Distributed Computation of Web Queries using Automata

Marc Spielmann
Departement WNI
University of Limburg
Universitaire Campus
B-3590 Diepenbeek, Belgium
marc.spielmann@luc.ac.be

Jerzy Tyszkiewicz[*]
Institute of Informatics
Warsaw University
Banacha 2
02-097 Warsaw, Poland
jty@mimuw.edu.pl

Jan Van den Bussche
Departement WNI
University of Limburg
Universitaire Campus
B-3590 Diepenbeek, Belgium
jan.vandenbussche@luc.ac.be

ABSTRACT

We introduce and investigate a distributed computation model for querying the Web. Web queries are computed by interacting automata running at different nodes in the Web. The automata which we are concerned with can be viewed as register automata equipped with an additional communication component. We identify conditions necessary and sufficient for systems of automata to compute Web queries, and investigate the computational power of such systems.

1. INTRODUCTION

Much attention has recently been paid to querying the Web [5]. A salient feature of queries and computations on the Web is their browsing nature: unlike a conventional database, the Web is usually explored navigationally, starting from a particular node in the Web (e.g., the user's homepage). This has led Abiteboul and Vianu [2] to formally define a *Web query* as a mapping from pairs $(\mathcal{I}, s)$ to sets of nodes in $\mathcal{I}$, where $\mathcal{I}$ is a Web instance and s is the *source of the query*, i.e., the node from which we start exploring the Web. Various kinds of machines specially tailored for computations on the Web have been introduced and studied by Abiteboul and Vianu in this context, in particular *browser machines*, which can be viewed as Turing machines navigating the Web by following links.

Another recent development is that of Internet supercomputing [6, 7], where many individual computers linked to the Internet collaborate in a distributed computation. An appealing and popular example is the SETI@home project, which scans radio signals from space for signs of extraterrestrial intelligence [12].

In this paper, we combine these two lines of research. More precisely, we investigate the possibilities and limitations of *Web automata*, a computation model for querying the Web, which is—like the browser machine model—purely navigational, but which is also distributed. Starting at a given source node, a finite portion of the Web, reachable from the source by following links, is populated with lightweight processes. Typically, this portion is determined by specifying a maximum number of links that can be followed (as commonly done in tools for off-line browsing and Web mirroring). The processes run concurrently and follow a program specified essentially as a finite register automaton [11]. They report back to the source process by sending messages upwards along the edges of a spanning tree, a standard network topology used in computer networks and distributed computation [3, 18].

Our investigation offers the following contributions:

(i) We define a fair, efficient, and easy to enforce communication protocol by which distributed computations proceed in rounds. Each round of a distributed computation has a layered structure according to the levels of a spanning tree. The Web automata (or, processes) at each level of the spanning tree run concurrently, and each level takes only constant parallel time. After each round, the source automaton, i.e., the Web automaton running at the source node, is guaranteed to have received enough data in order to decide whether to continue for another round, or to terminate the computation. In addition, the source automaton may produce output. We identify a decidable property of Web automata, called *productivity*, which enables this protocol. Testing productivity is PSPACE-complete.

(ii) Since the order of upward communications within a layer is not fixed, there may exist many different distributed runs for a given spanning tree. On top of that, the spanning tree itself arises out of the computation and is thus not a priori fixed. We call a Web automaton *sound* if it produces the same output for every possible distributed run on every possible spanning tree. Every sound Web automaton computes a well-defined Web query. We show that soundness is undecidable. (This is not entirely evident, given the finite nature of Web automata and the rather rigid communication protocol which they must follow.)

(iii) Although a sound Web automaton computes a Web query, one may have to wait many rounds before seeing any *new* output. This can be quite undesirable in practice. We call a Web automaton *continuous* if it produces

[*]Supported by Polish KBN grant 7 T11C 007 21

new output in every round. Note the analogy with Abiteboul and Vianu's distinction between *finitely* and *eventually* computable Web queries: in the case of a query which is only eventually computable, one can never be sure that the entire output has actually been output. Note also that a continuous Web automaton which computes a boolean query (i.e., a query with yes/no answers) already knows the correct answer after the first round. We show that continuity is also undecidable.

(iv) Assuming that at every node there is an ordering of the outgoing links available (a very natural assumption in the context of the Web), we can show that every logarithmic-space computable Web query is computable by a Web automaton.

(v) We furthermore introduce *decide-and-forward automata*, which constitute a natural, syntactic subclass of Web automata. A decide-and-forward (DF) automaton makes all the crucial decisions already after the first round; the subsequent rounds are pure forwarding rounds which merely flush the remaining contents of the communication queues to the output. We show that soundness and continuity of DF automata becomes decidable in the monadic case, i.e., when the tests performed by automata are based on unary predicates of Web nodes only. We also give a characterization of the Web queries continuously computable by monadic DF automata in terms of a fragment of first-order logic.

(vi) Finally, we study *browser stack machines*, a restricted variant of Abiteboul and Vianu's browser machines. Our restricted browser machines have only a finite work memory and use their Turing tape in a stack-like manner, similar to the three familiar surf actions of common Web browsers: 'follow this link', 'go back', and 'go forward'. We show that, if the depth of the stack is bounded (so that the machine cannot get 'lost in hyperspace'), then every Web query computable by a browser stack machine is computable by a Web automaton.

Related Work. Distributed Web querying systems (similar to our theoretical model) have already been implemented, e.g., the DIASPORA system [8, 15]. As for theoretical work, we are aware of only few publications on navigational Web querying, most notably the original papers by Abiteboul and Vianu [2], and Mendelzon and Milo [13]. Both papers focus on computational completeness, while we work with a limited computation model and focus on distribution and efficiency. Abiteboul and Vianu also proposed a distributed evaluation algorithm for regular path queries [1]. A very recent proposal of a formal model for Web querying using concurrent agents was made by Sazonov [16]. His model is based not on finite automata but on a set-theoretic term language. Of course, finite automata working over abstract domains (in our case, Web nodes) rather than over finite alphabets have been considered before, e.g., in the study of regular languages over infinite alphabets [11, 14]. Finally, we mention that our definition of distributed runs of Web automata is inspired by Gurevich's definition of partially ordered runs of distributed abstract state machines [9].

Outline. In the next section, we recall the definition of Web queries, slightly adapted for our purposes. In Sections 3 and 4, we introduce our automaton model and define distributed runs of systems of automata. In Sections 5 and 6, we consider automata suitable for computing Web queries and introduce DF automata. We conclude in Section 7 by drawing a connection between Abiteboul and Vianu's browser machines and our computation model.

2. WEB QUERIES

We consider Web queries in the spirit of Abiteboul and Vianu [2]. Since in this paper we are mainly concerned with computations on finite portions of the Web, we focus on Web queries on finite instances of the Web.

Web Instances. In the following, Υ denotes a finite, relational vocabulary containing (at least) the binary relation symbol *Link*. A *Web instance* $\mathcal{I}$ a finite structure over Υ. We view the elements of $\mathcal{I}$ as abstractions of Web pages, Web sites, or other objects on the Web, and call them *nodes*. The ordered pairs in the binary relation $Link^{\mathcal{I}}$ represent links in the Web. The other relations of $\mathcal{I}$ are abstractions of semantic predicates which a Web query may apply to nodes. For example, a unary relation $R_1(x)$ could stand for "Web page x contains the keyword Madison", a binary relation $R_2(x, y)$ could stand for "the link from x to y is labeled Madison", and a ternary relation $R_3(x, y, z)$ could stand for "on page x, all links to y precede all links to z". The vocabulary will thus vary from query to query.

Although a Web instance is nothing but a standard relational database with at least one binary relation, there is a crucial difference between querying a relational database and querying the Web: to answer a Web query, one can examine the 'database' only by following links, starting at some source node. This leads us to the next basic definition.

Web Queries. A *Web query* Q over Υ is a mapping that assigns to every pair $(\mathcal{I}, s)$, where $\mathcal{I}$ is a Web instance over Υ and s is a node in $\mathcal{I}$, a set of nodes in $\mathcal{I}$. The node s is also called the *source* (of the query). Following the standard genericity criterion for database queries, we require that Q preserves isomorphisms: if $(\mathcal{I}', s')$ is isomorphic to $(\mathcal{I}, s)$ via an isomorphism ι, then $Q(\mathcal{I}', s') = \iota(Q(\mathcal{I}, s))$. Furthermore, since we will consider purely navigational computation models only, it is only fair to accordingly require that $Q(\mathcal{I}, s) = Q(\text{Reach}(\mathcal{I}, s), s)$, where $\text{Reach}(\mathcal{I}, s)$ denotes the substructure of $\mathcal{I}$ generated by the nodes reachable from the source s by following links.

Notice that every first-order formula $\varphi(s, x)$ over Υ with $\text{free}(\varphi) = \{s, x\}$ defines a Web query Q_φ over Υ:

$$Q_\varphi(\mathcal{I}, s) := \{n : \text{Reach}(\mathcal{I}, s) \models \varphi[s, n]\}.$$

3. WEB AUTOMATA

We begin the introduction of our computation model by defining

- *Web automata*, a variant of register automata equipped with an additional communication component, and
- runs of Web automata at individual Web nodes, which we refer to as *local runs*.

Distributed runs of systems of Web automata are subject of the next section.

Web Automata. Our automata are specified by simple, rule-based programs defined as follows. Expand Υ to a vocabulary Υ^+ by adding three constant symbols 0, 1, and $\perp$, and a unary relation symbol *Source*. Intuitively, 0 and 1 represent the boolean values *false* and *true*, respectively, $\perp$ denotes the empty queue (of an automaton), and *Source* indicates the source node. Fix some tuple $\bar{r} = (r_1, \ldots, r_\ell)$ of variables (representing the registers of an automaton).

A *guard* (of a rule) is a quantifier-free first-order formula $\varphi(x, y, \bar{r})$ over Υ^+ with $\text{free}(\varphi) \subseteq \{x, y, \bar{r}\}$. As will become clear soon, x will be interpreted as the node at which an automaton is running locally, and y will be interpreted as the head of the automaton's queue. The queue will contain incoming messages sent by automata running at other nodes.

A *rule* is an expression of the form

$$\texttt{if}\ \varphi\ \texttt{then}\ \textit{action}$$

where φ is a guard, and *action* is an *update action* given by an expression of the form $(r_i := t)$, or a *send action* given by an expression of the form $\textit{send}(\bar{t})$. Here, t is a term in $\{x, y, r_1, \ldots, r_\ell, 0, 1\}$, and $\bar{t}$ is a finite (possibly empty) sequence of such terms.

A *program* is now simply a finite set of rules.

Example 1. Suppose that Υ contains a unary relation symbol *Interesting*. Below, we display a program which employs a register r to distinguish between the first computation step and all subsequent steps. In the first step, the program checks whether it is running at an 'interesting' node. If so, it sends a message containing this node. In all subsequent steps, it basically forwards the contents of its queue: in each step, the first item in the queue is sent in a message of length 1, after which the item is removed from the queue. In the program, "*this_node*" and "*head_q*" stand for the variables x and y, respectively.

```
if  $(r = 0) \wedge \textit{Interesting}(\textit{this\_node})$  then  $\textit{send}(\textit{this\_node})$
if  $(r = 0)$  then  $r := 1$
if  $(r = 1) \wedge (\textit{head\_q} \neq \perp)$  then  $\textit{send}(\textit{head\_q})$
if  $(r = 1) \wedge (\textit{head\_q} = \perp)$  then  $\textit{send}()$
```

Definition 1. A *Web automaton A* is a triple $(\Upsilon, \bar{r}, \Pi)$ consisting of a vocabulary Υ, a tuple $\bar{r}$ of (register) variables, and a program Π over Υ^+ and $\bar{r}$.

Next, we define a notion of run which reflects the behavior of a Web automaton when observed at a particular node.

Local Runs. Let $A = (\Upsilon, \bar{r}, \Pi)$ be an automaton, let $\mathcal{I}$ an instance over Υ, and let s be a (source) node in $\mathcal{I}$. Expand $\mathcal{I}$ to a structure $\mathcal{I}^+$ over Υ^+ by adding three new elements 0, 1, and $\perp$, and by interpreting the unary relation symbol *Source* as the singleton set $\{s\}$. In the following, the words *queue* and *message* both refer to a finite sequence of bits and nodes (in $\mathcal{I}$). If q is a queue, then $\text{head}(q)$ denotes the first element of q; the sequence of the remaining elements is denoted by $\text{tail}(q)$. The head of the empty queue is defined to be $\perp$.

Consider a node n in $\mathcal{I}$. A *configuration of A at n* is a triple $(n, q, \bar{a})$ where q is a queue and $\bar{a}$ is a tuple $(a_1, \ldots, a_\ell)$ of bits and nodes (where we assume that ℓ is the number of registers of A). Intuitively, a_i is the content of register r_i in this particular configuration.

Consider a configuration $(n, q, \bar{a})$. A program rule in Π with guard $\varphi(x, y, \bar{r})$ is said to be *enabled in* $(n, q, \bar{a})$ if $\mathcal{I}^+ \models \varphi[n, \text{head}(q), \bar{a}]$.

The *successor configuration of* $(n, q, \bar{a})$ is the configuration $(n, q', \bar{a}')$ where $q' = \text{tail}(q)$ and for each $i \in \{1, \ldots, \ell\}$ the following condition is satisfied: if there is precisely one r_i-update rule in Π which is enabled in $(n, q, \bar{a})$, and $(r_i := t)$ is the right-hand side of this rule, then $a'_i = t[x/n,\ y/\text{head}^*(q),\ \bar{r}/\bar{a}]$; otherwise, $a'_i = a_i$. Here, head^* maps the empty queue to 0, but is otherwise defined as head.

We say that A *sends a message m in* $(n, q, \bar{a})$ if there is precisely one send rule in Π which is enabled in $(n, q, \bar{a})$ and, if $\textit{send}(\bar{t})$ is the right-hand side of this rule, $m = \bar{t}[x/n,\ y/\text{head}^*(q),\ \bar{r}/\bar{a}]$.

A *local run of A at node n* (in $\mathcal{I}$ with source s) is a finite or infinite sequence $(C_i)_{i \in \kappa}$ of configurations of A at n such that for every $i + 1 \in \kappa$

- C_{i+1} is a successor configuration of C_i, and
- if A sends the empty message in C_i, then C_{i+1} is the last configuration of $(C_i)_{i \in \kappa}$.

Remark 1. The reader may wonder why our automata need to be able to send messages of length longer than 1. After all, an automaton could send a message component-wise, i.e., bit by bit, node by node, as messages of length 1? However, in the next section, we will consider systems of communicating automata where a receiving automaton may obtain messages from many different automata, and these messages can be intermingled during communication. In particular, the order in which messages occur in the queue of the receiving automaton can be arbitrary. If an automaton sends a message component-wise, the receiving automaton may not be able to reconstruct the original message from its components. In this context, note that messages longer than 1 can always be flanked by separators (e.g., special bit sequences), enabling a receiving automaton to distinguish between different messages.

Notice that a local run can start in *any* configuration. This is because in a distributed scenario an automaton may receive some messages even before it starts its own, local computation.

We are particularly interested in automata where the time between two send actions is bounded by a constant.

Definition 2. Let $k \geqslant 1$ be a natural number. A Web automaton A is *k-productive* if in every local run $(C_i)_{i<k}$ of A, there is at least one configuration in which A sends a message. A is *productive* if it is k-productive for some k.

As an example, recall the program in Example 1, and verify that the automaton defined by this program is 2-productive.

THEOREM 1. *Deciding productivity is* PSPACE*-complete.*

PROOF SKETCH. For containment in PSPACE, consider an arbitrary Web automaton A, and suppose that A has (at most) ℓ registers. It is easily verified that A is productive iff A is 3^ℓ-productive. Furthermore, there is a straightforward, non-deterministic algorithm which accepts a given A iff there exists a run of A of length $3^\ell - 1$ during which A does not send a message. This algorithm runs in space polynomial in the size of A. Since NPSPACE = PSPACE, we conclude that deciding non-productivity is in PSPACE. This immediately implies that deciding productivity is in PSPACE as well.

Hardness for PSPACE is proved via a reduction from a restriction of FIN-SAT(E+TC), the finite satisfiability problem for existential transitive-closure logic (see, e.g., [4, 17]). A formula of the form $[\mathrm{TC}_{\bar{x},\bar{x}'}\varphi](\bar{t},\bar{t}')$ is called *simple* if $\bar{t} = \bar{0}$, $\bar{t}' = \bar{1}$, and φ is a quantifier-free formula over the vocabulary $\{0, 1, =\}$ of the form $\varphi' \wedge \bar{x}' \in \{0, 1\}$. The problem of deciding whether a given simple TC formula has a (finite) model is PSPACE-complete [17]. We reduce this problem to the problem of deciding non-productivity.

Consider a simple TC sentence $\psi = [\mathrm{TC}_{\bar{x},\bar{x}'}\varphi](\bar{0},\bar{1})$, and suppose that $\bar{x}$ (and thus $\bar{x}'$) consists of k variables. One can define a Web automaton A_ψ which is not productive iff ψ is satisfiable. The idea is to let A_ψ interpret its queue as an encoding of a φ-path from $\bar{0}$ to $\bar{1}$. In k consecutive steps, A_ψ reads k bits from its queue, stores the bits in registers $\bar{x}'$, and then checks whether $\varphi(\bar{x}, \bar{x}')$ holds. If the test is successful, it sets $\bar{x} = \bar{x}'$; otherwise, it discards $\bar{x}'$. After 2^k repetitions, it knows whether an initial segment of its queue encodes a path model of ψ, or not. If it finds a model, it does not send any messages; otherwise, it sends some dummy message. $\square$

4. DISTRIBUTED COMPUTATIONS

Before we define distributed runs of systems of Web automata formally, we provide some intuition. A productive automaton A, when started at some source node s in an instance $\mathcal{I}$, begins by distributing copies of itself to all other nodes, using a straightforward recursive procedure: upon creation at a node n, A equips every node which n links to with a copy of itself, except if the node is already equipped with a copy. This procedure traces out some spanning tree of the link graph of $\mathcal{I}$. Of course, in the 'real' Web, it is virtually impossible to equip all nodes with copies of A. Instead, we propose to visit nodes only up to a certain level in the spanning tree. An upper bound on the levels could, e.g., be specified by the user. Also, all automata running at nodes which are located on the same server may still be implemented by a single process running at that server. Once the spanning tree is set up, all automata start running concurrently. Each automaton sends its messages to the automaton which created it, following a simple protocol based on two principles:

1. Start computing the next message only if you have 'enough' input (see below).
2. Stop once you have sent a message.

This naturally organizes a distributed computation in *rounds*, where in each round, every automaton (which is still active) sends precisely one message. For instance, automata at leaf nodes (of the spanning tree) never receive any messages and can thus start a round by computing their own messages (in parallel). Each leaf automaton, when finished, sends its message to its parent automaton, and then waits for the next round. Automata at inner nodes, on the other hand, consume messages from their queues each time they move. Hence, an inner automaton must wait until it has received enough messages (or knows that no new message will arrive in this round), such that it can run long enough in order to compute its own message. When finished, it also sends its message to its parent automaton, and then waits for the next round.

Since our automaton program is productive, every round can be performed in parallel time linear in the depth of the spanning tree. In every new round, an automaton continues its local run where it has stopped during the previous round. If an automaton sends the empty message, it exits the computation and will not participate in later rounds. If the source automaton exits, the whole computation terminates. The output produced during a computation is the set of nodes sent by the source automaton.

We proceed to the formal definition of distributed runs. Let $\mathcal{I}$ be an instance with node set N and link set L, and let s be a (source) node in $\mathcal{I}$. We assume that $\mathrm{Reach}(\mathcal{I}, s) = \mathcal{I}$; if this is not the case, replace $\mathcal{I}$ with $\mathrm{Reach}(\mathcal{I}, s)$ in what follows. Let $\mathcal{T}$ be a spanning tree of the link graph (N, L) such that s is the root of $\mathcal{T}$.

A *global configuration* of A is a mapping γ that assigns to each node n a configuration of A at n. The *initial global configuration* maps each n to $(n, \varnothing, \bar{0})$.

Consider two global configurations γ and γ', and let n be a node. γ' is called a *successor configuration of γ via a move at n* if the following three conditions hold:

1. There exists a finite local run $(C_0, \dots, C_r)$ of A at n such that (i) $C_0 = \gamma(n)$, (ii) $r \geqslant 1$, (iii) no message is sent during this local run before C_{r-1}, (vi) A does send some message m in C_{r-1}, and (v) $C_r = \gamma'(n)$.
2. If $n \neq s$, let p be the parent of n in $\mathcal{T}$ and suppose that $\gamma(p) = (p, q, \bar{a})$. Then, $\gamma'(p) = (p, qm, \bar{a})$. (That is, m is appended to the queue of the parent automaton.)
3. For every node o different from n and p (if p exists), $\gamma'(o) = \gamma(o)$.

Let d be the depth of $\mathcal{T}$. For every $i \in \{0, \dots, d\}$, let $\mathrm{level}(i)$ denote the set of nodes whose distance from s in $\mathcal{T}$ is i. Let

M be a subset of N containing s. An *M-round* (along $\mathcal{T}$) is a finite sequence $(\gamma_i)_{i \leqslant k}$ of global configurations such that

- for every $i+1 \leqslant k$, γ_{i+1} is a successor configuration of γ_i via a move at some node n_{i+1}
- the sequence $n_1 \dots n_k$ is an enumeration of M, and
- for each $i \in \{0, \dots, d\}$ there exists an enumeration $\bar{e}_i$ of level$(i) \cap M$ such that $\bar{e}_d \dots \bar{e}_0 = n_1 \dots n_k$.

The *output* produced during this round is the set of nodes occurring in the message sent by A at s.

A *one-round run* ρ (on $\mathcal{I}$ with source s) is an N-round which starts with the initial global configuration. Finally, a *multiple-round run* ρ^* (on $\mathcal{I}$ with source s) is a finite or infinite sequence $(\rho_i)_{i \in \kappa}$ of rounds along the same spanning tree such that ρ_0 is a one-round run and for every $i+1 \in \kappa$

- ρ_{i+1} starts with the last configuration of ρ_i, and
- if ρ_i is an M-round and $M_0 \subseteq M$ is the set of nodes at which A has sent the empty message during ρ_i, then ρ_{i+1} is an $(M - M_0)$-round.

Since M-rounds are only defined when $s \in M$, ρ^* is finite iff the root automaton sends the empty message during some round ρ_i, in which case ρ_i is the last round of ρ^*. The *output* produced during ρ^* is the union of the outputs produced during the rounds of ρ^*.

Notice that, since A was assumed to be productive, condition (1) in the above definition of a successor configuration is always satisfied. As a consequence, for every choice of $\mathcal{I}$, s, and $\mathcal{T}$, there exists a multiple-round run of A on $(\mathcal{I}, s)$ along $\mathcal{T}$.

Remark 2. It is worth noticing that productivity of A is only a *sufficient* criterion for the existence of a multiple-round run of A on any $(\mathcal{I}, s)$. In fact, there are many automata which are not productive in the sense of Definition 2 but which nevertheless always satisfy condition (1), and thus always have a multiple-round run. Unfortunately, it is undecidable whether a given automaton satisfies condition (1) in every possible distributed scenario. However, we only mention here that Definition 2 can be relaxed so that the obtained notion of productivity becomes strictly weaker, still enables multiple-round runs, and is also decidable in PSPACE.

In the remainder of the paper, we focus on productive Web automata.

Example 2. Recall the Web automaton defined in Example 1. When run on a pair $(\mathcal{I}, s)$, this automaton outputs all 'interesting' nodes in $\mathcal{I}$ reachable from s. In each round, the source automaton outputs precisely one node. Notice that the source automaton does not output any node twice and that the output order depends on the choice of the spanning tree and on the order in which the various automata communicate during each round.

Remark 3. Distributed runs as defined in this section are in fact *linearizations of partially-ordered runs* in the spirit of Gurevich [9]. This explains why the intuitive description of distributed computations at the beginning of this section may not entirely conform to the formal definition of distributed runs: the intuitive description refers to (genuine) partially-ordered runs. It is possible to give an alternative definition of distributed runs which does not make use of linearizations and which can easily be implemented.

5. AUTOMATA COMPUTING QUERIES

In general, a Web automaton can exercise many different distributed runs on one and the same Web instance (and source node). This is because both the choice of the spanning tree and the order of communications during rounds can be arbitrary. As a result, different runs may produce different output sets.

Example 3. Consider the following program. For the sake of readability, we display the program in a slightly relaxed syntax, using nested `if-then-else` rules (with the obvious meaning).

```
if ($r = 0$) then
    $r := 1$
    if ($head\_q \neq \bot$) then
        $send(head\_q)$
    else
        $send(this\_node)$
else
    $send()$
```

On the 3-node instance $n_1 \leftarrow s \rightarrow n_2$, the output can be either $\{n_1\}$ or $\{n_2\}$, depending which of the two children of s gets its message first in the queue of s. By adding the two links $n_1 \rightarrow n_2$ and $n_2 \rightarrow n_1$, we obtain also dependence on the choice of the spanning tree. If the tree $s \rightarrow n_1 \rightarrow n_2$ is selected, the output is $\{n_2\}$, while if the tree $s \rightarrow n_2 \rightarrow n_1$ is selected, the output is $\{n_1\}$.

On the other hand, there are automata whose output is independent of the choice of the spanning tree and the order of communications. To see an example, consider again the automaton in Example 1. Here, the *output set* is always the same, although the *output order* can differ from run to run. This motivates the following definition.

Definition 3. A Web automaton A is *sound* if for every pair $(\mathcal{I}, s)$, every multiple-round run of A on $(\mathcal{I}, s)$ produces the same output. In that case, we can speak of *the Web query computed by A*, which maps a pair $(\mathcal{I}, s)$ to the output (produced during any run) of A on $(\mathcal{I}, s)$.

Unfortunately, we cannot decide whether a given Web automaton is sound. (Nevertheless, there exists an interesting class of Web automata for which soundness is decidable, as we will see in the next section.)

THEOREM 2. *Soundness is undecidable.*

Proof Sketch. The proof is by reduction from the emptiness problem for deterministic one-way two-head automata (2-DFAs). It suffices to consider *simple 2-DFAs*, i.e., 2-DFAs whose input alphabet is $\{0, 1\}$ and whose program ensures that every computation progresses in two distinguished phases. During the first phase, a simple 2-DFA M uses its first input head to scan an initial segment of the input tape. The second input head remains idle. After each computation step, M may or may not switch to the second phase, depending on its current configuration. If and when M switches to the second phase, the first input head is placed somewhere on the tape, while the second input head is still on the first tape cell. During the second phase, M can do whatever 2-DFAs are entitled to do, with the restriction that, in every computation step, M must move both input heads, each one to the next tape cell. A computation of M stops if the input is accepted or if the first input head reaches the end of the input tape. One can show that for simple 2-DFAs the emptiness problem is undecidable (by reduction from the word problem for Turing machines).

Let M be a simple 2-DFA. Recall that by Q_{true} we denote the Web query which maps a pair $(\mathcal{I}, s)$ to the set of those nodes in $\mathcal{I}$ which are reachable from s. One can construct a Web automaton A_M over $\{Link\}$ such that

- if $L(M) = \varnothing$, then A_M computes Q_{true}, and
- if A_M is sound, then $L(M) = \varnothing$.

This reduces the emptiness problem for simple 2-DFAs to the problem of deciding soundness. Some details of the construction follow. In the first round, A_M performs the following two tasks in parallel. First, it checks whether it is executed along a spanning tree which has the form of a path. Second, it pretends that the first test was successful, views the spanning tree (which is now assumed to be a path) as an input tape (where link self loops represent set input bits), and simulates the first phase of M on that input tape. If the first test fails, the source instance of A_M switches to a 'forwarding' mode, which means that in every subsequent round it just outputs all nodes (reachable from the source node). The same happens if during the simulation of M the first input head reaches the end of the (virtual) input tape.

If the source automaton survives the first round without switching to forwarding mode, then, in all subsequent rounds, A_M simulates the second phase of M and, in parallel, outputs all nodes. Except if the source automaton discovers during the simulation that M accepts. In that case, the source automaton switches to a 'spoiling' mode, which means that it stops outputting nodes and instead sends some dummy messages. $\square$

An indication of the querying power of Web automata is provided by the next result. We call a Web instance $\mathcal{I}$ *locally ordered* if it contains a distinguished ternary relation $\prec$, typically written $x \prec_z y$, such that for every node n in $\mathcal{I}$, the binary relation $x \prec_n y$ is a total order on the children of n in $\mathcal{I}$.

Theorem 3. *Any logarithmic-space computable Web query on locally ordered instances is computable by a Web automaton.*

Proof Sketch. Let $\varphi(x_1, \ldots, x_k)$ be a formula of deterministic transitive-closure logic (see, e.g., [4]). One can construct a Web automaton A_φ which, on every pair $(\mathcal{I}, s)$ with a locally ordered $\mathcal{I}$, enumerates $\{\bar{a} : \text{Reach}(\mathcal{I}, s) \models \varphi[\bar{a}]\}$ in the following sense. In every round, A_φ at s sends either a 'wait' message or a message $(a_1, \ldots, a_k)$ satisfying φ. Eventually, all messages satisfying φ are sent by A_φ at s. The theorem is then implied by a well-known result due to Immerman [10], namely that a query on finite ordered structures is logarithmic-space computable iff it is expressible in deterministic transitive-closure logic (see also [4]). The construction of A_φ is based on the following observation. There exists a Web automaton A_{enum} such that every multiple-round run $(\rho_i)_i$ of A on $(\mathcal{I}, s)$ satisfies the following three conditions:

1. $(\rho_i)_i$ is infinite.
2. During each ρ_i, there is at most one node output.
3. Let $(n_j)_j$ be the node sequence produced during $(\rho_i)_i$ where rounds with empty output are omitted. There exists an enumeration $\bar{e}$ of the reachable nodes such that $(n_j)_j$ can be seen as an infinite repetition of $\bar{e}$.

A_φ can now be defined by induction on φ. For instance, if $\varphi(\bar{x}) = R(\bar{x})$, then A_φ simulates A_{enum}, turns the repetitive enumeration of all nodes into an enumeration of all k-tuples of nodes, and checks whether $R(\bar{x})$ holds for each k-tuple. $\square$

An undesirable behavior of Web automata, even of sound ones, is that one may have to wait many rounds before seeing any new output (e.g., a node which has not been output yet). In the worst case, one may even wait only to learn later that there is no new output at all. Since each round takes only linear parallel time in the depth of the portion of the Web which we are exploring, it would be particularly interesting to have the following behavior.

Definition 4. A Web automaton A is called *continuous* if every multiple-round run $(\rho_i)_i$ of A satisfies the following condition: during each round ρ_i, except of the last round, there is at least one node output which has not been output during any round preceding ρ_i.

Theorem 4. *Continuity is undecidable.*

Proof Sketch. The proof is similar to the proof of Theorem 2. In fact, A_M can be constructed so that (i) if $L(M) = \varnothing$, then A_M *continuously* computes Q_{true}, and (ii) if A_M is continuous, then $L(M) = \varnothing$. This reduces the emptiness problem for simple 2-DFAs to the problem of deciding continuity. $\square$

Remark 4. Theorems 2 and 4 hold already for automata which test only one unary relation (in particular, which do not test the link relation). Both theorems remain true if we focus on automata which test only the link relation (and no other relation). Moreover, undecidability is encountered even if we restrict our attention to tree-like Web instances (which have a unique spanning tree). Finally, both theorems remain true for *finite* Web automata, i.e., Web automata which cannot store nodes in their registers and therefore have only a finite number of different internal states. Note that the automaton in Example 1 is finite in that sense.

The next result provides a class of Web queries computable by continuous Web automata, in terms of a fragment of first-order logic, which we call *at-most-at-least logic*. The fragment may seem artificial at first, but later we will see that it is associated to a natural subclass of the class of Web automata (see Theorem 8).

Let $\alpha(x)$ be quantifier-free formula with $\mathrm{free}(\alpha) = \{x\}$, and let k be a natural number. Subsequently, we write $(\exists x \in \alpha)$ instead of $\exists x \alpha(x)$. An *α-at-most formula* is a formula of the form $(|\alpha| \leqslant k) \wedge \gamma_\alpha(s, x)$ where

- $(|\alpha| \leqslant k)$ abbreviates the first-order formula $\neg(\exists^{>k} x \in \alpha)$, and
- $\gamma_\alpha(s, x)$ is a boolean combination of formulas of the form $(\exists y_1 \in \alpha) \dots (\exists y_r \in \alpha)\beta(s, x, \bar{y})$ with β quantifier-free and $r \geq 0$.

An *at-most-at-least formula* is a formula of the form $\alpha(x) \wedge \delta_\alpha(s, x)$ where $\delta_\alpha(s, x)$ is a boolean combination of α-at-most formulas and atomic formulas $\beta(s)$.

Example 4. Suppose that the vocabulary Υ contains two unary relation symbols *Red* and *Green*. Verify that the following formula is a *Red*-at-most formula over Υ:

$$|\mathit{Red}| \leqslant 42 \wedge \neg\big(\mathit{Green}(x) \wedge (\exists y \in \mathit{Red})\mathit{Link}(x, y)\big).$$

The negation of this formula is equivalent to $|\mathit{Red}| \leqslant 42 \rightarrow \neg\,\gamma_{\mathit{Red}}(x)$ where $\gamma_{\mathit{Red}}(x)$ stands for the second conjunct of the above formula. The following formula is now an example of an at-most-at-least formula. Intuitively, the formula says that, if there are more than 42 red nodes, then output all red nodes; otherwise, output all red nodes which are also green and which link to another red node.

$$\mathit{Red}(x) \wedge \big(|\mathit{Red}| \leqslant 42 \rightarrow \neg\,\gamma_{\mathit{Red}}(x)\big).$$

Theorem 5. *Any Web query definable in at-most-at-least logic is computable by a continuous Web automaton.*

The proof of this theorem is presented in a more general context in the appendix (see also Remark 8). The converse direction of the theorem does not hold, however. For instance, the query defined by the following formula is continuously computable, but the formula is not equivalent to any at-most-at-least formula:

$$\big(\mathit{Red}(x) \wedge |\mathit{Red}| \geqslant 42\big) \vee \big(\mathit{Green}(x) \wedge |\mathit{Green}| \geqslant 10\big).$$

Remark 5. There is an interesting variant of Theorem 5, which reads as follows. A *generalized at-most-at-least formula* is simply a boolean combination of at-most formulas and quantifier-free formulas $\beta(s, x)$. In particular, the at-most subformulas of a generalized at-most-at-least formula do not need to be defined w.r.t. the same α. One can show that any Web query definable by a generalized at-most-at-least formula is computable by a Web automaton *with discard action*, i.e., a Web automaton which can discard its queue, in addition to performing update and send actions.

The next observation gives an example of a query which is computable by a Web automaton, but not by a continuous one.

Proposition 1. *Let R be a binary relation symbol. The Web query $Q_R : (\mathcal{I}, s) \mapsto \{n : \mathit{Reach}(\mathcal{I}, s) \models R(s, n)\}$ is not computable by a continuous Web automaton.*

Indeed, to be continuous, the source automaton must start outputting already in the first round. However, by productivity, it can see only a constant number of nodes in each round. If the communication order is unfortunate, none of the nodes seen in the first round qualify for output.

Remark 6. A similar argument shows that the Web query defined by the at-most-at-least formula $\mathit{Red}(x) \wedge |\mathit{Red}| \leqslant 2$, while computable by a continuous Web automaton, is not computable by a continuous Web automaton that can send messages of length at most 1 (recall Remark 1).

6. DECIDE AND FORWARD AUTOMATA

In this section, we introduce a natural, syntactic subclass of the class of Web automata, for which it is decidable whether a given automaton is sound and continuous. The subclass is closely tied to the at-most-at-least logic defined in the previous section.

A Web automaton is called *link-free* if the link relation does not occur in its program. Link-freeness of sound automata simplifies things considerably, as exemplified by the following 'flat-tree' property. For every pair $(\mathcal{I}, s)$, let $\mathrm{flat}(\mathcal{I}, s)$ denote the Web instance obtained from $\mathcal{I}$ by changing the link graph of $\mathcal{I}$ into a flat tree with root s, i.e., a tree where all nodes, except of s, are children of s.

Proposition 2. *Let A be a sound, link-free Web automaton, and let Q be the Web query computed by A. For every pair $(\mathcal{I}, s)$, $Q(\mathcal{I}, s) = Q(\mathit{flat}(\mathcal{I}, s), s)$.*

Proof. Consider a pair $(\mathcal{I}, s)$. Let $\mathcal{I}'$ be obtained from $\mathcal{I}$ by adding links from s to all other nodes. Every run of A on $(\mathcal{I}, s)$ is also a run of A on $(\mathcal{I}', s)$, and by soundness, $Q(\mathcal{I}, s) = Q(\mathcal{I}', s)$. Likewise, every run of A on $(\mathrm{flat}(\mathcal{I}, s), s)$ is also a run of A on $(\mathcal{I}', s)$, hence $Q(\mathrm{flat}(\mathcal{I}, s), s) = Q(\mathcal{I}', s) = Q(\mathcal{I}, s)$. $\square$

One can easily show that, on flat trees, every sound Web automaton computes a logarithmic-space computable query. The above proposition then implies that any Web query computable by a link-free Web automaton is logarithmic-space computable. Whether this also holds in general is an open problem; the proposition certainly does not hold in general.

Remark 7. Referring back to the introduction, we point out (only half seriously though) that Proposition 2 provides some kind of a-posteriori justification of the way Internet supercomputing works, where in fact the standard mode of operation is so that all computers which participate in a distributed computation report directly to a central source computer.

A *decide-and-forward automaton* (or DF automaton for short) is a Web automaton whose program has the form

```
if $first\_round$ then
    $\Pi$
else
    $\Pi_{forward}$
```

where *first_round* is a boolean register initialized with *true*, Π is a program in which every send rule has the form

$$\texttt{if } \varphi \texttt{ then } send(\bar{t});\ first\_round := false$$

where φ is a guard of the form $\varphi' \wedge \bar{t} \notin \{0,1\}$, and $\Pi_{forward}$ is the program

```
if ($head\_q \neq \bot$) then
    $send(head\_q)$
else
    $send()$
```

In other words, a DF automaton makes all the crucial decisions in the first round, sends only nodes (no bits), and acts in all subsequent rounds (if any) merely as a forwarder that flushes the remaining contents of the communication queues to the output.

Note that the program displayed in Example 1 can easily be rewritten into an equivalent DF form (see also the explanation in Example 2).

A Web automaton is called *monadic* if in its program no relation of arity $\geqslant 2$ occurs. In particular, a monadic automaton is link-free. For monadic DF automata, we obtain the following two decidability results plus a characterization in terms of monadic at-most-at-least logic. The corresponding proofs are rather technical and sketched in the appendix.

THEOREM 6. *For monadic DF automata, emptiness is decidable. (In general, emptiness is undecidable.)*

THEOREM 7. *The problem of deciding whether a monadic DF automaton is sound and continuous is decidable.*

THEOREM 8. *A Web query is computable by a continuous monadic DF automaton iff it is definable in monadic at-most-at-least logic.*

Recall the query which shows that the converse direction of Theorem 5 does not hold in general (see after Theorem 5). Since this query is monadic, we obtain the following corollary of Theorem 8.

COROLLARY 1. *Continuous DF automata are strictly weaker than (general) continuous Web automata.*

7. BROWSER STACK MACHINES

For the work by Abiteboul and Vianu [2] was one of the main inspiration for the present paper, we conclude our investigation by drawing a connection between Web automata and Abiteboul and Vianu's browser machines. More precisely, we introduce *browser stack machines*, a restricted variant of browser machines, and show that browser stack machines can be simulated by Web automata in depth-bounded regions of the Web.

Browser Stack Machines. There are two main restrictions which we impose on browser machines:

- the work tape is replaced with a finite number of registers, and
- the browsing tape is organized like a stack, forcing a machine to explore the Web only by means of the three familiar surf actions of common Web browsers: 'follow this link', 'go back', and 'go forward'.

Formally, browser stack machines (BSMs for short) are defined as follows. Let Υ and $\bar{r}$ be as in the definition of Web automata. A *guard* (of a rule) is a quantifier-free formula $\varphi(x, \bar{r})$ over $\Upsilon \cup \{0,1\}$ with free$(\varphi) \subseteq \{x, \bar{r}\}$. (This time, x will denote the stack element which the cursor of a BSM is currently pointing to.) A *BSM program* Π is a finite set of rules of the form (`if` φ `then` *action*) where φ is a guard, and *action* is an expression of the form *up*, *down*, *expand*, $(r_i := t)$ or *output*(t). Here, t is a term in $\{x, \bar{r}, 0, 1\}$.

Definition 5. A *browser stack machine* M is a triple $(\Upsilon, \bar{r}, \Pi)$ consisting of a vocabulary Υ, a tuple $\bar{r}$ of (register) variables, and a BSM program Π over $\Upsilon \cup \{0,1\}$ and $\bar{r}$.

Runs. Let $M = (\Upsilon, \bar{r}, \Pi)$ be a BSM, let $\mathcal{I}$ be a locally ordered instance over Υ (recall our convention prior to Theorem 3), and let s be a (source) node in $\mathcal{I}$. Subsequently, the word *stack* refers to a finite sequence of nodes and 0's. An occurrence of 0 on a stack will serve as a separator between different segments of the stack. If *st* is a stack of length k, then by a *cursor on st* we mean a natural number between 1 and k.

A *configuration of M* is a quadruple $(st, c, \bar{a}, O)$ where *st* is a stack, c is a cursor on *st*, $\bar{a}$ is a register assignment (as in the case of Web automata), and O is a set of nodes. Intuitively, O is the output produced so far.

Consider a configuration $(st, c, \bar{a}, O)$. The *successor configuration* $(st', c', \bar{a}', O')$ of this configuration is defined in the obvious way. Here, we only give some details concerning the actions *up*, *down*, and *expand*. Suppose that there is precisely one stack rule in Π which is enabled in $(st, c, \bar{a}, O)$. If the right-hand side of the rule is *up* or *down*, then $st' = st$ and c' is obtained from c as usual. If the right-hand side of the rule is *expand*, partition st into st_{low} and st_{high} such that $st = st_{\text{low}}\, st_{\text{high}}$ and the length of st_{low} is c. If c points to a node, say, n, and $\bar{e}$ denotes the enumeration of all children of n in the order $x \prec_n y$, then $st' = st_{\text{low}}\, 0\, \bar{e}$ and $c' = c$. Otherwise, $st' = st$ and $c' = c$.

A *run ρ of M* (in $\mathcal{I}$ with source s) is a finite or infinite sequence $(C_i)_{i\in\kappa}$ of configurations of M such that $C_0 = (s, 1, \bar{0}, \varnothing)$ and for every $i + 1 \in \kappa$

- C_{i+1} is a successor configuration of C_i, and
- C_{i+1} is the last configuration of ρ if M attempts to move the cursor below the stack bottom in C_i.

Note that ρ is uniquely determined by $(\mathcal{I}, s)$. We say that *M halts on* $(\mathcal{I}, s)$ if ρ is finite. In that case, the output component of the final configuration of ρ is called the *output* of M on $(\mathcal{I}, s)$. If M halts on every pair $(\mathcal{I}, s)$, we can speak of *the Web query computed by M*, which maps a pair $(\mathcal{I}, s)$ to the output of M on $(\mathcal{I}, s)$.

The next theorem gives an indication of the querying power of BSMs.

THEOREM 9. *Any logarithmic-space computable Web query on locally ordered instances is computable by a BSM.*

The proof of this theorem is similar to the proof of Theorem 3. The only difficult part is to find a BSM which repetitively enumerates all nodes (similar to A_{enum} in the proof of Theorem 3). We omit the details.

Depth-Bounded BSMs. Let d be a natural number. A BSM M is called *d-bounded* if it maintains a counter of the number of separators between the stack bottom and the current cursor position. Whenever this counter equals d, M ignores all expand actions.

We can now draw a connection between Web automata and browser machines.

THEOREM 10. *Any Web query computable by a depth-bounded BSM is computable by a Web automaton.*

PROOF SKETCH. Due to Theorem 3, it suffices to show that any depth-bounded BSM can be simulated by a logarithmic-space bounded Turing machine (with separate input and output tapes). Consider a d-bounded BSM M. We describe a logarithmic-space bounded Turing machine T_M such that for every input $(\mathcal{I}, s)$, T_M on an encoding of $(\mathcal{I}, s)$ simulates M on $(\mathcal{I}, s)$.

First observe that, because T_M has a separate output tape, it does not need to store the output of M. If $(st, c, \bar{a}, O)$ is a configuration of M (on some fixed input $(\mathcal{I}, s)$), then the triple $(st, c, \bar{a})$ is called a *reduced configuration of M*. We show that T_M can store a representation of any reduced configuration in logarithmic space (in the size of $\mathcal{I}$). This clearly holds for the contents $\bar{a}$ of the registers of M. Consider the stack st. By the *i-th segment of st* we mean the segment which

- starts with the node following the $(i-1)$-th separator, and
- ends with the i-th separator.

For example, the first segment of any stack consists of the source node and the first separator. To represent st, T_M employs d registers, called *stack registers*. Each stack register either holds a node or is undefined, and thus requires only logarithmic space. During a computation, the i-th stack register holds the last node of the i-th segment of st (i.e., the node before the i-th separator). This node was expanded when the $(i+1)$-th segment was placed on the node stack.

To represent the cursor c, T_M employs another register, called *cursor register*, and a counter ranging in $\{1, \ldots, d+1\}$. During a computation, the cursor register holds the node currently read by the cursor; it is undefined iff the cursor is currently placed on a separator. The counter specifies in which segment the cursor is currently roaming. Both, the counter and the cursor register require only logarithmic space. (Verify that, because no node occurs twice in the same segment, the counter and the cursor register together uniquely determine the position of the cursor.)

Using this representation of reduced configurations, T_M can simulate transitions of M from reduced configurations to reduced configurations in logarithmic space. $\square$

8. REFERENCES

[1] S. Abiteboul and V. Vianu. Regular path queries with constraints. *Journal of Computer and System Sciences*, 58(3):482–452, 1999.

[2] S. Abiteboul and V. Vianu. Queries and computation on the web. *Theoretical Computer Science*, 239(2):231–255, 2000.

[3] H. Attiya and J. Welch. *Distributed Computing: Fundamentals, Simulations and Advanced Topics.* McGraw-Hill, 1998.

[4] H. Ebbinghaus and J. Flum. *Finite Model Theory.* Springer-Verlag, 1995.

[5] D. Florescu, A. Levy, and A. Mendelzon. Database techniques for the World-Wide Web: A survey. *SIGMOD Record*, 27(3):59–74, 1998.

[6] I. Foster. Internet computing and the emerging grid. *Nature*, Dec. 2000.

[7] I. Foster and C. Kesselman. *The Grid: Blueprint for a New Computing Infrastructure.* Morgan-Kaufmann, 1998.

[8] N. Gupta, J. Haritsa, and M. Ramanath. Distributed query processing on the Web. In *Proceedings of 16th International Conference on Data Engineering*, page 84. IEEE Computer Society, 2000.

[9] Y. Gurevich. Evolving algebras 1993: Lipari guide. In E. Börger, editor, *Specification and Validation Methods*, pages 9–36. Oxford University Press, 1995.

[10] N. Immerman. Languages that capture complexity classes. *SIAM Journal of Computing*, 16(4):760–778, 1987.

[11] M. Kaminski and N. Francez. Finite-memory automata. *Theoretical Computer Science*, 134(2):329–363, 1994.

[12] E. Korpela, D. Werthimer, D. Anderson, J. Cobb, and M. Lebofsky. SETI@HOME—massively distributed computing for SETI. *Computing in Science and Engineering*, 3(1):78–83, 2001.

[13] A. Mendelzon and T. Milo. Formal models of web queries. *Information Systems*, 23(8):615–637, 1998.

[14] F. Neven, T. Schwentick, and V. Vianu. Towards regular languages over infinite alphabets. In *Proceedings of 26th International Symposium on Mathematical Foundations of Computer Science (MFCS 2001)*, volume 2136 of *Lecture Notes in Computer Science*, pages 560–572. Springer-Verlag, 2001.

[15] M. Ramanath and J. Haritsa. DIASPORA: A highly distributed web-query processing system. *World Wide Web*, 3(2):111–124, 2000.

[16] V. Sazonov. Using agents for concurrent querying of web-like databases via a hyper-set-theoretic approach. In *Proceedings of 4th International Conference on Perspectives of System Informatics (PSI 2001)*, volume 2244 of *Lecture Notes in Computer Science*, pages 378–394. Springer-Verlag, 2001.

[17] M. Spielmann. *Abstract State Machines: Verification Problems and Complexity.* PhD thesis, RWTH Aachen, 2000.

[18] A. Tanenbaum. *Computer Networks.* Prentice-Hall, 3rd edition, 1996.

APPENDIX

Below, we sketch the proofs of Theorems 6 and 7. The proofs of Theorems 5 and 8 are subject of the second part of this section.

Decidability Results

Fix a DF automaton A. We assume that A is k-productive. A Web instance $\mathcal{I}$ is called *tree-like* if the link graph of $\mathcal{I}$ is a tree (where each leaf is reachable from the root). By a run of A on a tree-like $\mathcal{I}$ we mean a run of A on $(\mathcal{I}, r)$ where r is the root of $\mathcal{I}$.

LEMMA 1. *Suppose that A is monadic. There exists a (computable) constant $c_{A,k}$ such that for every one-round run ρ of A there exists a one-round run ρ' of A on a tree-like Web instance of size at most $c_{A,k}$ such that the message sent by the source automaton during ρ' is identical with the message sent by the source automaton during ρ.*

Using this observation, one can prove the following theorem.

THEOREM 11. *For monadic Web automata the first-round emptiness problem is decidable.*

PROOF OF THEOREM 6. Verify that the identity is a reduction from the emptiness problem for DF automata to the first-round emptiness problem for Web automata. Theorem 6 then follows from the above theorem. □

The main idea in the proof of Theorem 7 is to reduce the problem of deciding soundness to the problem of deciding soundness on 'flat' trees. We call a tree-like $\mathcal{I}$ *flat* if the link depth of $\mathcal{I}$, measured from the root, is at most 1. Note that, for every tree-like $\mathcal{I}$ with root r, $\mathrm{flat}(\mathcal{I}, r)$ is by definition flat (recall the definition of $\mathrm{flat}(\mathcal{I}, r)$ prior to Proposition 2).

Definition 6. A is *flat-tree sound* if for every flat $\mathcal{I}$, every multiple-round run of A on $\mathcal{I}$ produces the same output. A is *flat-invariant* if it is flat-tree sound and for every tree-like $\mathcal{I}$, every multiple-round run of A on $\mathcal{I}$ produces the same output as A on $\mathrm{flat}(\mathcal{I}, r)$ where r denotes the root of $\mathcal{I}$.

LEMMA 2. *Flat-tree soundness is a decidable property of DF automata.*

LEMMA 3. *Suppose that A is monadic. A is sound iff A is flat-invariant.*

The remainder of the construction concerns a procedure for deciding flat-invariance.

Alpha Nodes. Let $\alpha(x)$ be a quantifier-free formula with $\mathrm{free}(\alpha) = \{x\}$ such that for every tree-like $\mathcal{I}$ and for every leaf node n in $\mathcal{I}$, $\mathcal{I} \models \alpha[n]$ iff A at n sends a non-empty message during a one-round run of A on $\mathcal{I}$. We call a node n in $\mathcal{I}$ *α-node* if $\mathcal{I} \models \alpha[n]$.

Alpha-Sending Automata. A is called *α-sending* if during every one-round run of A, every non-empty message sent by A at a non-source node contains pairwise distinct α-nodes only.

LEMMA 4. *If A is α-sending, then A is continuous.*

LEMMA 5. *It is decidable whether a given Web automaton is α-sending.*

Alpha-Outputting Automata. A is called *α-outputting* if for every $(\mathcal{I}, s)$, every multiple-round run of A on $(\mathcal{I}, s)$ produces a subset of $\{s\} \cup N_\alpha$ as output, where N_α is the set of α-nodes in $\mathcal{I}$.

LEMMA 6. *If A is monadic and sound, then A is α-outputting.*

LEMMA 7. *It is decidable whether a given α-sending DF automaton is α-outputting.*

We call a tree-like $\mathcal{I}$ *sparse* if there are at most k non-root α-nodes in $\mathcal{I}$.

Definition 7. A is *sparse-tree sound* if for every sparse $\mathcal{I}$, every multiple-round run of A on $\mathcal{I}$ produces the same output.

LEMMA 8. *Sparse-tree soundness is a decidable property of α-sending DF automata.*

The next lemma is central to the construction. Its proof is based on Lemma 1.

LEMMA 9. *Flat-invariance is a decidable property of flat- and sparse-tree sound, α-sending and -outputting, monadic DF automata.*

We are now in the position to sketch the proof of our main decidability result.

PROOF OF THEOREM 7. Consider a monadic DF automaton A. We call A *bounded* if for every flat $\mathcal{I}$ with precisely k non-root α-nodes, and for every one-round run ρ of A on $\mathcal{I}$, ρ is terminating (i.e., the source automaton sends the empty message during ρ). Otherwise, we call A *unbounded*.

First determine whether A is bounded or unbounded (simply by testing all non-isomorphic small flat trees). Suppose that A is unbounded. One can show that, if A is sound and continuous, then A must be α-sending. Check whether A is α-sending (see Corollary 5). If the test fails, reject A. Otherwise, check whether A is α-outputting (see Lemma 7). If this test fails, reject A (because A is not sound according to Lemma 6). Otherwise, check whether A is flat- and sparse-tree sound (see Lemmata 2 and 8). If one of the two tests fails, reject A (clearly, A cannot be sound in that case). Otherwise, check whether A is flat-invariant (see Lemma 9). If this test fails, reject A (because A is not sound according to Lemma 3). Otherwise, accept A, for it is sound and continuous due to Lemmata 3 and 4.

Now suppose that A is bounded. Note that A may not be α-sending in this case. An analysis similar to the one outlined above leads to a decision procedure for bounded automata. $\square$

Expressibility Results

This second part of the appendix concerns the proofs of Theorems 5 and 8. In the following, $\lambda_\alpha(s,x)$ denotes an *α-at-most literal*, i.e., a formula of the form $|\alpha| \leq k \wedge \gamma_\alpha(s,x)$ or the form $|\alpha| \leq k \rightarrow \gamma_\alpha(s,x)$.

PROPOSITION 3. *Any conjunction of α-at-most literals (in the variables s and x) is equivalent to an α-at-most literal. The same holds true for disjunctions of α-at-most literals.*

LEMMA 10. *Let $\varphi(s,x)$ be a formula of the form $\alpha(x) \wedge \lambda_\alpha(s,x)$. The Web query defined by φ, Q_φ, is computable by a continuous Web automaton.*

PROOF SKETCH. Suppose that λ_α is a positive literal, say, $\lambda_\alpha = |\alpha| \leq k \wedge \gamma_\alpha(s,x)$. We describe briefly a continuous automaton A_φ which computes Q_φ. A_φ is $(k+1)$-productive and sends messages of length $\leq k+1$. If A_φ is running at a node different from the source node, it forwards in each round as many as possible (but at most $k+1$) nodes satisfying α to its parent automaton. If A_φ is running at the source node, it attempts to see $(k+1)$ nodes satisfying α. If it succeeds, it sends the empty message, thereby terminating the computation. Otherwise, it knows all (reachable) nodes satisfying α. In particular, there are at most k such nodes. For each such node n, the source automaton checks whether $\gamma_\alpha(s,n)$ holds and, if successful, outputs n.

Now suppose that λ_α is a negative literal, say, $\lambda_\alpha = |\alpha| \leq k \rightarrow \gamma_\alpha(s,x)$. Modify A_φ as described above so that, if the source automaton discovers that there are at least $(k+1)$ nodes satisfying α, then, instead of sending the empty message, it outputs all nodes in its queue, plus the source node if the source node satisfies α. $\square$

Color Types. Let x be a variable. A *color type in x* is a maximal consistent set of atomic and negated atomic formulas in x. Observe that every quantifier-free formula $\alpha(x)$ with free$(\alpha) = \{x\}$ is equivalent to a disjunction of color types in x.

PROOF OF THEOREM 5. Let $\varphi(s,x)$ be an at-most-at-least formula. We construct a continuous automaton A_φ which computes Q_φ. Suppose that $\varphi(s,x) = \alpha(x) \wedge \delta_\alpha(s,x)$. Using Proposition 3, one can show that δ_α is equivalent to a formula of the form

$$\bigvee_i \left(c_i(s) \wedge \lambda_{\alpha,i}(s,x)\right) \qquad (1)$$

where each $c_i(s)$ is a color type in s such that $c_i \equiv c_j$ iff $i = j$. According to Lemma 10, for each index i in formula (1), there exists a continuous automaton computing $Q_{\alpha \wedge \lambda_{\alpha,i}}$. It is now an easy exercise to combine these automata to a continuous automaton A_φ computing Q_φ. $\square$

Remark 8. The proofs of both Lemma 10 and Theorem 5 can be arranged so that the constructed automata are link-free and DF.

PROOF OF THEOREM 8. Let Q be a Web query. Suppose that Q is definable by a monadic at-most-at-least formula. According to Theorem 5, Q is computable by a continuous Web automaton. By Remark 8, this automaton is monadic and DF.

Now suppose that Q is computable by a continuous monadic DF automaton A. Furthermore, suppose that A is $(k+1)$-productive. Let s and x be two variables, and let $c_1(s)$, ..., $c_\ell(s)$ be an enumeration of all color types in s (over the vocabulary of A, and up to isomorphism). Clearly,

$\bigvee_i c_i(s) \equiv (s = s)$. We are going to construct a quantifier-free formula $\alpha(x)$, and for each $i \in \{1, \ldots, \ell\}$, an α-at-most literal $\lambda_{\alpha,i}(s, x)$ such that the formula

$$\alpha(x) \wedge \bigvee_i \big(c_i(s) \wedge \lambda_{\alpha,i}(s, x)\big)$$

defines Q.

Let $\alpha(x)$ be define as in the previous subsection (see below Lemma 3). Intuitively, α specifies those (colorings of) leaf nodes which the source automaton can possibly see during any computation (recall Lemma 3).

The definition of $\lambda_{\alpha,i}(s, x)$ is based on various tests revealing the behavior of A when executed at c_i-colored source nodes. Choose pairwise distinct color types $c'_1(x), \ldots, c'_m(x)$ from the set $\{c_1(x), \ldots, c_\ell(x)\}$ so that $\alpha(x) \equiv \bigvee_j c'_j(x)$. Let $\mathcal{I}$ be a flat instance such that

- the root node r of $\mathcal{I}$ satisfies $c_i(s)$, and
- for each $j \in \{1, \ldots, m\}$, there are at least $(k + 1)$ leaf nodes satisfying $c'_j(x)$.

We are going to execute A at r (in $\mathcal{I}$) on various queues consisting of α-nodes.

By a *coordinate* $\bar{k}$ we mean a tuple $(k_1, \ldots, k_m)$ such that $k_1, \ldots, k_m \leq (k + 1)$. Let $\bar{k}$ be a coordinate. A *$\bar{k}$-queue* is a sequence of leaf nodes in $\mathcal{I}$ such that

- the length of the sequence is $\sum_{j=1}^m k_j$, and
- for each $j \in \{1, \ldots, m\}$, the sequence contains precisely k_j pairwise distinct nodes satisfying $c'_j(x)$.

Let q be a $\bar{k}$-queue. We say that A at r *accepts* q if the first message sent by A at r on q is not empty (i.e., contains a node).

Verify that for any two $\bar{k}$-queues q and q', A at r accepts q iff A at r accepts q'. Hence, we can define an m-dimensional table T_i as follows: at coordinate $\bar{k}$, T_i contains "accept" if A at r accepts any $\bar{k}$-queue; otherwise it contains "reject". By D_i we denote the diagonal plane of T given by all coordinates satisfying $\sum_{j=1}^m k_j = (k + 1)$. One can show that D_i has either only accept entries or only reject entries.

Next, observe that the definition of T_i does not depend on the choice of $\mathcal{I}$. We obtain the same table for any flat $\mathcal{I}$ whose root node satisfies $c_i(s)$, and which contains enough leaf nodes satisfying $c'_j(x)$ (for each j). This shows that the decision of whether A at a c_i-colored source node is going to output or not is entirely determined by the entries on and below the diagonal plane D_i, i.e., all entries at coordinates satisfying $\sum_j k_j \leq (k + 1)$.

Suppose that D_i has only reject entries. Let S be the set of those coordinates which satisfy $\sum_j k_j \leq k$ and where T_i has an accept entry. Define $\lambda_{\alpha,i}(s, x)$ to be

$$(|\alpha| \leq k) \wedge \bigvee_{\bar{k} \in S} \big(\gamma'_{\bar{k}} \wedge \gamma''_{\bar{k}}(s, x)\big) \tag{2}$$

where $\gamma'_{\bar{k}}$ and $\gamma''_{\bar{k}}$ are as follows. If $c_i(x)$ does not occur among $c'_1(x), \ldots, c'_m(x)$, set $\gamma'_{\bar{k}} = \bigwedge_j (|c'_j| = k_j)$. Otherwise, suppose that $c_i(x) = c'_p(x)$, and set $\gamma'_{\bar{k}} = (|c'_p| = k_1 + 1) \wedge \bigwedge_{j \neq p} (|c'_j| = k_j)$. By testing A, one can define $\gamma''_{\bar{k}}$ so that it specifies (i) those (colorings of) nodes in a $\bar{k}$-queue which are output, and (ii) whether or not s is output.

Now suppose that D_i has only accept entries. In that case, replace the first conjunction symbol in formula (2) with an implication symbol. $\square$