Algorithmica (2000) 27: 292–315
DOI: 10.1007/s004530010021

Algorithmica

Dynamic Algorithms for Graphs of Bounded Treewidth[1]

T. Hagerup[2]

Abstract. The formalism of monadic second-order (MS) logic has been very successful in unifying a large number of algorithms for graphs of bounded treewidth. We extend the elegant framework of MS logic from static problems to dynamic problems, in which queries about MS properties of a graph of bounded treewidth are interspersed with updates of vertex and edge labels. This allows us to unify and occasionally strengthen a number of scattered previous results obtained in an ad hoc manner and to enable solutions to a wide range of additional problems to be derived automatically.

As an auxiliary result of independent interest, we dynamize a data structure of Chazelle for answering queries about products of labels along paths in a tree with edges labeled by elements of a semigroup.

Key Words. Dynamic algorithms, Graph algorithms, Treewidth, Monadic second-order logic, Path queries.

1. Introduction. Many graph properties can be expressed via formulas in a suitable logic. For example, for given vertices s and t in a directed graph, the fact that the subgraph spanned by a set A of edges contains a path from s to t can be expressed by saying that every vertex set U containing s, but not t, can be left via an edge in A, i.e., by the formula

$$\begin{aligned} \mathit{Joins}(A,s,t) \;\equiv\; \forall U((s \in U \wedge t \notin U) \\ \Rightarrow \exists e \exists u \exists v(\mathit{tail}(u,e) \wedge \mathit{head}(v,e) \wedge e \in A \wedge u \in U \wedge v \notin U)), \end{aligned}$$

where e ranges over all edges, u and v range over all vertices, U ranges over all sets of vertices, and $\mathit{tail}(u,e)$ and $\mathit{head}(v,e)$ express that u and v are the tail and the head of e, respectively. If we want the graph spanned by A to be just a single (simple) path from s to t, we can additionally require A to be minimal; i.e.,

$$\mathit{Path}(A,s,t) \equiv \mathit{Joins}(A,s,t) \wedge \forall B((B \subseteq A \wedge \mathit{Joins}(B,s,t)) \Rightarrow B = A),$$

where B ranges over all sets of edges.

Expressing computational problems such as "Is there a path from s to t?" in a formal framework holds out the prospect of deriving algorithms to solve such problems in an automatic way. Indeed, every graph property expressible in first-order logic can be decided in polynomial time. The catch is that first-order logic is too weak to express most graph properties of interest (see, e.g., [18]). It allows variables ranging over vertices and edges, existential and universal quantification over such variables, the usual logic connectives $\wedge$, $\vee$, and $\neg$, and predicates such as *tail* and *head* for accessing the basic

[1] A preliminary version of this paper was presented at the 24th International Colloquium on Automata, Languages and Programming (ICALP) in Bologna, Italy, in July 1997.

[2] Fachbereich Informatik, Robert-Mayer-Straße 11–15, Johann Wolfgang Goethe-Universität Frankfurt, D-60054 Frankfurt am Main, Germany. hagerup@informatik.uni-frankfurt.de. The work was done while the author was with the Max-Planck-Institut für Informatik in Saarbrücken, Germany.

Received February 10, 1997. Communicated by H. L. Bodlaender.

connectivity structure of the graph under consideration. Very frequently, however, one is led, as in the examples above, to introduce variables ranging not over individual vertices or edges, but over sets of vertices or edges. Extending first-order logic with this possibility, we arrive at *monadic second-order* (*MS*) *logic*. MS logic as a graph-description language is defined more precisely in the following section; for the time being, the examples above provide a sufficient flavor. Strictly speaking, the examples are not formulated in pure MS logic, but it is easy to reformulate them to satisfy the formal definition (e.g., "$B \subseteq A$" would be rendered as "$\forall e(e \in A \vee \neg\, e \in B)$," where e ranges over all edges). As noted by many researchers, MS logic is a powerful language that allows the expression of a wide range of graph properties. Indeed, the collection of decision problems on graphs that can be defined by MS formulas is so large that it includes many NP-complete problems, leaving little hope of obtaining efficient algorithms for the general case. Rather than reverting to a less expressive logic, one can try to evade this problem by restricting the class of input graphs. Arnborg et al. [3] argue that a particularly felicitous combination is to consider problems definable by an MS formula on graphs of *bounded treewidth*, i.e., on graphs drawn from a class with a uniform upper bound on the *treewidth* of all graphs in the class. Loosely speaking, the treewidth of a graph is a measure of how far the graph deviates from being a tree. The details of the definition are provided in the next section, but are not crucial to the present discussion, except that several important classes of graphs are of bounded treewidth.

Consider a single MS formula Φ with l free set variables (such as "A" in the formula "$Path(A, s, t)$") and without free simple variables. Φ gives rise to several computational graph problems: First, there is the *decision problem* of determining whether there are sets $A_1, \ldots, A_l$ of vertices or edges that satisfy Φ if substituted for its free variables (e.g., "Is there a path from s to t?"). For this first type of problem it is not necessary to allow Φ to have free variables—we might as well quantify them existentially; still, we keep the present formulation for the sake of uniformity. Second, the *counting problem* of detecting the number of such tuples (e.g., "How many paths are there from s to t?"). Third, if the input additionally associates each vertex or edge a with an l-tuple $(f_1(a), \ldots, f_l(a))$ of real numbers, whose ith element is interpreted as the cost of including a in A_i, for $i = 1, \ldots, l$, the *optimization problem* of computing the minimal cost of a tuple $(A_1, \ldots, A_l)$ that satisfies Φ (e.g., "What is the distance from s to t?"). Fourth, in the same setting, the *construction problem* (this is not a standard term) of actually computing a tuple $(A_1, \ldots, A_l)$ satisfying Φ and of minimal cost (e.g., "Which path from s to t is shortest?"). Fifth, if $f_i(a)$ is reinterpreted as the probability of a stepping into A_i, for $i = 1, \ldots, l$, with each vertex or edge entering each set independently of all other such random decisions, the *reliability problem* of computing the probability of obtaining a tuple $(A_1, \ldots, A_l)$ that satisfies Φ (e.g., "What is the probability of having an operational path from s to t?").

Results by Courcelle [19] and Bodlaender [9] imply that every decision problem defined by an MS property can be solved in linear time on graphs of bounded treewidth. Generalizations of these and related earlier results to counting, optimization, construction, and reliability problems were investigated by a number of authors [3], [4], [7], [12], [20], [22], [31]. One of the simplest and most general extensions was suggested by Courcelle and Mosbah [20], and we essentially use their framework. In our formulation, a generic algorithm is instantiated by choosing a particular *commutative semiring*

$\mathcal{R} = (R, \oplus, \otimes, \bar{0}, \bar{1})$, i.e., an algebraic structure consisting of a set R, equipped with two associative and commutative operations $\oplus$ and $\otimes$ with neutral elements $\bar{0}$ and $\bar{1}$, respectively, such that $\otimes$ distributes over $\oplus$ (i.e., $a \otimes (b \oplus c) = (a \otimes b) \oplus (a \otimes c)$ for all $a, b, c \in R$) and $a \otimes \bar{0} = \bar{0}$ for all $a \in R$. Given an input graph $G = (V, E)$ with associated functions $f_1, \ldots, f_l$: $V \cup E \to R$ (which are called *cost functions*, independently of their interpretation), the generic algorithm computes the *value* of G under Φ and $\mathcal{R}$, defined as the quantity

$$|G|_{\Phi,\mathcal{R}} = \bigoplus_{G \models \Phi[A_1,\ldots,A_l]} \bigotimes_{i=1}^{l} \bigotimes_{a \in A_i} f_i(a),$$

i.e., the "sum," over all tuples $(A_1, \ldots, A_l)$ that satisfy Φ, of the "products," over the sets A_i, of the appropriate costs. As we will see in Section 3, this solves the problems mentioned above as well as a number of additional problems. For example, with $\mathcal{R} = (\mathbb{N} \cup \{0\}, +, \cdot, 0, 1)$, where $\mathbb{N} = \{1, 2, \ldots\}$, we obtain a solution to the counting problem.

The problem of computing $|G|_{\Phi,\mathcal{R}}$ for fixed Φ and $\mathcal{R}$ is called *static*, meaning that the entire input as well as the question to be answered are known from the outset. The focus of this paper is to extend the elegant framework of MS logic to a dynamic setting in which, following a certain *initialization* or *preprocessing* based on the input graph, a sequence of *attribute updates* and queries must be executed on-line; i.e., each query must be answered before the next operation to be executed is revealed. An (attribute) update changes a single attribute of a vertex or edge without affecting the structure of the graph. One might also consider *structural updates* that insert or delete edges. The data structures and algorithms described here can easily be extended to allow deletions, but supporting insertions of vertices and edges appears to be considerably more difficult. Bodlaender [8] and Cohen et al. [16] describe dynamic algorithms that admit structural updates in the special case of graphs of treewidth 2 or 3. A construction of Frederickson [21] allows structural updates for every treewidth, but requires a user to specify structural updates to a graph through updates to its tree decomposition (defined in the next section), which places the burden of maintaining the tree decomposition on the user.

We allow *boolean attributes*, which take values in {*false*, *true*}, indicate (non)membership in "user-defined" sets, and may be tested in Φ through corresponding predicates, and *ring attributes*, which take values in R, together define the cost functions, and cannot be referred to in Φ. A query temporarily (for the duration of the query) carries out a constant number of updates of boolean and/or ring attributes, thereby changing G into G', and then computes and returns $|G'|_{\Phi,\mathcal{R}}$, after which all attributes revert to their values before the query. This view of a query operation may be unfamiliar, but it is general and permits a convenient statement of our results.

Our running example centered around the MS formula $Path(A, s, t)$ is used to clarify some of the concepts introduced above. We have already seen that $Path(A, s, t)$ expresses that the edges in A span a path from s to t, and if we give each edge e a ring attribute $f(e)$ equal to its length, the length of the path spanned by A is $\sum_{e \in A} f(e)$, which can be minimized by choosing $\mathcal{R} = (\mathbb{R} \cup \{\infty\}, \min, +, \infty, 0)$. What is lacking is that we would like to support queries asking for the distance from s to t (call this an (s, t)-query), where s and t are variable. We can achieve this effect within the general framework by introducing two "user-defined" sets, S and T, both initialized to $\varnothing$, letting an (s, t)-query

temporarily change two boolean vertex attributes to make $S = \{s\}$ and $T = \{t\}$, and using instead of the original formula the formula

$$\exists s \exists t (Origin(s) \wedge Destination(t) \wedge Path(A, s, t)),$$

where *Origin* and *Destination* are predicate symbols corresponding to the sets S and T, respectively. It should be clear that other traditional types of queries can be formulated in a similar way. We show that for all $\tau \geq 1$, the dynamic version of every problem defined by an MS formula Φ and a commutative semiring $\mathcal{R}$ whose operations can be carried out in constant time (call such a semiring *efficient*) can be solved on n-vertex graphs of bounded treewidth with initialization time $O(n)$, (attribute-)update time $O(\tau n^{1/\tau})$, and query time $O(\tau + \alpha(n))$, where α is a slowly growing "inverse Ackermann" function. Alternatively, for arbitrary integer $k \geq 1$, with the same update time, but initialization time $O(nI_k(n))$ and query time $O(\tau + k)$, where I_k, for every integer $k \geq 1$, is another slowly growing function. Both α and the functions I_k are defined in the next section. In the special case of the dynamic *distance* and *shortest-path* problems considered above, this result was obtained previously by Chaudhuri and Zaroliagis [13] for $\tau = O(1)$ as well as with a worse tradeoff between initialization time, update time, and query time. In more detail, Chaudhuri and Zaroliagis indicate the following bounds, for all integers $r \geq 1$: initialization time $O(c^r n)$, update time $O(c^{2r} n^{2^{1-r}})$, and query time $O(c^{2r}\alpha(n))$, where $c = \Theta(3^r)$. In order to compare these bounds with ours, take $\tau = c^{2r}$ to observe that our bounds associate a query time of $O(c^{2r}\alpha(n))$ with an update time of the form $c^{2r} n^{2^{-\Omega(r^2)}}$. For example, in order to achieve an update time of $O(2^{\sqrt{\log n}})$ with the bounds of Chaudhuri and Zaroliagis, it is necessary to choose r larger than $\frac{1}{2}\log\log n$, which yields a query time of $(\log n)^{\Omega(\log\log n)}$, whereas our bounds associate an update time of $O(2^{\sqrt{\log n}})$ with a query time of $O(\sqrt{\log n})$. Our bounds are never worse than those of Chaudhuri and Zaroliagis, and strictly better for all nonconstant τ and r. One end of the tradeoff, with update and query times both $O(\log n)$, was realized previously by Bodlaender [8] and Cohen et al. [16].

If only queries but no updates are to be supported, we achieve initialization time $O(n)$ and query time $O(\alpha(n))$ or, for every integer $k \geq 1$, initialization time $O(nI_k(n))$ and query time $O(k)$. This result was found previously by Chaudhuri and Zaroliagis [13] for the distance and shortest-path problems and by Arikati et al. [2] for the problem of computing (the value of) a minimum cut separating two given vertices. (The value of a minimum cut separating s and t can be found by minimizing $\sum_{e \in A} f(e)$, where $f(e)$ denotes the capacity of the edge e, subject to $\forall B(Path(B, s, t) \Rightarrow (A \cap B \neq \varnothing))$, where A and B range over all sets of edges.)

In some cases, queries may become cheaper if they can be batched. We consider queries that (temporarily) change at most d boolean attributes and no ring attributes and use the term *exhaustive d-dimensional query* to denote a set of all possible queries of this type (e.g., the well-known all-pairs shortest-paths problem is to answer an exhaustive two-dimensional query). We show that for all $d \geq 1$, exhaustive d-dimensional queries defined by an MS formula and an efficient commutative semiring can be answered in $O(n^d)$ time for n-vertex input graphs of bounded treewidth. This was proved previously for $d = 1$ and $d = 2$ for the distance problem by Radhakrishnan et al. [27] and for $d = 2$ for the problem of computing (the value of) a minimum cut by Arikati et al. [2].

All of the algorithms described above translate into algorithms for the EREW PRAM, a parallel machine described in the next section. In all cases, the parallel algorithms execute as many operations as (i.e., have the same time–processor product as) their sequential counterparts, up to a constant factor, and the running times on n-vertex input graphs G are typically $O(\log n)$ if a minimum-width tree decomposition of G is provided as part of the input, and $O((\log n)^2)$ if not. The static problem is solved within these time bounds, and the queries-only problem is solved with these time bounds for the initialization, while queries are executed by a single processor within the same time bounds as in the sequential case. For the dynamic problem the initialization and update times are both $O(\tau \log n)$, where τ is the parameter introduced above, and queries are again executed by a single processor as in the sequential case. For the distance and shortest-path problems, most of these results were found previously by Chaudhuri and Zaroliagis [14].

The next section provides some preliminary details, mainly related to MS logic and treewidth. In subsequent sections we consider computational problems of the kind discussed above, progressing from the purely static problem to the fully dynamic one. Section 3 describes a generic algorithm, essentially due to previous authors, for solving the static problem. In Section 4 we explain how to modify the static algorithm to allow queries, but not yet updates. Section 5 is devoted to exhaustive queries and Section 6, finally, deals with both queries and updates. The final section of the paper contains some additional observations related to the shortest-path problem, an important special case.

2. Preliminaries. MS logic, specialized to the description of graphs, is a language, i.e., a set of symbol strings, called *formulas*. In order to define the language, we begin by fixing three disjoint infinite sequences $\mathcal{P}$, $\mathcal{V}_1$, and $\mathcal{V}_2$ without repetitions of *unary predicate symbols*, *simple variables*, and *set variables*, respectively. We assume that $\mathcal{P}$ contains the special symbols *vertex*, *arc*, and *edge*. A symbol string is a formula exactly if this can be deduced from the following rules: (1) If $P \in \mathcal{P}$ and $x \in \mathcal{V}_1$, then $P(x)$ is a formula. (2) If $x, y \in \mathcal{V}_1$, then $\mathit{tail}(x, y)$, $\mathit{head}(x, y)$, $\mathit{incident}(x, y)$, and $x = y$ are formulas. (3) If $x \in \mathcal{V}_1$ and $X \in \mathcal{V}_2$, then $x \in X$ is a formula. (4) If φ and ψ are formulas, then so are $(\varphi \wedge \psi)$, $(\varphi \vee \psi)$, and $\neg\varphi$. (5) If φ is a formula, $x \in \mathcal{V}_1$, and $X \in \mathcal{V}_2$, then $\exists x\varphi$, $\forall x\varphi$, $\exists X\varphi$, and $\forall X\varphi$ are all formulas.

The *free variables* of a formula are defined as usual: If a formula φ can be constructed using only rules (1)–(3) above, a variable is free in φ exactly if it occurs in φ. The free variables of $\neg\varphi$ are those of φ, a variable is free in $\varphi \wedge \psi$ and $\varphi \vee \psi$ iff it is free in at least one of φ and ψ, and a variable is free in any one of $\exists x\varphi$, $\forall x\varphi$, $\exists X\varphi$, and $\forall X\varphi$ iff it is free in φ and distinct from the variable x or X that follows the quantifier. A formula without free variables is called a *sentence*.

In order to allow an MS formula to be interpreted as a statement about a graph, one views the graph as a relational structure, i.e., a *ground set* Ω together with a set of relations defined on Ω. We consider the relations to be *predicates*, i.e., functions with range $\{\mathit{false}, \mathit{true}\}$. Following Arnborg et al. [3], we allow mixed graphs containing directed and/or undirected edges and view a graph G with vertex set V, edge set E, and a tuple of l boolean attributes associated with each element of $V \cup E$ as a relational structure $(\Omega, \mathit{vertex}, \mathit{arc}, \mathit{edge}, \mathit{tail}, \mathit{head}, \mathit{incident}, P_1, \ldots, P_l)$ with ground set $\Omega = V \cup E$, unary predicates *vertex*, *arc*, and *edge* on Ω taking the value *true* precisely on those elements of

the ground set that are vertices, directed edges, and undirected edges, respectively, binary predicates *tail*, *head*, and *incident* on Ω taking the value *true* precisely on those pairs $(v, e) \in \Omega \times \Omega$ for which v is the tail of the directed edge e, the head of the directed edge e, and an endpoint of the undirected edge e, respectively, and unary predicates $P_1, \ldots, P_l$ on Ω such that the value of P_i on a coincides with the value of the ith boolean attribute of a, for all $a \in \Omega$ and for $i = 1, \ldots, l$.

A graph G, viewed as a relational structure with ground set Ω, is said to be *appropriate* for an MS formula Φ if it defines (contains) a predicate on Ω for each predicate symbol occurring in Φ (with the correspondence between predicate symbols and predicates fixed in some manner). Even if G is appropriate for Φ, before we can speak of G as (not) satisfying Φ, we have to fix values for all free variables occurring in Φ. An *assignment* is a function that maps each variable in $\mathcal{V}_1$ to an element of Ω and each variable in $\mathcal{V}_2$ to a subset of Ω. The assertion that G satisfies Φ under an assignment σ, written $G \models \Phi[\sigma]$, is defined as one expects. Formally, the definition is by structural induction: (1) If P is a predicate symbol of arity $r \in \{1, 2\}$ and $x_1, \ldots, x_r \in \mathcal{V}_1$, then $G \models P(x_1, \ldots, x_r)[\sigma]$ iff $\bar{P}(\sigma(x_1), \ldots, \sigma(x_r))$, where $\bar{P}$ is the predicate on Ω corresponding to P. (2) If $x, y \in \mathcal{V}_1$, then $G \models x = y[\sigma]$ iff $\sigma(x) = \sigma(y)$. (3) If $x \in \mathcal{V}_1$ and $X \in \mathcal{V}_2$, then $G \models x \in X[\sigma]$ iff $\sigma(x) \in \sigma(X)$. (4) If φ and ψ are formulas, then $G \models (\varphi \wedge \psi)[\sigma]$ iff $G \models \varphi[\sigma]$ and $G \models \psi[\sigma]$, $G \models (\varphi \vee \psi)[\sigma]$ iff $G \models \varphi[\sigma]$ or $G \models \psi[\sigma]$, and $G \models \neg\varphi[\sigma]$ iff $G \not\models \varphi[\sigma]$. (5) If φ is a formula and $x \in \mathcal{V}_1$, then $G \models \exists x\varphi[\sigma]$ iff there exists $a \in \Omega$ such that $G \models \varphi[\sigma_{|x=a}]$, where $\sigma_{|x=a}$ is the assignment that coincides with σ, except that $\sigma_{|x=a}(x) = a$. Similarly, $G \models \forall x\varphi[\sigma]$ iff $G \models \varphi[\sigma_{|x=a}]$ for all $a \in \Omega$. If φ is a formula and $X \in \mathcal{V}_2$, then $G \models \exists X\varphi[\sigma]$ iff there exists $A \subseteq \Omega$ such that $G \models \varphi[\sigma_{|X=A}]$, and $G \models \forall X\varphi[\sigma]$ iff $G \models \varphi[\sigma_{|X=A}]$ for all $A \subseteq \Omega$.

It is easy to see that for an arbitrary graph G appropriate for an MS formula Φ, the truth of $G \models \Phi[\sigma]$, where σ is an assignment, depends only on the images under σ of the free variables in Φ. If Φ has k free simple variables and l free set variables, we therefore write $G \models \Phi[\sigma]$ as $G \models \Phi[a_1, \ldots, a_k, A_1, \ldots, A_l]$, where $a_1, \ldots, a_k$ and $A_1, \ldots, A_l$ are the images under σ of the free simple and set variables in Φ, respectively, in the order in which they appear in the sequences $\mathcal{V}_1$ and $\mathcal{V}_2$. For the sake of convenience, we usually assume that $k = 0$, since free simple variables, for our purposes, can be eliminated without loss of expressivity; this involves replacing each free simple variable by a set variable and requiring the corresponding set X to contain exactly one element through a formula $\mathit{Singleton}(X)$ that can obviously be formulated in MS logic without introducing free simple variables.

It is often convenient not to stick rigidly to the definition of MS logic given above. For example, in the Introduction we freely used symbols such as "$\Rightarrow$" that have straightforward definitions in terms of the basic symbols, and it often improves readability to add or omit certain parentheses. Although, in principle, the sequences of predicate symbols and variables are predetermined and fixed, in practice we can use any convenient names. The symbol "$\equiv$" introduces abbreviated names for formulas, with free variables indicated between parentheses; rather than prescribing a specific mechanism of "parameter substitution" for the free variables, we rely on the reader's good sense of what is intended. Although quantification always is over (all subsets of) the entire ground set, and all elements of the ground set have the same types of attributes, it is often desirable to quantify over only vertices or only edges or to associate attributes with only vertices

or only edges. This effect is easily achieved by using the predicates *vertex*, *arc*, and *edge* on the one hand, and by setting unused boolean attributes to *false* and unused ring attributes to the additive neutral element on the other hand. A large collection of graph properties that can be formulated as MS sentences can be found in [3] and [5]. We offer a few additional examples of queries that fit our framework. In all cases, lowercase letters, with or without subscripts, denote vertices specified in a query.

- A decision problem: Is there a depth-first-search tree in which u but not v is an ancestor of w?
- A counting problem: How many Hamilton paths are there from s to t?
- An optimization problem: What is the capacity of a maximum cut separating s and t?
- A construction problem: Which are the disjoint paths from s_1 to t_1 and from s_2 to t_2 of minimum total length?
- A reliability problem: What is the probability of having operational paths starting at s and t and ending at the same vertex?

When considering a graph G with vertex set V and edge set E from an algorithmic rather than a logic point of view, we may write G as (V, E, Δ), where Δ specifies the boolean and ring attributes of the vertices and edges in G in some way that will be of no concern here. We occasionally apply the term *labeled graph* to G to emphasize that we consider the attributes associated with elements of $V \cup E$ to be an integral part of G. When the attributes are absent or irrelevant, we may omit Δ and write G as (V, E). Standard graph terminology for unlabeled graphs extends to labeled graphs in an obvious way on which we do not elaborate.

As introduced by Robertson and Seymour [29], a *tree decomposition* of a graph $G = (V, E)$ is a pair $(T_D, \mathcal{U})$, where $T_D = (V_D, E_D)$ is an undirected tree and $\mathcal{U} = \{U_x \mid x \in V_D\}$ is a family of subsets of V called *bags*, one for each node in T_D, such that

(1) $\bigcup_{x \in V_D} U_x = V$ (every vertex in G occurs in some bag);
(2) for all $u, v \in V$, if u and v are the endpoints of some edge in E, then there exists an $x \in V_D$ with $\{u, v\} \subseteq U_x$ (every edge in G is "internal" to some bag);
(3) for all $x, y, z \in V_D$, if y is on the path from x to z in T_D, then $U_x \cap U_z \subseteq U_y$ (every vertex in G occurs in the bags in a connected part of T_D, i.e., in a subtree).

The *width* of a tree decomposition $(T_D = (V_D, E_D), \{U_x \mid x \in V_D\})$ is $\max_{x \in V_D} |U_x| - 1$. The *treewidth* of a graph G is the smallest treewidth of any tree decomposition of G. Many important graph classes are of bounded treewidth, including those of outerplanar and series-parallel graphs; for surveys of results of this kind, see [10] and [33].

Define $I_0 \colon \mathbb{N} \to \mathbb{N}$ by $I_0(n) = \lceil n/2 \rceil$, for all $n \in \mathbb{N}$. Inductively, for $k = 1, 2, \ldots$, define $I_k \colon \mathbb{N} \to \mathbb{N}$ by $I_k(n) = \min\{i \in \mathbb{N} \mid I_{k-1}^{(i)}(n) = 1\}$, for all $n \in \mathbb{N}$, where superscript (i) denotes i-fold repeated application. Finally, for all $n \in \mathbb{N}$, take $\alpha(n) = \min\{k \in \mathbb{N} \mid I_k(n) \leq 3\}$. $I_1(n) = \lceil \log_2 n \rceil$ and $I_2(n) = \log^* n$ for all $n \geq 2$, the function I_k grows very slowly for large k, and the "inverse Ackermann" function α grows even more slowly.

A *PRAM* is a synchronous parallel machine consisting of processors numbered $1, 2, \ldots$ and a global memory accessible to all processors. An *EREW PRAM* disallows concurrent access to a memory cell by more than one processor, a *CREW PRAM* allows concurrent reading, but not concurrent writing, and a *CRCW PRAM* allows both

concurrent reading and concurrent writing. If a PRAM algorithm uses p processors and t time steps, it is said to use pt *operations*. Our exposition assumes basic familiarity with the design of PRAM algorithms as can be gained, e.g., from the first chapters of [25].

3. Static Algorithms. Given an MS formula Φ with l free set variables and without free simple variables and a commutative semiring $\mathcal{R} = (R, \oplus, \otimes, \bar{0}, \bar{1})$, we say that a labeled graph $G = (V, E, \Delta)$ is appropriate for the pair $(\Phi, \mathcal{R})$ if it is appropriate for Φ and each $a \in V \cup E$ has l ring attributes $f_1(a), \ldots, f_l(a) \in R$. The (static) *RMS problem* defined by Φ and $\mathcal{R}$ is, given a graph G appropriate for $(\Phi, \mathcal{R})$, to compute the value

$$|G|_{\Phi,\mathcal{R}} = \bigoplus_{G \models \Phi[A_1,\ldots,A_l]} \bigotimes_{i=1}^{l} \bigotimes_{a \in A_i} f_i(a)$$

of G under Φ and $\mathcal{R}$. Here $\oplus$ and $\otimes$ over an empty index set are supposed to yield $\bar{0}$ and $\bar{1}$, respectively. Courcelle and Mosbah [20] show that every RMS problem defined by an MS formula Φ without free simple variables and an efficient commutative semiring $\mathcal{R}$ can be solved in linear time on graphs of bounded treewidth. In this section we first give a different proof of their result, modeled on the approach of [3], then review interesting special cases obtained by choosing $\mathcal{R}$ appropriately, and finally show that a simple extension of our algorithm for the RMS problem can be used to solve problems of random generation. The reason for providing a nearly self-contained proof of Theorem 3.1 is that we refer to details of the proof frequently in what follows.

THEOREM 3.1 [20]. *For all constants $t \geq 1$ and all integers $n \geq 1$, every RMS problem defined by an MS formula Φ without free simple variables and an efficient commutative semiring $\mathcal{R}$ can be solved in $O(n)$ time on n-vertex input graphs appropriate for $(\Phi, \mathcal{R})$ and of treewidth at most t.*

PROOF. Let $G = (V_G, E_G, \Delta)$ be an n-vertex input graph appropriate for $(\Phi, \mathcal{R})$ and of treewidth at most t and take $\Omega = V_G \cup E_G$. We first show that $O(n)$ time suffices to construct an MS formula Ψ with the same free variables as Φ, an undirected tree $T^* = (V^*, E^*, \Delta^*)$ of degree ≤ 3 and appropriate for $(\Psi, \mathcal{R})$, and an injective function $\pi\colon \Omega \to V^*$ so that the following holds: Suppose that Φ has l free set variables. Then, for all $A_1, \ldots, A_l \subseteq \Omega$, $G \models \Phi[A_1, \ldots, A_l]$ if and only if $T^* \models \Psi[\pi(A_1), \ldots, \pi(A_l)]$; moreover, $T^* \not\models \Psi[B_1, \ldots, B_l]$ whenever $\bigcup_{i=1}^{l} B_i \not\subseteq \pi(\Omega)$. Intuitively, if we identify a and $\pi(a)$, for all $a \in \Omega$, then we want T^* to satisfy Ψ under a particular assignment if and only if G satisfies Φ under the same assignment. In the construction of Ψ, it is convenient to modify the MS language slightly by replacing all of the binary predicates *tail*, *head*, and *incident* by a single binary predicate *adjacent* with the value *true* on a pair of vertices exactly if they are adjacent; this prevents us from expressing facts about edges, but we shall never need to.

Our construction is a variant of one described by Arnborg et al. [3]. The first step is to compute a tree decomposition $(T_{\mathrm{D}} = (V_{\mathrm{D}}, E_{\mathrm{D}}), \mathcal{U} = \{U_x \mid x \in V_{\mathrm{D}}\})$ of G of width

bounded by t. We obtain the structure of T^* by replacing each node $x \in V_D$ by a tree $T_x = (V_x, E_x)$ of degree ≤ 3, called the *piece* of x, with at least $3|U_x|^2$ vertices and at least d leaves, where d is the degree of x in T_D, and then connecting the pieces with a set of endpoint-disjoint edges that consists of, for each edge $\{x, y\} \in E_D$, an undirected edge between a leaf of the piece of x and a leaf of the piece of y.

In order to define the attributes of T^*, we begin by rooting T_D at an arbitrary node r_D. We then process T_D from the root to the leaves (i.e., process each node before all of its children), marking each node in T_D with its depth (i.e., distance from r_D), marking each vertex $v \in V_G$ with the node in V_D of minimal depth whose bag contains v, called the *high point* of v (it follows from properties (1) and (3) of a tree decomposition that there is a unique such node), and coloring the vertices in V_G with at most $2(t+1)$ colors such that distinct vertices that belong to a common bag or to bags of neighboring nodes in T_D have distinct colors—a greedy strategy that colors a vertex when its high point is processed suffices. Subsequently, it is an easy matter to determine for each edge $e \in E_G$ a node $x \in V_D$ whose bag contains both endpoints of e (it follows from properties (2) and (3) of a tree decomposition that the high point of one of the endpoints of e will do); we think of e as assigned to x and define A_x, for all $x \in V_D$, as the set of edges of G assigned to x. For each $x \in V_D$, we now fix an arbitrary injective mapping g_x from $U_x \cup A_x$ to V_x; this is possible since $|V_x| \geq 3|U_x|^2$. For each $a \in \Omega$, define a *representative* of a to be any vertex in V^* of the form $g_x(a)$, where $x \in V_D$ and $a \in U_x \cup A_x$. Each edge has exactly one representative, while a vertex may have many representatives. For each $a \in \Omega$, choose $\pi(a)$ as an arbitrary, but fixed representative of a and copy all attributes of a in G to $\pi(a)$.

We next mark each vertex $v \in V^*$ with a boolean attribute *EvenDepth*$[v]$, whose value is *true* exactly if v belongs to the piece of a node in T_D of even depth. Two vertices $u, v \in V^*$ belong to the same piece if and only if there is a path from u to v, all of whose vertices have the same value of *EvenDepth*; adapting the formula *Path* defined earlier to the case of undirected graphs and to the use of *adjacent* instead of *tail* and *head* (which only makes things easier), we can obviously express this via an MS formula *SamePiece*(u, v).

For each $v \in V_G$, we color each representative of v with the color of v computed above, while leaving vertices in V^* that are not representatives of vertices in V_G uncolored; in actuality, the at most $2(t+1)$ colors are expressed through $2(t+1)$ "color predicates." Now two vertices $u, v \in V^*$ are representatives of the same vertex in V_G if and only if they have the same color C and are connected by a path, each of whose vertices belongs to a piece containing a vertex of color C. Again, using *Path* and *SamePiece*, this is easily expressed in the form of an MS formula *SameVertex*(u, v).

We next define a constant number of "tail predicates" on V^* such that for all $x \in V_D$ and all $v \in U_x$ and $e \in A_x$, $g_x(v)$ and $g_x(e)$ *share* a tail predicate, i.e., some tail predicate has the value *true* on both $g_x(v)$ and $g_x(e)$, if and only if v is the tail of e. This is possible since $U_x \cup A_x$ is of bounded size for all $x \in V_D$. A constant number of "head predicates" and "incident predicates" are defined analogously. Now for all $u, e \in \Omega$, *tail*(u, e) holds in G if and only if $u \in V_G$, $e \in E_G$, and *SameVertex*$(\pi(u), v)$ holds in T^* for some vertex $v \in V^*$ that belongs to the same piece as $\pi(e)$ and shares a tail predicate with $\pi(e)$, clearly a condition on $\pi(u)$ and $\pi(e)$ that can be expressed via an MS formula *Tail*. Predicates *Head* and *Incident* are defined analogously.

With a final predicate *GroundSet* on $V^* \cup E^*$, taking the value *true* precisely on the elements of $\pi(\Omega)$, we are ready to describe the formula Ψ. It is identical to Φ, except that all occurrences of *tail*, *head*, and *incident* are replaced by occurrences of *Tail*, *Head*, and *Incident*, respectively, and that all variables of Φ are constrained to belong to $\pi(\Omega)$. The latter requirement is enforced by, e.g., replacing each subformula of the form $\exists x\varphi$ in Φ, where x is a simple variable, by $\exists x(\mathit{GroundSet}(x) \wedge \varphi)$.

It is easy to see that T^*, Ψ, and π have the desired properties, which implies that $|T^*|_{\Psi,\mathcal{R}} = |G|_{\Phi,\mathcal{R}}$ (recall that all ring attributes are copied from a to $\pi(a)$, for all $a \in \Omega$). The only nontrivial part of the construction, computing a tree decomposition of the input graph, can be carried out in linear time [9]. It therefore now suffices to show how to obtain $|T^*|_{\Psi,\mathcal{R}}$ as the output of a simple tree automaton constructed from Ψ and operating on T^*. For this purpose root T^* at an arbitrary vertex of degree at most 2 and define an arbitrary left-to-right order among the (at most) two children of each vertex in T.

Informally, a finite tree automaton is the natural generalization of a usual finite automaton from inputs that are strings to inputs that are binary trees. Formally, we can take a finite tree automaton to be a 5-tuple $(S, \Sigma, \delta, s_0, F)$, where S is a finite set of *states*, Σ is a finite *alphabet*, δ is a *transition function* from $S \times S \times \Sigma$ to S, $s_0 \in S$ is a distinguished *initial state*, and $F \subseteq S$ is a distinguished set of *accepting states*. Given a binary tree, each of whose vertices is labeled with an element of Σ, the tree automaton assigns a state to each vertex in the tree, working from the leaves to the root (i.e., processing each vertex after all of its children). If the left and right children of a vertex v are assigned states s and t, respectively, and v is labeled σ, the state $\delta(s, t, \sigma)$ is assigned to v; if one or both children are missing, the initial state s_0 is used in place of their states. The tree automaton *accepts* the input tree exactly if the state assigned to the root belongs to F. Arnborg et al. [3] show how to construct a tree automaton $M = (S, \Sigma, \delta, s_0, F)$ with the following property: Suppose that the unary predicates appearing in Ψ are $P_1, \ldots, P_k$. Then $\Sigma = \{\mathit{false}, \mathit{true}\}^{k+l}$, and for arbitrary subsets $A_1, \ldots, A_l$ of V^*, if each vertex $v \in V^*$ is labeled with the bit vector $(P_1(v), \ldots, P_k(v), b_1, \ldots, b_l) \in \Sigma$, where $b_i = \mathit{true}$ iff $v \in A_i$, for $i = 1, \ldots, l$, then M accepts T^* exactly if $T^* \models \Psi[A_1, \ldots, A_l]$. (Strictly speaking, Arnborg et al., instead of our predicate *adjacent*, assume predicates L and R corresponding to the left-child and right-child relations of an *ordered* binary tree; but it is easy either to modify their construction to fit our application or to express *adjacent* in terms of L and R (namely, $\mathit{adjacent}(u, v) \equiv L(u, v) \vee L(v, u) \vee R(u, v) \vee R(v, u)$.)

We show how to derive from M another tree automaton $M' = (S', \Sigma', \delta', s_0')$ to solve the RMS problem at hand. M' is not a finite automaton, since both its alphabet and its state set may be infinite, and it will compute a value (namely, $|T^*|_{\Psi,\mathcal{R}}$) rather than just accepting or rejecting, for which reason it has no set of accepting states; in other respects, M' behaves exactly as a finite tree automaton.

Write $\mathcal{R} = (R, \oplus, \otimes, \bar{0}, \bar{1})$, let $m = |S|$, and identify the states of M with the integers $1, \ldots, m$, with 1 being the initial state. We take the state set S' of M' to be R^m, the set of vectors of length m with components in R, and define the initial state s_0' as $(\bar{1}, \bar{0}, \ldots, \bar{0})$. The alphabet of M' is $\Sigma' = \{\mathit{false}, \mathit{true}\}^k \times R^l$, and the label of a vertex $v \in V^*$ is $(P_1(v), \ldots, P_k(v), f_1(v), \ldots, f_l(v))$, where $f_1, \ldots, f_l$ are the cost functions copied to T^* from the input graph G. We next define the transition function δ'. Assume that the states of the (possibly fictitious) left and right children of a vertex $u \in V^*$ are $(s_1, \ldots, s_m)$

and $(t_1, \ldots, t_m)$, respectively. Then the state of u is $(r_1, \ldots, r_m)$, where

$$r_j = \bigoplus_{p=1}^{m} \bigoplus_{q=1}^{m} \bigoplus_{\substack{(b_1,\ldots,b_l)\in\{false,true\}^l \\ \delta(p,q,(P_1(u),\ldots,P_k(u),b_1,\ldots,b_l))=j}} \left(s_p \otimes t_q \otimes \bigotimes_{\substack{1\le i\le l \\ b_i=true}} f_i(u) \right),$$

for $j = 1, \ldots, m$. It can be seen that the sum for the jth component (corresponding to the jth state of M) is over those pairs of states of M and those choices of (non)membership of u in $A_1, \ldots, A_l$ that would lead the original automaton to give u the state j, for $j = 1, \ldots, m$. It is then easy to prove by induction that for each vertex $u \in V^*$ and for $j = 1, \ldots, m$, the jth component of the state assigned to u is

$$\bigoplus_{\substack{A_1,\ldots,A_l\subseteq U \\ M(u,A_1,\ldots,A_l)=j}} \bigotimes_{i=1}^{l} \bigotimes_{a\in A_i} f_i(a),$$

where U is the set of descendants of u in T^* and $M(u, A_1, \ldots, A_l)$ denotes the state assigned to u by M if the vertex labels are set according to $A_1, \ldots, A_l$. Indeed, suppose that v and w are the left and right children of a vertex u in T^*, respectively, and let U, V, and W be the sets of descendants of u, v, and w, respectively (thus $V \cap W = \varnothing$ and $U = V \cup W \cup \{u\}$). By the induction hypothesis, applied to v and w, and the fact that $\otimes$ distributes over $\oplus$, the jth component of the state of u, for $j = 1, \ldots, m$, is

$$\begin{aligned}
&\bigoplus_{p=1}^{m} \bigoplus_{q=1}^{m} \bigoplus_{\substack{(b_1,\ldots,b_l)\in\{false,true\}^l \\ \delta(p,q,(P_1(u),\ldots,P_k(u),b_1,\ldots,b_l))=j}} \left(\left(\bigoplus_{\substack{B_1,\ldots,B_l\subseteq V \\ M(v,B_1,\ldots,B_l)=p}} \bigotimes_{i=1}^{l} \bigotimes_{a\in B_i} f_i(a) \right) \right. \\
&\qquad \left. \otimes \left(\bigoplus_{\substack{C_1,\ldots,C_l\subseteq W \\ M(w,C_1,\ldots,C_l)=q}} \bigotimes_{i=1}^{l} \bigotimes_{a\in C_i} f_i(a) \right) \otimes \bigotimes_{\substack{1\le i\le l \\ b_i=true}} f_i(u) \right) \\
&= \bigoplus_{p=1}^{m} \bigoplus_{q=1}^{m} \bigoplus_{\substack{B_1,\ldots,B_l\subseteq V \\ M(v,B_1,\ldots,B_l)=p}} \bigoplus_{\substack{C_1,\ldots,C_l\subseteq W \\ M(w,C_1,\ldots,C_l)=q}} \bigoplus_{\substack{(b_1,\ldots,b_l)\in\{false,true\}^l \\ \delta(p,q,(P_1(u),\ldots,P_k(u),b_1,\ldots,b_l))=j}} \\
&\qquad \left(\bigotimes_{i=1}^{l} \left(\left(\bigotimes_{a\in B_i} f_i(a) \right) \otimes \left(\bigotimes_{a\in C_i} f_i(a) \right) \right) \otimes \bigotimes_{\substack{1\le i\le l \\ b_i=true}} f_i(u) \right) \\
&= \bigoplus_{\substack{A_1,\ldots,A_l\subseteq U \\ M(u,A_1,\ldots,A_l)=j}} \bigotimes_{i=1}^{l} \bigotimes_{a\in A_i} f_i(a).
\end{aligned}$$

If the state of the root of T^* computed by M' is $(s_1, \ldots, s_m)$, this observation shows that $|G|_{\Phi,\mathcal{R}} = |T^*|_{\Psi,\mathcal{R}} = \bigoplus_{j\in F} s_j$, which the automaton therefore computes and returns. Since m is a constant (for fixed Φ), each application of δ' takes constant time, so that the entire processing of T^* by M' can be carried out in $O(n)$ time. This ends the proof of Theorem 3.1. □

As shown below, it is very easy to parallelize the algorithm of Theorem 3.1, since essentially all of the work has been done elsewhere.

THEOREM 3.2. *For all constants $t \geq 1$ and all integers $n \geq 2$, every RMS problem defined by an MS formula Φ without free simple variables and an efficient commutative semiring $\mathcal{R}$ can be solved on an EREW PRAM using $O(\log n)$ time and $O(n)$ operations on each n-vertex input graph G appropriate for $(\Phi, \mathcal{R})$ and of treewidth bounded by t, provided that an $O(n)$-node tree decomposition of G of width at most t is supplied as part of the input.*

PROOF. We begin by *balancing* the given tree decomposition, i.e., turning it into a rooted, binary tree decomposition T of the input graph of depth $O(\log n)$, size $O(n)$, and width bounded by a constant. Building on previous work by Bodlaender [6], Bodlaender and Hagerup [11] show how to do this on an EREW PRAM using $O(\log n)$ time and $O(n)$ operations. It is easy to see that the remaining computation of the algorithm of Theorem 3.1 can be structured as a constant number of passes over T from the root to the leaves or vice versa, where the processing of each node requires constant time and a single processor. Thus with one processor for each node in T, it is a simple matter to carry out the computation in time proportional to the depth of T, i.e., in $O(\log n)$ time. If we compute the depths of all nodes in T by the well-known Euler-tour technique [32], sort the nodes in T by depth, and use the result of the sorting to guide the allocation of processors (a similar procedure is described in Section 2.3 of [17]), we can carry out the same computation in $O(\log n)$ time using only $O(n)$ operations. □

If no tree decomposition is provided as part of the input, we can compute one with $O(n)$ nodes and minimum width either in $O(\log n)$ time using $O(n^{3t+4})$ processors on a CRCW PRAM [6] or in $O((\log n)^2)$ time using $O(n)$ operations on an EREW PRAM [11].

We next discuss a number of corollaries of Theorem 3.1 obtained by choosing specific semirings $\mathcal{R}$. Most of these were also considered by Courcelle and Mosbah [20]. Assume throughout that the MS formula Φ under consideration has l free set variables and no free simple variables and let G be an input graph with ground set Ω. G may specify l cost functions $f_1, \ldots, f_l$, or these will be defined independently of the input. We call a tuple $(A_1, \ldots, A_l)$ of Ω a *solution* if $G \models \Phi[A_1, \ldots, A_l]$.

1. Choosing $\mathcal{R}$ as the semiring $(\{false, true\}, \vee, \wedge, false, true)$ and taking $f_i(a) = true$ for all $a \in \Omega$ and $i = 1, \ldots, l$, we can solve the decision problem of determining whether at least one solution exists.

2. Choosing $\mathcal{R}$ instead as the semiring $(\mathbb{N} \cup \{0\}, +, \cdot, 0, 1)$ and taking $f_i(a) = 1$ for all $a \in \Omega$ and $i = 1, \ldots, l$, we can solve the counting problem of determining the number of solutions.

3. Choosing $\mathcal{R}$ as the semiring $(\mathbb{R} \cup \{\infty\}, \min, +, \infty, 0)$, we can solve the optimization problem of minimizing $\sum_{i=1}^{l} \sum_{a \in A_i} f_i(a)$ over all solutions $(A_1, \ldots, A_l)$. Replacing min with max and ∞ with $-\infty$ allows us to solve maximization rather than minimization problems.

4. Taking $\mathcal{R} = (\mathbb{R} \cup \{-\infty, \infty\}, \min, \max, \infty, -\infty)$, we can solve the different optimization problem of minimizing $\max_{1 \leq i \leq l} \max_{a \in A_i} f_i(a)$ over all solutions $(A_1, \ldots, A_l)$. Of course, it is possible to interchange the roles of min and max.

5. In this example, for ease of notation, we assume that $l = 1$ and write f for f_1. Choosing $\mathcal{R} = (2^{2^\Omega}, \cup, \uplus, \varnothing, \{\varnothing\})$ and taking $f(a) = \{\{a\}\}$ for all $a \in \Omega$, we can solve the problem of computing the set of all solutions. Here 2^{2^Ω} is the set of all sets of subsets of the ground set of the input graph (if we insist that $\mathcal{R}$ should be independent of the input graph, we can instead take Ω to be some universal ground set that is a superset of every actual ground set), and $\mathcal{A} \uplus \mathcal{B}$ denotes the set consisting of all unions of a set in $\mathcal{A}$ and a set in $\mathcal{B}$ (e.g., $\{\{a\}, \{b, c\}\} \uplus \{\varnothing, \{d, e\}\} = \{\{a\}, \{b, c\}, \{a, d, e\}, \{b, c, d, e\}\}$). In contrast with the semirings of Examples 1–4, the semiring $\mathcal{R}$ of this example is not efficient.

6. We call an operation $\odot$ on a set S *selective* if $a \odot b \in \{a, b\}$ for all $a, b \in S$, and we call a semiring $(R, \oplus, \otimes, \bar{0}, \bar{1})$ selective if its additive operation $\oplus$ is selective (equivalently, $\oplus$ is the minimum operator induced by some total order on R). This is the case in Examples 1, 3, and 4 above. When dealing with a selective commutative semiring $\mathcal{R} = (R, \oplus, \otimes, \bar{0}, \bar{1})$, we can ask for an *optimal solution*, a solution $(A_1, \ldots, A_l)$ with

$$\bigotimes_{i=1}^{l} \bigotimes_{a \in A_i} f_i(a) = |G|_{\Phi, \mathcal{R}}.$$

In particular, with $\mathcal{R} = (\mathbb{R} \cup \{\infty\}, \min, +, \infty, 0)$, this solves the construction problem discussed in the Introduction.

Define the *cost* of an l-tuple $(A_1, \ldots, A_l)$ of subsets of Ω as $\bigotimes_{i=1}^{l} \bigotimes_{a \in A_i} f_i(a)$ and say that a value $a \in R$ is *minimal* within a finite subset B of R containing a if $a = \bigoplus_{b \in B} b$. Recall that Example 5 shows how to compute the set of all solutions. In order to compute just a single optimal solution, we proceed analogously but whenever the algorithm produces a set containing more than one subset of Ω, discard all of these except one of minimal cost. The fact that $a \oplus b = a$ ("a is cheaper than b") implies $(a \otimes c) \oplus (b \otimes c) = a \otimes c$ ("$a \otimes c$ is cheaper than $b \otimes c$") by distributivity, for all $a, b, c \in R$, can be used to show by induction that this "greedy" approach yields a globally optimal solution. For $l = 1$, we thus take $R' = R \times 2^\Omega$, $f'(a) = (f(a), \{a\})$ for all $a \in \Omega$, and, for all $(a, A), (b, B) \in R'$,

$$(a, A) \oplus' (b, B) = \left(a \oplus b, \begin{cases} A \text{ or } B, & \text{if } a = b \\ A, & \text{if } a \neq b \text{ and } a \oplus b = a \\ B, & \text{if } a \neq b \text{ and } a \oplus b = b \end{cases}\right),$$

$$(a, A) \otimes' (b, B) = (a \otimes b, A \cup B).$$

Because of the nondeterminism implied by the construction "A or B," we cannot construe $\oplus'$ as an algebraic operator in the usual sense, but only as a (nondeterministic) computational procedure. This is merely a technical obstacle that prevents a direct application of the semiring formalism, and carrying out the computation of the tree automaton M' with R, f, $\oplus$, and $\otimes$ replaced by R', f', $\oplus'$, and $\otimes'$ indeed computes an optimal solution (and its cost). The freedom of choice inherent in "A or B" makes it possible to choose between optimal solutions according to secondary criteria; e.g., among all shortest paths from s to t, we can compute one with a minimum (or maximum) number of edges.

Even though the operations $\oplus'$ and $\otimes'$ are more efficient than the operations of Example 5, it is not as clear as for our other examples that they can be executed in constant time, the question of course hinging on how to represent subsets of Ω and how to form

the union of two such subsets. As observed, e.g., by Bern et al. [4] and Arikati et al. [2], however, the usual dynamic-programming device of "back pointers" can be employed. We represent all subsets of Ω currently in use by the algorithm through a directed acyclic graph H. H has exactly $|\Omega|$ vertices of outdegree zero, called *leaves*, each labeled with a different element of Ω, and each nonempty subset A of Ω is represented via a pointer to a vertex v in H, called the *entry point* of A, with the property that the set of labels of leaves reachable from v is precisely A; the empty set is represented via a *null* pointer. In order to form the union of two nonempty subsets A and B of Ω, we add to H a new vertex v and directed edges from v to the entry points of A and B and return a pointer to v. If one of A and B is empty, the union of A and B is computed as a copy of the opposite set, obtained simply by copying the pointer that represents it.

It is obvious that this implementation of subsets of Ω is correct, and that the union of two subsets of Ω can be formed in constant time. What is missing is an efficient way to convert a set A represented as above to a standard representation in which the elements of A appear as consecutive entries in an array or a list. We show that, for each set A that arises in our application, this conversion can be carried out in time $O(|A| + 1)$. The crucial observation is that the algorithm never forms the union of two sets A and B unless they are disjoint (A and B will be sets of vertices in disjoint subtrees). Because of this, there are never two distinct paths in H between two given vertices. Put differently, if $A \neq \varnothing$, a standard graph search started at the entry point of A and ignoring incoming edges will explore a tree, each internal vertex of which has outdegree exactly 2. The time needed for such a search is at most proportional to the number of leaves encountered, i.e., to $|A|$. All that remains is to output the labels of the leaves as they are encountered.

The conversion algorithm just described, while working in linear sequential time, is not satisfactory in a parallel setting. In order to enable an efficient parallel search of the subgraph of H reachable from a given entry point, we augment the algorithm constructing H to mark each vertex in H with the number of its leaf descendants. Then, provided that every path in H is of length $O(\log n)$, as is certainly the case if H was constructed by our parallel algorithm, a set A can be converted in $O(\log n)$ time using $\lceil |A|/\lfloor \log_2 n \rfloor \rceil$ processors by a method very similar to the algorithm for searching in 2-3-trees described by Paul et al. [26]. We leave the details to the reader, except for noting that once a processor reaches a subtree with $O(\log n)$ leaves, it will explore this subtree sequentially.

7. Recall that the reliability problem is defined as follows: For all $a \in \Omega$ and $i = 1, \ldots, l$, a steps into A_i with probability $f_i(a)$, and all such decisions are taken independently. What is the probability that the resulting tuple $(A_1, \ldots, A_l)$ satisfies Φ? In order to solve this problem, we introduce l new free set variables $A_{l+1}, \ldots, A_{2l}$, modify Φ to require A_{l+i} to be the complement of A_i with respect to the ground set Ω, for $i = 1, \ldots, l$, and take $f_{l+i}(a) = 1 - f_i(a)$, for all $a \in \Omega$ and for $i = 1, \ldots, l$. Finally, let $\mathcal{R} = (\mathbb{R}, +, \cdot, 0, 1)$. Then

$$\bigoplus_{G \models \Phi[A_1, \ldots, A_l]} \bigotimes_{i=1}^{2l} \bigotimes_{a \in A_i} f_i(a) = \sum_{G \models \Phi[A_1, \ldots, A_l]} \prod_{i=1}^{l} \left(\prod_{a \in A_i} f_i(a) \right) \left(\prod_{a \in \Omega \setminus A_i} (1 - f_i(a)) \right),$$

which is the desired probability.

We finally show how to solve the problem of generating a random tuple $(A_1, \ldots, A_l)$ that satisfies Φ, with all such tuples being equiprobable. In our usual example based on the *Path* formula, we would ask for a random path from s to t.

We take $\mathcal{R} = (\mathbb{N} \cup \{0\}, +, \cdot, 0, 1)$ and $f_i(a) = 1$ for all $a \in \Omega$ and for $i = 1, \ldots, l$ and use the notation of the proof of Theorem 3.1. In particular, the tree T^*, the formula Ψ, and the machines M and M' are as in that proof. Define a *solution* as a tuple $(A_1, \ldots, A_l)$ of subsets of V^* with $T^* \models \Psi[A_1, \ldots, A_l]$ (which, modulo the mapping π, is the same as a tuple of subsets of Ω that satisfies Φ) and consider the random experiment consisting in drawing a solution at random from the uniform distribution over the set of all solutions (assumed, of course, to be nonempty) and executing M on the resulting labels. For all $u \in V^*$, let X_u be a random variable equal to the state assigned by M to u in this experiment. We can assume without loss of generality that the state of each vertex in V^* includes complete information about its (non)membership in $A_1, \ldots, A_l$, so that our task is to sample from the joint distribution of the random variables X_u, where $u \in V^*$.

Recall that we identified the state set of M with $\{1, \ldots, m\}$. It follows from the proof of Theorem 3.1 that for $u \in V^*$ and $j = 1, \ldots, m$, the jth component N_{uj} of the state vector assigned to u by M' equals the number of tuples of subsets of the set of descendants of u in T^* that cause M to assign the state j to u. In particular, $\Pr(X_{r^*} = j) = N_{r^*j} / \sum_{p=1}^{m} N_{r^*p}$ for $j = 1, \ldots, m$, where r^* is the root of T^*. It is easy to see that if v and w are distinct children of a vertex u in T^* and $j, p, q \in \{1, \ldots, m\}$ such that $\Pr(X_u = j) > 0$, then $\Pr(X_v = p \wedge X_w = q \mid X_u = j) = 0$ if $\delta(p, q, (P_1(u), \ldots, P_k(u), b_1, \ldots, b_l)) \neq j$, where $b_1, \ldots, b_l$ are bits representing the (non)membership of u in $A_1, \ldots, A_l$ implied by u being in state j. We claim that if $\delta(p, q, (P_1(u), \ldots, P_k(u), b_1, \ldots, b_l)) = j$, then $\Pr(X_v = p \wedge X_w = q \mid X_u = j) = N_{vp} N_{wq} / N_{uj}$. Moreover, this relation remains valid if the condition $(X_u = j)$ is replaced by any condition of the form $(X_u = j \wedge D)$ with positive probability, where the event D is independent of X_z for all descendants z of u in T^*. To see this, note that for any setting with $X_u = j$ with positive probability of the variables X_z, where $z \in V^*$, varying X_z arbitrarily for all proper descendants z of u while keeping the remaining values fixed generates exactly N_{uj} solutions, of which $N_{vp} N_{wq}$ additionally have $X_v = p$ and $X_w = q$.

In order to draw a value from the joint distribution of the variables X_u, where $u \in V^*$, we first carry out the algorithm of Theorem 3.1 in order to compute N_{uj} for all $u \in V^*$ and $j = 1, \ldots, m$, after which the distribution of X_{r^*} and the conditional joint distributions of X_v and X_w determined above are readily available. We then draw a random value x_{r^*} from the distribution of X_{r^*}. Finally we process T^* from the root to the leaves, at each node u with two children v and w computing random values x_v and x_w for X_v and X_w from the known value of X_u according to the relevant conditional distribution; the value for a nonroot node without a sibling is obtained by pretending the existence of a sibling w with $(N_{w1}, \ldots, N_{wm}) = (1, 0, 0, \ldots, 0)$ (recall that the initial state of M is identified with 1). Writing out the probability of obtaining a particular collection of values x_u, for $u \in V^*$, one can see that this yields the correct distribution.

A similar algorithm with a similar proof, using the approach of Example 7 above, allows us to draw random tuples from the distribution obtained by repeatedly letting a step into A_i with some probability $f_i(a)$, for $a \in \Omega$ and $i = 1, \ldots, l$, all such random choices being independent, until the resulting tuple $(A_1, \ldots, A_l)$ satisfies $G \models \Phi[A_1, \ldots, A_l]$.

4. Data Structures for Queries. In this section we describe data structures that support queries efficiently, but not updates. Given an MS formula Φ without free simple variables, a commutative semiring $\mathcal{R}$, and an integer constant $d \geq 1$, the d-dimensional *RMS query problem* defined by Φ and $\mathcal{R}$ is, given a graph G appropriate for $(\Phi, \mathcal{R})$, to preprocess G for subsequent queries for quantities of the form $|G'|_{\Phi,\mathcal{R}}$, where G' is obtained from G by (temporarily) changing at most d boolean and/or ring attributes.

Let the tree T^* and the machines M and M' be as in the proof of Theorem 3.1 and again consider a vertex $u \in V^*$ with left and right children v and w, respectively. Let $(r_1, \ldots, r_m)$, $(s_1, \ldots, s_m)$, and $(t_1, \ldots, t_m)$ be the states assigned by M' to u, v, and w, respectively. Then, by definition of the transition function δ', we have $r_j = \bigoplus_{p=1}^{m} (c_{jp} \otimes s_p)$ for $j = 1, \ldots, m$, where

$$c_{jp} = \bigoplus_{q=1}^{m} \bigoplus_{\substack{(b_1,\ldots,b_l)\in\{false,true\}^l \\ \delta(p,q,(P_1(u),\ldots,P_k(u),b_1,\ldots,b_l))=j}} \left(t_q \otimes \bigotimes_{\substack{1\leq i\leq l \\ b_i=true}} f_i(u) \right),$$

for $p = 1, \ldots, m$. In other words, provided that the state of w remains constant, the function that maps the state of v to the state of u is premultiplication with an $m \times m$ matrix (over the semiring $\mathcal{R}$). We call this function the *relay function* of the edge $\{u, v\}$ (for the input graph G). The relay functions of edges between vertices and their right children are defined in complete analogy and have the same form.

Suppose that G is changed into G' by modifying a single boolean or ring attribute of some vertex or edge $a \in \Omega$. This translates into a change of a single boolean or ring attribute of $v = \pi(a)$ in T^* or, as seen from the point of view of M', into a change of the label of v. We can compute $|G'|_{\Phi,\mathcal{R}}$ by simulating the execution of M' on the new label settings. One way to do this is to compose the relay functions of all edges on the path from v to the root r^* of T^*, and then to apply the resulting function to the new state of v; this yields the new state of the root, from which $|G'|_{\Phi,\mathcal{R}}$ can be computed in constant time. Similarly, a query that changes the labels of two vertices v and w can be handled by composing relay functions along the two paths from v and w to the children of the lowest common ancestor (LCA) u of v and w, using the result to compute the new state of u, and then propagating the change to r^* by composing the relay functions on the path from u to r^*. Answering queries therefore essentially reduces to composing functions along paths in T^*, a problem that has been studied in a more general setting.

We call a semigroup $\mathcal{S} = (S, \odot)$ *efficient* if $a \odot b$ can be computed from a and b in constant time (by a single processor) for all $a, b \in S$. We can assume without loss of generality that $\mathcal{S}$ contains a neutral element. Define a *bidirected tree* to be a directed graph whose underlying undirected graph is a tree and that, with each edge (u, v), contains also the reverse edge (v, u). In the context of a bidirected tree T, each of whose edges is labeled by an element of a semigroup $(S, \odot)$ called its *weight*, we define the weight of a path in T of length k as the neutral element of $\mathcal{S}$ if $k = 0$, and otherwise as the quantity $\lambda_1 \odot \cdots \odot \lambda_k$, where λ_i is the weight of the ith edge on the path, for $i = 1, \ldots, k$, and we define a *path-weight query* as a query that specifies two vertices u and v and asks for the weight of the (unique) path in T from u to v. The following lemma was proved by Chazelle [15].

LEMMA 4.1. *For all integers $n, k \geq 1$, an n-vertex bidirected tree with edge weights drawn from an efficient semigroup $(S, \odot)$ can be preprocessed for path-weight queries with preprocessing time $O(nI_k(n))$ and query time $O(k)$.*

A particularly interesting special case of preprocessing time $O(n)$ and query time $O(\alpha(n))$ is obtained by choosing $k = \alpha(n)$. The lemma can be applied to undirected trees as well as bidirected trees by replacing each undirected edge $\{u, v\}$ by the two directed edges (u, v) and (v, u), each with the same weight as $\{u, v\}$. Similar remarks apply in the following.

THEOREM 4.1. *For all integer constants $t \geq 1$ and all integers $n, k \geq 1$, every t-dimensional RMS query problem defined by an MS formula Φ without free simple variables and an efficient commutative semiring $\mathcal{R}$ can be solved on n-vertex input graphs appropriate for $(\Phi, \mathcal{R})$ and of treewidth bounded by t with preprocessing time $O(nI_k(n))$ and query time $O(k)$.*

PROOF. We preprocess the tree T^* of the proof of Theorem 3.1 according to Lemma 4.1, the weight of each edge being its relay function and $\odot$ being function composition (i.e., matrix multiplication over $\mathcal{R}$). We also preprocess T^* so that subsequent queries for the LCA of two arbitrary vertices can be answered in constant time; it is known how to do this in $O(n)$ time [24], [30].

Suppose that a query changes the labels of the vertices in some set $U \subseteq V^*$ (thus $|U| \leq t$). Let $Q = U \cup W \cup \{r^*\}$, where W is the set of all LCAs of two vertices in U and r^* is the root of T^*; Q is still of bounded size. Let $T = (V, E)$ be the tree obtained from T^* by contracting each vertex in $V^* \setminus Q$ into its closest ancestor whose parent belongs to Q. With the aid of still more LCA queries, to determine for all $u, v \in Q$ whether u is an ancestor of v in T^*, T can be constructed in constant time. We now process T from the leaves to the root, for each vertex v in T computing the new state assigned to v by M' after the label changes caused by the update. For a vertex v in Q, this is trivial, since the new states of its children in T^*, if any, will be known when v is processed. For a vertex v in $V \setminus Q$, on the other hand, all descendants of v in T^* that belong to U, if any, are descendants of a single vertex in Q whose new state is known when v is processed. Thus the new value of v can be computed in $O(k)$ time by composing relay functions according to Lemma 4.1. Once the new state of r^* is known, the query can be answered in constant time. □

If the algorithm of Theorem 4.1 is used to solve construction problems with the approach described in Example 6 in Section 3, each query will add new vertices to the graph H. In order to ensure that the size of H remains $O(n)$, the queries can be extended to clean up after themselves by restoring H to its former state.

Again, a parallelization is effortless. Since queries can be answered in nearly constant time by one processor anyway, it is natural to limit the parallelization to the preprocessing. Suitable EREW PRAM algorithms for preprocessing for path-weight and LCA queries, working in $O(\log n)$ time using as many operations as the corresponding sequential algorithms, up to constant factors, were given by Alon and Schieber [1] and Schieber

and Vishkin [30], respectively (see also [23]). All other parts of the preprocessing either parallelize trivially or were already discussed in the proof of Theorem 3.2.

THEOREM 4.2. *For all integer constants $t \geq 1$ and all integers $n, k \geq 1$, every t-dimensional RMS query problem defined by an MS formula Φ without free simple variables and an efficient commutative semiring $\mathcal{R}$ can be solved on each n-vertex input graph G appropriate for $(\Phi, \mathcal{R})$ and accompanied by an $O(n)$-node tree decomposition of G of width at most t on an EREW PRAM with preprocessing time $O(\log n)$, preprocessing operations $O(nI_k(n))$, and sequential query time $O(k)$.*

Queries that (temporarily) insert and/or delete a constant number of vertices of bounded degree and/or edges can be handled within the framework of this section. In order to allow a query to insert a single undirected edge, e.g., one would introduce predicates corresponding to sets *Endpoints*, initialized to $\varnothing$, and *NewEdge*, permanently equal to $\{e^*\}$, where e^* is a dummy edge that is not part of the input graph. A query that wishes to insert an edge $\{v, w\}$ would set $\mathit{Endpoints} = \{v, w\}$ and store the boolean and ring attributes of the new edge with e^*, and an assertion such as $\mathit{incident}(u, e)$ would be replaced by

$$(e \notin \mathit{NewEdge} \wedge \mathit{incident}(u, e)) \vee (e \in \mathit{NewEdge} \wedge u \in \mathit{Endpoints}).$$

In this manner we can solve problems such as "Given a 3-colorable graph of bounded treewidth, can 3-colorability be destroyed through the insertion of a single edge?"

5. Exhaustive Queries. This section shows that exhaustive queries can be answered in time linear in the size of the output. Given an MS formula Φ without free simple variables, a commutative semiring $\mathcal{R}$, and an integer constant $d \geq 1$, the d-dimensional *RMS exhaustive-query problem* defined by Φ and $\mathcal{R}$ is, given a graph G appropriate for $(\Phi, \mathcal{R})$, to compute all quantities of the form $|G'|_{\Phi,\mathcal{R}}$, where G' differs from G in at most d boolean and no ring attributes.

LEMMA 5.1. *Given an n-vertex undirected tree $T = (V, E)$ with root r and with edges labeled by weights drawn from an efficient semigroup, the weight $\Lambda(u)$ of the path in T from u to r can be computed for all $u \in V$ in $O(n)$ time.*

PROOF. Carry out a depth-first search of T, starting at r. When a vertex $v \neq r$ is reached for the first time via an edge (u, v), compute $\Lambda(v)$ in constant time from $\Lambda(u)$. □

THEOREM 5.1. *For all integer constants $d, t \geq 1$ and all integers $n \geq 1$, every d-dimensional RMS exhaustive-query problem defined by an MS formula Φ without free simple variables and an efficient commutative semiring $\mathcal{R}$ can be solved in $O(n^d)$ time on n-vertex input graphs appropriate for $(\Phi, \mathcal{R})$ and of treewidth at most t.*

PROOF. It suffices to prove the claim for $d = 1$, since an exhaustive d-dimensional query, for $d \geq 2$, can be carried out as n^{d-1} independent exhaustive one-dimensional queries, each with its own $O(n)$-time initialization and with $d - 1$ query variables

considered as constants. For $d = 1$, compute the relay functions of all edges in T^* as described in the previous section. The claim now follows easily by an application of Lemma 5.1, the weight of each edge being its relay function and the semigroup operation being function composition. □

The following parallelization is straightforward.

THEOREM 5.2. *For all integer constants $d, t \geq 1$ and all integers $n \geq 2$, every d-dimensional RMS exhaustive-query problem defined by an MS formula Φ without free simple variables and an efficient commutative semiring $\mathcal{R}$ can be solved on an EREW PRAM using $O(\log n)$ time and $O(n^d)$ operations on each n-vertex input graph G appropriate for $(\Phi, \mathcal{R})$ and accompanied by an $O(n)$-node tree decomposition of G of width at most t.*

6. Dynamic Data Structures. In this section we dynamize the path-weight-query data structure of Lemma 4.1 to allow updates of edge weights as well as path-weight queries, state the implications for dynamic RMS problems, and finally describe an alternative simple solution to a special case of the latter.

THEOREM 6.1. *For all integers $n, k, \tau \geq 1$, an n-edge tree with edge weights drawn from an efficient semigroup $(S, \odot)$ can be preprocessed for path-weight queries with preprocessing time $O(nI_k(n))$, query time $O(\tau + k)$, and update time $O(\tau n^{1/\tau})$.*

PROOF. We reuse part of a scheme developed by Chazelle [15] in order to prove Lemma 4.1. For a parameter m with $1 \leq m \leq n$ to be chosen below, we partition the edge set E of the input tree $T = (V, E)$ into at most $3n/m$ sets, each of which spans a subtree of T, called a *piece*, with at most m edges. Chazelle shows how to do this in $O(n)$ time (Lemma 3 in [15]). Call a vertex of T a *fringe vertex* if it is shared between two or more pieces. In order to make what follows clearer, assume that we separate the pieces by replacing each fringe vertex v, shared between d pieces, by a star consisting of a central vertex, which we identify with v, connected to d new vertices; each of the d new vertices is associated with a different piece containing v, is called the *representative* of v in that piece, and replaces v as an endpoint of each edge belonging to the piece and incident on v (see Figure 1). Provided that each star edge is given a weight of $\bar{0}$, the neutral element of $(S, \odot)$, this transformation does not change the weight of the path between any two vertices in T. It at most triples the number of edges and is easily carried out in $O(n)$ time.

The number of fringe vertices is bounded by $3n/m$, and Chazelle shows how to construct an edge-weighted bidirected tree T^* with at most $12n/m$ edges that contains all fringe vertices and assigns the same weight as T to the path between any two fringe vertices; T^* is obtained in $O(n)$ time from T by removing all nonfringe vertices that have fewer than three incident edges lying on paths between fringe vertices and replacing paths of such vertices by single edges with the same weight.

Each piece is preprocessed independently for path-weight queries, and the global tree T^* is processed recursively as just described. For all $u, v \in V$, denote by $\Lambda(u, v)$ the

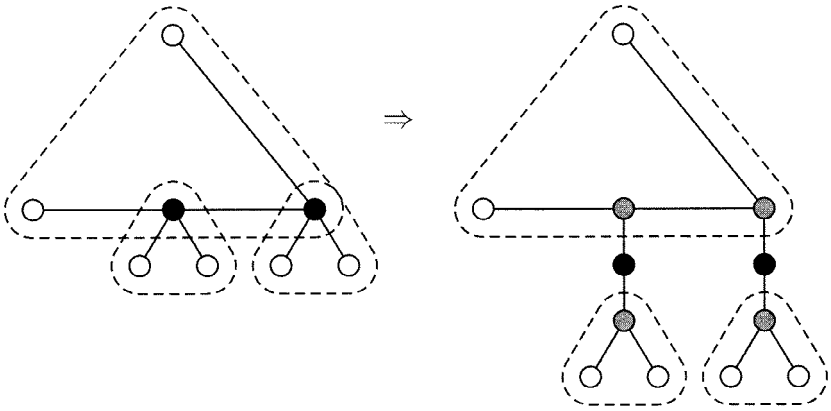

Fig. 1. Separation of the pieces, indicated with dashed lines. Fringe vertices and their representatives are shown in black and gray, respectively.

weight of the path in T from u to v. Consider two vertices u and v in T and let x and y be the first and the last fringe vertices on the path in T from u to v, respectively, if any. If x and y do not exist, u and v belong to the same piece, and the weight λ of the path from u to v can be obtained from the data structure maintained for that piece. Otherwise $\lambda = \Lambda(u,x) \odot \Lambda(x,y) \odot \Lambda(y,v)$. If $x = u$ (u is a fringe vertex), $\Lambda(u,x) = \bar{0}$; otherwise $\Lambda(u,x) = \Lambda(u,r_x)$, where r_x is the representative of x in the piece of u, and the latter quantity can be obtained from the data structure maintained for the piece of u. $\Lambda(y,v)$ is computed similarly, and $\Lambda(x,y)$ is obtained recursively from the data structures maintained for T^*. One small issue, how to determine x and y and possibly r_x and r_y, is resolved with the help of yet another tree T^+, obtained from T by replacing all edges within each piece by edges from each (nonfringe) vertex in the piece to a new vertex representing the piece. The vertices of interest occur among the first four and the last four vertices on the path in T^+ from u to v and can be identified by two applications of the algorithm of Lemma 4.1: the weight of each edge is its identity, considered as a string of length 1, and $\odot$ is "truncated concatenation," which concatenates its two arguments but, if the resulting string is of length ≥ 4, keeps only its suffix of length 3.

Without loss of generality assume that $k \geq 2$. On the first recursive level we choose $m = m_0 = \lceil \sqrt{I_1(n)} \rceil$ and preprocess the pieces for path-weight queries according to Lemma 4.1. This needs a total of $O(nI_k(n))$ time and provides a query time of $O(k)$. On all subsequent recursive levels we choose $m = m_1 = \max\{\lceil n^{1/(2\tau)} \rceil, 24\}$ and preprocess the pieces for path-weight queries according to Lemma 4.1 with $k = 2$, ending the recursion when the number of edges drops below 24. This provides a query time of $O(1)$ per recursive level, and since $m_0 = \Omega(I_2(n))$, the preprocessing effort sums to $O(n)$ over all levels. Because the recursive depth is $O(\log n / \log m_1) = O(\tau)$, the overall query time is $O(\tau + k)$. An update of an edge weight requires recomputation of data structures maintained for a single piece on each recursive level, and thus needs $O(m_0 I_k(n))$ time on the first level and $O(m_1 I_2(m_1))$ time on all subsequent levels, resulting in an overall update time of $O(\tau n^{1/\tau})$. □

Given an MS formula Φ without free simple variables, a commutative semiring $\mathcal{R}$, and an integer constant $d \geq 1$, the d-dimensional *dynamic RMS problem* defined by Φ and $\mathcal{R}$ is, given a graph G appropriate for $(\Phi, \mathcal{R})$, to preprocess G for subsequent updates of single boolean or ring attributes and queries for quantities of the form $|G'|_{\Phi,\mathcal{R}}$, where G' is obtained from (the current) G by (temporarily) changing at most d boolean and/or ring attributes. As an immediate consequence of Theorem 6.1 and the methods introduced in Section 4, we obtain the following result.

THEOREM 6.2. *For all integer constants $t \geq 1$ and all integers $n, k, \tau \geq 1$, every t-dimensional dynamic RMS query problem defined by an MS formula Φ without free simple variables and an efficient commutative semiring $\mathcal{R}$ can be solved on n-vertex input graphs appropriate for $(\Phi, \mathcal{R})$ and of treewidth bounded by t with preprocessing time $O(nI_k(n))$, query time $O(\tau + k)$, and update time $O(\tau n^{1/\tau})$.*

In the case of construction problems solved with the algorithm of Theorem 6.2, the simple cleaning-up approach outlined in Section 4 for keeping the size of the graph H under control no longer suffices. Still, by keeping reference counts and using standard techniques for on-the-fly garbage collection, it is possible to ensure that the size of H remains $O(n)$. We omit the details.

Theorem 6.2 describes a tradeoff going from query time $O(\alpha(n))$ and update time $O(n)$ to query and update times both $O(\log n)$. The last point on the tradeoff curve can be realized in a much simpler way, as indicated by Bodlaender [8]. The idea is to apply the methods of Section 4 to a tree T^* of depth $O(\log n)$, obtained by starting the construction from a balanced tree decomposition of the input graph (recall that a balanced tree decomposition is one of logarithmic depth). Queries are answered by composing relay functions from the affected vertices to the root in a straightforward manner, and an update is handled by recomputing all relay functions along the single affected path.

Except for the division of T into pieces, all parts of the preprocessing and update algorithms of Theorem 6.1 parallelize using standard techniques (in particular, tree contraction) to spend $O(\log n)$ time on each recursive level and as many operations as the sequential algorithms; we omit the details. As for the division of T into pieces, we can use a slight modification of the division into m-bridges described in Section 3.3.5 of [28]. We apply the modified algorithm to a binary tree obtained by replacing each vertex v with $d \geq 3$ children by a tree with $d - 1$ vertices, connected to the descendants of v via d edges, whose root is identified with v. Labeling each original vertex present in T with 1 and each dummy vertex introduced in the binarization with 0, we define $W(v)$ for each vertex v as the sum of the labels of the descendants of v. (The original algorithm defines $W(v)$ to be the number of descendants of v.) We then continue with the algorithm of [28], which divides the binary tree into $O(n/m)$ edge-disjoint subtrees, the vertex labels of each of which sum to at most $m+1$ (the label of a vertex may be counted in several subtrees). This division of the binary tree induces the required division of T into pieces (except that the number of pieces may be larger, by a constant factor, than in the sequential case), and we obtain the following result.

THEOREM 6.3. *For all integer constants $t \geq 1$ and all integers $n, k, \tau \geq 2$, every t-dimensional dynamic RMS problem defined by an MS formula Φ without free simple*

variables and an efficient commutative semiring $\mathcal{R}$ can be solved on each n-vertex input graph G appropriate for $(\Phi, \mathcal{R})$ and accompanied by an $O(n)$-node tree decomposition of G of width at most t on an EREW PRAM with preprocessing time $O(\tau \log n)$, preprocessing operations $O(nI_k(n))$, sequential query time $O(\tau + k)$, update time $O(\tau \log n)$, and update operations $O(\tau n^{1/\tau})$.

7. Distances and Shortest Paths. This section contains some special considerations concerning the computation of distances and shortest paths that do not come out of the general framework.

A *negative cycle* in a graph G with real edge weights is a simple cycle in G, the total weight of whose edges is negative. We say that a pair (s, t) of vertices is *affected by* a negative cycle C if there is a path from s to some vertex on C and a path from some vertex on C to t. Our discussion of queries about the distance from s to t until now implicitly assumed that (s, t) is not affected by any negative cycle; exactly then does G contain a shortest walk (allowing repeated vertices) from s to t, whose length, by definition, is the distance from s to t. The traditional shortest-path problem in general does not ask for a shortest path, but for a shortest walk, and for an appropriate message if no shortest walk exists. What our algorithms actually compute, however, is a shortest (simple) *path* from s to t or its length, which is also a shortest walk from s to t or its length only if (s, t) is not affected by a negative cycle. This observation motivates the present section, which shows how to use our methods to solve the shortest-path problem in its usual formulation.

In order to test whether (s, t) is affected by a negative cycle, we can maintain, along with G, a second graph G' consisting of three disjoint copies of G, *Layers* 1–3, together with directed edges of zero length from each vertex in Layer i to the corresponding vertex in Layer $i + 1$, for $i = 1, 2$. Queried about the length of a shortest path from s to t in G (or about the path itself), we also query about the length of a shortest path from s_1 to t_3 in G', where s_1 and t_3 are the vertices corresponding to s in Layer 1 and to t in Layer 3, respectively. It is easy to see that the two answers coincide if and only if (s, t) is not affected by a negative cycle in G, which furnishes the desired test, since a negative cycle in G affecting (s, t) can be "traversed" at least once in G' by switching to different layers during the traversal of the cycle.

Algorithms for computing distances and shortest paths traditionally give up once they detect the presence of a negative cycle, independently of whether the negative cycle actually affects the vertex pairs occurring in queries; e.g., the algorithms in [13] work in this manner. The algorithm discussed above, in contrast, answers all meaningful queries and indicates the nonexistence of a shortest walk in the remaining cases. The traditional behavior does not seem very convenient in a practical setting. If that convention is desired, however, we can easily support it, since the presence of a negative cycle can be formulated as an RMS problem followed by a test, so that the initial presence of a negative cycle can be detected in linear time on graphs of bounded treewidth, while a new negative cycle created by decreasing the length of an edge (u, v) certainly affects (u, v), so that the creation of a negative cycle can be detected through a usual query.

Given an edge-weighted directed graph $G = (V, E, \Delta)$, a vertex $s \in V$, and the length $d(v)$ of a shortest walk in G from s to v for all v in the set V' of vertices for which

such a shortest walk exists, it is easy to create a shortest-path tree T for s in G, i.e., an out-tree on the vertex set V' rooted at s in which the path from s to each vertex $v \in V'$ is a shortest path from s to v in G. If $l(u, v)$ denotes the length of the edge (u, v), for all $(u, v) \in E$, one simply constructs the graph $G' = (V', E')$, where E' is the subset of E of those edges (u, v) with $d(v) = d(u) + l(u, v)$, and takes T to be the tree of any search (such as depth-first search) of G' starting at s.

References

[1] N. Alon and B. Schieber, Optimal preprocessing for answering on-line product queries, Tech. Rep. No. 71/87, Tel Aviv University, 1987.

[2] S. R. Arikati, S. Chaudhuri, and C. D. Zaroliagis, All-pairs min-cut in sparse networks, in *Proc.* 15*th Conference on Foundations of Software Technology and Theoretical Computer Science* (P. S. Thiagarajan, ed.), Lecture Notes in Computer Science, Vol. 1026, Springer-Verlag, Berlin, 1995, pp. 363–376.

[3] S. Arnborg, J. Lagergren, and D. Seese, Easy problems for tree-decomposable graphs, *J. Algorithms*, **12** (1991), 308–340.

[4] M. W. Bern, E. L. Lawler, and A. L. Wong, Linear-time computation of optimal subgraphs of decomposable graphs, *J. Algorithms*, **8** (1987), 216–235.

[5] H. L. Bodlaender, Dynamic programming on graphs with bounded treewidth, in *Proc.* 15*th International Colloquium on Automata, Languages and Programming* (T. Lepistö and A. Salomaa, eds.), Lecture Notes in Computer Science, Vol. 317, Springer-Verlag, Berlin, 1988, pp. 105–118.

[6] H. L. Bodlaender, NC algorithms for graphs with small treewidth, in *Proc.* 14*th International Workshop on Graph-Theoretic Concepts in Computer Science* (J. van Leeuwen, ed.), Lecture Notes in Computer Science, Vol. 344, Springer-Verlag, Berlin, 1989, pp. 1–10.

[7] H. L. Bodlaender, On reduction algorithms for graphs with small treewidth, in *Proc.* 19*th International Workshop on Graph-Theoretic Concepts in Computer Science* (J. van Leeuwen, ed.), Lecture Notes in Computer Science, Vol. 790, Springer-Verlag, Berlin, 1994, pp. 45–56.

[8] H. L. Bodlaender, Dynamic algorithms for graphs with treewidth 2, in *Proc.* 19*th International Workshop on Graph-Theoretic Concepts in Computer Science* (J. van Leeuwen, ed.), Lecture Notes in Computer Science, Vol. 790, Springer-Verlag, Berlin, 1994, pp. 112–124.

[9] H. L. Bodlaender, A linear-time algorithm for finding tree-decompositions of small treewidth, *SIAM J. Comput.*, **25** (1996), 1305–1317.

[10] H. L. Bodlaender, A partial k-arboretum of graphs with bounded treewidth, *Theoret. Comput. Sci.*, **209** (1998), 1–45.

[11] H. L. Bodlaender and T. Hagerup, Parallel algorithms with optimal speedup for bounded treewidth, *SIAM J. Comput.*, **27** (1998), 1725–1746.

[12] R. B. Borie, R. G. Parker, and C. A. Tovey, Automatic generation of linear-time algorithms from predicate calculus descriptions of problems on recursively constructed graph families, *Algorithmica*, **7** (1992), 555–581.

[13] S. Chaudhuri and C. D. Zaroliagis, Shortest path queries in digraphs of small treewidth, in *Proc.* 22*nd International Colloquium on Automata, Languages and Programming* (Z. Fülöp and F. Gécseg, eds.), Lecture Notes in Computer Science, Vol. 944, Springer-Verlag, Berlin, 1995, pp. 244–255.

[14] S. Chaudhuri and C. D. Zaroliagis, Shortest paths in digraphs of small treewidth. Part II: Optimal parallel algorithms, *Theoret. Comput. Sci.*, **203** (1998), 205–223.

[15] B. Chazelle, Computing on a free tree via complexity-preserving mappings, *Algorithmica*, **2** (1987), 337–361.

[16] R. F. Cohen, S. Sairam, R. Tamassia, and J. S. Vitter, Dynamic algorithms for optimization problems in bounded tree-width graphs, in *Proc.* 3*rd Integer Programming and Combinatorial Optimization Conference* (G. Rinaldi and L. A. Wolsey, eds.), Ciaco, Louvain-la-Neuve, 1993, pp. 99–112.

[17] R. Cole and U. Vishkin, Deterministic coin tossing with applications to optimal parallel list ranking, *Inform. and Control*, **70** (1986), 32–53.

[18] B. Courcelle, Graph rewriting: an algebraic and logic approach, in *Handbook of Theoretical Computer Science*, *Vol. B*: *Formal Models and Semantics* (J. van Leeuwen, ed.), Elsevier, Amsterdam and The MIT Press, Cambridge, MA, 1990, Chap. 5, pp. 193–242.
[19] B. Courcelle, The monadic second-order logic of graphs. I. Recognizable sets of finite graphs, *Inform. and Comput.*, **85** (1990), 12–75.
[20] B. Courcelle and M. Mosbah, Monadic second-order evaluations on tree-decomposable graphs, *Theoret. Comput. Sci.*, **109** (1993), 49–82.
[21] G. N. Frederickson, Maintaining regular properties dynamically in *k*-terminal graphs, Preprint, 1995.
[22] A. Habel, H.-J. Kreowski, and W. Vogler, Decidable boundedness problems for sets of graphs generated by hyperedge-replacement, *Theoret. Comput. Sci.*, **89** (1991), 33–62.
[23] T. Hagerup, Parallel preprocessing for path queries without concurrent reading, *Inform. and Comput.*, to appear.
[24] D. Harel and R. E. Tarjan, Fast algorithms for finding nearest common ancestors, *SIAM J. Comput.*, **13** (1984), 338–355.
[25] J. JáJá, *An Introduction to Parallel Algorithms*, Addison-Wesley, Reading, MA, 1992.
[26] W. Paul, U. Vishkin, and H. Wagener, Parallel computation on 2-3-trees, *RAIRO Theoret. Informatics*, **17** (1983), 397–404.
[27] V. Radhakrishnan, H. B. Hunt III, and R. E. Stearns, Efficient algorithms for solving systems of linear equations and path problems, in *Proc. 9th Annual Symposium on Theoretical Aspects of Computer Science* (A. Finkel and M. Jantzen, eds.), Lecture Notes in Computer Science, Vol. 577, Springer-Verlag, Berlin, 1992, pp. 109–119.
[28] M. Reid-Miller, G. L. Miller, and F. Modugno, List ranking and parallel tree contraction, in *Synthesis of Parallel Algorithms* (J. H. Reif, ed.), Morgan Kaufmann, San Mateo, CA, 1993, Chap. 3, pp. 115–194.
[29] N. Robertson and P. D. Seymour, Graph Minors. II. Algorithmic aspects of tree-width, *J. Algorithms*, **7** (1986), 309–322.
[30] B. Schieber and U. Vishkin, On finding lowest common ancestors: simplification and parallelization, *SIAM J. Comput.*, **17** (1988), 1253–1262.
[31] R. E. Stearns and H. B. Hunt III, An algebraic model for combinatorial problems, *SIAM J. Comput.*, **25** (1996), 448–476.
[32] R. E. Tarjan and U. Vishkin, An efficient parallel biconnectivity algorithm, *SIAM J. Comput.*, **14** (1985), 862–874.
[33] J. van Leeuwen, Graph algorithms, in *Handbook of Theoretical Computer Science, Vol. A*: *Algorithms and Complexity* (J. van Leeuwen, ed.), Elsevier, Amsterdam and The MIT Press, Cambridge, MA, 1990, Chap. 10, pp. 525–631.