Balanced Queries: Divide and Conquer*

Dmitri Akatov[1] and Georg Gottlob[1,2]

[1] Oxford University Computing Laboratory, University of Oxford
[2] Oxford Man Institute of Quantitative Finance, University of Oxford
{dmitri.akatov,georg.gottlob}@comlab.ox.ac.uk

Abstract. We define a new hypergraph decomposition method called *Balanced Decomposition* and associate *Balanced Width* to hypergraphs and queries. We compare this new method to other well known decomposition methods, and analyze the complexity of finding balanced decompositions of bounded width and the complexity of answering queries of bounded width. To this purpose we define a new complexity class, allowing recursive divide and conquer type algorithms, as a resource-bounded class in the nondeterministic auxiliary stack automaton computation model, and show that finding decompositions of bounded balanced width is feasible in this new class, whereas answering queries of bounded balanced width is complete for it.

1 Introduction

The aim of the study of *hypergraph decompositions* is to find tractable subclasses of the *Boolean Conjunctive Query* (BCQ) evaluation problem in databases and the *Constraint Satisfaction Problem* in AI. Both these problems are equivalent and well known to be NP-complete [6,19] with the cyclicity of the hypergraphs causing the state explosion. A *hypergraph decomposition* transforms a hypergraph into an acyclic structure (a labelled tree), reducing the complexity of the associated problem. The tractability results of these problems rely on the acyclicity of the tree on the one hand and on certain properties of its labels on the other. Probably the most prominent decomposition method is the *tree decomposition* of [22], originally developed for graphs, but also applicable to hypergraphs. [10,7,16,17] provide an overview of more recent decomposition methods including *(generalized) hypertree decompositions*, spread cut decompositions and fractional hypertree decompositions. An important notion in most decompositions is their *width*, which often ensures tractability if it is independent of the hypergraph under consideration. Thus the two main complexity-theoretic problems usually considered are the following:

– **Decomposition problem**: What is the complexity of recognizing hypergraphs admitting a decomposition of fixed width?

* Work funded by EPSRC Grant EP/G055114/1 "Constraint Satisfaction for Configuration: Logical Fundamentals, Algorithms and Complexity. G. Gottlob would also like to acknowledge the Royal Society Wolfson Research Merit Award.

P. Hliněný and A. Kučera (Eds.): MFCS 2010, LNCS 6281, pp. 42–54, 2010.

– **BCQ evaluation problem**: What is the complexity of evaluating BCQs for the class of queries with a decomposition of fixed width?

A particularly nice property for queries with bounded tree and (generalized) hypertree width, thus also including acyclic queries, is that the BCQ evaluation problem is not only tractable, but also complete for the complexity class LOGCFL [12]. This complexity class lies very low within the NC-AC hierarchy (and hence within P) between NC^1 and AC^1 and hence is highly parallelizable. In [11] Gottlob et al. present a parallel algorithm for the BCQ evaluation problem[1] which is optimal under its complexity restrictions, and whose running time does not depend on the shape of the hypertree. The problem of recognizing hypergraphs of bounded hypertree width is also in LOGCFL [12],[2] however, most efficient sequential algorithms, see e.g. [15], compute the decomposition node by node in a top-down manner, following the shape of the resulting tree, which for most hypergraphs is often deep (linear in the size of the hypergraph) and narrow (branching factor of 1 for most nodes). This has negative effects on the parallelization of such hypertree computation algorithms, which is easiest when the *computation tree* is balanced, indicating a *division* of the problem into smaller independent subproblems which can be *conquered* recursively.

While looking for better, more "balanced", algorithms, we decided to analyze hypertrees which are balanced *a priori*, but are not necessarily valid (generalized) hypertree decompositions. These *balanced decompositions* constitute an entirely new hypergraph decomposition method in its own right, and hence deserve further analysis. In particular they possess beneficial properties for parallelization, they capture wider classes of hypergraphs than other known decomposition methods, and provide more insight into the structure of NP-complete problems.

In section 3 we provide the formal definition of balanced decompositions and compare them to generalized hypertree decompositions. In section 4 we characterize balanced decompositions game-theoretically by defining the *Robber and Sergeants Game* for hypergraphs. To better understand the complexity of the decomposition and the BCQ evaluation problems, we define a **new complexity class $\mathbf{DC}^1$** in section 5 by limiting resources of Nondeterministic auxiliary Stack Automata [18], and identify its lower bounds as LOGCFL and $\mathrm{GC}(\log^2 n, \mathrm{NL})$ in the Guess-and-Check model and its upper bound as $\mathrm{NTiSp}(n^{O(1)}, \log n)$, the space-bounded subclass of NP. In section 6 and section 7 we show that recognizing hypergraphs of bounded *balanced width* (BW) is feasible in DC^1, while the **BCQ evaluation problem** for the class of queries of **bounded balanced width** is **complete for $\mathbf{DC}^1$**. We conclude the paper in section 8.

Omitted proofs can be found in the full version of this paper [1].

[1] Actually, the algorithm was developed for acyclic BCQs, but can easily be adapted to generalized hypertree decompositions.

[2] Unfortunately recognizing hypergraphs of generalized hypertree width at least 3 is NP-complete [14].

2 Preliminaries

All sets in this paper are finite.

We assume the reader to be familiar with the standard formalizations of rooted and ordered trees. We use T to denote a tree, its node set and its "child function", we write T_p for a subtree rooted at a node p, $O(T)$ for the "root" of T, and $\sqsubseteq_T$ for the "ancestor relation", with $O(T) \sqsubseteq_T p$ for all other $p \in T$.

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 *edges* or *hyperedges* of H. Given a hypergraph H and $R, S \subseteq E(H)$, we say $Q \subseteq R \setminus S$ is an $[S]$-component of R iff either $Q = \{e\}$ with $e \subseteq \bigcup S$, or for any two edges in Q there exists a sequence (an $[S]$-*path*) of edges in Q between them, such that every two consecutive edges share some vertex not covered by $\bigcup S$.

Given a hypergraph H, a *hypertree for* H is a triple (T, χ, λ), where T is a rooted tree, and χ and λ are labeling functions which associate to each vertex $p \in T$ two sets $\chi(p) \subseteq V(H)$ and $\lambda(p) \subseteq E(H)$.
Given $p \in T$ we define $\chi(T_p) = \bigcup\{\chi(q) | q \in T_p\}$ and $\lambda(T_p) = \bigcup\{\lambda(q) | q \in T_p\}$.
The *width* of a hypertree is $\max_{p \in T} |\lambda(p)|$.

A *hypertree decomposition* is a hypertree satisfying the following conditions:

1. For all $e \in E(H)$, there exists $p \in T$, such that $e \subseteq \chi(p)$,
2. for all $v \in V(H)$, the set $\{p \in T | v \in \chi(p)\}$ induces a connected subtree of T,
3. for each $p \in T$, $\chi(p) \subseteq \bigcup \lambda(p)$,
4. for each $p \in T$, $(\bigcup \lambda(p)) \cap \chi(T_p) \subseteq \chi(p)$.

A *generalized hypertree decomposition* is a hypertree only satisfying the first three of these conditions. The width of a (generalized) hypertree decomposition is the width of its hypertree. The *(generalized) hypertree width* (GHW / HW) of a hypergraph H is the minimal width over all its (generalized) hypertree decompositions [12].

The *monotone Robber and Marshals Game* and its equivalence with hypertree decompositions is studied in [13] and [3].

A *Database* is a relational structure over a schema (signature). A *Boolean Conjunctive Query* (BCQ) is also a relational structure (over the same schema) containing no constants. We call every tuple occurring in some relation also an *atom*, and the objects of the base set occurring in the query or an atom its *variables*. We write atoms(q) for the set of atoms of a query q and var(a) for the set of variables occurring in an object a (e.g. an atom or a query). We say that a database D *satisfies* a BCQ q ($D \models q$) iff there exists a homomorphism from q to D.[3]

The *underlying hypergraph* of a BCQ q is a the hypergraph $H(q)$, with $V(H(q)) = var(q)$ and $E(H(q)) = \{\text{var}(a) | a \in \text{atoms}(q)\}$. A decomposition of a BCQ is simply a decomposition of its underlying hypergraph.

[3] An alternative definition of databases and BCQs can be found e.g. in [2].

3 Balanced Decompositions

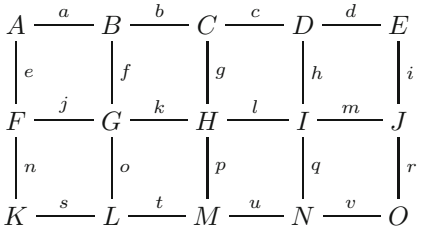
Definition 1. *Let H be a hypergraph. A* hypercut decomposition of H *is a hypertree (T, χ, λ) for H which satisfies the following conditions:*

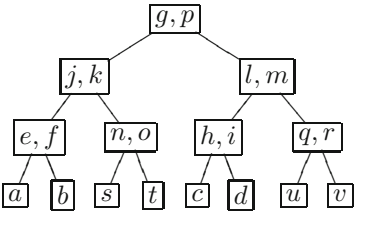
1. *For each $e \in E(H)$ there exists $p \in T$ such that $e \in \lambda(p)$,*
2. *for each $Y \in V(H)$, the set $\{p \in T | Y \in \chi(p)\}$ contains its $\sqsubseteq_T$-meet,*
3. *for each vertex $p \in T$, $\chi(p) = \bigcup \lambda(p)$.*

A hypercut decomposition is a shallow decomposition *if additionally* $\text{depth}(T) \leq \log |E(H)|$ *holds. A hypercut decomposition is a* balanced decomposition *if additionally $|\lambda(T_q)| \leq |\lambda(T_p)|/2$ holds for all $p \in T, q \in T(p)$. The* width *of a shallow decomposition or a balanced decomposition is the width of its hypertree. The* shallow width*, resp.* balanced width*, of H is the minimum width over all its shallow, resp. balanced, decompositions. We write* $\text{SW}(H)$ *for the shallow width and* $\text{BW}(H)$ *for the balanced width of H.*

Notice that a hypercut decomposition is uniquely defined by T and λ alone, while the χ-labels are only required for the second condition. We do not define a hypercut width, since then every hypergraph would trivially have width one.

Example 1. The balanced and shallow widths of the following hypergraph are two and we also show a balanced decomposition, which is also shallow (in the decomposition we only present the λ-labels):





For any such "grid graph" of width k and length m, it is easy to construct a balanced decomposition of width k. The generalized hypertree width of such hypergraphs, however, is always $k+1$, and any decomposition tree will have depth at least m (linear in the size of the hypergraph), and will generally contain a long "chain" with no branching.

Proposition 1. *Let H be a hypergraph. The following holds:*

$$SW(H) \leq BW(H) \leq GHW(H) \leq SW(H) \log |E(H)| .$$

A shallow tree will have "good" branching at some point, however this branching does not have to occur at every internal node, as in a (perfectly) balanced tree. Hence also the distinction between balanced and shallow decompositions, which also capture slightly different classes of hypergraphs. For instance, every graph consisting of a single cycle has shallow width one, whereas its balanced width is always two. However, for all complexity-theoretic purposes, the distinction between balanced and shallow decompositions is not relevant, since our results apply to both of them.

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, [24,13,16]. We shall define the *Robber & k Sergeants game* on a hypergraph H ($\mathrm{R\&S}^k(H)$). It resembles the Robber and Marshals game [13], but has two important differences: The robber is positioned on edges rather than vertices (an escape space hence becomes a set of edges rather than a set of vertices). Also, the sergeants only have to cover any edge once and 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 ($\mathrm{R\&S}^k(A)$) 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 $\mathrm{R\&S}^k(B)$. If the game is shallow, then player $\mathcal{R}$ wins $\mathrm{R\&S}^k(A)$ 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 $\mathrm{R\&S}^k(H)$ is the game $\mathrm{R\&S}^k(E(H))$ (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. This leads to the formal definition of a winning strategy:

Definition 2. *Let k be a positive integer, let H be a connected hypergraph. A* winning strategy *for $R\&S^k(H)$ is a tuple (T, ρ, λ), where T is a rooted tree and $\rho, \lambda : T \to \mathcal{P}(A)$ are labelling functions (escape space and sergeants' moves, respectively) such that the following conditions hold:*

1. Initial Condition: $\rho(O(T)) = E(H)$.
2. Boundedness: *For all* $t \in T$, $1 \leq |\lambda(t)| \leq k$.
3. Completeness: *For all* $s \in T$, $\rho(s) = \mu(s) \cup \bigcup_{t \in T(s)} \rho(t)$.
4. Separation: *For all* $s \in T, t \neq u \in T(s)$, $\rho(t) \cap \rho(u) = \emptyset$.
5. 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 $\mathrm{depth}(T) \leq \log|E(H)|$. *A winning strategy in the balanced $R\&S^k$ game additionally satisfies* $|\rho(t)| \leq |\rho(s)|/2$, *for all* $s \in T, t \in T(s)$.

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 $\lambda(s) \subseteq \rho(s)$.

Lemma 1. *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.*

5 The DC Hierarchy

An *auxiliary pushdown automaton* (AuxPDA) is a generalization of both the Turing Machine (TM) and the Pusdown Automaton (PDA) — it possesses both a tape and a pushdown. Adding a pushdown makes these machines more powerful than Turing Machines, since they admit *recursive algorithms*, which push "temporary variables" before a recursive call and pop them after the call returns. For instance, a nondeterministic TM using simultaneously logarithmic space and polynomial time (in $\mathrm{NTiSp}(n^{O(1)}, \log n)$, see e.g. [20]) can solve problems precisely in NL. A nondeterministic AuxPDA (NauxPDA) with the same time and space bound on the worktape can precisely solve problems in LOGCFL, which contains the latter complexity class. We do not usually limit the space on the pushdown (the *maximal pushdown height*), however, Ruzzo showed that problems in LOGCFL only require $O(\log^2 n)$ space on the pushdown. We write $\mathrm{NTiSpPh}(T(n), S(n), H(n))$ for the class of problems solvable by a NauxPDA which is simultaneously bounded by time $O(T(n))$, worktape space $O(S(n))$ and maximal pushdown height $O(H(n))$.

A *stack* acts like a pushdown for writing (pushing and popping), but like a tape for reading (any cell can be read). Thus, *Stack Automata* (SA), introduced by Ginsburg et al. [8], are more powerful than PDAs. Analogously to extending TMs with a pushdown, Ibarra proposed to do the same with SAs [18], yielding the model of the *auxiliary stack automaton* (AuxSA). 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. We write $\mathrm{NTiSpPh}(T(n), S(n), H(n))$ for the class of problems solvable by a nondeterministic SA (NauxSA) which is simultaneously bounded by time $O(T(n))$, worktape space $O(S(n))$ and maximal stack height $O(H(n))$.

Since a pushdown can be simulated by a stack, and a stack can in turn be simulated by a worktape we have the following relationship of complexity classes:

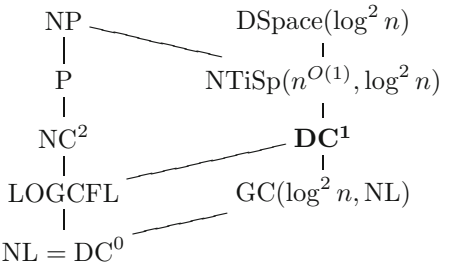
$$\begin{aligned}&\mathrm{NTiSpPh}(T(n), S(n), H(n))\\ \subseteq&\mathrm{NTiSpSh}(T(n), S(n), H(n))\\ \subseteq&\mathrm{NTiSp}(T(n), \max(S(n), H(n)))\,.\end{aligned}$$

Definition 3. *For integers* $k \geq 0$*, let* $\mathrm{DC}^k = \mathrm{NTiSpSh}(n^{O(1)}, \log n, \log^{k+1} n)$.

This new hierarchy of complexity classes lies between $\mathrm{NL}{=}\mathrm{DC}^0$ and NP. 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. In this paper 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. We have $\mathrm{LOGCFL} \subseteq \mathrm{DC}^1 \subseteq \mathrm{NTiSp}(n^{O(1)}, \log^2 n)$.

We can use SAs to simulate the Guess-and-Check model of [5], by nondeterministically guessing an "advice string" and placing it on the stack. In particular, we have $\mathrm{GC}(\log^{k+1} n, \mathrm{NL}) \subseteq \mathrm{DC}^k$, 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. Note that $\mathrm{GC}(\log^2 n, \mathrm{NL})$ is **not known or believed to be contained in** P, which strongly indicates that neither is DC^1.

We get the following inclusion diagram of complexity classes:



6 Membership in $\mathbf{DC^1}$

Winning strategies in the R&S^{k}game give us an easy way to find balanced decompositions. Consider the algorithm k-robber-sergeants:

Algorithm 1. k-robber-sergeants

```
1 : input Hypergraph H
2 : fixed parameter k: Integer
3 : check-win(firstEdge(H), |V(H)|)
4 : accept

5 : procedure check-win(Edge r, Integer size)
6 :    guess Edge[1...k] sergs
7 :    for each Edge f ∈ E(H) do
8 :       if connected(r,f) then
9 :          Integer n := count-connected-edges(f, sergs)
10 :         if n > size/2 then reject
11 :         else if n > 0 then check-win(f, sergs)
```

Here the function *count-connected-edges(e, sergs)* counts the number of edges $[S]$-connected to e, where S is the set of all sergeant hyperedges already on the stack plus *sergs*. This can obviously be done in NL, and even in L by using the undirected st-connectivity algorithm of [21].

Theorem 1. *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.

k-robber-sergeants repeats a lot of work, however, this affects neither its correctness nor its complexity bounds. A deterministic algorithm would of course trade space for time, avoiding any redundancy. It is an easy exercise to adapt k-robber-sergeants to check for shallow decompositions instead.

As for the bounded BW BCQ evaluation problem, consider the algorithm k-hd-bcq:

Algorithm 2. k-hd-bcq

```
 1 : fixed parameter k: Integer
 2 : input Database d
 3 : input Query q with a k-width hypercut decomposition
 4 : satisfiable(root(q))
 5 : accept

 6 : procedure satisfiable(Node u)
 7 :   for each Atom a ∈ λ(u) do                  // at most k repetitions
 8 :     guess Tuple t ∈ table(d, name(a))
 9 :     for each (Atom, Tuple) (b, s) on stack do
10 :       if not compatible((a, t), (b, s)) then reject
11 :     push (a, t)
12 :   for each Node v ∈ children(u) do satisfiable(v)
13 :   pop all (a, t) pairs which were pushed during current function call
```

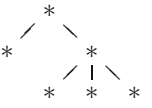
We assume that the tree of the hypercut decomposition is an ordered tree, and that we can access its root using the function *root*, and that given a node u, we can iterate through *children(u)* using a single pointer. Given an atom a the function *name(a)* returns the "schema" s of that atom, and we 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. whether all shared variables of a and b have the same values in t and u.

Theorem 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* $\mathrm{NTiSpSh}(n^{O(1)}, \log n, d \log n)$*, where n is the size of the whole input and $d = \mathrm{depth}(T)$.*

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

7 DC[1]-Completeness

To show hardness for DC[1] of the BCQ evaluation problem for queries of fixed balanced width we will need some preliminary results about NauxSAs. For any AuxSA, AuxPDA, PDA or SA we define the *push-pop tree* to be an ordered tree representing the sequence of the non-read stack operations of the machine. The root of the tree represents an empty pushdown/stack, any one node represents some distinct state of the pushdown/stack, and a new child is added to that node whenever the machine pushes while the pushdown/stack is in that state. For example, if the sequence is Push, Pop, Push, Push, Pop, Push, Pop, Push, Pop, Pop, then the push-pop tree looks like this:



Definition 4. *We call a NauxSA M* regular *if it has the following properties:*

1. *The push-pop tree of M is a full binary tree.*
2. *After every push the entire contents of the stack is read.*
3. *Whenever M doesn't push, it acts deterministically (the only nondeterministic steps are the pushes).*

Lemma 2. *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'.*

Theorem 3. *Let M be a regular NauxSA running in polynomial time, logarithmic space and with a push-pop tree of height $k = O(\log n)$ and x a string. Then there exists a database B and a Boolean Conjunctive Query Q with a Balanced Decomposition of width* 8 *such that $B \vDash Q$ iff M accepts x. Moreover there exists a* log*-space reduction from (M, x) to (B, Q).*

Proof. The database B has 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 $O_i(C_1, C_2)$.[4] The table I contains the initial configuration of M as the only tuple. The table A contains all accepting configurations of M. 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. 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. A tuple (c, s, d) is in U_i iff M starting in configuration c would next push s into the ith cell and end up in configuration d. Similarly a

[4] For a schema, $T(A, B, C)$ here indicates that we have a table called T which has three *attributes* (with "types" A, B and C), such that every tuple in that table has exactly three elements.

tuple (c, s, d) is in R_i iff M starting in configuration c would next read s from the ith cell and end up in configuration d. All these tables can be computed in log-space.

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. For a node N let $p^j(N)$, $l(N)$ and $r(N)$ denote the j-th ancestor, left child and right child of N, respectively. For each $N \in T \setminus O(T)$, define the query Q_N^R in the following way: if $\mathrm{depth}_T(N) = 1$, 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, then

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

if N is a leaf, then

$$Q_N = D(X_N^1, X_N^2) \wedge Q_N^R \wedge D(X_N^3, X_N^4) \wedge O_{d(N)}(X_N^4, X_N^7)\,,$$

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

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

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 has a hyperedge for every atom in the query, since they are all different. In particular, every subquery Q^R_N will correspond to $2\text{depth}_T(N) - 1$ hyperedges. Additionally we have 5 hyperedges for the root, 3 hyperedges for each leaf, 7 hyperedges for 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 decomposition in the following way: For every Q^R_N 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 decomposition of Q^R_N of width 1, since Q^R_N 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^R_N. 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 subtree 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 obvious that (T', λ) is balanced, since every node has two or four children and is built absolutely symmetrically. □

Corollary 2. *The BCQ evaluation problem for queries of bounded balanced width is complete for* DC^1.

8 Determinization, Parallelization and Future Work

A standard technique to make a nondeterministic algorithm deterministic is a brute-force search of the computation tree, trying out all nondeterministic choices. 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($\log^2 n$). The deterministic algorithm works its way along the push-pop tree, backtracking its steps whenever some "nondeterministic" choice leads to rejection. Notice, that once a node in the push-pop tree has several descendants, we can split the work to different processors, since the results for the subtasks do not depend on each other. The only downside is the amount of work that needs to be done at every such node, since there are $O(n^{O(1)\log n})$ possibilities for the contents of the stack (at the deepest level). Hence the algorithm, even with parallelization, remains superpolynomial. It is however still *quasipolynomial*.

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 [23], [9] and [4]. 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. Finally, an important aspect of future work is the implementation and testing of the parallelization methods described above in practice.

References

1. `http://benner.dbai.tuwien.ac.at/staff/gottlob/DAGG-MFCS10.pdf`
2. Abiteboul, S., Hull, R., Vianu, V.: Foundations of Databases. Addison Wesley, Reading (November 1994)
3. Adler, I.: Marshals, monotone marshals, and hypertree-width. Journal of Graph Theory 47(4), 275–296 (2004)
4. Beigel, R., Fu, B.: Molecular computing, bounded nondeterminism, and efficient recursion. In: Proceedings of the 24th International Colloquium on Automata, Languages, and Programming, vol. 25, pp. 816–826 (1998)
5. Cai, L., Chen, J.: On the amount of nondeterminism and the power of verifying. SIAM Journal on Computing 26, 311–320 (1997)
6. Chandra, A.K., Merlin, P.M.: Optimal implementation of conjunctive queries in relational data bases. In: STOC 1977: Proceedings of the ninth annual ACM symposium on Theory of computing, pp. 77–90. ACM, New York (1977)
7. Cohen, D., Jeavons, P., Gyssens, M.: A unified theory of structural tractability for constraint satisfaction and spread cut decomposition. In: IJCAI 2005: Proceedings of the 19th international joint conference on Artificial intelligence, pp. 72–77. Morgan Kaufmann Publishers Inc., San Francisco (2005)
8. Ginsburg, S., Greibach, S.A., Harrison, M.A.: Stack automata and compiling. J. ACM 14(1), 172–201 (1967)
9. Goldsmith, J., Levy, M.A., Mundhenk, M.: Limited nondeterminism (1996)
10. Gottlob, G., Leone, N., Scarcello, F.: A comparison of structural csp decomposition methods. Artificial Intelligence 124(2), 243–282 (2000)
11. Gottlob, G., Leone, N., Scarcello, F.: The complexity of acyclic conjunctive queries. J. ACM 48(3), 431–498 (2001)
12. Gottlob, G., Leone, N., Scarcello, F.: Hypertree decompositions and tractable queries. Journal of Computer and System Sciences 64(3), 579–627 (2002)
13. Gottlob, G., Leone, N., Scarcello, F.: Robbers, marshals, and guards: Game theoretic and logical characterizations of hypertree width. J. Comput. Syst. Sci. 66(4), 775–808 (2003)

14. Gottlob, G., Miklos, Z., Schwentick, T.: Generalized hypertree decompositions: Np-hardness and tractable variants. In: PODS 2007: Proceedings of the twenty-sixth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems, pp. 13–22. ACM Press, New York (2007)
15. Gottlob, G., Samer, M.: A backtracking-based algorithm for hypertree decomposition. J. Exp. Algorithmics 13 (2009)
16. Grohe, M., Marx, D.: Constraint solving via fractional edge covers. In: SODA 2006: Proceedings of the seventeenth annual ACM-SIAM symposium on Discrete algorithm, pp. 289–298. ACM, New York (2006)
17. Hlineny, P., Oum, S.-i., Seese, D., Gottlob, G.: Width parameters beyond tree-width and their applications. The Computer Journal 51(3), 326–362 (2007)
18. 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), 88–117 (1971)
19. Mackworth, A.: Consistency in networks of relations. Artificial Intelligence 8(1), 99–118 (1977)
20. Monien, B., Sudborough, I.H.: Bandwidth constrained np-complete problems. Theoretical Computer Science 41, 141–167 (1985)
21. Reingold, O.: Undirected st-connectivity in log-space. In: STOC 2005: Proceedings of the thirty-seventh annual ACM symposium on Theory of computing, pp. 376–385. ACM, New York (2005)
22. Robertson, N., Seymour, P.: Graph minors. ii. algorithmic aspects of tree-width. Journal of Algorithms 7(3), 309–322 (1986)
23. Ruzzo, W.L.: Tree-size bounded alternation. Journal of Computer and System Sciences 21(2), 218–235 (1980)
24. Seymour, P., Thomas, R.: Graph searching and a min-max theorem for tree-width. Journal of Combinatorial Theory, Series B 58(1), 22–33 (1993)