm 32.

First, we check whether *EState task* was suspended by another thread and set the flag *suspend* accordingly. If it was not, we check whether it was cancelled. If it was, we simply delete all the newly computed configurations (*newUstate* and *newEstates*), since they are not required any more. There is no point in storing any of the *newEstates* in the table, since they either already exist there, or another *UState* will create them again, and we would not save any work.

If *task* was not cancelled, we first set its *state* to WAITING, so that the methods *Scheduler.fail* and *Scheduler.succeed* can operate on it properly. We then check whether *newUstate* was created at all. If it was not, we can either fail *task* immediately (if it has no other children), or leave it WAITING for other children to return.

If *newUstate* was created, however, we add it to the configuration graph as a child

Algorithm 31 *Thread.update()* — Part I

```
1: method Thread.update() {
2:   var fails: Boolean := false
3:   var suspend: Boolean := false
4:
5:   if (task.state = SUSPEND) {
6:     suspend := true
7:   }
8:
9:   if (task.state = CANCEL) {
10:    delete newUstate and all elements of newEstates
11:    scheduler.cancel(task)
12:    return
13:  }
14:
15:  task.state := WAITING
16:
17:  if (newUstate = nil) {
18:    if (task.children.empty()) {
19:      scheduler.fail(task)
20:    }
21:    return
22:  }
23:
24:  task.children.add(newUstate)
25:  newUstate.parent := task
26:
27:  for all (var e: EState ∈ newEstates) {
28:    if (table.contains(e.getID())) {
29:      if (fails) {
30:        delete e
31:      } else if (table.value(e.getID()) = FAILURE) {
32:        fails := true
33:      } else if (table.value(e.getID()) ≠ SUCCESS) {
34:        newUstate.children.add(table.value(e.getID()))
35:        table.value(e.getID()).parents.add(newUstate)
36:      }
37:    } else {
38:      newUstate.children.add(e)
39:      e.parents.add(newUstate)
40:    }
41:  }
42:  ...
43: }
```

▷ continued on next page

Algorithm 32 *Thread.update()* — Part II

```
1: method Thread.update() {  
2:   ...  
3:   ▷ continued from previous page  
4:  
5:   if (fails) {  
6:     scheduler.fail(newUstate)  
7:     return  
8:   }  
9:  
10:  if (newUstate.children.empty()) {  
11:    scheduler.succeed(task, newUstate)  
12:    return  
13:  }  
14:  
15:  if (suspend) {  
16:    return  
17:  }  
18:  
19:  if (|task.children| = 1) {  
20:    newUstate.priority := task.priority  
21:  } else {  
22:    newUstate.priority := scheduler.newLowestPriority()  
23:  }  
24:  scheduler.resume(newUstate)  
25: }
```

of *task*. We then process its children, stored in *newEstates*. We add each such child, or its “duplicate” already existing in the table, to the *children* list of *newUstate*. We ignore children for which the result is SUCCESS or FAILURE, but set the flag *fails*, if one child is known to fail. Once one child is known to fail, we start deleting all the remaining children directly, for the same reason as when *task* was cancelled.

After we added all the children, if the flag *fails* is set, we can fail *newUstate* directly, taking care of any previously added children. Remember, that *Scheduler.fail* will add *task* back to the queue, if necessary.

If, on the other hand, *newUstate* has no children and *fails* is not set, this means that *newUstate* succeeds. Hence we can call *Scheduler.succeed(task, newUstate)*.

If we cannot succeed or fail immediately, we either just leave the configuration graph as it is, if *suspend* is set, or we update the priority of *newUstate*, according to our rules described previously, if it is not. Again, remember, that calling *Scheduler.resume(newUstate)* will add any children or descendants of *newUstate* to the queue, if necessary.

5.6.4 Starting and Stopping the Scheduler

Finally, we show how the scheduler is started, and how it signals its threads when to stop. The pseudo-code is presented in Algorithm 33.

Algorithm 33 *Scheduler.start()* and *Scheduler.finish()*

```

1: method Scheduler.start() {
2:   root := atm.initialConfiguration()
3:   addTask(root)
4:   table.store(root.getID(), root)
5:   for all (var t: Thread ∈ threads) {
6:     t.start()
7:   }
8:   for all (var t: Thread ∈ threads) {
9:     t.join()
10:  }
11: }
12: method Scheduler.finish() {
13:   result := table.value(root.getID())
14:   addTask(nil)
15: }
```

After the method *Scheduler.start()* returns (that is, after one of the threads calls *Scheduler.finish()* and eventually all threads finish executing the method *Thread.run()* and join the main execution thread), the result is available in the field *Scheduler.result*.

5.7 Reflections

We have analyzed an alternative definition of valid computations for an Alternating Turing Machine, as a certain set of paths in the configuration graph, rather than

an accepting computation tree. We then looked at the structure of such graphs for problems in the complexity class LOGCFL. We then devised an algorithm which finds such paths, and showed its correctness by giving semantics to partial computations and showing that the algorithm converges to a fixed point. We also briefly indicated how this algorithm could be implemented in practice, by taking inspiration from the algorithm *det- k -decomp*, presented in Subsection 4.3.2.

We then analyzed which search strategies could yield better runtime results, and indicated how parallelization could be introduced into the search. We concluded that an asynchronous shared-memory approach would fit this task best.

In the last section, we develop such a parallel algorithm, also providing an interface, which a programmer can use to execute any ATM algorithms in LOGCFL, or, more generally, any alternating algorithms fulfilling the conditions on the configuration graph described earlier. The parallel speed-up relies on the fact that different portions of the configuration graph get explored quickly, and once the same configuration is reached via different paths, it will already be well explored. The speed-up will heavily depend on the alternating algorithm itself, and most of all on whether the problem instance succeeds or fails. In case of FAILURE, the speed-up will generally be at most linear, since the configuration graph will have to be explored in almost its entirety. If the instance succeeds, the speed-up can be much more significant, even when using few **physical** processors.³

There are several directions for future research:

One could extend the scheduler to be able to handle **any** ATM program. The modifications may include changes to the interfaces, changes to the table (we will have to include universal configurations as well), and possibly even changes to the search heuristics. It is however questionable whether such a scheduler would be of any advantage, since it should be able to support any alternating program, including

³ Running two to four times as many threads as there are physical processors available seems to give the best results, as the overhead of switching between threads is still negligible.

PTIME-complete problems (in $\text{ASpace}(\log n)$), which are believed to be inherently sequential. The parallel speed-up reachable in this case would at best be linear, but probably much worse.

Independently of this, we would like to extend the framework to clusters of multi-core systems, in which a scheduler is run on each individual node, and the nodes use message passing to communicate and solve the problem together. The first approach would be using MPI [34] and synchronization at certain “heartbeat” intervals. Later on we would like to have an entirely asynchronous network “scheduler” leading to an optimal speed-up. An inherent problem to this is how to distribute the table over the individual nodes of the cluster, such as to avoid too much repetition of work, but also to avoid too much communication and the exceeding of each node’s memory bounds.

Going in another direction, one could rewrite such a synchronized “distributed scheduler” to work on PUBWCL, the Java implementation of BSP [10], and tackle ATM problems using potentially a very large number of computers on the internet. The parallelization will now be fully distributed, with every node running possibly only a single thread.

Finally, another direction of research would be to implement the scheduler in Java (once Java 7 comes out, supposedly allowing more control over the physical mapping of logical Threads), and extend the *Alter-Java* compiler [20] to produce code directly runnable on this scheduler.

Chapter 6

Implementation and Experimental Results

6.1 Roadmap

In Section 6.2 we discuss the particulars of the implementation of the scheduler described in Chapter 5. In particular, in Subsection 6.2.1 we present several languages and libraries that were considered for the implementation and discuss their advantages and disadvantages. In Subsection 6.2.2 we briefly mention several unsuccessful approaches undertaken in the early stages of this project that were abandoned partly due to insufficient research performed at the time, partly due to bad planning, but also due to deficiencies of the relevant technologies. In Subsection 6.2.3 we then describe the final implementation that was actually used for the experiments presented in Section 6.4. We also mention performance issues in Subsection 6.2.4 which have to be considered in a program using our scheduler.

We then proceed to discuss in Subsection 6.3.1 the data structures used to implement the algorithms from Section 4.3 which are the ingredients of both sequential *det- k -decomp* and **par-det- k -decomp** — the parallelized version of *det- k -decomp* using the above scheduler. In Subsection 6.3.2 we discuss several performance issues — in particular the memory requirements — that arise when using these data structures for *par-det- k -decomp*.

Finally, we present the experimental results of par-det- k -decomp in Section 6.4 and conclude the chapter in Section 6.5.

6.2 The Scheduler

6.2.1 Languages and Libraries

We tested Java, C and C++ as languages for the implementation of the scheduler and par-det- k -decomp. We also came across the following libraries for these languages, which we deemed useful for parallelization: PUBWCL and the Standard Java threading library for Java; MPI, OpenMP, Intel TBB and the Pthreads library for C/C++.

Java is a flexible and easy platform for development with very good libraries, however, as an interpreted language running in a virtual machine it leaves little control over the mapping of green threads¹ to system threads and no control over the memory management. C is a relatively low level language leaving control over both memory and thread management to the programmer giving him more freedom, but also more room for error. C++ is more or less a high level extension to C, most importantly including object-orientation, which still leaves a lot of control to the programmer.

PUBWCL [10] is a BSP (bulk synchronous parallel) style library for Java developed specifically for distributed computing over the internet. The Pthreads library [5] is a low level library for C that allows the programmer to create threads which are then mapped to system threads directly. It is the programmers responsibility to take care of any synchronization issues like race conditions and deadlock through mutexes. MPI (Message Passing Interface) [34] is a library developed for high-performance distributed computing in a cluster of computers. OpenMP (Multi-Processing) [44] is an extension of C/C++ with capabilities to parallelize many loop structures on a multi-core shared-memory machine, with minimal “programming overhead”. These extensions are written out as preprocessor instructions (*pragmas*) and the main idea

¹Threads scheduled by the virtual machine

behind OpenMP is that the same source code should compile with or without them and OpenMP support producing programs with the same external behaviour. Intel TBB (Threading Building Blocks) [3] is a more flexible library than OpenMP which allows to write programs taking advantage of multi-core processors, while abstracting away most of the thread management. It is only available for C++, since it uses object-orientation extensively.

6.2.2 Abandoned Approaches

After some initial tests with PUBWCL, we decided to abandon distributed parallelism altogether and to concentrate on the Shared-Memory approach. We then soon abandoned the idea of using Java as the main development platform, since the Java threads in Java 6 did not map well to system threads, thus not taking any advantage of multiple cores.²

We then considered OpenMP as a quick way to parallelize the simulation of an alternating program like the computation of a hypertree decomposition. Our initial approach was to use an OpenMP-enabled loop to compute new nodes in the configuration graph. We would pick a number of nodes for which to compute new children (the set C mentioned in Section 5.4), put them in an array and then execute *ATM.nextChild()* on every element of the array in an OpenMP-enabled loop. After that, a single thread updated the configuration graph and the table, processing the array elements one by one. The advantage of this approach is that we did not need to bother about synchronization of threads.

It carried a number of problems, however. One problem is the overhead linked to creating new threads to run the loop once. OpenMP loops work well when the range of the index variable is a very large number, that is when the execution of the whole loop takes a long time, whereas the loop itself is not called very often. However,

² Waiting for Java 7 to come out (which promises to fix these shortcomings) would have meant a substantial delay to the completion of this dissertation.

in our approach running a loop once would correspond to processing all *EStates* in the queue. Thus, the arrays would be rather small in the beginning. Even if the algorithm provides high branching at universal configurations, the most number of configurations being processed by a loop at any one time would still be linear in the size of the problem, if we use breadth-first search on universal configurations only. To get a large number of *EStates* into the queue we would have to perform breadth-first search on *EStates* extensively (and not only when processors are idle as in the asynchronous approach). In particular, we would often work on an *EState* and all its *EState* ancestors simultaneously. This potentially leads to work being done “in vain”. For example if the first node in the array of the loop delivers a negative or positive result, which, after propagation, would imply the success or failure of some of its ancestors, which are also in the array, we would not need any of the computations performed on them. However, the loop will have to be finished first, before the configuration graph is updated, thus wasting a lot of valuable processor time.

But probably the most important problem with the OpenMP loop approach is that a single repetition of the loop (a single call to *nextChild()*) will vary hugely in execution time for different elements of the array. OpenMP is not aware of this heterogeneity and will simply divide the *EStates* evenly between the cores, such that each core processes the same number of *EStates*. It is then very well possible that one core will be unlucky and have all those *EStates* for which it takes the longest to compute a child, and all the other cores will be sitting idle waiting for that one core to finish.

Empirical results showed that the speed-up achieved with OpenMP on a quad-core machine over sequential computation was negligible (in fact, on many instances the OpenMP version was slower than the sequential version, due to overhead). Together with OpenMP we also abandoned the pure C approach, and opted out for the flexibility of C++ instead.

We investigated the Intel Threading Building Blocks as a candidate to implement the scheduler of Chapter 5. Providing a wide range of thread models and very clean interfaces, this is an excellent C++ library suited for many parallel computation tasks requiring more flexible scheduling than loop partitioning.³ In particular, one of the facilities it provides is a task-scheduler and an interface for task-based parallelism. This is exactly what we need in our approach, except for one caveat.

A TBB task can spawn several children and then wait for the children to finish execution and return some results. It can then either do some more computations, possibly spawning some new children, or it can process the results, compute its own result and terminate. But the one thing it always does is wait for **all** children to return results before resuming its own computation.

We, however, need configurations to be able to process **any** one result, without having to wait for its other children. We also want to be able to cancel any potentially existing descendant tasks.

Testing the TBB library revealed that if we are lucky enough for the children of universal states high up in the configuration graph to complete computation in roughly the same time, we can still get linear speed-up.⁴ This indicated the soundness of the *Task Parallelism* approach. However, we could not hope for a super-linear speed-up, because parallelization at existential configurations would mean we had to wait for all its children to finish processing, and this is the parallelization that can deliver super-linear speed-up (due to different parts of the configuration graph being explored right from the beginning). Moreover, we were only allowed to have a *tree* of tasks, rather than a *DAG*, which would heavily complicate parallelization at existential configurations.

³ Intel Threading Building Blocks provides easy interfaces for loop partitioning as well.

⁴ As examples we used highly symmetric hypergraphs, which were separated into isomorphic connected components quickly.

6.2.3 Successful Approach

Apart from these restrictions, the TBB task scheduler implements some very useful paradigms which we adopted in our task scheduler as well. One of them is using a fixed number of threads that work on a single task, whereas the tasks are managed in a global task pool. This is generally better than spawning entirely new threads every time we want to fork a computation, because of the overhead linked to the creation of new threads. Another paradigm we adopted is that of “depth-first work, breadth-first theft”, which precisely represents the idea of performing depth-first search on existential configurations and breadth-first search on universal configurations generally and on existential configurations, when a processor is idle, as was described in Chapter 5.

We thus decided to create a task scheduler “from scratch”, using the very low-level but efficient Pthreads library [5].⁵

Every one of our *Thread* objects has a field *thread* of type *pthread_t* and a private function called *thread_func()*, which are passed to *pthread_create*, thus creating a new thread, when *Thread.start()* is called. The function *thread_func()* simply calls *Thread.run()*. *Thread.join()* initiates *pthread_join* on *thread*, thus resulting in the main thread waiting for the forked thread to finish execution.

The *Scheduler* object has fields *mutex* and *cond* of types *pthread_mutex_t* and *pthread_cond_t*, respectively. These fields are used in the methods *Scheduler.lock()*, *Scheduler.unlock()*, *Scheduler.wait()* and *Scheduler.notify()*, by calling the functions *pthread_mutex_lock* and *pthread_mutex_unlock* on *mutex*, and *pthread_cond_wait* and *pthread_cond_broadcast* on *cond*, respectively.

Perhaps one final remark about the granularity of synchronization should be made at this point. There are many ways to use mutexes for synchronization. A very fine-grained approach would be to introduce a mutex for every task. The advantage of

⁵ In principle it should be very easy to change the code to use the *Boost Thread Library* [1] instead, thus making the scheduler even more portable.

this approach is the possibility to parallelize even the updating of the configuration graph. However, this approach requires great care in the design of the graph update algorithm and is very hard to implement. Also, as discussed earlier, most computation time should be spent on creating new configurations. Hence, updating the graph only takes a proportionally small amount of time. Locking and unlocking of mutexes brings with it an overhead, and when updating a long chain of configurations in the graph, a thread would have to lock and unlock quite a few of them incurring a rather large overhead. Thus, we chose to use a single mutex for the whole scheduler. This coarse-grained approach proved itself very appropriate for hypertree decompositions for instance, since in this case computing new configurations takes a lot longer than updating the configuration graph, and there was very little idle time in any of the threads.

The implementation of all the other structures in our algorithm from Chapter 5 is fairly straightforward in C++, using the *Standard Template Library* [52]. The only “non-trivial” data structure is the table of partial results. We decided to use the *hash_map* template from SGI’s STL extensions, which are included in common libraries such as the GNU C++ Library [2], and are scheduled to become part of the C++0x standard. Every object derived from the class *EStateID* will be required to override the method *hash()*, returning an integer, and the method *equalsTo(EStateID)*, returning a boolean. These methods are then used by the hash-table to compute hash values and compare keys in the table.

6.2.4 Performance Issues

As discussed before, creating new children for configurations should take up most of the computation time. Hence, the programmer needs to make sure a single computation of *nextChild()* is as efficient as possible.

As processors become ever faster, one of the greatest problems in hardware today

is probably memory bandwidth. Fast memory is expensive, so computer memory is usually built in several layers. Working from the processor core to the outside, each next layer of memory is usually larger than the previous one by an order of magnitude, but at the same time, the access time — from an instruction call, to data being available — is usually also slower by an order of magnitude.

We present as an example the Intel(R) Core(TM)2 QUAD Q8200 CPU [4]. It operates at a frequency of 2.33 GHz and has 4 cores. The fastest memory are the registers on the core, with instant access by the ALUs. The next outer layer is the Level 1 (L1) Cache, holding 32 KB of data memory and 32 KB of instruction memory. Each core has its own L1 Cache. Further away from the cores is the Level 2 (L2) Cache, holding 4MB of memory. In this particular processor 2 cores each share 2 MB of L2 Cache memory. Even further away is the Random Access Memory, which in modern machines is usually several gigabytes. Finally, if we run out of RAM, there is a possibility of using swap space on the hard drive, which can potentially be one terabyte or larger, but is very slow compared to RAM.⁶

The program data (and the program itself) will usually be stored in RAM. When a core is accessing data, it first checks whether a copy of the data is already stored in its cache. If it is not, then it locates the data in the RAM, and copies it from that location into its cache. Each such event is called a *cache miss*. The core then proceeds to work on this copy, until it needs further data which is again outside its cache. It then flushes the current contents of the cache back to the RAM at its old location and reads the data from the new RAM location back into the cache. Such a copy operation is expensive, and can become a speed bottleneck, if the program provokes too many cache misses. We thus would want a single task to be small enough to fit into the cache, to prevent any cache misses, while the core is working on that

⁶ One could even include external storage tapes in this hierarchy. In this case, however, instead of having a transparent abstraction of the various parts of memory through a contiguous program code, the programmer would need to give verbal instructions to an engineer to swap external tapes manually, and synchronization issues would take on a whole new meaning.

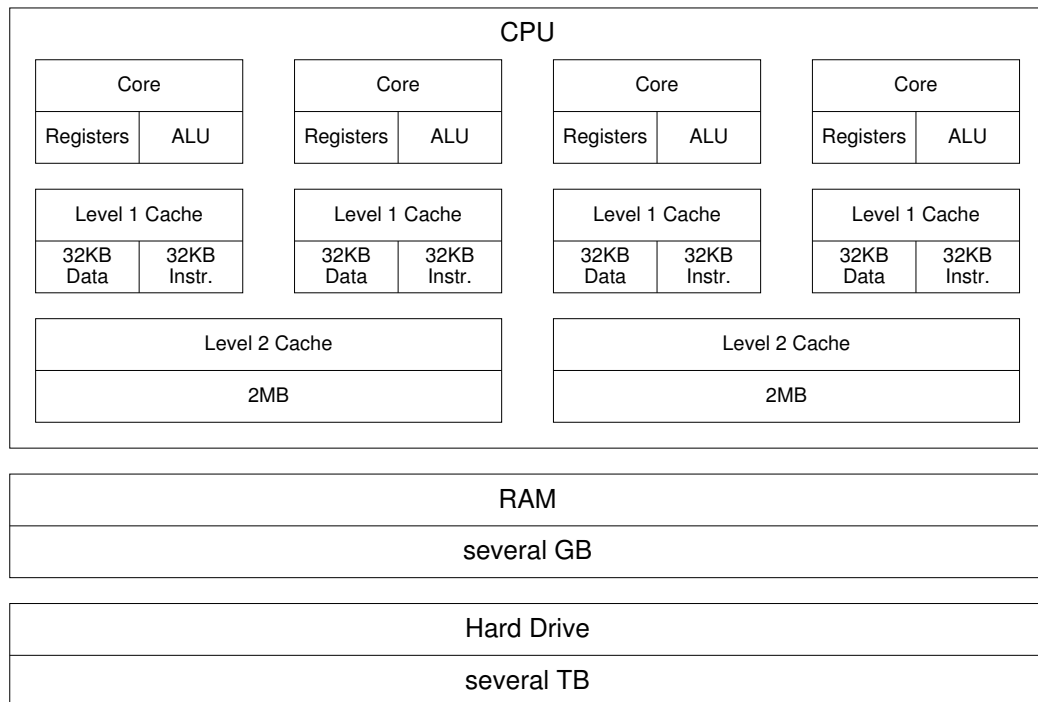


Figure 6.1: Memory Levels in an x86 Computer

task. Ideally, if the task is small enough to fit into the L1 Cache, we can increase performance even further, as this reduces the number of times data is copied between the L1 Cache and the L2 Cache, which is not as expensive as copying between L2 Cache and RAM, but still incurs a small overhead.

Another important point about how memory is copied to and from the cache is in “pages”, that is **contiguous chunks of memory** in RAM. A bad example of data to work on would be some structure having pointers to other objects which need to be accessed intensively by the algorithm, while the objects themselves are at a completely different address region in RAM. In this case, the program would provoke a lot of cache misses, slowing down the program by several orders of magnitude. Hence, implementations of *ATM*, *UState* and *EState* should also occupy small, contiguous chunks of memory, and *ATM.nextChild()* should not require any other data structures for a single execution.

6.3 Hypertree Decompositions

6.3.1 Data Structures

We generally adopted the habit of representing sets of vertices and sets of edges as *bitsets* of length m or n , where m is the number of vertices and n is the number of edges in the hypergraph. Only in a few instances do we represent a set of edges or a set of vertices as an integer array. We chose to represent a bitset as an integer pointer, and provided three macros to either read the value of an individual bit, to set it to 0 or to set it to 1. We also provided functions to create a bitset of specified length, and to compute the logical OR and AND of two bitsets. Bitsets turned out to be the best compromise between time and space requirements.

The *Hypergraph* object holds information about the number of edges and vertices (*numVs* and *numEs*), and the width of the hypertree decomposition we are looking for (*width*). The structure of the hypergraph itself is represented by two incidence matrices, *eToVsBool* and *vToEsBool*, where *eToVsBool[i]* is the bitset representing vertices contained in edge i , and *vToEsBool[j]* is the bitset representing edges containing vertex j . For convenience, we also include arrays of integer arrays for these two sets, called *vToEs* and *eToVs*, together with the arrays of their respective lengths, called *vToEsLength* and *eToVsLength*. Thus, for example, *vToEs[j][k]* is the k -th edge covering vertex j . Remember, that we sort every array *vToEs[j]* by the size of the edges. We also include the edge and vertex names, for output, stored as *vectors* of *strings*.

The *ComponentID* object uses a bitset representing the connecting set, and a vertex with the lowest number from inside the component. We chose this representation, rather than using the set of edges of the “old” separator, since in several instances different separators will produce the same component. This representation, even though it takes up a little bit more space, can save us from repeating work quite a few times.

The *SeparatorID* object simply contains the number of edges used in the separator and the integer array containing the numbers of the edges in ascending order.

Rather than having the *Separator* object hold much of the same information as its parent *Component* object, as described in Subsection 4.3.2, it will be purely nominal here. We will solely need *Separator* to override the method *getID()* properly. Thus, a *Separator* merely holds its *SeparatorID*. Returning new components using this separator will be all done through its “parent” *Component* object. Thus, executing *Hypergraph.nextChild(sep)*, will call and return *Hypergraph.nextComponent(sep.parent)*. The latter method returns a new component using the latest separator stored internally in *Component*, computed using *Hypergraph.nextSeparator(Component)*. Hence, we have to make sure these functions are called in the proper order (which is ensured by the scheduler, as well as by *det-k-decomp*).

A *Component* contains the bitsets *compVs*, *compEs*, *connVs*, *forbiddenEs*, the integer array *emphselectedEs*, and integers *index* and *oldIndex*. All these fields and their relevance were described in Subsection 4.3.3. Additionally, we have integers *numEs* and *numVs* (to know the length of the bitsets), *numCompEs*, indicating the number of 1’s in *compEs* (used for the short-cut heuristics), and *numSelectedEs*, indicating the number of edges selected for the separator in *selectedEs*. Finally, we also have the auxiliary variables *v*, used to go through the vertices of the hypergraph, and the bitsets *coveredVs*, indicating vertices currently covered by edges in *selectedEs*, and *componentCovered*, a boolean array, such that *componentCovered[numSelectedEs]* is true, whenever a component edge is included in the currently selected separator edges. These auxiliary variables are used to speed up the methods *Hypergraph.nextSeparator(Component)* and *Hypergraph.coverVertex(Component)* and *Hypergraph.removeEdge(Component)*. We update *coveredVs* and *componentCovered*, whenever we add or remove an edge from our separator. The variable *v* will always point to the first vertex which is in *connVs*, but not in *coveredVs*, or will be equal to

$numVs$, if this set is empty.

Once we have a valid separator, we point v at the first vertex in $compVs$ which is not in $coveredVs$. Whenever we call $Hypergraph.nextComponent(Component)$, we start the depth-first search from that vertex, and update $coveredVs$ to also contain all vertices selected for the new component. Once we return a new component, we set v again to point to the first vertex in $compVs$, which is not in $coveredVs$. Thus, once v is equal to $numVs$, we know that we already returned all new components.

Assuming that all hypergraphs have less than 2^{16} edges or vertices, we can use *short ints* to represent integers (taking up 16 bits = 2 bytes of memory). A *ComponentID* then takes up $n/8 + 4 + p$ bytes of memory, a *SeparatorID* takes up $2k + 2 + p$ bytes of memory, and a *Component* takes up $14 + 7p + 2k + k/8 + 3n/8 + 2m/8$ bytes of memory, where n is the number of edges in the hypergraph, m is the number of vertices in the hypergraph, k is the width of the hypertree decomposition, and p is the size of a pointer in bytes. Using the values $m = 1024, n = 1024, k = 24$ and $p = 4$ as an upper bound, we get maximal sizes of 136 bytes for *ComponentID*, 54 bytes for *SeparatorID* and 725 bytes for a *Component*. Thus, every entry in the hash-table will be under 1KB in size, and we could create up to a million components for every 1GB of RAM.

The size of a *Hypergraph* object (excluding the edge and vertex names) is $6 + 6p + n/8 + m/8 + 2m + 2n + np + mp + 4x$, where x is the number of 1's in the incidence matrix. Using the same values as before we get $12,574 + 4x$ bytes. Depending on the density of the hypergraph, x can vary substantially. An upper bound is mn , giving 4MB for the hypergraph in the worst case, but usually, it will be much less. Since $md_v = x = nd_e$, where d_e is the average degree of all edges and d_v is the average degree of all vertices, we can in fact get hypergraphs of size less than 30KB, if the average degree is at most 4. Thus, in an ideal case, the hypergraph **and** a component could fit into the processor's level 1 cache, speeding up the computation substantially.

Of course, a trick has to be used to make sure the memory occupied by the hypergraph and a component is contiguous: instead of initializing memory for every array independently, we initialize memory for all arrays in one go, and then set the array pointers to the correct positions.

6.3.2 Performance

As was mentioned earlier, our hypertree decomposition algorithm (as well as the original $\text{det-}k\text{-decomp}$) takes much longer to run, when the hypergraph cannot be decomposed with the specified width. The reason for this is that practically the entire configuration graph needs to be explored. As we increase the parameter k , the running time increases dramatically (remember that k was in the exponent of the upper bound on running time). The worst case happens, when for a hypergraph H we try to decompose it with $k = \text{HW}(H) - 1$. However, even setting $k = 3$ has catastrophic consequences for large hypergraphs.

Memory usage also becomes an issue in this case. Remember that for a hypergraph with 1024 edges and 1024 vertices, a *ComponentID* requires 136 bytes of storage space, while a *SeparatorID* requires 12 bytes, if $k = 3$. Thus using one Gigabyte of memory we could store around 7.3×10^6 entries in our table. However, the upper bound on the number of possible components is 10^{12} . Even though the real number of distinct components is usually much less than that (closer to 10^9), we would still need almost 140 Gigabytes of main memory. We could of course change the representation of *ComponentID* to consist of a separator and a component edge or vertex, thus reducing its space usage to 14 bytes. We could then store up to 3.8×10^7 entries for every Gigabyte, and we would “merely” need 27 Gigabytes of RAM for 10^9 entries.

Machines with this sort of memory specifications are not uncommon in modern supercomputing centres. However, from the moment a *Component* is created, it is stored in the table until its result is eventually computed, or the program terminates,

even if it becomes suspended during the computation. While trying to construct a decomposition of a width which is smaller than the actual hypertree width, it is not unlikely that many components will be suspended, especially if many separators actually partition the hypergraph into two or more components. A *Component* needs 689 bytes of space, and one Gigabyte would only give us 1.5×10^6 table entries with a *Component* value. Thus even 64 Gigabytes of RAM can only cope with at most 10^7 “working” table entries.

On the other hand, if $k \geq \text{HW}(H)$, then in many instances increasing k can reduce the running time substantially. The reason for this is that larger separators can cover larger connecting sets, and hence it is easier to compute *Hypergraph.nextSeparator()*, since valid separators can be found quicker.

Thus, for large examples, it is sometimes very difficult to determine the exact hypertree width of a given hypergraph.

6.4 Experimental Results

For our experiments we used three different classes of hypergraph instances [23, 19]. The first class is from DaimlerChrysler and consists of hypergraphs extracted from adder and bridge circuits, the NewSystem examples, and a model of a jet propulsion system. The first class of benchmarks are hypergraphs extracted from twodimensional grids. These instances are interesting since their hypertree-width is known by construction, and an optimal decomposition of these examples can easily be constructed by hand. The other class of benchmarks are hypergraphs extracted from circuits of the ISCAS89 (International Symposium on Circuits and Systems) benchmark suite. The ISCAS benchmarks are examples from industry, and the exact hypertree of most of them is still unknown.

Our evaluations were performed on a machine with an Intel® Core™2 Quad Processor Q8200 (4×2.33 GHz), 6 GB RAM, and running Ubuntu 9.10 Linux. We

used 16 threads to maximize the coverage of the hypergraphs straight from the start. Using more threads than 16 threads (four times the number of cores) leads to a decline in performance due to increased overhead linked to thread scheduling by the kernel.

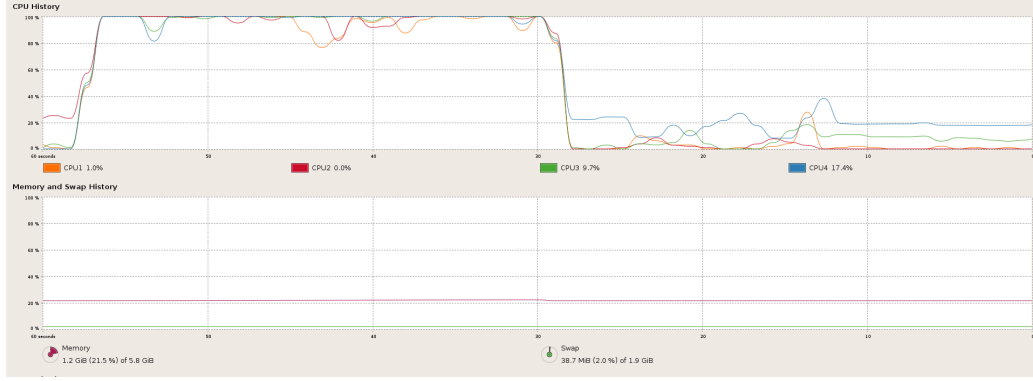


Figure 6.2: CPU and Memory Usage

Figure 6.2 illustrates how all four cores are operating at maximum capacity during a test run of the algorithm.

For comparison we also make reference to the experimental results of [32], in which $\text{det-}k\text{-decomp}$ was executed on a 2.2 GHz Intel® Xeon® processor (dual) with 2 Gigabytes of RAM running SuSe Linux 9.2. The original $\text{det-}k\text{-decomp}$ algorithm is *randomized* (it uses a new random ordering of the edges and vertices at every run) to assure that the running time does not depend on the representation of the hypergraph. Our parallel algorithm always sorts its edges in a specific order, as was described in Subsection 4.5.3. To get better consistency between different runs, we did not randomize the order of the edges or vertices before sorting them. Our algorithm is thus to some extent dependent on the representation of the input, however it would be very easy to add randomization. There is still a degree of nondeterminism, linked to the scheduling of the threads by the operating system, over which the user has no control.

We present the test results in Table 6.1 and Table 6.2. The given execution times are in seconds. The third column (“Min”) shows the minimal hypertree width

Instance (Vertices/Edges)	Min	det- k -decomp		par-det- k -decomp	
		width	time	width	time
grid2d_10 (50/50)	4	4	0	4	0
grid2d_15 (113/112)	6	6	3		
grid2d_20 (200/200)	7	7	3140	7	35
grid2d_25 (313/312)	9	10	2000	10	28
grid2d_30 (450/450)	11	13	1566	13	941

Table 6.1: Grid2D Benchmarks

of a given hypergraph, if it is known. Sometimes we show a one-sided limit (e.g. $\text{HW}(H) \geq 3$), and a dash (—) represents no prior information about the hypertree width.

When decomposing “grid” hypergraphs the relative speed-up is astonishing, provided the decomposition width is small. In particular for the examples *grid2d_20* and *grid2d_25* our parallel algorithm outperforms the sequential one by factors 90 and 70, respectively. Since we are using four computational cores, the parallel efficiency is 22, respectively 17. For higher decomposition widths our algorithm’s performance declines drastically. The reason for this is a much higher execution time of a single call of *Hypergraph.nextSeparator*. This is due to the time-space tradeoff described in Section 6.2.

For the ISCAS benchmarks, we tested hypergraphs which are known to have a relatively small hypertree width. In most of these cases the relative speed-up is even higher than for “grid” hypergraphs. In particular, for the example *s713* we get a factor of 138 and hence a parallel efficiency of 35. Unfortunately, example *s386* illustrates that there exist exceptions to this sort of super-linear speed-up. The fact that the original det- k -decomp performed better for this example may be due to a “lucky” random representation of the hypergraph that was chosen for that particular run. The algorithm par-det- k -decomp is apparently stuck to an unlucky representation of the hypergraph, preventing it from finding a decomposition quickly (the different

Instance (Vertices/Edges)	Min	det- k -decomp		par-det- k -decomp	
		width	time	width	time
s208 (115/104)	≥ 3	6	0	6	0
s298 (139/133)	≥ 3	4	462	4	0
s344 (184/175)	≥ 3	5	730	5	0
s349 (185/176)	≥ 3	5	4	5	0
s382 (182/179)	≥ 3	5	722	5	0
s386 (172/165)	—	7	1824	7	2818
s400 (186/183)	≥ 3	5	273	5	0
s420 (231/212)	≥ 3	8	454	8	1
s444 (205/202)	≥ 3	5	385	5	0
s526 (217/214)	≥ 3	7	1715	7	21
s641 (433/398)	—	7	1611	7	16
s713 (447/412)	—	7	1800	7	13

Table 6.2: ISCAS89 Benchmarks

threads of computation do not join in together nicely.) Moreover, the slightly higher hypertreewidth is another important factor for slowing down par-det- k -decomp substantially.

Our algorithm is particularly well-suited to test upper bounds for “non-existence” of decompositions, because its data structures, in particular *Component* are small. We could show that both *s298* and *s444* as well as the *NewSystem3* example from the DaimlerChrysler hypergraphs do not have hypertree decompositions of width 3. The execution times of the runs were 20, 65 and 447 seconds respectively. We thus determined the exact hypertree width of *s298* and *NewSystem3* as 4.

Interestingly enough, the relative speed-up of our parallel algorithm for such “negative instances” is less than linear. The reason for this is that the complete configuration graph needs to be explored, and the program cannot stop early, so it does not really matter which thread does what, as long as they do different things. The thread management overhead, which becomes more apparent with an increasing number of threads, reduces the speed-up even further, in particular for small examples, where synchronization occurs frequently.

6.5 Reflections

We managed to demonstrate a super-linear speedup in par-det- k -decomp over det- k -decomp for many examples of hypergraphs and hypertree decomposition widths. In particular, for small widths (≤ 5) the results are remarkable.

For larger widths and big hypergraphs, however, the parallel speedup seems to decline quickly. This is mostly due to the specific hypertree decomposition algorithms and data structures used in our implementation of par-det- k -decomp. Also, memory becomes the biggest issue for large examples, since our scheduler keeps all configurations for which a result is not yet known in memory.

There is certainly still room for improvement. One could for instance improve the running time of the hypertree decomposition specific algorithms by increasing the size of the data structures. On the other hand one would then need better ways to deal with the space requirements. One possible approach is to keep only the information necessary to reconstruct a specific configuration in a particular state, rather than the whole configuration itself. Another approach would be to purge suspended configurations from memory after a certain time period in which they have not been resumed. In this case we could end up repeating the same work multiple times, however with a good heuristic for this sort of *garbage collection* on the configuration graph we could reduce such occasions to a minimum.

Chapter 7

Conclusion and Future Work

7.1 Conclusion

Answering BCQs is NP-hard, therefore it is important to study tractable subclasses of queries. Hypergraph decompositions, in particular hypertree decompositions, are a powerful tool in this respect.

Recent trends in modern hardware technologies make it important to exploit not only polynomial tractability of computational problems, but also their parallelizability. The membership of the recognition problem for hypertree decompositions of bounded width in the complexity class LOGCFL paves the road to the exploration of parallel algorithms.

In the process of finding such algorithms we explored new types of decomposition, the balanced and shallow decompositions, initially studied as a heuristic for hypertree decomposition computation. Complexity-theoretic analysis of the recognition and BCQ evaluation problems led us to define a new hierarchy of complexity classes, the DC-hierarchy, and we could show membership in DC^1 for the former problem and DC^1 -completeness for the latter.

We analyzed how the ideas of balanced decompositions could be used to compute hypertree decompositions, by guessing future nodes in the hypertree, rather than proceeding linearly, like the algorithm k -decomp, or its deterministic equivalent, $\text{det-}k$ -decomp. We proposed the alternating algorithm k -divide-decomp, which apart

from being in LOGCFL needs only $O(\log n)$ alternations, and whose deterministic equivalent is easily parallelizable, due to the shallowness and good branching of the associated computation tree. Unfortunately this deterministic algorithm's exuberant running time and space requirements ruled it out as a practical one. We could, however, still combine the divide and conquer approach with the original $\text{det-}k\text{-decomp}$ algorithm to devise $\text{par-det-}k\text{-decomp}$, which combines low asymptotic bounds with excellent parallelization.

We then proceeded to analyze how $\text{par-det-}k\text{-decomp}$ could be implemented using only a constant number of processors. To this end we exhibited a certain structure of configuration graphs of Alternating Turing Machines solving computational problems in LOGCFL, showed how accepting computation trees correspond to a set of paths in this graph, and discussed the effectiveness of various search strategies in such graphs. We then developed an asynchronous parallel algorithm implementable on shared-memory systems, which finds such sets of paths, and hence also an accepting computation tree for a problem in LOGCFL. We provided a programming interface, allowing a programmer to run this parallel algorithm on a variety of LOGCFL problems, by implementing a couple of classes and methods, while the parallelization process itself remains hidden in the library.

Finally, we used this API and the task scheduler, which was implemented in C++ using the Pthreads library, to create a program, which computes hypertree decompositions in parallel. We tested this program on a number of example hypergraphs, and could exhibit a super-linear speed-up in many cases.

7.2 Future Work

The purely theoretical part of this dissertation still leaves a few questions unanswered. In particular, it would be interesting to see, where exactly the DC-hierarchy fits its classes between NL and NP. For this, one would need to further explore the

relationship between resource-bounded (N)AuxSAs and other models of computation. As for balanced decompositions, the exact complexity of the recognition problem still remains unknown.

As for the practical aspect of this dissertation, we have unveiled a number of future directions:

As for the task scheduler itself, one could, for instance, extend it to handle more general alternating algorithms. Another possibility, is the development of a *distributed* algorithm, rather than a shared-memory one. Finally, of course, one could combine the distributed and the multi-core approach, by devising a *hybrid* parallel algorithm.

A much easier area to explore would be the application of the presented API to a whole range of problems solvable by low-complexity alternating algorithms.

As for the computation of hypertree decompositions themselves, one could try to find even better heuristics for $\text{det-}k\text{-decomp}$, and thus also for its parallelization, which might be based on even stricter types of hypertree decompositions, than the ones in strong normal form. Of course, there may (and will) be completely different parallel algorithms altogether.

Finally, an interesting topic to explore is the *combination of decomposition methods*. For instance, one could combine balanced decompositions with hypertree decompositions: A few steps in the Robber and Sergeants game from Section 3.4 will split up a large hypergraph into several components. Each such component could be decomposed into a hypertree decomposition. As long as the number of steps in the Robber and Sergeants game is constant, we can still guarantee the tractability of the BCQ evaluation problem. As for finding balanced decompositions, it would also greatly help to develop good algorithms and heuristics for finding balanced separators in a hypergraph.

Another possibility is to combine several hypertree decompositions of different parts of the hypergraph into a generalized hypertree decomposition.

Let us define a *partial strategy* in the $R\&M^k$ game on H for a subset $F \subseteq E(H)$ of edges as (T, ρ, λ) , which is a strategy in the usual sense, except that we do not have to cover edges outside of F : There is a node $p \in T$, such that $\rho(p) = E(H) \setminus F$, and we do not require a λ -label for that node, nor do we need a sub-tree rooted at p decomposing $E(H) \setminus F$. Moreover, none of the λ -labels are allowed to use edges outside F . The connecting set at p is $F \cap (E(H) \setminus F)$, and the λ -label of the parent of p needs to cover that set.

Theorem 7.1. *Let H be a hypergraph and k a positive integer. Let F_1, F_2 be a partition of $E(H)$. If there exist partial strategies (T_1, ρ_1, λ_1) and (T_2, ρ_2, λ_2) in $R\&M^k$ for F_1 and F_2 , respectively, then there exists a generalized hypertree decomposition of H of width k .*

Proof. (Sketch.) Let p_1 and p_2 be the nodes in T_1 and T_2 , respectively, such that $\rho_1(p_1) = F_2$ and $\rho_2(p_2) = F_1$, and let q_1 and q_2 be the parent nodes of p_1 and p_2 , respectively. By joining the trees T_1 and T_2 by setting $p_1 = q_2$ and $p_2 = q_1$ we get an (un-rooted) tree T (remember, that in generalized hypertree decompositions the tree does not have to be rooted). If we keep the λ -labels from the previous two trees, it is a mere technicality to show that we can define the χ -labels to get a valid generalized hypertree decomposition. $\square$

The above theorem can be adjusted to any number of components and partial decompositions (possibly having several “connecting nodes”), provided that every connecting set is the border between precisely two components. One could thus adjust our parallel algorithm to recognize when two partial decompositions have the same connecting set, and combine them into a new partial decomposition. The advantage of this approach, is that we are not restricted to finding decompositions only going one way, and a parallel algorithm working in different directions simultaneously could

cover the hypergraph twice as fast, as a parallel algorithm having to wait to reach another branch going in the same direction.

Of course, the recognition of hypergraphs with generalized hypertree width at least 3 remains NP-hard, and this is not a valid method to construct minimal-width generalized hypertree decompositions. However, we could still compute generalized hypertree decompositions of strictly smaller width than the hypertree width of the hypergraph.

Bibliography

- [1] Boost C++ Libraries. <http://www.boost.org>.
- [2] GCC, the GNU Compiler Collection. <http://gcc.gnu.org>.
- [3] Intel Threading Building Blocks. <http://www.threadingbuildingblocks.org>.
- [4] Intel® Core2™Quad CPU Q8200.
<http://ark.intel.com/Product.aspx?id=36547>.
- [5] POSIX Threads. http://en.wikipedia.org/wiki/POSIX_Threads.
- [6] ABITEBOUL, S., HULL, R., AND VIANU, V. *Foundations of Databases*. Addison Wesley, November 1994.
- [7] ADLER, I. Marshals, monotone marshals, and hypertree-width. *Journal of Graph Theory* 47, 4 (2004), 275–296.
- [8] ADLER, I., GOTTLOB, G., AND GROHE, M. Hypertree width and related hypergraph invariants. *European Journal of Combinatorics* 28, 8 (November 2007), 2167–2181.
- [9] BEIGEL, R., AND FU, B. Molecular computing, bounded nondeterminism, and efficient recursion. In *In Proceedings of the 24th International Colloquium on Automata, Languages, and Programming* (1998), vol. 25, pp. 816–826.

- [10] BONORDEN, O., GEHWEILER, J., AND MEYER AUF DER HEIDE, F. A web computing environment for parallel algorithms in java. In *Parallel Processing and Applied Mathematics*, R. Wyrzykowski, J. Dongarra, N. Meyer, and J. Waśniewski, Eds., vol. 3911. Springer Berlin Heidelberg, Berlin, Heidelberg, 2006, ch. 96, pp. 801–808.
- [11] BRUSS, A. R., AND MEYER, A. R. On time-space classes and their relation to the theory of real addition. In *STOC '78: Proceedings of the tenth annual ACM symposium on Theory of computing* (New York, NY, USA, 1978), ACM, pp. 233–239.
- [12] CAI, L., AND CHEN, J. On the amount of nondeterminism and the power of verifying. *SIAM Journal on Computing* 26 (1997), 311–320.
- [13] CHANDRA, A. K., KOZEN, D. C., AND STOCKMEYER, L. J. Alternation. *J. ACM* 28, 1 (January 1981), 114–133.
- [14] CHANDRA, A. K., AND MERLIN, P. M. Optimal implementation of conjunctive queries in relational data bases. In *STOC '77: Proceedings of the ninth annual ACM symposium on Theory of computing* (New York, NY, USA, 1977), ACM, pp. 77–90.
- [15] COHEN, D., JEAUVONS, P., AND GYSSENS, M. A unified theory of structural tractability for constraint satisfaction and spread cut decomposition. In *IJCAI'05: Proceedings of the 19th international joint conference on Artificial intelligence* (San Francisco, CA, USA, 2005), Morgan Kaufmann Publishers Inc., pp. 72–77.
- [16] COOK, S. A. Characterizations of pushdown machines in terms of time-bounded computers. *J. ACM* 18, 1 (January 1971), 4–18.

- [17] COOK, S. A. Deterministic cfl's are accepted simultaneously in polynomial time and log squared space. In *STOC '79: Proceedings of the eleventh annual ACM symposium on Theory of computing* (New York, NY, USA, 1979), ACM, pp. 338–345.
- [18] CORMEN, T. H., LEISERSON, C. E., RIVEST, R. L., AND STEIN, C. *Introduction to Algorithms, Second Edition*, 2nd ed. The MIT Press, September 2001.
- [19] DERMAKU, A., GANZOW, T., GOTTLOB, G., MCMAHAN, B., MUSLIU, N., AND SAMER, M. Heuristic methods for hypertree decomposition. In *MICAI 2008: Advances in Artificial Intelligence*, A. Gelbukh and E. F. Morales, Eds., vol. 5317 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008, ch. 1, pp. 1–11.
- [20] DVOŘÁK, W., GOTTLOB, G., PICHLER, R., AND WOLTRAN, S. Alternation as a programming paradigm. In *PPDP '09: Proceedings of the 11th ACM SIGPLAN conference on Principles and practice of declarative programming* (New York, NY, USA, 2009), ACM, pp. 61–72.
- [21] FLOYD, R. W. Algorithm 97: Shortest path. *Commun. ACM* 5, 6 (June 1962), 345+.
- [22] FLUM, J., FRICK, M., AND GROHE, M. Query evaluation via tree-decompositions. *J. ACM* 49, 6 (2002), 716–752.
- [23] GANZOW, T., GOTTLOB, G., MUSLIU, N., AND SAMER, M. A CSP Hypergraph Library. Tech. rep., Database and Artificial Intelligence Group, June 2005.
- [24] GINSBURG, S., GREIBACH, S. A., AND HARRISON, M. A. Stack automata and compiling. *J. ACM* 14, 1 (1967), 172–201.

- [25] GOLDSMITH, J., LEVY, M. A., AND MUNDHENK, M. Limited nondeterminism, 1996.
- [26] GOTTLÖB, G., LEONE, N., AND SCARCELLO, F. On tractable queries and constraints. In *Database and Expert Systems Applications*, T. J. M. Bench-Capon, G. Soda, and A. M. Tjoa, Eds., vol. 1677. Springer Berlin Heidelberg, Berlin, Heidelberg, June 1999, ch. 1, p. 810.
- [27] GOTTLÖB, G., LEONE, N., AND SCARCELLO, F. A comparison of structural csp decomposition methods. *Artificial Intelligence* 124, 2 (December 2000), 243–282.
- [28] GOTTLÖB, G., LEONE, N., AND SCARCELLO, F. The complexity of acyclic conjunctive queries. *J. ACM* 48, 3 (May 2001), 431–498.
- [29] GOTTLÖB, G., LEONE, N., AND SCARCELLO, F. Hypertree decompositions and tractable queries. *Journal of Computer and System Sciences* 64, 3 (May 2002), 579–627.
- [30] GOTTLÖB, G., LEONE, N., AND SCARCELLO, F. Robbers, marshals, and guards: game theoretic and logical characterizations of hypertree width. *J. Comput. Syst. Sci.* 66, 4 (June 2003), 775–808.
- [31] GOTTLÖB, G., MIKLOS, Z., AND SCHWENTICK, T. Generalized hypertree decompositions: np-hardness and tractable variants. Tech. rep., Institut für Informationssysteme, Technische Universität Wien, March 2007.
- [32] GOTTLÖB, G., AND SAMER, M. A backtracking-based algorithm for hypertree decomposition. *J. Exp. Algorithmics* 13 (2009).

- [33] GROHE, M., AND MARX, D. Constraint solving via fractional edge covers. In *SODA '06: Proceedings of the seventeenth annual ACM-SIAM symposium on Discrete algorithm* (New York, NY, USA, 2006), ACM, pp. 289–298.
- [34] GROPP, W., LUSK, E., AND THAKUR, R. *Using MPI-2: Advanced Features of the Message Passing Interface (Scientific and Engineering Computation)*, 1st ed. The MIT Press, November 1999.
- [35] HLINENY, P., OUM, S.-I., SEESE, D., AND GOTTLOB, G. Width parameters beyond tree-width and their applications. *The Computer Journal* 51, 3 (September 2007), bxm052–362.
- [36] IBARRA, O. H. Characterizations of some tape and time complexity classes of turing machines in terms of multihead and auxiliary stack automata. *Journal of Computer and System Sciences* 5, 2 (April 1971), 88–117.
- [37] KNUTH, D. E. *Art of Computer Programming, Volume 1: Fundamental Algorithms (3rd Edition)*, 3 ed. Addison-Wesley Professional, July 1997.
- [38] LEONE, N., MAZZITELLI, A., AND SCARCELLO, F. Cost-based query decompositions. In *SEBD* (2002), pp. 390–403.
- [39] MACKWORTH, A. Consistency in networks of relations. *Artificial Intelligence* 8, 1 (February 1977), 99–118.
- [40] MCMAHAN, B. Bucket elimination and hypertree decomposition. Tech. rep., Technische Universität Wien, Database and Artificial Intelligence Group, 2004.
- [41] MIKLOS, Z. *Understanding tractable decompositions for constraint satisfaction*. PhD thesis, University of Oxford, 2008.
- [42] MONIEN, B., AND SUDBOROUGH, I. H. Bandwidth constrained np-complete problems. *Theoretical Computer Science* 41 (1985), 141–167.

- [43] PAPADIMITRIOU, C. H. *Computational Complexity*. Addison Wesley, December 1993.
- [44] QUINN, M. *Parallel Programming in C with MPI and OpenMP*, 1 ed. McGraw-Hill Science/Engineering/Math, June 2003.
- [45] REINGOLD, O. Undirected st-connectivity in log-space. In *STOC '05: Proceedings of the thirty-seventh annual ACM symposium on Theory of computing* (New York, NY, USA, 2005), ACM, pp. 376–385.
- [46] ROBERTSON, N., AND SEYMOUR, P. Graph minors. ii. algorithmic aspects of tree-width. *Journal of Algorithms* 7, 3 (September 1986), 309–322.
- [47] RUZZO, W. L. Tree-size bounded alternation. *Journal of Computer and System Sciences* 21, 2 (October 1980), 218–235.
- [48] SAVITCH, W. J. Relationships between nondeterministic and deterministic tape complexities. *J. Comput. Syst. Sci.* 4, 2 (1970), 177–192.
- [49] SCARCELLO, F., GRECO, G., AND LEONE, N. Weighted hypertree decompositions and optimal query plans. *J. Comput. Syst. Sci.* 73, 3 (May 2007), 475–506.
- [50] SEYMOUR, P., AND THOMAS, R. Graph searching and a min-max theorem for tree-width. *Journal of Combinatorial Theory, Series B* 58, 1 (May 1993), 22–33.
- [51] SIPSER, M. *Introduction to the Theory of Computation*. Course Technology, December 1996.
- [52] STROUSTRUP, B. *C++ Programming Language, The (3rd Edition)*, 3 ed. Addison-Wesley Professional, June 1997.

- [53] SUDBOROUGH, I. H. The complexity of the membership problem for some extensions of context-free languages. *International Journal of Computer Mathematics* 6, 3 (1977), 191–215.
- [54] TARR, G. Hypertree Decomposition. Master’s thesis, University of Oxford, 2008.
- [55] VOLLMER, H. *Introduction to Circuit Complexity: A Uniform Approach (Texts in Theoretical Computer Science. An EATCS Series)*, 1 ed. Springer, July 1999.