UNIVERSITY OF CALIFORNIA
SANTA CRUZ

RANKED ENUMERATION OVER CIRCUITS VIA IMPLICIT REPRESENTATIONS

A thesis submitted in partial satisfaction
of the requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER SCIENCE AND ENGINEERING

by

Jiayin Hu

August 2026

The Thesis of Jiayin Hu
is approved:

Professor Nikolaos Tziavelis

Professor Sungjin Im

Professor Ioannis Demertzis

Donald Smith
Interim Vice Provost and
Dean of Graduate Studies

Contents

List of Figures

Abstract

Ranked Enumeration over Circuits

via Implicit Representations

by

Jiayin Hu

Keywords: ranked enumeration, d-DNNF circuits, implicit representation, enumeration delay

We study the problem of enumerating the satisfying assignments of smooth multi-valued d-DNNF circuits in sum order. We measure the cost in terms of a preprocessing phase and the delay between consecutive answers, charging for both the size of the circuit and the number of variables. Producing each answer as a complete assignment takes time linear in the number of variables. We propose an algorithmic framework that avoids this cost by producing answers in an implicit representation. After preprocessing quasilinear in the size of the circuit, the delay for tree-shaped circuits is logarithmic in the number of answers already produced and independent of the number of variables. For general circuits, preprocessing incurs an additional factor of the number of variables, and one further term remains in the delay, logarithmic in the number of variables. An answer can be recovered explicitly from its implicit representation in additional linear time.

Chapter 1

Introduction

§ 1.1 Background

For query evaluation in databases, a fundamental challenge is that the size of intermediate or final query output can far exceed the size of the input database [7]. For complex join queries, fully materializing the answer set is often too slow and requires too much memory. This motivates representing the answer set in a compressed form, thereby allowing evaluation tasks to be solved efficiently without fully unpacking the answers.

A simple example conveys the idea. Consider the Cartesian product of $A = \{a_1, \ldots, a_n\}$ and $B = \{b_1, \ldots, b_m\}$. Listing the answers explicitly takes $n \cdot m$ tuples. Instead, read a_i as the assertion *the first component takes value* a_i, and likewise for b_j; the answers are exactly the satisfying assignments of $(a_1 \vee \cdots \vee a_n) \ \wedge \ (b_1 \vee \cdots \vee b_m)$, an expression of size $n + m$. Nothing is lost, since every tuple can be recovered by picking one from each side.

Each answer is a satisfying assignment of a formula built from the query and the database: deciding whether a query has an answer is satisfiability, and counting its

answers is model counting. Knowledge compilation [21] studies representations of Boolean functions on which such tasks become tractable, and its tools have since been adapted to the database setting. It represents functions by *circuits*: rooted DAGs whose gates are labeled by $\wedge$, $\vee$, or a literal. A circuit represents the answer space without loss, and can be exponentially smaller than the answer set itself [34]. Crucially, circuits with certain structural properties (e.g., d-DNNFs) let query tasks such as counting, enumeration [24] or sampling [19] be performed by traversing the circuit, so their cost is governed by the size of the circuit rather than by that of the answer set.

When the size of a query answer is large enough, producing all the answers at once is expensive and consumes too much time. *Enumeration* avoids this by outputting the answers one by one. An enumeration algorithm first runs a preprocessing phase, whose cost is measured in the size of the input, and then produces the answers in sequence; the quality of an enumeration algorithm is measured by the *delay*, the time that elapses between two consecutive outputs. The ideal is linear preprocessing and constant delay [9, 25]. Producing answers in an arbitrary order, however, is not always satisfactory. This motivates *ranked enumeration*, where a ranking function assigns a score to each answer, and the answers must be returned in the designated order given by the score.

These tasks have both been studied on circuits. Amarilli et al. [2] showed that the satisfying assignments of a d-DNNF circuit can be enumerated with linear preprocessing and constant delay. Ranked enumeration on circuits was also studied [1], where the answers are produced in order of score with logarithmic delay.

§ 1.2 Motivations

Let us look at the algorithm of Amarilli et al. [1] more closely. They consider ranked enumeration over *smooth multivalued d-DNNF* circuits on a variable set X, under subset-monotone ranking functions. After $O(|C|)$ preprocessing, where $|C|$ denotes the circuit size, their algorithm returns the k-th ranked assignment with a delay of $O(\log k)$.

This deserves a second look: in the enumeration phase, each answer is produced by a recursive descent through the circuit, and every priority-queue operation along this descent is charged once per variable in the worst-case scenario. Accounting for $|X|$ explicitly, the delay is therefore $O(|X| \log k)$, and the stated $O(\log k)$ delay holds only under the assumption that the size $|X|$ of the variable set is a *constant*. The assumption is acceptable in the setting they target: monadic second-order logic (MSO) with first-order free variables. It is still a limitation, because $|X|$ can be part of the input in other settings, such as for MSO with second-order free variables.

This dependence on $|X|$ appears unavoidable at first, as simply materializing an answer requires specifying a value for each of the variables, imposing an $\Omega(|X|)$ cost. However, the lower bound can be bypassed by avoiding a full answer. If two answers agree on most variables, one can be described by the variables on which it differs from the other, together with its values there. This is the paradigm exploited by Eppstein for the k shortest paths problem [26], where the k-th path is stored *implicitly* as a sequence of deviations from a shortest path tree. The idea of enumerating the implicit representation of each solution has also been used in formal language [6] and combinatorial generation problems [5]. We adopt a similar discipline for ranked enumeration over circuits: an answer is encoded by its deviations from a reference answer.

For the k-th answer we return an implicit representation from which its score is available exactly at no additional cost, and the explicit answer can be materialized with some extra cost on demand.

§ 1.3 Results

As in the problem of Amarilli et al. [1], we study ranked enumeration of the satisfying assignments over a smooth multivalued d-DNNF circuit, with $|X|$ treated as part of the input. We restrict the ranking function to *sum order*: every input gate carries a score, and the score of an assignment is the sum of the scores of the gates it uses. An output is a representation from which the score is available at no cost and the assignment is materialized in time $O(|C|)$ on demand.

Within this setting, we distinguish two types of circuits based on their structure: *tree-shaped circuits*, whose underlying graph is a tree, and *general circuits*, in which one gate may feed several others. Tree-shaped circuits enjoy simpler algorithms because each gate has a unique parent. General circuits allow gates to be shared, requiring algorithms to track additional information at the expense of higher preprocessing and delay costs. Our results are as follows.

Theorem 1.3.1. *The satisfying assignments of a smooth multivalued d-DNNF circuit C over X under a sum order can be enumerated in nondecreasing order of score, each answer being output as an implicit representation from which its score is read off in constant time and the assignment itself is materialized in time $O(|C|)$ on demand:*

- *after preprocessing in time and space $O(|C| \cdot \log |C|)$, the k-th answer is produced with delay $O(\log k)$, if C is tree-shaped;*

- *after preprocessing in time and space* $O(|C| \cdot |X| \cdot \log |C|)$*, the* k*-th answer is produced with delay* $O(\log k + \log |X|)$ *if* C *is a general circuit.*

The time to obtain the first k answers, $\mathrm{TT}(k)$, is defined as the preprocessing time plus the sum of the first k delays [36, 38]. The algorithm of Amarilli et al. [1] gives $\mathrm{TT}(k) = P' + O(k \cdot |X| \log k)$, where the enumeration term is the bound discussed in § 1.2 and $P' = \Omega(|C|)$. We do not state an upper bound on P', since it leaves the representation of the explicit assignment stored at each gate unspecified. Copying gives $O(|C| \cdot |X|)$; sharing by reference avoids this, but shifts the cost to materializing complete assignments during enumeration. Their algorithm abstracts away from variable-size dependencies.

For tree-shaped circuits, our algorithm gives $\mathrm{TT}(k) = O(|C| \cdot \log |C| + k \log k)$ for the first k implicit answers, saving a factor of $|X|$ on every answer produced. For general circuits, it gives $\mathrm{TT}(k) = O(|C| \cdot |X| \cdot \log |C| + k(\log k + \log |X|))$, where $|X|$ contributes an additive $\log |X|$ instead of multiplying $\log k$ in enumeration.

Organization. Chapter 2 reviews enumeration paradigms and the application of circuits in database theory. Chapter 3 introduces the notations for circuits and the problem. Chapter 4 develops the algorithmic framework and the data structures that both algorithms share. Chapters 5 and 6 are dedicated to tree-shaped and general circuits, respectively.

Chapter 2

Related Work

§ 2.1 Enumeration and its Variants

The answers to a query may be far more numerous than the database itself, so rather than returning them as a single block, enumeration algorithms and their advanced variants, ranked enumeration and direct access, produce them one at a time and let the caller stop after any number. Algorithm complexity is analyzed in terms of *preprocessing*, performed before enumeration begins, and *delay*, the time interval between outputting any two consecutive answers.

2.1.1 Enumeration

An enumeration algorithm must produce each answer exactly once; however, the output order is arbitrary. For conjunctive queries, the central question has been which queries admit linear preprocessing and constant delay. Bagan et al. [9] showed that free-connex acyclic queries do, and that for self-join-free queries this condition is also necessary, subject to hypotheses from fine-grained complexity. Later work has revisited this picture

for conjunctive queries with negation [13, 14], for unions of conjunctive queries [17], and in a dynamic setting where the instance is subject to updates [10].

Another line restricts the class of structures, so that an entire logic becomes tractable. Bagan [8] enumerates the answers to an MSO query over structures of bounded treewidth, with linear preprocessing and delay linear in the size of each answer. For queries of fixed arity, the delay was brought down to a constant [28].

2.1.2 Ranked Enumeration

Ranked enumeration adds a ranking function on answers and requires the enumeration to follow the order. Tziavelis et al. [36, 38] study this for conjunctive queries under a class of ranking functions that includes sum of weights; for free-connex queries, they reach the k-th answer in time $O(n + k \log k)$. Deep and Koutris [23] obtain logarithmic delay for ranking functions whose value can be computed incrementally as an answer is assembled. Other work covers projections [22], join predicates other than equality [37], and probabilistic databases under updates [11]. Outside the relational setting, Bourhis et al. [12] study ranked enumeration for monadic second-order queries on words.

2.1.3 Direct Access

A relevant problem is *direct access*, which asks for the answer at a given position in the order without producing those preceding it. Since direct access allows for retrieving the k-th answer on-demand, it inherently supports ranked enumeration by accessing answers in increasing order of k. Carmeli et al. [18] focus on conjunctive queries, identifying the queries and output orders for which optimal enumeration is achievable after linear

preprocessing; matching lower bounds are established by Bringmann et al. [15].

§ 2.2 Circuits in Database Theory

The above-mentioned enumeration algorithms operate on the database directly. A different approach is to first compile the answers into a circuit, and then execute enumeration over that representation instead. Tractable circuits, developed in knowledge compilation [21], separate these concerns: all query-dependent work goes into compiling a circuit of a designated class, and the enumeration algorithm then operates on the circuit alone. Database theory has used such circuits in two ways, differing in what the circuit represents [4].

The *Boolean provenance* of a Boolean query treats each fact of the instance as a variable: a valuation selects a subinstance, and the function records whether the query holds on it. Its variables are the facts, and its models are subinstances, not answers. Probabilistic query evaluation then amounts to weighted model counting over this function, which is tractable on circuits in suitable classes [1]. Olteanu and Huang [32] compiled provenance into ordered binary decision diagrams. Jha and Suciu [27] studied which queries admit provenance representations of polynomial size, and characterized several such classes. Amarilli et al. [3] gave a construction for monadic second-order queries over trees and over instances of bounded treewidth, producing a provenance circuit in linear time. On the systems side, ProvSQL [35] maintains provenance circuits inside PostgreSQL and passes them to existing compilers.

The second use represents the answers rather than the facts. Factorized representations [34, 33] build the answer relation from singleton relations using union and product

alone, and give size bounds in terms of structural parameters of the query. Their *f-representations* are trees, while their *d-representations* allow a subexpression to be named and referenced several times, and are thus DAGs.

2.2.1 Enumeration on Circuits

Once the answers are represented by a circuit, the specifications of Section 2.1 can be addressed on the representation, and the resulting algorithms apply to any query language that compiles into the class. Amarilli et al. [2] did this for enumeration, obtaining linear preprocessing and output-linear delay on d-SDNNF and recovering earlier results for conjunctive queries and for monadic second-order queries as instances. Muñoz and Riveros [30] obtain comparable guarantees for nested documents through a related structure developed independently.

Ranked enumeration on circuits was studied by Amarilli, Bourhis, Capelli and Monet [1], who enumerate the satisfying assignments of a smooth multivalued d-DNNF in the order given by a subset-monotone ranking function. This is the setting of the present thesis, and their algorithm is the starting point for ours; the comparison is made in Chapter 1. Capelli and Irwin [16] study direct access on circuits and apply it to conjunctive queries with negation.

Chapter 3

Preliminaries

§ 3.1 Multivalued Circuits

A *multivalued circuit* C over a variable set X is a rooted directed acyclic graph (DAG) with each vertex labeled as a *gate*. The root without outgoing edges is the *output gate* r of C. Each gate without incoming edges is an *input gate* assigned a pair $\langle x : d \rangle$ where $x \in X$ and $d \in \mathsf{dom}_x$ (the domain of the variable x). Every other gate is labeled as either $\wedge$ (AND-gate) or $\vee$ (OR-gate). The set of AND-gates (resp. OR-gates) in C is denoted AND (resp. OR).

The gate g' is an *input* of the gate g if there is a directed edge $g' \to g$; g is an *output* of g'. The set of inputs of a gate g is denoted $\mathsf{inputs}(g)$, and we write $[\mathsf{inputs}(g)] = (c_1, \ldots, c_n)$ for its inputs ordered according to an arbitrary but fixed order. To make each gate meaningful, we assume AND-gates and OR-gates both have at least two inputs (or they can be removed from the circuit directly in linear time).

A gate g' is *reachable* from g (equivalently, g *reaches* g') if C contains a directed path from g' to g; in particular, every gate reaches itself via the path of length zero.

Reachability is taken against the direction of the edges, so that the gates reachable from g are exactly those gates feeding into g, directly or transitively. We say that g' lies *below* g or that g lies *above* g' if g' is reachable from g.

We write $\mathsf{gates}(g)$ for the set of gates reachable from g (below g), C_g for the *subcircuit rooted at* g, i.e., the subgraph of C induced by $\mathsf{gates}(g)$, and $\mathsf{vars}(g) \subseteq X$ for the set of variables x that occur in the label of some input gate in $\mathsf{gates}(g)$. Then C_g is itself a multivalued circuit with output gate g and variable set $\mathsf{vars}(g)$.

Observation 3.1.1. C is a finite DAG, so every gate reaches at least one input gate; hence, $\mathsf{vars}(g) \neq \emptyset$ for every gate g. Moreover, since C has a single root r, every gate is reachable from r.

Observation 3.1.2. Let g_1, g_2 be gates of C with $\mathsf{vars}(g_1) \cap \mathsf{vars}(g_2) = \emptyset$. Then $\mathsf{gates}(g_1) \cap \mathsf{gates}(g_2) = \emptyset$.

Assignments. Let $X' \subseteq X$. An *assignment over* X' is a set τ of pairs $\langle x : d \rangle$ with $x \in X'$ and $d \in \mathsf{dom}_x$, containing exactly one pair for each $x \in X'$. We write $\mathsf{vars}(\tau) = X'$, and $\tau(x)$ for the unique d such that $\langle x : d \rangle \in \tau$.

Two assignments τ, τ' are *compatible*, written $\tau \simeq \tau'$, if $\tau(x) = \tau'(x)$ for every $x \in \mathsf{vars}(\tau) \cap \mathsf{vars}(\tau')$; equivalently, the union $\tau \cup \tau'$ is an assignment as well, in which case $\tau \cup \tau' \in \mathsf{Assign}(\mathsf{vars}(\tau) \cup \mathsf{vars}(\tau'))$. This extends to sets of assignments by the natural join: for $R \subseteq \mathsf{Assign}(Y)$ and $S \subseteq \mathsf{Assign}(Z)$, $R \bowtie S = \{\, \tau \cup \tau' \;\big|\; \tau \in R,\ \tau' \in S,\ \tau \simeq \tau' \,\} \subseteq \mathsf{Assign}(Y \cup Z)$.

Every gate g of C accepts a set of assignments $[\![g]\!] \subseteq \mathsf{Assign}(\mathsf{vars}(g))$, defined

inductively by

$$[\![g]\!] = \begin{cases} \{\{\langle x : d\rangle\}\} & \text{if } g \text{ is an input gate } \langle x : d\rangle, \\ [\![c_1]\!] \bowtie \ldots \bowtie [\![c_m]\!] & \text{if } g \in \mathsf{AND},\ [\mathsf{inputs}(g)] = (c_1, \ldots, c_m), \\ \bigcup_{i=1}^{m} [\![c_i]\!] & \text{if } g \in \mathsf{OR},\ [\mathsf{inputs}(g)] = (c_1, \ldots, c_m). \end{cases}$$

We write $[\![C]\!] := [\![r]\!]$ and call its elements the *satisfying assignments* of C. We assume $\mathsf{vars}(r) = X$, which is without loss of generality because we may discard variables that do not occur in any input gate; the satisfying assignments of C are then total.

Definition 3.1.3. In this thesis, we focus on a multivalued circuit C with the following *structural properties*:

- (a) **Decomposability**: For every AND-gate g in C, the variable sets of its inputs are pairwise disjoint. Formally, $\mathsf{vars}(c_1) \cap \mathsf{vars}(c_2) = \emptyset$ for all $g \in \mathsf{AND}$ and all $c_1, c_2 \in \mathsf{inputs}(g)$ with $c_1 \neq c_2$. We say every AND-gate g is *decomposable*.

- (b) **Determinism**: For every OR-gate g in C, no assignment is accepted by two different inputs. Formally, for all $g \in \mathsf{OR}$ and all $c_1, c_2 \in \mathsf{inputs}(g)$ with $c_1 \neq c_2$, $\tau_1 \neq \tau_2$, for all $\tau_1 \in [\![c_1]\!]$, $\tau_2 \in [\![c_2]\!]$, i.e., there is some $x \in \mathsf{vars}(c_1) \cap \mathsf{vars}(c_2)$ with $\tau_1(x) \neq \tau_2(x)$. We say every OR-gate g is *deterministic*.

- (c) **Smoothness**: For every OR-gate g, all of its inputs share the same variable set, i.e., $\mathsf{vars}(c_i) = \mathsf{vars}(g)$ for all $g \in \mathsf{OR}$ and all $c_i \in \mathsf{inputs}(g)$. We say every OR-gate g is *smooth*.

By default, we assume circuit C satisfies the structural properties in Definition 3.1.3.

Observation 3.1.4. Let C satisfy Definition 3.1.3, let g be a gate of C and, if g is not an input gate, let $[\mathsf{inputs}(g)] = (c_1, \ldots, c_m)$.

1. *(Decomposability)* If $g \in \mathsf{AND}$, any $\tau_i \in [\![c_i]\!]$ and $\tau_j \in [\![c_j]\!]$ with $i \neq j$ are compatible, so the join is a product: $(\tau_1, \ldots, \tau_m) \mapsto \bigcup_i \tau_i$ is a bijection from $\prod_i [\![c_i]\!]$ onto $[\![g]\!]$. Conversely every $\tau \in [\![g]\!]$ decomposes uniquely as $\tau = \bigcup_i \tau|_{\mathsf{vars}(c_i)}$ with $\tau|_{\mathsf{vars}(c_i)} \in [\![c_i]\!]$. Moreover, no gate is reachable from both c_i and c_j with $i \neq j$, i.e., $\mathsf{gates}(c_i) \cap \mathsf{gates}(c_j) = \emptyset$, and the subcircuits C_{c_i}, C_{c_j} are vertex-disjoint.

2. *(Determinism)* If $g \in \mathsf{OR}$, then $[\![c_i]\!] \cap [\![c_j]\!] = \emptyset$ for $i \neq j$, so $[\![g]\!] = \biguplus_{i=1}^{m} [\![c_i]\!]$: every $\tau \in [\![g]\!]$ is accepted from exactly one input of g.

3. *(Smoothness)* $\mathsf{vars}(\tau) = \mathsf{vars}(g)$ for every $\tau \in [\![g]\!]$, i.e., $[\![g]\!] \subseteq \mathsf{Assign}(\mathsf{vars}(g))$.

The circuits we focus on are exactly the *smooth d-DNNF* circuits of Darwiche and Marquis [21], transposed to the multivalued setting.

§ 3.2 Subcircuit Sharing in a Circuit

In a DAG, a gate g may have more than one output, in which case g is *shared*: a single copy of subcircuit C_g is used by all outputs of g. Sharing propagates downwards: every gate in $\mathsf{gates}(g)$ is then also used by several outputs of g, even if it has a single output itself. Sharing is what makes a DAG more succinct than a tree [34], and it is also the property that separates the two enumeration algorithms in this thesis. We therefore distinguish the two settings.

Lemma 3.2.1. *Let g be a gate of C and let $c_1 \neq c_2$ be two of its inputs. If* $\mathsf{gates}(c_1) \cap \mathsf{gates}(c_2) \neq \emptyset$*, then* $g \in \mathsf{OR}$*. Consequently, if two distinct paths of C have the same endpoints, then every gate at which these paths diverge is an* OR *-gate.*

Proof. The first statement is the contrapositive of Observation 3.1.4(1).

For the second statement, let $p_1 \neq p_2$ be directed paths from g_0 to h, and let g be a gate where they diverge, say p_1 continues to c_1 and p_2 to c_2 with $c_1 \neq c_2$. Both paths reach h, so $h \in \mathsf{gates}(c_1) \cap \mathsf{gates}(c_2)$ and the first statement applies. □

Definition 3.2.2. A circuit C is *tree-shaped* if every gate other than the output gate r has exactly one output, i.e., if the underlying DAG is a tree rooted at r. When no such restriction is imposed, we call C a *general* circuit.

Observation 3.2.3. If C is tree-shaped, then $\mathsf{gates}(c_1) \cap \mathsf{gates}(c_2) = \emptyset$ for every gate g and all $c_1, c_2 \in \mathsf{inputs}(g)$ with $c_1 \neq c_2$.

In a tree-shaped circuit, each gate is reached from r by a unique path, while in a general circuit, this fails. The two cases are therefore treated separately: Chapter 5 exploits the uniqueness of paths, while Chapter 6 must address how each gate is reached.

§ 3.3 Problem Statement

Each assignment $\langle x : d \rangle$ (with $x \in X$, $d \in \mathsf{dom}_x$) in an input gate has a score $\omega(\langle x : d \rangle) \in \mathbb{R}$. The *score* of an assignment $\tau \in \mathsf{Assign}(X')$ is the sum of the scores of its elements,

$$\omega(\tau) \;=\; \sum_{\langle x:d \rangle \in \tau} \omega(\langle x : d \rangle).$$

We rank assignments in nondecreasing order of score and call this the *sum ordering*; it is a special case of subset-monotone rankings [1].

Definition 3.3.1. The *ranked enumeration problem* takes as input a smooth multivalued d-DNNF C with output gate r, together with a score $\omega(\langle x : d \rangle) \in \mathbb{R}$ for each one-variable assignment. It must enumerate all the assignments in $[\![C]\!]$ without duplicates in nondecreasing order of score (ties broken by any fixed rule).

An enumeration algorithm consists of a *preprocessing phase*, run before the first satisfying assignment is produced, and an *enumeration phase*, producing the satisfying assignments one after the other. *Delay* measures the time between two consecutive outputs, or between the last satisfying assignment and the announcement of the end of the enumeration.

Ranking in nondecreasing order is without loss of generality: due to the additivity of the weight function, replacing ω by $-\omega$ exchanges the two orders and swaps min with max throughout, so the algorithms can be applied to the reversed ranking.

Chapter 4

Algorithmic Framework

§ 4.1 Idea from Eppstein's Algorithm

Recall the outline of Eppstein's algorithm [26] for the k shortest paths between the designated start s and target t in a weighted digraph. First, it computes a shortest-path tree towards t, labeling every vertex v with its shortest distance $d(v)$ to t. Every s–t path then decomposes as the tree path plus a sequence of *sidetracks* (edges that leave the tree). A sidetrack $e = (u, v)$ with weight $w(e)$ carries the nonnegative "detour cost" $\delta(e) = w(e) + d(v) - d(u)$, and the cost of a path equals the shortest-path cost plus the sum of the detour costs of its sidetracks. Enumerating paths in nondecreasing cost thus reduces to enumerating the *sequences of sidetracks* in nondecreasing total detour cost, which Eppstein's algorithm organizes with heaps.

The same decomposition idea drives the algorithms to enumerate satisfying assignments over the circuit. The single cheapest satisfying assignment/weight is easy to obtain by a single pass from the input gates to the output gate: at each OR-gate one simply keeps a cheapest branch, and these local choices compose into a global cheap-

est assignment, which is the analogue of the all-cheapest path. Every other satisfying assignment can then be found as this cheapest one modified at some of the OR-gates, where it takes a branch other the cheapest (*deviation*). The cost of a deviation measures how much is lost by taking that branch instead of the cheapest one available. The weight of an assignment is the score of the cheapest assignment plus the costs of its deviations, so ranked enumeration becomes the enumeration of *sets of deviations* in nondecreasing total cost.

One structural difference keeps Eppstein's construction from carrying over directly. In Eppstein's problem, one enumerates every *path*, a sequence of edges, along which the sidetracks themselves naturally fall into a sequence. In our setting, however, an assignment corresponds to an *upward tree*: at each OR-gate it selects a single input, but at each AND-gate it combines independent assignments over disjoint sets of variables into a single assignment over their union [2]. Each input of an AND-gate admits its own deviations, independently of the others. Deviations therefore combine across the inputs of an AND-gate, in addition to accumulating along a single branch as in the path setting.

§ 4.2 Implicit Representation and Champions

Fix a circuit C with output gate r holding the structural properties of Definition 3.1.3 (decomposability, determinism, and smoothness). For a gate g, recall that $[\![g]\!] \subseteq \mathsf{Assign}(\mathsf{vars}(g))$ is the set of satisfying assignments of the subcircuit rooted at g, and $[\![C]\!] = [\![r]\!]$.

4.2.1 Implicit Representations

An assignment is described explicitly by listing its variable-value pairs; alternatively, it can be represented by recording, at each OR-gate it reaches, which input branch it takes.

Definition 4.2.1. A *branch choice* β assigns to each OR-gate g an input, $\beta(g) \in \mathsf{inputs}(g)$. Given β, the *trace* $\mathsf{trace}(g, \beta)$ is defined by recursion on g:

$$\mathsf{trace}(g,\beta) = \{g\} \cup \begin{cases} \emptyset & \text{if } g \text{ is an input gate,} \\ \bigcup_{c \in \mathsf{inputs}(g)} \mathsf{trace}(c,\beta) & \text{if } g \in \mathsf{AND}, \\ \mathsf{trace}\big(\beta(g),\beta\big) & \text{if } g \in \mathsf{OR}, \end{cases}$$

and the *reconstruction* $\mathsf{rec}(g, \beta) \in \mathsf{Assign}(\mathsf{vars}(g))$ collects the labels of the input gates in $\mathsf{trace}(g, \beta)$ to represent the *explicit* assignment. We write $\mathsf{trace}(\beta)$ and $\mathsf{rec}(\beta)$ for $\mathsf{trace}(r, \beta)$ and $\mathsf{rec}(r, \beta)$, and say that the gates in $\mathsf{trace}(\beta)$ are *visited* by β.

Only the values of β at the visited OR-gates enter the recursion, so an assignment is determined by the restriction of β to these gates, which is the *implicit representation* of $\mathsf{rec}(\beta)$.

Observation 4.2.2. Let β be a branch choice. Every gate of $\mathsf{trace}(g, \beta)$ is reached from g by a unique path inside $\mathsf{trace}(g, \beta)$.

Observation 4.2.3. For every gate g, reconstruction induces a bijection between the satisfying assignments in $[\![g]\!]$ and the implicit representations rooted at g. In particular, every satisfying assignment of C has a unique implicit representation, so enumerating implicit representations enumerates $[\![C]\!]$ exactly once.

By Observation 4.2.3, every $\tau \in [\![C]\!]$ is $\mathsf{rec}(\beta)$ for some branch choice β, written β_τ, unique at the OR-gates it visits. So, enumerating branch choices is equivalent to enumerating assignments, reconstructing an explicit assignment only when needed.

Computing $\mathsf{rec}(\beta)$ by the recursion visits each gate in $\mathsf{trace}(\beta)$ once, so takes $O(|\mathsf{trace}(\beta)|)$ time; in general $O(|\mathsf{trace}(\beta)|)$ can be $O(|C|)$. The number of input gates in $\mathsf{trace}(\beta)$ is exactly $|X|$, and since no gate has a single input, the AND-gates in $\mathsf{trace}(\beta)$ are fewer than its input gates. However, the number of OR-gates is not controlled by $|X|$: an OR-gate may have an OR-gate as an input, and such a *chain* of OR-gates contributes one gate to $\mathsf{trace}(\beta)$ without contributing any input gate. Call a directed path of C whose gates are all OR-gates an *OR-chain*, and write λ for the maximal number of gates on an OR-chain; we then have $|\mathsf{trace}(\beta)| = O(|X| \cdot \lambda)$, because following β maps each OR-gate of $\mathsf{trace}(\beta)$ to a non-OR-gate, with at most λ OR-gates mapped to the same gate. In particular, reconstruction takes $O(|X|)$ time when C has no chain of OR-gates, i.e., when $\lambda = 1$.

Definition 4.2.4. Fix a linear order on $\mathsf{inputs}(g)$ for every gate g. For a branch choice β and a gate x, we denote by $\prec_{x,\beta}$ the depth-first order induced by $\mathsf{trace}(x, \beta)$: for distinct gates g_1, g_2 of $\mathsf{trace}(x, \beta)$, let g be the last gate common to the paths from x to g_1 and to g_2, which are unique by Observation 4.2.2; then $g_1 \prec_{x,\beta} g_2$ if $g_1 = g$, or if g_1 and g_2 are reachable from distinct inputs c_1, c_2 of g respectively, with c_1 preceding c_2 under the fixed linear order. We say that g_2 comes *after* g_1 when $g_1 \prec_{x,\beta} g_2$.

This order plays a structural role in the following chapters. A trace is obtained from another by changing the branch choice at some of its gates, and a linear order on its gates lets any such set of changes be written as an increasing sequence, which is unique since a set admits exactly one increasing enumeration. Depth-first order is used because

it groups the gates below a gate g together: every gate coming after g is either below g or comes after all gates below g. Changing the branch choice at g replaces the gates below g and leaves all other gates of the trace unchanged.

4.2.2 Optimum score and champion

Definition 4.2.5. For a gate g, the *optimum score* $\omega^*(g)$ is the least score of a satisfying assignment of the subcircuit rooted at g:

$$\omega^*(g) \;=\; \min_{\tau \in [\![g]\!]} \omega(\tau).$$

Proposition 4.2.6. *The optimum score satisfies*

$$\omega^*(g) = \begin{cases} \omega(\langle x : d \rangle) & \text{if } g \text{ is an input gate } \langle x : d \rangle, \\ \sum_{c \in \mathsf{inputs}(g)} \omega^*(c) & \text{if } g \in \mathsf{AND}, \\ \min_{c \in \mathsf{inputs}(g)} \omega^*(c) & \text{if } g \in \mathsf{OR}\,. \end{cases}$$

Proof. By induction on g. If g is an input gate, $[\![g]\!]$ contains only the singleton assignment $\langle x : d \rangle$, so $\omega^*(g) = \omega(\langle x : d \rangle)$. If g is an AND-gate, decomposability gives $[\![g]\!] = \bowtie_c [\![c]\!]$ over pairwise disjoint variable sets, and scores add under $\bowtie$; minimizing each score for each input independently yields $\omega^*(g) = \sum_c \omega^*(c)$. If g is an OR-gate, then $[\![g]\!] = \bigcup_c [\![c]\!]$, so $\omega^*(g) = \min_c \min_{\tau \in [\![c]\!]} \omega(\tau) = \min_c \omega^*(c)$. □

The recurrence is evaluated in one pass in topological order, in time $O(|C|)$. At each OR-gate we select *one* score-minimal input as a champion input.

Definition 4.2.7. Fix an order on elements in $[\mathsf{inputs}(g)]$ for every OR-gate g. The *champion input* $c^*(g)$ is the ω^*-minimal input, ties broken by that order. We have

$\omega^*\big(c^*(g)\big) = \omega^*(g)$.

Taking the champion input everywhere in a d-DNNF C gives a specific branch choice, whose reconstruction is the cheapest satisfying assignment.

Definition 4.2.8. *Champion choice* β^* sets $\beta^*(g) = c^*(g)$ at every OR-gate g. We write $T_g := \mathsf{trace}(g, \beta^*)$ for the *champion trace* of g, the gates reached from g by expanding every AND-gate and following the champion input at every OR-gate; thus *champion* at g is $\tau^*(g) := \mathsf{rec}(g, \beta^*)$, collecting the labels of the input gates; in particular, the global champion at r, i.e., $\tau^*(r)$.

Remark. The algorithm stores both ω^* and c^*, taking $O(1)$ space and time per gate; it never caches any $\tau^*(g)$, leading to $O(|X|)$ time to construct and $O(|X|)$ space to store. Additionally, the optimum score $\omega^*(g)$ is unique, whereas a score-minimal assignment may not be; $\tau^*(g)$ is the *specific representative* determined by the tie-breaking order of Definition 4.2.7.

§ 4.3 Deviation

A deviation takes a non-champion branch at an OR-gate; its cost measures how much more that branch costs than the champion input.

Definition 4.3.1. A *deviation* d is a pair (g, c) where g is an OR-gate and $c \in \mathsf{inputs}(g)$ with $c \neq c^*(g)$; it records taking the non-champion branch c at g. We write $\mathsf{source}(d) = g$ and $\mathsf{target}(d) = c$ and call them *source (gate)* and *target (gate)* of d respectively. Its *cost* is

$$dev(g, c) \;:=\; \omega^*(c) - \omega^*(g) \;\geqslant\; 0.$$

$dev(g, c)$ is nonnegative because $\omega^*(g) = \min_{c' \in \mathsf{inputs}(g)} \omega^*(c') \leqslant \omega^*(c)$.

Definition 4.3.2. For $\tau \in [\![C]\!]$, define *deviation set*

$$\mathsf{Dev}(\tau) \;:=\; \left\{ (g, \beta_\tau(g)) \;\middle|\; g \in \mathsf{trace}(\beta_\tau) \cap \mathsf{OR} \text{ and } \beta_\tau(g) \neq c^*(g) \right\}.$$

$\mathsf{Dev}(\tau)$ lists only the places where τ differs from the champion. At an OR-gate where τ takes the champion branch, nothing is recorded; a deviation is recorded only where τ takes a branch other than the champion branch. The global champion $\tau^*(r)$ has an empty deviation set, and every other assignment τ is described by the set $\mathsf{Dev}(\tau)$, recording the branches at which it departs from the champion.

Definition 4.3.3. Let D be a set of deviations containing at most one pair for each OR-gate. The *induced branch choice* β_D is given by $\beta_D(g) = c$ if $(g, c) \in D$, and $\beta_D(g) = c^*(g)$ otherwise. We call D *well-formed* if every g with $(g, c) \in D$ is visited by β_D.

Lemma 4.3.4. $\tau \mapsto \mathsf{Dev}(\tau)$ *is a bijection from* $[\![C]\!]$ *onto the set of well-formed deviation sets, with inverse* $D \mapsto \mathsf{rec}(\beta_D)$.

Definition 4.3.5. A *deviation record* is either $\perp$ or a pair $\langle s', d \rangle$ where s' is a deviation record and d a deviation. We write $\sigma(s)$ for the sequence of deviations obtained by following s back to $\perp$. Records are immutable and shared between states, so extending one takes constant time.

Lemma 4.3.6. *For every* $\tau \in [\![C]\!]$,

$$\omega(\tau) \;=\; \omega^*(r) \;+\; \sum_{d \in \mathsf{Dev}(\tau)} dev(d).$$

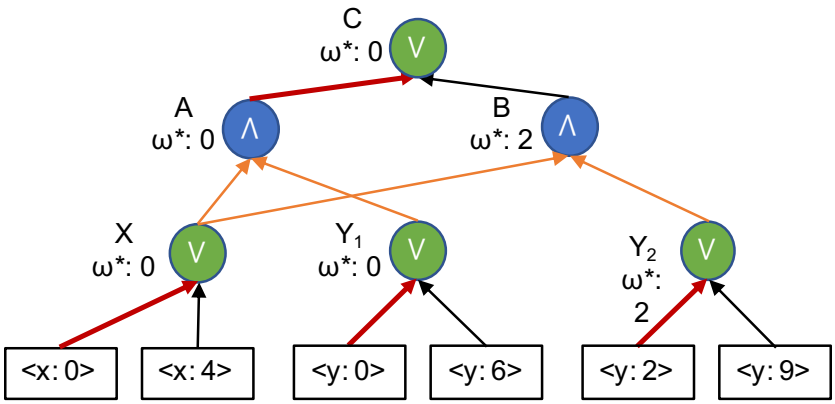


Figure 4.1: A smooth d-DNNF circuit with scores on the input gates. Red edges mark the edges connecting to a champion input; ω^* is the optimum score at a gate.

Intuitively, the champion is the cheapest assignment possible, with a baseline of score $\omega^*(r)$. Any other τ is more expensive only because it departs from the champion at some OR-gate, and each deviation d carries a nonnegative cost $dev(d)$; summing these costs onto the baseline recovers the score of τ. The champion's sum of costs is empty and its score is exactly $\omega^*(r)$. This is an analogue of Eppstein's decomposition of a path into a shortest path with a sequence of sidetracks.

This decomposition helps optimal-first enumeration. Since $\omega^*(r)$ is fixed, producing assignments in nondecreasing score is equivalent to generating deviation sets in nondecreasing total cost, starting from the empty set. Building on this reduction, the following data structures and algorithms are developed to perform the full enumeration.

§ 4.4 Persistent Meldable Heap

Figure 4.1 shows a small circuit whose champion assignment is $(x{:}0,\ y{:}0)$ of score 0. This answer is collected from the trace $= \{C, A, X, Y_1, \langle x{:}0\rangle, \langle y{:}0\rangle\}$. Ask the question

the enumeration will keep asking: starting from the champion, what can be changed in *one* step, and at what cost? At X, we can switch to $\langle x{:}4\rangle$ costing 4; at Y_1, switch to $\langle y{:}6\rangle$ costing 6; at C there is also the switch away from the champion input A to input B, which does not collect data from Y_1 and switches to the $\langle y{:}2\rangle$ below gate Y_2, costing $\omega(B) - \omega(C) = 2$. Collecting these at C gives three one-step departures, of costs 2, 4 and 6. Reading the figure this way already fixes what the structure at a gate has to do.

- **Ordered access.** The enumeration wants the cheapest departure first, so a gate's departures cannot sit in a list: they must be ordered by cost. At C the order is $2, 4, 6$.

- **Combination.** A gate does not compute its departures from scratch. At an AND-gate, they are those of all inputs taken together: at A, those of X and those of Y_1. By decomposability the inputs have disjoint variable sets, so they do not affect each other. At an OR-gate they are those of the champion inputs, together with one new entry per non-champion input: at C, those of A *plus* "take B instead". Note that the deviations below B do not appear at C: switching to B and then deviating inside is two steps. Either way, the operation needed is a union of collections already built, plus $O(1)$ fresh entries.

- **Costs carry over unchanged.** The entry of X costs 4 at X, and still 4 at A and at B, although $\omega(A) = 0$ and $\omega(B) = 2$. This is no accident: at an AND-gate the optima of the inputs add up to the optimum of the gate, and at an OR-gate the optimum of the gate is that of its champion input. The union therefore never has to rebuild the key for anything.

- **Operands must survive.** (1) In the heap-building phase, X feeds both A and

> B, so its collection is combined twice, and what A does while building its own must leave that of X intact for B. (2) In enumeration, an answer may depart at X and then depart at Y_1 as well: taking the second departure means consulting the collection at Y_1, unchanged, long after A's collection was assembled out of it. The first reason is particular to general circuits with subcircuit sharing; the second applies to both.

The first requirement asks for a heap, the second and third for a *meldable* one, and the fourth for a *persistent* one. We use *leftist heaps* [20] [29, Section 5.2.3] to satisfy the conditions. The following two features guarantee the complexity of our algorithms:

- A persistent Meld of two leftist heaps of size n takes $O(\log n)$ time; it rewrites only the nodes on the merged right spine and shares the rest, so the operands survive at no extra cost [31, Section 3.1].
- A leftist heap is a heap-ordered binary tree, so every node has at most two children.

We benefit from the second property in the enumeration phase. Heap order says that the key of a node is at least the key of its parent, so the entries are already arranged in a tree in which every root-to-node path has nondecreasing cost. That is enough to visit them in nondecreasing order without deleting anything: begin at the root, and each time an entry is taken, its at most two children become new candidates. The technique is similar to Eppstein's [26] though it also uses other special structures. Two things follow: each entry taken adds a constant number of candidates, and the heap is only ever read, which is what lets one heap be shared across the branches of the enumeration.

Two questions are deliberately left open here. First, an answer may depart from the champion in several places at once: in Figure 4.1, one may change both X and Y_1,

reaching the assignment of score 10, which is no entry of any heap we currently have. Meanwhile, combining departures raises the problem of generating each combination once only: changing X then Y_1 and changing Y_1 then X must not both be produced. That is the reason we need the order of Definition 4.2.4. Second, once a departure has been taken, which further departures are still available depends on where one now is: after switching to take $\langle y{:}2\rangle$ below B as a deviation, we should have an approach to obtain the deviation on top of the committed deviation. Both questions will be addressed differently for tree-shaped and for general circuits; we will return to them in Chapter 5 and Chapter 6.

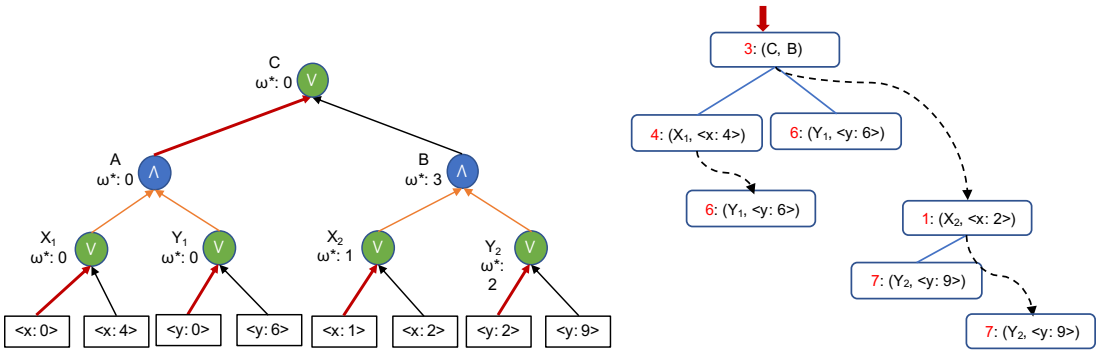
Chapter 5

Ranked Enumeration on Tree-shaped Circuits

A tree-shaped circuit has no sharing of subcircuits: every gate has a single output. The assignments accepted by gate g come from the subcircuit rooted at g and do not require knowing how the gate was reached, whether in a tree or in a DAG. What may differ is the part that remains *above* g: the sibling branches of the AND-gates above g, and the deviations they offer. In a tree-shaped circuit these are fixed once the circuit is specified, since g is reached from output gate r by a unique path.

§ 5.1 A Small Example

Let us begin with a small tree-shaped circuit. For the circuit of Figure 5.1a, what the preprocessing phase builds is the structure of Figure 5.1b. Its topmost entry is the deviation (C, B), of cost 3 relative to the champion. From an entry, the enumeration follows a solid blue edge (*heap edge*) to *replace* the last deviation by another one, and a dashed black edge (*cross pointer*) to *append* a further deviation. Every set of deviations reachable in this way represents an answer, and the answers are produced in nondecreasing order of score. The rest of this chapter makes both the construction and



(a) (b)

Figure 5.1: (a) A tree-shaped circuit over $X = \{x, y\}$, with $C = A \vee B$, $A = X_1 \wedge Y_1$ and $B = X_2 \wedge Y_2$; each gate is labeled ω^*. The three available deviations cost +4 at X_1, +6 at Y_1, and +3 at C (switching the whole subcircuit from A to B). (b) Sketch map of the deviations of Figure 5.1a held in heaps ordered by cost (we ignore persistency here). An entry $\delta : (g, c)$ records the deviation at gate g towards its input c, of cost δ.

the enumeration precise.

§ 5.2 Continuation and Deviation Heap

Suppose all deviations of C were collected in a single heap, keyed by cost, one entry per deviation. Its minimum would be the cheapest single deviation, hence the second-best assignment overall; the heap children of that entry would give the candidates for the third, and so on. In this way, the assignments whose deviation set has size one are produced in the correct order.

However, assignments with more deviations are not reached. After taking a deviation (g, c) at an OR-gate g, the assignment is no longer the global champion, and the deviations that remain available fall into two categories: those inside C_c exposed by following c, and those at the gates the trace still has to visit *after* g.

Three components are therefore needed. First, an order on the gates, so that *after some gate* is meaningful. Second, a heap for each gate, holding the deviations available below that gate together with those that come after it; those already passed are excluded, which is what makes the deviations of a set occur in a fixed order. Third, a pointer on each entry leading to the heap of the deviations that remain available once that deviation is taken. Each entry of the resulting heaps stands for one deviation that may be added to the deviation set.

Throughout this chapter, we write $\prec$ for $\prec_{r,\beta}$ of Definition 4.2.4. Since C is a tree, the paths from r are those of C itself and $\prec$ does not depend on β: it is one fixed order on the gates of C.

Definition 5.2.1. Let β be a branch choice. The *continuation* of $g \in \mathsf{trace}(\beta)$ is

$$\mathsf{Cont}_\beta(g) := \{h \in \mathsf{trace}(\beta) \setminus \mathsf{trace}(g, \beta) \mid g \prec h\},$$

the gates of $\mathsf{trace}(\beta)$ that come after the whole of $\mathsf{trace}(g, \beta)$. For simplicity, we write $\mathsf{Cont}(g)$ for $\mathsf{Cont}_{\beta^*}(g)$.

The order $\prec$ is linear on the gates of C, and it is in fact the depth-first order on the gates of C. However, $\mathsf{trace}(\beta)$ depends on β: taking a non-champion input at an OR-gate brings the gates of the corresponding subcircuit into the trace, together with the deviations they carry. The continuation of g contains the gates of $\mathsf{trace}(\beta)$ that come after the whole of $\mathsf{trace}(g, \beta)$.

Definition 5.2.2. Let β be a branch choice and $g \in \mathsf{trace}(\beta)$. The deviations *available* at g are the pairs (h, c) where h is an OR-gate in $\mathsf{trace}(g, \beta) \cup \mathsf{Cont}_\beta(g)$ and $c \in \mathsf{inputs}(h) \setminus \{c^*(h)\}$. The *deviation heap* $H_\beta(g)$ contains one entry e per available deviation (h, c),

consisting of its cost $e.\delta = dev(h, c)$ as key, the deviation itself $e.d = (h, c)$, and a *cross* pointer to $e.cross = H_{\beta'}(c)$ where $\beta' = \beta_{D\cup\{(h,c)\}}$ and D is the deviation set of β. Moreover, for a set Y of gates we write $H_\beta(Y)$ for the heap of the deviations at the OR-gates of Y. We write $H(g)$ for $H_{\beta^*}(g)$.

Remark. Note that the two parts of $\mathsf{trace}(g, \beta) \cup \mathsf{Cont}_\beta(g)$ play different roles. The first is what lies below g, and Algorithm 1 always expands it along the champion; the second is what remains outside, and it is the algorithm's second argument. Only the latter depends on β in the calls the algorithm performs, whereas the first is always T_g.

Within a heap, the children of an entry offer alternatives to it to explore deviation sets of the same size. The cross pointer instead leads to the heap of the deviations that remain available once that deviation is taken, searching the deviation sets with one more element.

§ 5.3 Preprocessing

5.3.1 Bottom-up Construction

The preprocessing consists of two bottom-up passes over C. The first evaluates the recurrences of § 4.2.2, producing $\omega^*(g)$ and $c^*(g)$ for every gate. The second is Algorithm 1 to build the deviation heaps.

Algorithm 1 Deviation heap construction

```
1:  function BuildHeap(g, H_cont)
2:      if g is an input gate then
3:          return H_cont                    ▷ no choice here; only the continuation remains
4:      else if g is an AND-gate with inputs (c_1, ..., c_m) then
5:          H ← H_cont
6:          for j = m downto 1 do
7:              H ← BuildHeap(c_j, H)        ▷ Cont(c_j) = {c_{j+1}, ..., c_m} ∪ Cont(g)
8:          return H
9:      else if g is an OR-gate with inputs (c_1, ..., c_n) and champion c* then
10:         H ← BuildHeap(c*, H_cont)        ▷ deviations from Cont(g) and C_{c*}
11:         for all c ∈ {c_1, ..., c_n} \ {c*} do
12:             H_c ← BuildHeap(c, H_cont)
13:             e_c ← (dev(g, c), (g, c), H_c)   ▷ deviation entry; cross pointer to H_c
14:             H ← Meld(H, {e_c})           ▷ meld the entry into the heap
15:         return H
```

We briefly describe the calls along the champion choice, so we have $\beta = \beta^*$ and the subscript is omitted. Calls at other inputs will be covered by Lemma 5.3.1. At output gate r, the initial call is $\textsc{BuildHeap}(r, \emptyset)$ for $\mathsf{Cont}(r) = \emptyset$.

At an input gate no deviation is available and the argument is returned unchanged.

At an AND-gate, the continuation of c_j consists of $\mathsf{Cont}(g)$ together with $c_{j+1}, \ldots, c_m$ and the gates below them. The inputs are processed from the last to the first in the fixed order, so that the heap passed to the call at c_j has accumulated the deviations contributed from $c_{j+1}, \ldots, c_m$.

At an OR-gate g, the entries created are those of the deviations available at g itself; the entries of $\mathsf{Cont}(g)$ are already present in H_{cont} and are inherited through Meld. The champion input receives H_{cont} unchanged, since g has a single champion input. Each remaining input c receives the same argument, and the resulting heap is stored as the pointer of e_c: it collects the deviations that remain available once (g, c) is taken, i.e.,

those inside C_c and those on $\mathsf{Cont}(g)$.

Note that all heap operations are persistent: MELD returns a new heap and leaves its arguments unchanged. Heaps built earlier remain the same and available to the entries that point to them.

Remark. In the calls performed by Algorithm 1, we always go along the champion trace, so $\mathsf{Cont}_\beta(g) = \mathsf{Cont}(g)$ throughout and the heap built at g depends on g alone. Figure 5.2 gives an example. The branch choice is carried in the statement only so that the cross pointer can be named: the heap reached once (g, c) is taken is $H_{\beta'}(c)$ for the branch choice β' recording the new deviation.

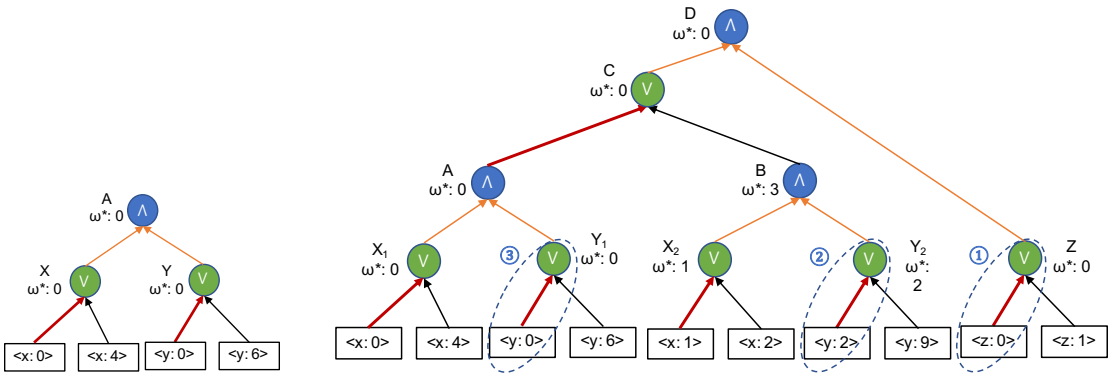


Figure 5.2: Circuit in (a) contains only one AND-gate. The continuation of gates in subcircuit C_X is $\{\mathrm{Y}, \langle y{:}2\rangle\}$.
Circuit in (b) has a more complex structure. For the gates in the subcircuits C_{Y_2} and C_{Y_1}, the continuation is the same, set marked with ①. For the gates in C_{X_2}, their continuation is the union of sets ① and ②. For gates in C_{X_1}, the continuation is the union of sets ① and ③.

5.3.2 Correctness

Lemma 5.3.1. *For every branch choice β and every $g \in \mathsf{trace}(\beta)$ with $\mathsf{trace}(g, \beta) = T_g$, the call $\textsc{BuildHeap}(g, H_\beta(\mathsf{Cont}_\beta(g)))$ returns $H_\beta(g)$, and the heap stored in the entry of a deviation (g, c) created at that call is $H_{\beta'}(c)$ for $\beta' = \beta_{\mathsf{Dev}(\beta)\cup\{(g,c)\}}$.*

Proof. By induction on g.

If g is an input gate, $\mathsf{trace}(g, \beta) = \{g\}$ carries no deviation, so $H_\beta(g) = H_\beta(\mathsf{Cont}_\beta(g))$ and the argument is returned unchanged.

Let $g \in \mathsf{AND}$ with inputs $(c_1, \ldots, c_m)$. The hypothesis $\mathsf{trace}(g, \beta) = T_g$ gives $\mathsf{trace}(c_j, \beta) = T_{c_j}$ for every j, so the lemma applies at each input. By Definition 5.2.1, $\mathsf{Cont}_\beta(c_j) = \mathsf{Cont}_\beta(g) \cup \bigcup_{j'>j} \mathsf{trace}(c_{j'}, \beta)$, and these sets are pairwise disjoint by decomposability, so melding their heaps yields one entry per deviation of the union. The backward loop ($j = m$ down to 1) ensures that when c_j is processed, the deviations of the later inputs have already been accumulated into the heap, which is therefore $H_\beta(\mathsf{Cont}_\beta(c_j))$; by induction the call at c_j returns $H_\beta(c_j)$. No deviation occurs at g itself, and $\mathsf{trace}(g, \beta) = \{g\} \cup \bigcup_j \mathsf{trace}(c_j, \beta)$, so the heap returned after the last iteration covers $\mathsf{trace}(g, \beta) \cup \mathsf{Cont}_\beta(g)$, that is, it is $H_\beta(g)$.

Let $g \in \mathsf{OR}$. The hypothesis gives $\beta(g) = c^*(g)$, so $\mathsf{trace}(g, \beta) = \{g\} \cup \mathsf{trace}(c^*(g), \beta)$; the two subtrees have the same successors and $\mathsf{Cont}_\beta(c^*(g)) = \mathsf{Cont}_\beta(g)$. The first call therefore receives the right argument and returns $H_\beta(c^*(g))$ by induction, a heap covering $\mathsf{trace}(c^*(g), \beta) \cup \mathsf{Cont}_\beta(g)$. Melding the entries e_c for $c \neq c^*(g)$ adds exactly the deviations occurring at g, each once; they were not present before, since g is in neither of these two sets. The result covers $\mathsf{trace}(g, \beta) \cup \mathsf{Cont}_\beta(g)$, that is, it is $H_\beta(g)$.

Then we identify the heap stored in e_c, i.e., the value of $\textsc{BuildHeap}(c, H_\beta(\mathsf{Cont}_\beta(g)))$.

Write $\beta' = \beta_{\mathsf{Dev}(\beta)\cup\{(g,c)\}}$. The choices β and β' differ only at g, so $\mathsf{trace}(g, \beta)$ is replaced by $\{g\} \cup \mathsf{trace}(c, \beta')$ and the gates coming after the whole of either subtree are the same; hence $\mathsf{Cont}_{\beta'}(c) = \mathsf{Cont}_{\beta}(g)$, and the call receives $H_{\beta'}(\mathsf{Cont}_{\beta'}(c))$. Moreover $\mathsf{Dev}(\beta)$ carries no deviation inside T_c, by the hypothesis $\mathsf{trace}(g, \beta) = T_g$, so $\mathsf{trace}(c, \beta') = T_c$ and the lemma applies at c. Induction gives $H_{\beta'}(c)$, as Definition 5.2.2 requires. □

5.3.3 Complexity

Proposition 5.3.2. *For a tree-shaped circuit C, ω^*, c^* and all deviation heaps are computed in time and space $O(|C| \log |C|)$.*

Proof. The first pass evaluates one recurrence per gate, in time $O(|C|)$. Since C is tree-shaped, Algorithm 1 recursively processes the circuit strictly top-down, so BuildHeap is called once per gate. Each call performs one Meld per non-champion input of an OR-gate, so $O(|C|)$ melds in total. Every heap holds at most one entry per deviation of C, so its size is $O(|C|)$. Each single Meld() on the leftist heap costs $O(\log |C|)$ time and allocates $O(\log |C|)$ for the copy of persistent nodes. Therefore, both time and space are $O(|C| \log |C|)$. □

§ 5.4 Enumeration

For every gate, Algorithm 1 gives the heap of the deviations available from that gate onwards (according to the $\prec$-order). The champion has empty deviation set and is output first. Every other assignment has a non-empty deviation set, and by Lemma 4.3.4 it is determined by that set. Next, it remains to enumerate the well-formed deviation sets in order of total cost.

5.4.1 State and Enumeration

Definition 5.4.1. A *state* is a triple (π, u, s), where s is a deviation record (we write $\sigma(s)$ for the deviation sequence), and u is an entry recording the last deviation of $\sigma(s)$. The total cost of the deviations $\pi = \sum_{d' \in \sigma(s)} dev(d')$ serves as the key for ordering and is used to compute the total score.

The enumeration phase takes ω^*, c^* and the deviation heaps produced by the preprocessing of § 5.3 as input.

Algorithm 2 Ranked enumeration on a tree-shaped circuit

```
1:  procedure ENUMERATE(r)
2:      output (ω*(r), ⊥)                          ▷ champion; empty deviation record
3:      Q ← ∅
4:      if H(r) ≠ ∅ then
5:          u ← FINDMIN(H(r))
6:          PUSH(Q, (u.δ, u, ⟨⊥, u.d⟩))
7:      while Q ≠ ∅ do
8:          (π, u, s) ← POPMIN(Q)
9:          output (ω*(r) + π, s)
            ▷ (Replace) another deviation in place of u
10:         for all u′ ∈ {u.left, u.right} \ {⊥} do
11:             PUSH(Q, (π − u.δ + u′.δ, u′, ⟨s.prev, u′.d⟩))
            ▷ (Append) descend into the deviation heap after u
12:         if u.cross ≠ ⊥ then
13:             v ← FINDMIN(u.cross)
14:             PUSH(Q, (π + v.δ, v, ⟨s, v.d⟩))
```

The champion is emitted first, with the empty deviation record and score $\omega^*(r)$; thereafter a priority queue Q keyed by π drives the search. Popping the cheapest state outputs its deviation record; the state then spawns its successors by following the tree structure of the leftist heaps and its cross pointer.

Replace (u.left, u.right). The entry u sits in a leftist heap, itself a heap-ordered binary tree; its two children are the next-cheapest deviations that could have been chosen in place of u. Replacing u by such a child u' keeps the length of the record unchanged, with the total deviation cost changing by $-u.\delta + u'.\delta \geqslant 0$.

Append (u.cross). Committing to u, we descend into the deviations that remain available once $u.d$ is taken: those inside C_c and from $\mathsf{Cont}(g)$. u.cross is the heap of the subcircuit entered by taking the deviation $u.d$. They are collected in u.cross by Algorithm 1. Its minimum v is the cheapest among them; pushing it extends the record by one element with added cost $v.\delta \geqslant 0$.

5.4.2 Correctness

Observation 5.4.2. We extend $\prec$ to deviations: $d \prec d'$ if $\mathsf{source}(d) \prec \mathsf{source}(d')$. On a well-formed deviation set, $\prec$ is linear. The deviations recorded in a state increase for $\prec$. This follows from an invariant: at every point of the generation, all deviations in the current heap come after all deviations already recorded. It holds initially for $H(r)$, where nothing is recorded yet. *Replace* keeps the current heap and exchanges u for another entry in it. *Append* commits the current deviation and moves to the heap of its cross pointer, whose deviations come after it. A deviation sequence generated in our algorithm is therefore determined by the set of deviations it records, and we write $\sigma(s)$ for either without distinction.

Lemma 5.4.3. *For every state (π, u, s) obtained by Algorithm 2, $\sigma(s)$ is well-formed, and $\pi = \sum_{d \in \sigma(s)} dev(d)$.*

Proof. By induction on the number of moves performed to reach the state. Together, we track the deviation heap where u sits. There is a deviation set D with $\sigma(s) = D \cup \{u.d\}$ and a gate x such that u is an entry of $H_{\beta_D}(x)$.

Initially, $\sigma(s)$ is empty, and the first element added is $u.d = (g, c)$ with u having the minimum cost in $H(r) = H_{\beta^*}(r)$. By Definition 5.2.2, $\mathsf{source}(u.d) \in \mathsf{trace}(\beta^*)$, so $\sigma(s)$ is well-formed, and $\pi = u.\delta = dev(u.d)$. u sits in $H_{\beta_D}(x)$ where $D = \emptyset$ and $x = r$.

For the step, let (π, u, s) satisfy the claims, and write $u.d = (h, c)$. By Definition 5.2.2, every entry e of $H_{\beta_D}(x)$ satisfies $\mathsf{source}(e.d) \in$
$trace(x, \beta_D) \cup \mathsf{Cont}_{\beta_D}(x)$, and these gates are contained in $\mathsf{trace}(\beta_D)$.

Replace takes an entry u' of $H_{\beta_D}(x)$ and yields $\sigma' = D \cup \{u'.d\}$. The deviations at the gates above $\mathsf{source}(u'.d)$ are those in D and remain unchanged, so $\mathsf{source}(u'.d) \in \mathsf{trace}(\beta_{\sigma'})$ and σ' is well-formed. The new cost $\pi - u.\delta + u'.\delta$ is the sum over σ'. u' sits in $H_{\beta_D}(x)$ with the same D and x.

Append takes the minimum v of $u.\mathrm{cross}$, which is $H_{\beta_{\sigma(s)}}(c)$, and yields $\sigma' = \sigma(s) \cup \{v.d\}$. Every entry e of that heap satisfies $\mathsf{source}(e.d) \in \{c\} \cup \mathsf{Cont}_{\beta_{\sigma(s)}}(c) \subseteq \mathsf{trace}(\beta_{\sigma(s)})$. The choices above $\mathsf{source}(v.d)$ are those recorded in $\sigma(s)$; hence $\mathsf{source}(v.d) \in \mathsf{trace}(\beta_{\sigma'})$ and σ' is well-formed. The new cost is $\pi + v.\delta$. v sits in $H_{\beta_D}(x)$ with $D = \sigma(s)$ and $x = c$. □

Lemma 5.4.4. *Every non-empty well-formed deviation set is $\sigma(s)$ for exactly one state (π, u, s) obtained by Algorithm 2.*

Proof. Let D be non-empty and well-formed, with elements $d_1 \prec \cdots \prec d_n$.

Existence. Since d_1 must be $\prec$-least in D, no deviation of D occurs above $\mathsf{source}(d_1)$, so $\mathsf{source}(d_1) \in \mathsf{trace}(\beta^*)$ and d_1 is an entry of $H(r)$. The initial state records the minimum of $H(r)$, and *Replace* moves exchange this single deviation for any other entry

of the heap; hence $\{d_1\}$ is generated. Take a generated state with $\sigma(s) = \{d_1, \ldots, d_i\}$ and entry u recording d_i. Since d_{i+1} comes after d_i and $\mathsf{source}(d_{i+1})$ is in $\mathsf{trace}(\beta_{\{d_1,\ldots,d_{i+1}\}})$, it comes from an entry of $u.\mathrm{cross}$ by Definition 5.2.2, reached by one *Append* followed by *Replace* moves.

Uniqueness. Let deviation record s be obtained with $\sigma(s) = D$. By Observation 5.4.2, the deviations were recorded in the order $d_1, \ldots, d_n$. The state generation must perform $n - 1$ *Append* moves with the i-th move committing d_i exactly. The *Replace* moves can traverse from the minimum of a heap to a fixed entry. The heap we use is a binary tree, so this path is unique. The whole generation is therefore determined by D. □

Lemma 5.4.5. *The cost of a state is at least the cost of the state it was generated from.*

Proof. *Replace* changes cost by $-u.\delta + u'.\delta$, which is nonnegative since u' is a child of u in a heap ordered by cost. *Append* changes cost by adding $v.\delta = dev(v.d) \geqslant 0$. □

Theorem 5.4.6. *Algorithm 2 outputs the assignments of $[\![C]\!]$ in nondecreasing order of score, each exactly once.*

Proof. The champion $\mathsf{rec}(\beta^*)$ is output first, with score $\omega^*(r)$. By Lemmas 4.3.4 and 5.4.4, the states generated by Algorithm 2 are in bijection with $[\![C]\!] \setminus \{\mathsf{rec}(\beta^*)\}$. Every obtained state is eventually pushed and popped, since a state is pushed when the state it is generated from is popped, so each assignment is output exactly once. Its score is $\omega^*(r) + \pi$ by Lemma 5.4.3.

For the order, recall the priority queue Q is keyed by π, so the state popped at any iteration has minimum cost among those in Q. Let Σ be the state popped at some iteration and Σ' the state popped at the next one. Either Σ' was already in Q when Σ was popped, and then $\pi(\Sigma) \leqslant \pi(\Sigma')$ because Σ has the minimum cost of Q; or Σ' was pushed

by Σ and still $\pi(\Sigma) \leqslant \pi(\Sigma')$ by Lemma 5.4.5. The costs of consecutively popped states are therefore nondecreasing. □

5.4.3 Complexity

Theorem 5.4.7. *After preprocessing, Algorithm 2 outputs the k-th assignment with delay $O(\log k)$ in the implicit representation.*

Proof. Each pop generates at most three successors: at most two by *Replace* and at most one by *Append*. Hence after k pops at most $3k$ states have been pushed, and Q has size $O(k)$. Producing the k-th output performs one PopMin and at most three Push operations on a queue of this size, each in $O(\log k)$ time. Reading the children of an entry or the root of a heap and updating the cost take constant time. So does extending a deviation record, which shares its prefix instead of copying it. The delay of the implicit representation is therefore $O(\log k)$. □

Chapter 6

Ranked Enumeration on General Circuits

A general d-DNNF, which does not forbid gate sharing, cannot guarantee that the continuation of a gate is fixed by the gate itself. A shared gate can be reached along different paths from the output gate of the circuit, and what must be satisfied after it depends on the path; a gate no longer has only one continuation, but several. This chapter recovers ranked enumeration in this setting. We must rebuild the algorithm that supplies each gate with its appropriate continuations for the search along diverse paths.

§ 6.1 Owner and Local Continuation

In a tree-shaped circuit the continuation of a gate is determined by the gate alone, because the gate is reached in a single way. In a DAG, it is different: a gate below a shared gate is reached from the root by several paths, and what remains to be completed above it differs from one path to the other. Therefore the deviation heaps of Chapter 5 can no longer be attached to gates.

The deviations available at a gate nevertheless split in two. Those taken inside the champion traversal below a gate never depend on how the gate was reached, while those

taken outside it do.

Recall the champion trace in Definition 4.2.8. Champion traces are nested: if $x \in T_o$ then $T_x \subseteq T_o$, both being generated by the same rules. By Observation 4.2.2, they also keep the tree-shape, even though C is a DAG. So, every gate of T_o is reached from o by a unique path inside T_o. Sharing is thus invisible from the inside of a champion trace, because sharing originates at OR -gates (Lemma 3.2.1) and a champion trace only retains a single input at each of them. The order of Chapter 5 can therefore be defined similarly on each champion trace separately, giving one order per owner instead of a single global one.

Definition 6.1.1. The *owner set* O is defined inductively: $r \in O$; and if $o \in O$, $g \in T_o \cap \mathsf{OR}$ and $c \in \mathsf{inputs}(g) \setminus \{c^*(g)\}$, then $c \in O$. An *owner* is thus the output gate of C, or the target of a deviation available in the champion trace of an owner.

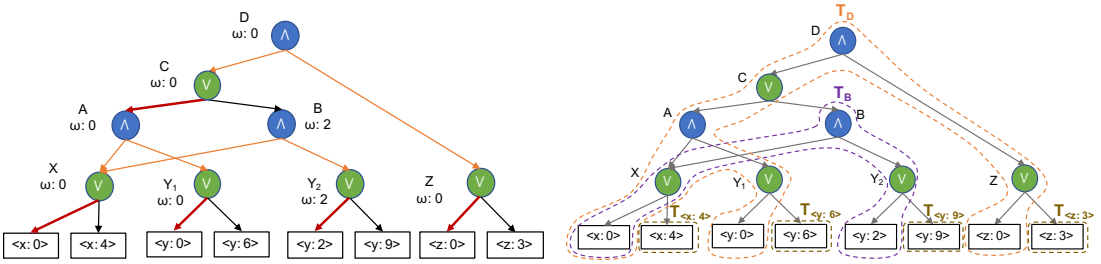


Figure 6.1: (a) shows a general circuit, where gate X is an input for both gate A and gate B.
(b) shows the champion traces for the owners in owner set. Owner set $O = \{D, B, \langle x{:}4\rangle, \langle y{:}6\rangle, \langle y{:}9\rangle, \langle z{:}3\rangle\}$. Each element o in O has one champion trace T_o, e.g., $T_D = \{C, A, \langle x{:}0\rangle, \langle y{:}0\rangle, Z, \langle z{:}0\rangle\}$.

To make it clear, the trace of an assignment is patched together from champion traces. It follows T_r until the first deviation, continues in T_c for the target c of that deviation, and

so on. Each deviation leaves the champion trace it occurs in and enters that of a new owner. Figure 6.1 shows a general circuit, and how owners and their champion traces are organized. It is worth mentioning that the deviations available in T_B come into play only once the deviation (C, B) has been taken, and the same holds for any champion trace.

In Chapter 5 the continuation of a gate collected the gates of the trace coming after the whole subtree below it. The same definition applies *inside a champion trace*, and there it depends on the owner. In Figure 6.1, gate X, inside T_D, has as its continuation the champion traces starting at gate Y_1 and gate Z.

Definition 6.1.2. For an owner o, write $\prec_o := \prec_{o,\beta^*}$ for the depth-first order of Definition 4.2.4 on the champion trace T_o. It records the order in which a traversal starting at o will visit the gates of T_o. The *local continuation* of $g \in T_o$ is

$$\mathsf{Cont}_o(g) \;:=\; \{\, h \in T_o \setminus T_g \;\mid\; g \prec_o h \,\},$$

the gates of T_o visited after all the gates of T_g.

Equivalently, Cont_o is given from top-down by

$$\begin{aligned}
&\mathsf{Cont}_o(o) = \emptyset,\\
&\mathsf{Cont}_o(c_j) = \mathsf{Cont}_o(g) \cup \bigcup_{j'>j} T_{c_{j'}} \quad g \in \mathsf{AND},\\
&\mathsf{Cont}_o(c^*(g)) = \mathsf{Cont}_o(g) \quad g \in \mathsf{OR}
\end{aligned}$$

where $c_1, \ldots, c_m$ are the inputs of an AND-gate g in their fixed order. An AND-gate passes each input the local continuation of the gate together with the champion traces of the later siblings; an OR-gate passes its champion input that continuation unchanged.

Definition 6.1.3. For an OR-gate g, the *champion path* from g is the sequence g, $c^*(g)$, $c^*(c^*(g))$, ... obtained by repeatedly following a champion input. The *optimum exit* $\mathsf{exit}(g)$ is the first non-OR-gate on this path:

$$\mathsf{exit}(g) = \begin{cases} c^*(g) & \text{if } c^*(g) \notin \mathsf{OR}, \\ \mathsf{exit}(c^*(g)) & \text{otherwise.} \end{cases}$$

Observation 6.1.4. For an OR-gate $g \in T_o$ we have $\mathsf{Cont}_o(g) = \mathsf{Cont}_o(\mathsf{exit}(g))$.

A chain of OR-gates contributes nothing to the local continuation, and the continuations along such a chain need to be recorded only once, under the key $\mathsf{exit}(g)$. Distinct chains have distinct optimum exits. Writing $\mathsf{Cont}(o, \mathsf{exit}(g))$ for the heap of the deviations available on $\mathsf{Cont}_o(g)$, Observation 6.1.4 makes this table well-defined; § 6.2 computes it.

§ 6.2 Preprocessing

The preprocessing phase produces two structures. The obstacle they address is that in a general circuit what remains after a gate depends on how the gate was reached, and a gate is reached in many ways; recording all of them is costly. The circuit is covered by champion traces, and the two structures describe each of them separately. The first structure attaches to every gate g the deviations available in T_g; the second attaches to every owner o and every OR-gate $g \in T_o$ the deviations available on $\mathsf{Cont}_o(g)$.

Similarly to Chapter 5, ω^* and c^* are still computed according to the recurrence in § 4.2.2 in one bottom-up pass, in time $O(|C|)$; the values $\mathsf{exit}(g)$ can be obtained in the same pass, each in constant time by reading $\mathsf{exit}(c^*(g))$.

6.2.1 Local Deviation Heap

Definition 6.2.1. For a gate g, the deviations *available in* T_g are the pairs (h, c) with $h \in T_g \cap \mathsf{OR}$ and $c \in \mathsf{inputs}(h) \setminus \{c^*(h)\}$. The *local deviation heap* $H(g)$ contains one entry e per deviation, recording $e.d = (h, c)$ and its cost $e.\delta = dev(h, c)$ as a key. For simplicity, we call it the *local heap* when no ambiguity arises.

The local deviation heaps are computed in a single pass in topological order in Algorithm 3, following the decomposition of T_g at each gate: an AND-gate collects the heaps of its inputs, an OR-gate inherits that of its champion input and adds one entry per remaining non-champion input.

The arrays S_g computed by Algorithm 3 are retained for Algorithm 4, which reads $S_g[i{+}1]$ when descending into the i-th input of an AND-gate g. They amount to one heap root per edge of C, so the additional space is $O(|C|)$; the heaps themselves are persistent and share their nodes with H.

Algorithm 3 Build local deviation heaps

for each gate g in topological order **do**
  if g is an input gate **then**
    $H(g) \leftarrow \bot$
  else if g is an AND-gate with inputs $(c_1, \ldots, c_m)$ **then**
    $S_g[m{+}1] \leftarrow \bot$
    for $i = m$ **down to** 1 **do**
      $S_g[i] \leftarrow \textsc{Meld}(H(c_i),\ S_g[i{+}1])$ ▷ deviations of $T_{c_i}, \ldots, T_{c_m}$
    $H(g) \leftarrow S_g[1]$
  else if g is an OR-gate with inputs $(c_1, \ldots, c_n)$ **then**
    $H(g) \leftarrow H(c^*(g))$
    for all $c \in \mathsf{inputs}(g) \setminus \{c^*(g)\}$ **do**
      $e_c \leftarrow \big(dev(g, c),\ (g, c)\big)$
      $H(g) \leftarrow \textsc{Meld}(H(g),\ \{e_c\})$

Lemma 6.2.2. *Algorithm 3 computes $H(g)$ for every gate g, together with $S_g[i] = H\left(\bigcup_{j \geqslant i} T_{c_j}\right)$ at every* AND *-gate g with inputs $(c_1, \ldots, c_m)$.*

Proof. By induction along the topological order.

If g is an input gate, $T_g = \{g\}$ carries no deviation and $H(g) = \bot$.

If $g \in \mathsf{AND}$, then $T_g = \{g\} \cup \bigcup_j T_{c_j}$, and the T_{c_j} are pairwise disjoint by decomposability, so every deviation available in T_g is available in exactly one T_{c_j}. The backward iteration gives $S_g[i] = \textsc{Meld}(H(c_i), S_g[i+1])$, hence $S_g[i] = H(\bigcup_{j \geqslant i} T_{c_j})$ by induction, and $H(g) = S_g[1]$.

If $g \in \mathsf{OR}$, then $T_g = \{g\} \cup T_{c^*(g)}$, so the deviations available in T_g are those of $T_{c^*(g)}$ together with the pairs (g, c) for $c \neq c^*(g)$. The algorithm melds exactly one entry for each of the latter into $H(c^*(g))$, and they were not present before, since $g \notin T_{c^*(g)}$. □

6.2.2 Local Continuation Table

Definition 6.2.3. For an owner o and an OR-gate $g \in T_o$, set *local continuation table* $\mathsf{Cont}(o, \mathsf{exit}(g)) := H(\mathsf{Cont}_o(g))$, the heap of the deviations available on $\mathsf{Cont}_o(g)$. By Observation 6.1.4 this does not depend on which OR-gate of a champion path the key comes from.

The continuation table is computed by a traversal of each champion trace, from the owner downwards, carrying the local continuation as a parameter. Local continuation grows at AND-gates by the champion traces of the later siblings, and is unchanged along a champion path of OR-gates. New owners are spawned at the OR-gates encountered on the way.

Algorithm 4 Continuation table

```
1:  Built ← {r};   Spawned ← ∅
2:  BuildCont(r, r, ⊥)

3:  procedure BuildCont(o, g, H_A)
4:      if g is an input gate then
5:          return
6:      else if g is an OR-gate then
7:          Cont(o, exit(g)) ← H_A            ▷ one key per champion path of OR-gates
8:          x ← g
9:          while x is an OR-gate and x ∉ Spawned do
10:             Spawned ← Spawned ∪ {x}
11:             for all c ∈ inputs(x) \ {c*(x)} do
12:                 if c ∉ Built then
13:                     Built ← Built ∪ {c}
14:                     BuildCont(c, c, ⊥)     ▷ new owner with empty continuation
15:             x ← c*(x)
16:         BuildCont(o, exit(g), H_A)         ▷ Cont_o is unchanged along the chain
17:     else
18:         Let (c_1, ..., c_m) be the inputs of the AND-gate g
19:         for i = 1 to m do
20:             BuildCont(o, c_i, Meld(H_A, S_g[i+1]))
```

Lemma 6.2.4. *Every call* BuildCont(o, g, H_A) *performed by the algorithm satisfies* $o \in O$, $g \in T_o$ *and* $H_A = H(\mathsf{Cont}_o(g))$.

Proof. By induction on the calls. The initial call is BuildCont$(r, r, \bot)$, and $\mathsf{Cont}_r(r) = \emptyset$. A call BuildCont$(c, c, \bot)$ inside the while loop has $c \in O$ by Definition 6.1.1, since c is a non-champion input of an OR-gate of T_o, and again $\mathsf{Cont}_c(c) = \emptyset$.

The remaining calls keep o and satisfy the recurrences of Definition 6.1.2. At an OR-gate $g \in T_o$ the call is on $\mathsf{exit}(g)$, which is in T_o because the champion path from g is contained in T_o; the parameter is unchanged, which is correct by Observation 6.1.4. At an AND-gate $g \in T_o$ with children $c_1, \ldots, c_m$, all children lie in T_o, and the call on c_i

carries $\textsc{Meld}(H_A, S_g[i+1])$. By Lemma 6.2.2, $S_g[i+1] = H\big(\bigcup_{j>i} T_{c_j}\big)$, so this heap is $H\big(\mathsf{Cont}_o(g) \cup \bigcup_{j>i} T_{c_j}\big) = H(\mathsf{Cont}_o(c_i))$ by the recurrence for AND-gates. The union is disjoint, so each deviation occurs once. □

Lemma 6.2.5. *On termination,* $\mathsf{Cont}(o, \mathsf{exit}(g)) = H(\mathsf{Cont}_o(g))$ *for every owner* o *and every* OR*-gate* $g \in T_o$.

Proof. Every owner is the subject of exactly one call $\textsc{BuildCont}(o, o, \bot)$: *Built* admits each gate at most once, and the gates it admits are exactly those of Definition 6.1.1, because the while loop spawns precisely the non-champion inputs of the OR-gates of T_o. Within such a call the recursion moves downward through T_o, jumping from each OR-gate g directly to $\mathsf{exit}(g)$. By Observation 6.1.4, the OR-gates skipped share the key and the value written at g. Therefore, every OR-gate of T_o is covered, and the value written is $H_A = H(\mathsf{Cont}_o(g))$ by Lemma 6.2.4. □

Observation 6.2.6. The owners spawned at an OR-gate g are the gates of $\mathsf{inputs}(g) \setminus \{c^*(g)\}$, independently of the owner whose champion trace contains g.

Spawned exploits this: a champion path of OR-gates is walked once over the whole run of the algorithm, not once per owner containing it. It affects neither which owners are created nor the values written into the table, both of which are settled by Lemma 6.2.5.

6.2.3 Complexity

Proposition 6.2.7. *Algorithm 3 and Algorithm 4 together run in time and space* $O(|C| \cdot |X| \cdot \log |C|)$.

Proof. A persistent meld for the local deviation heaps takes $O(\log |C|)$ time and creates $O(\log |C|)$ nodes by using leftist heaps.

Algorithm 3 performs one meld per edge, hence $O(|C|)$ in total.

In Algorithm 4, each owner is the subject of one call BuildCont$(o, o, \bot)$, and there are at most $|C|$ owners. The recursion started at o leaves every OR-gate at once for its optimum exit, so it visits only the AND-gates and input gates of T_o; contracting the OR-gates leaves a tree in which every internal gate has at least two children and the leaves are one per variable of $\mathsf{vars}(o)$, since $\mathsf{rec}(o, \beta^*) \in \mathsf{Assign}(\mathsf{vars}(o))$. Each owner thus accounts for $O(|X|)$ melds. *Spawned* guarantees that the while loops walk each OR-gate of C once overall without performing a meld.

In total, $O(|C| \cdot |X|)$ melds are performed, which gives the stated bound. □

Note that S_g and Cont add $O(|C|)$ and $O(|C| \cdot |X|)$ pointers besides the heap operations.

§ 6.3 Continuation Heap

6.3.1 Orders on General Circuits

Definition 6.3.1. For a well-formed deviation set σ, write $\prec_\sigma := \prec_{r,\beta_\sigma}$ for the depth-first order of Definition 4.2.4 on $\mathsf{trace}(\beta_\sigma)$. Similar to Chapter 5, we extend the order to deviations: $d \prec_\sigma d'$ if $\mathsf{source}(d) \prec_\sigma \mathsf{source}(d')$. A well-formed set contains at most one of them, so $\prec_\sigma$ is linear on σ.

$\prec_\sigma$ depends on the deviations taken: taking c at an OR-gate g replaces the whole of $\mathsf{trace}(g, \beta)$ by $\mathsf{trace}(c, \beta')$, so the gates compared below g are not the same before and after.

Observation 6.3.2. Let σ be a well-formed deviation set, o a gate of $\mathsf{trace}(\beta_\sigma)$ and $x \in T_o$ such that no deviation of σ has its source gate on the path from o to x inside T_o,

except possibly at x. That path is also the path from o to x in $\mathsf{trace}(\beta_\sigma)$, and for every $y \in T_o \cap \mathsf{trace}(\beta_\sigma)$ outside T_x,

$$x \prec_o y \quad \text{if and only if} \quad x \prec_\sigma y.$$

Moreover, if no deviation of σ has its source in $\mathsf{Cont}_o(x)$, then $\mathsf{Cont}_o(x) \subseteq \mathsf{trace}(\beta_\sigma)$.

Proof. The choices β^* and β_σ agree at every OR-gate carrying no deviation of σ, hence along the same path from o to x except possibly at x.

Let $y \in \mathsf{Cont}_o(x) \cap \mathsf{trace}(\beta_\sigma)$ and g the last gate common to the paths from o to x and to y. Taking r instead of o as the starting gate merely adds the same path prior to paths from o to x and from o to y, so g and those two inputs leading to x and y are the same, giving the same order. □

Lemma 6.3.3. *Let σ and $\sigma' = \sigma \cup \{d\}$ be well-formed deviation sets such that $\mathsf{source}(d') \prec_{\sigma'} \mathsf{source}(d)$ for every $d' \in \sigma$. Then $\prec_\sigma$ and $\prec_{\sigma'}$ induce the same order on σ, i.e., $d_1 \prec_\sigma d_2$ if and only if $d_1 \prec_{\sigma'} d_2$, for all $d_1, d_2 \in \sigma$.*

Proof. β_σ and $\beta_{\sigma'}$ differ only at $\mathsf{source}(d)$, so the two traces agree outside the subtree rooted at $\mathsf{source}(d)$. Let $d' \in \sigma$, then $\mathsf{source}(d')$ cannot be in that subtree: otherwise it would be reachable from $\mathsf{source}(d)$ inside the subtree then come after it, contrary to the hypothesis. □

6.3.2 Continuation Heap

Recall the question raised at the end of Chapter 4: after a deviation is committed, which deviations become available? In Chapter 5 the answer came in two parts. Committing (g, c) opened the subtree (subcircuit) of c, and left the gates after g still to be visited; the

heap of the first was built during preprocessing, and the parameter H_{cont} in Algorithm 1 carried the second.

The same two parts appear here, but only the first can be attached to a gate directly. The gates after g depend on how g was reached from r, and a gate is reached in more than one way. However, fixing one trace restores the answer and the continuation can be assembled during the enumeration. Such a trace follows champion inputs except at the gates carrying a deviation, so it is patched together from champion traces, each of them left unfinished where a deviation was taken. The deviations available are then those of the champion trace just entered, together with those of the unfinished parts, one part per champion trace involved. Which champion traces are still unfinished can be read off from σ: they are the ones entered at a gate on the path to the last deviation.

Consider the circuit of Figure 6.2 and the deviation set $\sigma = \{(C, B), (X, \langle x{:}4\rangle)\}$. Its trace leaves T_D at C for T_B, then leaves T_B at X for $T_{\langle x:4\rangle}$, which is a single input gate. Two champion traces are thus left unfinished, each at the source of the deviation that leaves it: T_D at C, and T_B at X.

$$\mathsf{Cont}_D(C) = \{Z, \langle z{:}0\rangle\}, \qquad \mathsf{Cont}_B(X) = \{Y_2, \langle y{:}2\rangle\}.$$

Each offers one deviation, $(Z, \langle z{:}3\rangle)$ of cost 3 and $(Y_2, \langle y{:}9\rangle)$ of cost 7, held in the table entries $\mathsf{Cont}(D, \mathsf{exit}(C))$ and $\mathsf{Cont}(B, \mathsf{exit}(X))$. The enumeration must hold both at once and be able to produce the cheaper one. Note that it comes from the outer of the two traces: the unfinished parts cannot simply be resumed innermost first; they must be ordered by cost. This is the role of the continuation heap.

Lemma 6.3.4. *Let σ be a non-empty well-formed deviation set, d its greatest element for $\prec_\sigma$, and P the path from r to* $\mathsf{source}(d)$ *in* $\mathsf{trace}(\beta_\sigma)$. *Call $d' \in \sigma$* active *if* $\mathsf{source}(d')$

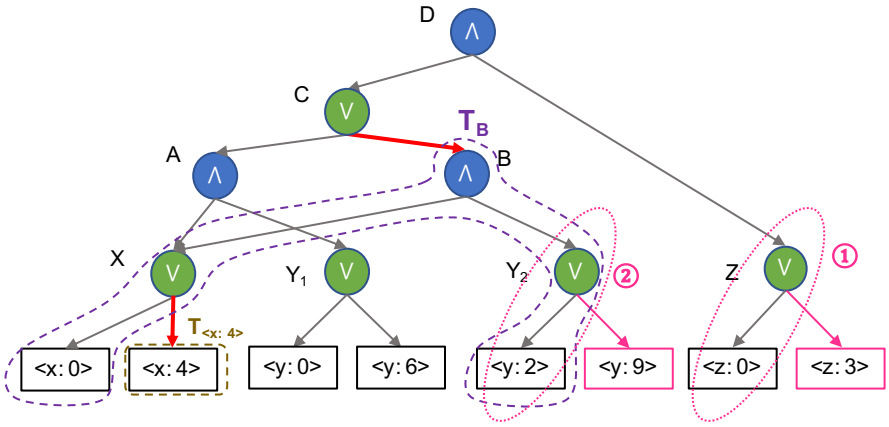


Figure 6.2: The circuit of Figure 6.1a, with only the champion traces T_B and $T_{\langle x:4\rangle}$ marked, for clarity. Red arrows denote the deviations taken. The regions ① and ② are $\mathsf{Cont}_D(C)$ and $\mathsf{Cont}_B(X)$. Once the deviation set $\{(C, B), (X, \langle x{:}4\rangle)\}$ has been taken, the deviations $(Z, \langle z{:}3\rangle)$ and $(Y_2, \langle y{:}9\rangle)$ are the ones still available in these two regions.

lies on P, and let $d'_1, \ldots, d'_k = d$ be the active deviations in the order in which their source gates occur along P. Set $o_1 = r$ and $o_{i+1} = \mathsf{target}(d'_i)$, then the gates of $\mathsf{trace}(\beta_\sigma)$ *coming after* $\mathsf{source}(d)$ *are*

$$T_{o_{k+1}} \uplus \mathsf{Cont}_{o_k}(\mathsf{source}(d'_k)) \uplus \cdots \uplus \mathsf{Cont}_{o_1}(\mathsf{source}(d'_1)),$$

and no deviation of σ has its gate in any of these sets.

Proof. As d is the greatest element of σ, a gate comes after $\mathsf{source}(d)$ if and only if it comes after the source of every deviation of σ. Throughout, note that $\mathsf{Cont}_o(x)$ consists of gates in branches of AND-gates *above* x other than the one containing x, so by decomposability it is disjoint from the subcircuit at x.

(1) Each $\mathsf{source}(d'_i)$ *is reached from o_i inside T_{o_i}.* The deviations of σ whose source lies on P are exactly the active ones, so P carries no deviation strictly between o_i and $\mathsf{source}(d'_i)$. Along that portion of P, β_σ takes the champion input at every OR-gate,

hence the portion lies in T_{o_i}.

(2) $\mathsf{Cont}_{o_i}(\mathsf{source}(d'_i))$ *does not contain any source of a deviation.* Suppose some $d'' \in \sigma$ occurred in $\mathsf{Cont}_{o_i}(\mathsf{source}(d'_i))$. Well-formedness gives $\mathsf{source}(d'') \in \mathsf{trace}(\beta_\sigma)$, then (1) and Observation 6.3.2 yield $\mathsf{source}(d'_i) \prec_\sigma \mathsf{source}(d'')$. $\mathsf{source}(d'')$ is outside $T_{\mathsf{source}(d'_i)}$, and hence also outside $\mathsf{trace}(\mathsf{source}(d'_i), \beta_\sigma)$. As $\mathsf{source}(d)$ lies in $\mathsf{trace}(\mathsf{source}(d'_i), \beta_\sigma)$, $\mathsf{source}(d) \prec_\sigma \mathsf{source}(d'')$, contradicting the maximality of d. Therefore, the gates of $T_{o_{k+1}}$ carry no deviation coming after $\mathsf{source}(d)$.

From right to left. The gates of $T_{o_{k+1}}$ are those of the champion trace entered at $\mathsf{source}(d)$, and hence come after $\mathsf{source}(d)$. For $i \leqslant k$, the gates of $\mathsf{Cont}_{o_i}(\mathsf{source}(d'_i))$ lie in $\mathsf{trace}(\beta_\sigma)$ and come after $\mathsf{source}(d'_i)$ by (2) and Observation 6.3.2; they are outside $\mathsf{trace}(\mathsf{source}(d'_i), \beta_\sigma)$, and $\mathsf{source}(d)$ lies in $\mathsf{trace}(\mathsf{source}(d'_i), \beta_\sigma)$, so they come after $\mathsf{source}(d)$.

From left to right. Let $y \in \mathsf{trace}(\beta_\sigma)$ come after $\mathsf{source}(d)$, and let g be the last gate common to P and to the path from r to y.

If $g = \mathsf{source}(d)$ then y lies in $\mathsf{trace}(\mathsf{source}(d'_i), \beta_\sigma)$. No deviation of σ occurs there, all its gates coming after $\mathsf{source}(d)$, so that subtree is $T_{\mathsf{target}(d)} = T_{o_{k+1}}$.

Otherwise g lies strictly before $\mathsf{source}(d)$ on P, and the two paths leave g by distinct inputs. The input leading to y follows the one leading to $\mathsf{source}(d)$, since $\mathsf{source}(d) \prec_\sigma y$. Let i be the index with g on the portion of P from o_i to $\mathsf{source}(d'_i)$; then $g \in T_{o_i}$ by (1). The path from g to y carries no deviation of σ by (2). Therefore that path takes the champion input at every OR-gate, so $y \in T_{o_i}$; and y comes after $\mathsf{source}(d'_i)$ and must be outside $T_{\mathsf{source}(d'_i)}$, i.e., $y \in \mathsf{Cont}_{o_i}(\mathsf{source}(d'_i))$.

Disjointness is obvious by definition and decomposability. □

Definition 6.3.5. A *continuation entry* $\gamma = \mathsf{ContEntry}(L, o, R)$ records a local heap

γ.heap = L taken from the table Cont, the owner γ.owner = o it belongs to, and a heap γ.rest = R of continuation entries. Its key is $\gamma.\delta = L.\delta$, the least cost in L, i.e., the cost of the root deviation of L. A *continuation heap* $\mathcal{H}$ is a heap of continuation entries.

An entry is created when a deviation is committed at a gate x of T_o: the traversal leaves T_o for the champion trace of the target, and the entry keeps what is left of T_o, namely $\mathsf{Cont}(o, \mathsf{exit}(x))$. Its third field holds the entries of the traces left even earlier. Melding onto that field rather than onto $\mathcal{H}$ therefore continues from a heap containing neither the entry itself nor any created after it.

§ 6.4 Enumeration

A state of the enumeration must record where in the circuit the next deviation may be taken. In Chapter 5 a single heap entry sufficed, because the deviation heap attached to a gate already covered everything that remained. Here the two halves of Definition 6.1.2 are stored apart: the champion trace of the current owner is covered by the tables of § 6.2, while the champion traces the assignment has already left behind must be carried along. A state therefore holds, besides its deviation record, a heap of *entries*, one for each such trace.

6.4.1 State and Enumeration

Definition 6.4.1. A *state* is a tuple $(\pi, u, o, \mathcal{H}, \gamma, s)$ where s is a deviation record, π is the total cost of $\sigma(s)$, u is an entry recording the last deviation of $\sigma(s)$, o is the owner whose tables u was taken from, $\mathcal{H}$ is a continuation heap, and γ is the entry of $\mathcal{H}$ that u was taken from, or $\perp$ if u was taken from $H(o)$.

Algorithm 5 Ranked enumeration on a general circuit

```
1:  procedure Enumerate(r)
2:      output (ω*(r), ⊥)                          ▷ champion; empty deviation record
3:      Q ← ∅
4:      if H(r) ≠ ⊥ then
5:          u ← FindMin(H(r))
6:          Push(Q, (u.δ, u, r, ⊥, ⊥, ⟨⊥, u.d⟩))
7:      while Q ≠ ∅ do
8:          (π, u, o, H, γ, s) ← PopMin(Q)
9:          output (ω*(r) + π, s)
        // (a) replace the last deviation within the same local heap
10:         for all u' ∈ {ℓ(u), r(u)} \ {⊥} do
11:             Push(Q, (π − u.δ + u'.δ, u', o, H, γ, ⟨s.prev, u'.d⟩))
        // (b) replace it by one taken from another continuation entry
12:         if γ ≠ ⊥ and u = γ.node then
13:             for all γ' ∈ {ℓ(γ), r(γ)} \ {⊥} do
14:                 w ← γ'.node
15:                 Push(Q, (π − u.δ + γ'.δ, w, γ'.owner, H, γ', ⟨s.prev, w.d⟩))
        // commit u: keep only the champion traces outside the current one
16:         R ← γ.rest if γ ≠ ⊥ else H
17:         L ← Cont(o, exit(source(u.d)))
18:         if L ≠ ⊥ then
19:             H' ← Meld(R, ContEntry(L, o, R))
20:         else
21:             H' ← R
        // (c) continue in a champion trace already left behind
22:         if H' ≠ ⊥ then
23:             γ* ← FindMin(H');   w ← γ*.node
24:             Push(Q, (π + γ*.δ, w, γ*.owner, H', γ*, ⟨s, w.d⟩))
        // (d) continue in the champion trace of the target
25:         if H(target(u.d)) ≠ ⊥ then
26:             v ← FindMin(H(target(u.d)))
27:             Push(Q, (π + v.δ, v, target(u.d), H', ⊥, ⟨s, v.d⟩))
```

Algorithm 5 outputs the champion first, with empty deviation record and score $\omega^*(r)$; thereafter a priority queue Q keyed by π drives the search.

Replacing the last deviation, (a) and (b). The deviations that may take the place of the last one form the union of the local heap u was taken from with the local heaps of the other entries of $\mathcal{H}$. Move (a) explores that union within one local heap, following its heap children; move (b) moves to another entry, whose least deviation is γ'.node. Both keep the length of $\sigma(s)$. The guard $u = \gamma$.node restricts (b) back to the least deviation of the current entry; without it, an entry could be reached both directly and after a move (a), and the same deviation set would be produced repeatedly.

Committing the last deviation, (c) and (d). Committing u keeps $u.d$ and appends a further deviation. The traversal now enters the champion trace of $\mathsf{target}(u.d)$, whose deviations are held in $H(\mathsf{target}(u.d))$; move (d) takes the least of them. What remains of the trace being left, namely $\mathsf{Cont}(o, \mathsf{exit}(\mathsf{source}(u.d)))$, becomes a new entry, and move (c) takes the least deviation offered by $\mathcal{H}'$. The new entry is melded onto $R = \gamma$.rest rather than onto $\mathcal{H}$. The entry u was taken from is thereby replaced: what it still offered after $\mathsf{source}(u.d)$ has passed to the new entry. The entries created after it disappear altogether, since the deviations they hold come before $\mathsf{source}(u.d)$.

Lemma 6.4.2. *Let $(\pi, u, o, \mathcal{H}, \gamma, s)$ be a state generated by Algorithm 5 and let $\mathcal{H}'$ be the heap computed at its commit step. With the notation of Lemma 6.3.4 applied to $\sigma(s)$, the entries of $\mathcal{H}'$ are in bijection with the indices $i \leqslant k$ such that $\mathsf{Cont}_{o_i}(\mathsf{source}(d'_i))$ carries a deviation; the entry of index i has γ.heap $= \mathsf{Cont}\big(o_i, \mathsf{exit}(\mathsf{source}(d'_i))\big)$ and γ.owner $= o_i$, and its field γ.rest holds the entries of the indices below i.*

Proof. By induction along the generation path. In the initial state $\sigma(s)$ is a singleton, so $k = 1$ and $o_1 = r$; the commit step melds the entry of $\mathsf{Cont}(r, \mathsf{exit}(\mathsf{source}(d)))$ onto $\mathcal{H} = \bot$, which is the claim.

For the step, let $\sigma = \sigma(s)$ with greatest element $d = u.d$, and assume the claim for the state at hand.

Move (d) appends a deviation of $T_{\mathsf{target}(d)}$. Its gate lies below $\mathsf{source}(d)$, so the path P of the new set extends the old one and its active deviations are those of σ together with d. The successor carries $\mathcal{H}'$, which holds the indices up to k by assumption, and its own commit step melds onto it the entry of the index $k + 1$, as claimed.

Move (c) appends the least deviation of an entry γ^* of $\mathcal{H}'$, of index j say. Its gate lies in $\mathsf{Cont}_{o_j}(\mathsf{source}(d'_j))$, hence outside $T_{\mathsf{source}(d'_j)}$, so the new path leaves the old one at $\mathsf{source}(d'_j)$: the active deviations of the new set are $d'_1, \ldots, d'_{j-1}$ together with the new one, those of index j and above being no longer active. The successor carries $\gamma = \gamma^*$, whose field rest holds the indices below j by assumption, and its commit step melds onto that, as claimed.

Moves (a) and (b) replace d without committing it, so $\mathcal{H}$ is unchanged. By Lemma 6.3.4 applied to $\sigma \setminus \{d\}$, the new deviation is drawn from the same union as d: move (a) from the same local heap, move (b) from another entry γ'. The active deviations of the new set are those of the index the new deviation belongs to, together with those of smaller index, which are held by the field rest of the entry carried by the successor; that entry is γ for move (a) and γ' for move (b). □

6.4.2 Correctness

Both groups of moves rest on the same question: once part of a deviation set has been fixed, which deviations may still be added to it? Two conditions are required. A deviation must keep the set well-formed, and it must come after those already fixed; otherwise, the same set would be assembled in several orders. The next lemma states

that the deviations satisfying both are exactly those the algorithm has.

Lemma 6.4.3. *Let $(\pi, u, o, \mathcal{H}, \gamma, s)$ be a state generated by Algorithm 5 and let $\mathcal{H}'$ be the heap computed at its commit step. Write $\sigma = \sigma(s)$ and $\sigma_d = \sigma \cup \{d\}$. The following are equivalent:*

1. *σ_d is well-formed and $d' \prec_{\sigma_d} d$ for every $d' \in \sigma$;*
2. *d occurs in $H(\mathsf{target}(u.d))$, or in the local heap of an entry of $\mathcal{H}'$.*

Moreover the two parts of (2) are disjoint, as are the local heaps of distinct entries of $\mathcal{H}'$.

Proof. Assume (1). By Lemma 6.3.3 the orders $\prec_\sigma$ and $\prec_{\sigma_d}$ agree on σ_d, so $\mathsf{source}(d)$ comes after every gate of σ in $\prec_\sigma$, in particular after $\mathsf{source}(u.d)$. By Lemma 6.3.4, $\mathsf{source}(d)$ is then an OR-gate of $T_{o_{k+1}}$ or of some $\mathsf{Cont}_{o_i}(\mathsf{source}(d'_i))$. In the first case d occurs in $H(\mathsf{target}(u.d))$ by Definition 6.2.1, since $o_{k+1} = \mathsf{target}(u.d)$. In the second, $\mathsf{Cont}_{o_i}(\mathsf{source}(d'_i))$ carries a deviation, so it has an entry in $\mathcal{H}'$ by Lemma 6.4.2, whose local heap is $\mathsf{Cont}\big(o_i, \mathsf{exit}(\mathsf{source}(d'_i))\big) = H(\mathsf{Cont}_{o_i}(\mathsf{source}(d'_i)))$ by Lemma 6.2.5 and contains d.

Conversely, a deviation occurring in one of those heaps has its gate in the corresponding set of Lemma 6.3.4, hence in $\mathsf{trace}(\beta_\sigma)$ and after $\mathsf{source}(u.d)$. The set σ_d is then well-formed, the choices above $\mathsf{source}(d)$ being those recorded in σ, and the orders agree on σ_d by Lemma 6.3.3, so $d' \prec_{\sigma_d} d$ for every $d' \in \sigma$.

Disjointness is that of Lemma 6.3.4, transported by Lemma 6.4.2. □

Lemma 6.4.4. *The deviations recorded in a state occur in increasing $\prec_{\sigma(s)}$-order. A sequence of deviation records is therefore determined by the set of deviations it records, and we write $\sigma(s)$ for either without distinction.*

Proof. By induction along the generation path.

Moves (c) and (d) append a deviation supplied by Lemma 6.4.3, so one coming after every deviation recorded in the order $\prec_{\sigma'}$ of the extended set. By Lemma 6.3.3 the earlier deviations keep the order there, so the extended sequence is increasing for $\prec_{\sigma'}$.

Moves (a) and (b) replace the last deviation by another one available after $\sigma(s) \setminus \{u.d\}$; the resulting sequence is again increasing for the order of its own set.

A linear order has a unique increasing enumeration, so the record is determined by the set. □

Lemma 6.4.5. *For every state generated by Algorithm 5, $\sigma(s)$ is a well-formed deviation set and $\pi = \sum_{d \in \sigma(s)} dev(d)$.*

Proof. The initial state records a deviation d from the entry of $H(r)$, where $\mathsf{source}(d)$ must be in $T_r = \mathsf{trace}(\beta^*)$ and $\pi = dev(d)$. Lemma 6.4.3 gives the well-formedness by induction along the generation path. The costs are from the recorded deviations, added at moves (c) and (d) and replaced by moves (a) and (b). □

Lemma 6.4.6. *Every non-empty well-formed deviation set is $\sigma(s)$ for exactly one state generated by Algorithm 5.*

Proof. Let D be non-empty and well-formed, with elements $d_1 \prec_D \cdots \prec_D d_n$.

Existence. We show by induction on i that some generated state has $\sigma(s) = \{d_1, \ldots, d_i\}$. As d_1 is $\prec_D$-least, no deviation of D lies above $\mathsf{source}(d_1)$, so $\mathsf{source}(d_1) \in T_r$ and d_1 is an entry of $H(r)$; the initial state records the least entry of $H(r)$, and move (a) exchanges it for any other entry of that heap. For the step, take a state with $\sigma(s) = \{d_1, \ldots, d_i\}$; by Lemma 6.4.3 the deviation d_{i+1} occurs in $H(\mathsf{target}(d_i))$ or in the local heap of an entry of $\mathcal{H}'$. In the first case one move (d) followed by moves (a)

reaches it; in the second, one move (c) reaches the least entry of $\mathcal{H}'$, moves (b) reach the entry containing d_{i+1}, and moves (a) reach d_{i+1} inside it.

Uniqueness. Let s be generated with $\sigma(s) = D$. By Lemma 6.4.4 the deviations were recorded in the order $d_1, \ldots, d_n$, so the generation performed exactly $n-1$ commits, the i-th committing d_i. Each commit is followed by a move (c) or a move (d), and not both: by Lemma 6.4.3 the deviation d_{i+1} lies either in $H(\mathsf{target}(d_i))$ or in the local heap of an entry of $\mathcal{H}'$, and these are disjoint. Which move is used is therefore determined by d_{i+1}. The moves between two consecutive commits are then determined as well. After a move (d) only moves (a) may follow, since $\gamma = \bot$; after a move (c) the guard allows moves (b) only before any move (a), so the segment consists of moves (b) followed by moves (a). Moves (b) travel from the root of $\mathcal{H}'$ to the entry holding d_{i+1}, and moves (a) from the root of a local heap to d_{i+1} itself; both heaps are binary trees, so each path is unique. The generation is therefore determined by D. □

Lemma 6.4.7. *The cost of a state is at least the cost of the state it was generated from.*

Proof. Move (a) changes the cost by $-u.\delta + u'.\delta \geqslant 0$, since u' is a heap child of u. Move (b) changes the value by $-u.\delta + \gamma'.\delta$, where γ' is a heap child of γ. This change is nonnegative: the guard guarantees $u = \gamma.\mathrm{node}$ and the heap property ensures $\gamma.\delta \leqslant \gamma'.\delta$. Moves (c) and (d) commit a new deviation and change π by adding a nonnegative cost. □

Theorem 6.4.8. *Algorithm 5 outputs every assignment of $[\![C]\!]$ exactly once, in nondecreasing order of score.*

Proof. The champion $\mathsf{rec}(\beta^*)$ is output first, with score $\omega^*(r)$. By Lemmas 4.3.4 and 6.4.6, the states generated by Algorithm 5 are in bijection with $[\![C]\!] \setminus \{\mathsf{rec}(\beta^*)\}$,

and by Lemma 6.4.5 the state of τ carries $\pi = \omega(\tau) - \omega^*(r)$. Every generated state is pushed, and every pushed state is eventually popped, so every assignment is output exactly once with its score.

Lemma 6.4.7 and the same argument as in Theorem 5.4.6 establish that the outputs are in nondecreasing order of score. □

6.4.3 Complexity

Proposition 6.4.9. *Every continuation heap at every point during the execution in Algorithm 5 holds at most $|X|$ continuation entries.*

Proof. By Lemma 6.4.2, the entries correspond to indices i for which $\mathsf{Cont}_{o_i}(\mathsf{source}(d_i'))$ carries a deviation. These sets and the variable sets they carry are both pairwise disjoint. So there are at most $|X|$ entries. □

Theorem 6.4.10. *After the preprocessing of Proposition 6.2.7, Algorithm 5 outputs the k-th assignment with delay $O(\log k + \log |X|)$ in the implicit representation.*

Proof. Each iteration performs one PopMin and at most six Push operations, so after k outputs Q holds $O(k)$ states and each queue operation costs $O(\log k)$. The commit step performs one Meld on a continuation heap, which holds at most $|X|$ entries by Proposition 6.4.9 and costs $O(\log |X|)$. All remaining work is constant: the table lookup $\mathsf{Cont}(o, \mathsf{exit}(\mathsf{source}(u.d)))$, the reads of heap roots and children, the cost updates, and the extension of a deviation record. □

Chapter 7

Conclusion

§ 7.1 Summary

This thesis studied ranked enumeration of the satisfying assignments over smooth multivalued d-DNNF circuits under a sum ordering, measuring delay in terms of both the circuit size and the number of variables.

The starting point of the algorithm was Eppstein's observation that the k shortest paths of a graph need not be stored as paths: each can be recorded as a sequence of deviations from a shortest one. Chapter 4 transferred this idea to circuits. An answer is represented implicitly by a set of deviations, and the enumeration manipulates these implicit representations rather than the explicit assignments.

Chapter 5 developed the tree-shaped case. Deviation heaps are built bottom-up, and enumeration traverses the deviation heaps connected by cross pointers. The resulting delay is logarithmic in the number of answers already produced and independent of the number of variables, after preprocessing in $O(|C| \log |C|)$ time.

Chapter 6 extended the framework to general circuits, where a gate can be reached

along more than one path from the output gate. Deviations and continuations are recorded locally in terms of owners and champion traces; the price is an additional structure, the continuation heap, manipulated during enumeration. The delay becomes $O(\log |X| + \log k)$ and preprocessing is $O(|C| \cdot |X| \log |C|)$.

These bounds hold for implicit output. An application that materializes every answer in full regains the dependence on $|C|$ as a traversal of the circuit is needed; the gain holds when only a few of the k answers need to be materialized.

§ 7.2 Future work

Several directions remain. The first is a lower bound for the general case: whether the additional logarithmic term is inherent or a consequence of the continuation mechanism used here. The second is to widen the ranking. Only the sum ordering is treated in this algorithm, and whether the results extend to other orderings is open. Finally, the results are stated for circuits rather than for the queries that produce them. Applications such as monadic second-order queries with free second-order variables, where the number of variables grows with the data, are reached only through a compilation step whose cost is not accounted for here. How the two costs combine is left open.

Bibliography

[1] Antoine Amarilli, Pierre Bourhis, Florent Capelli, and Mikaël Monet. Ranked Enumeration for MSO on Trees via Knowledge Compilation, January 2024. arXiv:2310.00731, doi:10.48550/arXiv.2310.00731.

[2] Antoine Amarilli, Pierre Bourhis, Louis Jachiet, and Stefan Mengel. A Circuit-Based Approach to Efficient Enumeration. In Ioannis Chatzigiannakis, Piotr Indyk, Fabian Kuhn, and Anca Muscholl, editors, *44th International Colloquium on Automata, Languages, and Programming (ICALP 2017)*, volume 80 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 111:1–111:15, Dagstuhl, Germany, 2017. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi:10.4230/LIPIcs.ICALP.2017.111.

[3] Antoine Amarilli, Pierre Bourhis, and Pierre Senellart. Provenance Circuits for Trees and Treelike Instances. In Magnús M. Halldórsson, Kazuo Iwama, Naoki Kobayashi, and Bettina Speckmann, editors, *Automata, Languages, and Programming*, pages 56–68, Berlin, Heidelberg, 2015. Springer. doi:10.1007/978-3-662-47666-6_5.

[4] Antoine Amarilli and Florent Capelli. Tractable Circuits in Database Theory, August 2024. arXiv:2407.01127, doi:10.48550/arXiv.2407.01127.

[5] Antoine Amarilli, Claire David, Nadime Francis, Victor Marsault, Mikaël Monet, and Yann Strozecki. Gray Codes with Constant Delay and Constant Auxiliary Space. In Sayan Bhattacharya, Danupon Nanongkai, Michael Benedikt, and Gabriele Puppis, editors, *53rd International Colloquium on Automata, Languages, and Programming (ICALP 2026)*, volume 374 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 160:1–160:22, Dagstuhl, Germany, 2026. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi:10.4230/LIPIcs.ICALP.2026.160.

[6] Antoine Amarilli and Mikaël Monet. Enumerating Regular Languages with Bounded Delay. In Petra Berenbrink, Patricia Bouyer, Anuj Dawar, and Mamadou Moustapha Kanté, editors, *40th International Symposium on Theoretical Aspects of Computer Science (STACS 2023)*, volume 254 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 8:1–8:18, Dagstuhl, Germany, 2023.

Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi:10.4230/LIPIcs.STACS.2023.8.

[7] Albert Atserias, Martin Grohe, and Dániel Marx. Size Bounds and Query Plans for Relational Joins. *SIAM Journal on Computing*, 42(4):1737–1767, January 2013. doi:10.1137/110859440.

[8] Guillaume Bagan. MSO Queries on Tree Decomposable Structures Are Computable with Linear Delay. In Zoltán Ésik, editor, *Computer Science Logic*, pages 167–181, Berlin, Heidelberg, 2006. Springer. doi:10.1007/11874683_11.

[9] Guillaume Bagan, Arnaud Durand, and Etienne Grandjean. On Acyclic Conjunctive Queries and Constant Delay Enumeration. In Jacques Duparc and Thomas A. Henzinger, editors, *Computer Science Logic*, pages 208–222, Berlin, Heidelberg, 2007. Springer. doi:10.1007/978-3-540-74915-8_18.

[10] Christoph Berkholz, Jens Keppeler, and Nicole Schweikardt. Answering Conjunctive Queries under Updates. In *Proceedings of the 36th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*, PODS '17, pages 303–318, New York, NY, USA, May 2017. Association for Computing Machinery. doi:10.1145/3034786.3034789.

[11] Christoph Berkholz and Maximilian Merz. Probabilistic Databases under Updates: Boolean Query Evaluation and Ranked Enumeration. In *Proceedings of the 40th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*, PODS'21, pages 402–415, New York, NY, USA, June 2021. Association for Computing Machinery. doi:10.1145/3452021.3458326.

[12] Pierre Bourhis, Alejandro Grez, Louis Jachiet, and Cristian Riveros. Ranked Enumeration of MSO Logic on Words. *LIPIcs, Volume 186, ICDT 2021*, 186:20:1–20:19, 2021. doi:10.4230/LIPICS.ICDT.2021.20.

[13] Johann Brault-Baron. De la pertinence de l'énumération: complexité en logiques propositionnelle et du premier ordre.

[14] Johann Brault-Baron. A Negative Conjunctive Query is Easy if and only if it is Beta-Acyclic. In Patrick Cégielski and Arnaud Durand, editors, *Computer Science Logic (CSL'12) - 26th International Workshop/21st Annual Conference of the EACSL*, volume 16 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 137–151, Dagstuhl, Germany, 2012. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi:10.4230/LIPIcs.CSL.2012.137.

[15] Karl Bringmann, Nofar Carmeli, and Stefan Mengel. Tight Fine-Grained Bounds for Direct Access on Join Queries. In *Proceedings of the 41st ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*, PODS '22, pages 427–436, New York, NY, USA, June 2022. Association for Computing Machinery. doi:10.1145/3517804.3526234.

[16] Florent Capelli and Oliver Irwin. Direct Access for Conjunctive Queries with Negations. In Graham Cormode and Michael Shekelyan, editors, *27th International Conference on Database Theory (ICDT 2024)*, volume 290 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 13:1–13:20, Dagstuhl, Germany, 2024. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi:10.4230/LIPIcs.ICDT.2024.13.

[17] Nofar Carmeli and Markus Kröll. On the Enumeration Complexity of Unions of Conjunctive Queries. *ACM Transactions on Database Systems (TODS)*, 46(2):5:1–5:41, May 2021. doi:10.1145/3450263.

[18] Nofar Carmeli, Nikolaos Tziavelis, Wolfgang Gatterbauer, Benny Kimelfeld, and Mirek Riedewald. Tractable Orders for Direct Access to Ranked Answers of Conjunctive Queries. *ACM Trans. Database Syst.*, 48(1):1:1–1:45, March 2023. doi:10.1145/3578517.

[19] Edith Cohen. Sampling Big Ideas in Query Optimization. In *Proceedings of the 42nd ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*, PODS '23, pages 361–371, New York, NY, USA, June 2023. Association for Computing Machinery. doi:10.1145/3584372.3589935.

[20] Clark Allan Crane. *Linear lists and priority queues as balanced binary trees*. PhD thesis, Stanford University, Stanford, CA, USA, 1972. AAI7220697.

[21] A. Darwiche and P. Marquis. A Knowledge Compilation Map. *Journal of Artificial Intelligence Research*, 17:229–264, September 2002. arXiv:1106.1819, doi:10.1613/jair.989.

[22] Shaleen Deep, Xiao Hu, and Paraschos Koutris. Ranked enumeration of join queries with projections. *Proceedings of the VLDB Endowment*, 15(5):1024–1037, January 2022. doi:10.14778/3510397.3510401.

[23] Shaleen Deep and Paraschos Koutris. Ranked Enumeration of Conjunctive Query Results. *LIPIcs, Volume 186, ICDT 2021*, 186:5:1–5:19, 2021. doi:10.4230/LIPICS.ICDT.2021.5.

[24] Arnaud Durand. Fine-Grained Complexity Analysis of Queries: From Decision to Counting and Enumeration. In *Proceedings of the 39th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*, PODS'20, pages 331–346, New York, NY, USA, June 2020. Association for Computing Machinery. doi:10.1145/3375395.3389130.

[25] Arnaud Durand and Etienne Grandjean. First-order queries on structures of bounded degree are computable with constant delay. *ACM Transactions on Computational Logic (TOCL)*, 8(4):21–es, August 2007. doi:10.1145/1276920.1276923.

[26] David Eppstein. Finding the k Shortest Paths.

[27] Abhay Jha and Dan Suciu. Knowledge Compilation Meets Database Theory: Compiling Queries to Decision Diagrams. *Theory of Computing Systems*, 52(3):403–440, April 2013. doi:10.1007/s00224-012-9392-5.

[28] Wojciech Kazana and Luc Segoufin. Enumeration of monadic second-order queries on trees. *ACM Transactions on Computational Logic (TOCL)*, 14(4):25:1–25:12, November 2013. doi:10.1145/2528928.

[29] Donald E. Knuth. *The Art of Computer Programming, Volume 3: Sorting and Searching*. Addison-Wesley, 2nd edition, 1998. Leftist trees: Section 5.2.3.

[30] Martín Muñoz and Cristian Riveros. Streaming Enumeration on Nested Documents. In Dan Olteanu and Nils Vortmeier, editors, *25th International Conference on Database Theory (ICDT 2022)*, volume 220 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 19:1–19:18, Dagstuhl, Germany, 2022. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi:10.4230/LIPIcs.ICDT.2022.19.

[31] Chris Okasaki. *Purely Functional Data Structures*. Cambridge University Press, 1998. doi:10.1017/CBO9780511530104.

[32] Dan Olteanu and Jiewen Huang. Using OBDDs for Efficient Query Evaluation on Probabilistic Databases. In Sergio Greco and Thomas Lukasiewicz, editors, *Scalable Uncertainty Management*, pages 326–340, Berlin, Heidelberg, 2008. Springer. doi:10.1007/978-3-540-87993-0_26.

[33] Dan Olteanu and Jakub Závodný. Factorised representations of query results: Size bounds and readability. In *Proceedings of the 15th International Conference on Database Theory*, ICDT '12, pages 285–298, New York, NY, USA, March 2012. Association for Computing Machinery. doi:10.1145/2274576.2274607.

[34] Dan Olteanu and Jakub Závodný. Size Bounds for Factorised Representations of Query Results. *ACM Transactions on Database Systems*, 40(1):1–44, March 2015. doi:10.1145/2656335.

[35] Pierre Senellart, Louis Jachiet, Silviu Maniu, and Yann Ramusat. ProvSQL: Provenance and probability management in postgreSQL. *Proceedings of the VLDB Endowment*, 11(12):2034–2037, August 2018. doi:10.14778/3229863.3236253.

[36] Nikolaos Tziavelis, Deepak Ajwani, Wolfgang Gatterbauer, Mirek Riedewald, and Xiaofeng Yang. Optimal algorithms for ranked enumeration of answers to full conjunctive queries. *Proceedings of the VLDB Endowment*, 13(9):1582–1597, May 2020. doi:10.14778/3397230.3397250.

[37] Nikolaos Tziavelis, Wolfgang Gatterbauer, and Mirek Riedewald. Beyond equi-joins: Ranking, enumeration and factorization. *Proceedings of the VLDB Endowment*, 14(11):2599–2612, July 2021. doi:10.14778/3476249.3476306.

[38] Nikolaos Tziavelis, Wolfgang Gatterbauer, and Mirek Riedewald. Any-k Algorithms for Enumerating Ranked Answers to Conjunctive Queries. *ACM Transactions on Database Systems*, 51(1):1:1–1:47, September 2025. doi:10.1145/3734517.