The navigational power of Web browsers*

Michał Bielecki,[1] Jan Hidders,[2] Jan Paredaens,[2]
Marc Spielmann,[3] Jerzy Tyszkiewicz,[1] Jan Van den Bussche[3]
[1]Warsaw University, Poland
[2]University of Antwerp, Belgium
[3]Hasselt University, Belgium

September 14, 2010

Navigare necesse est,
vivere non est necesse.
Pompeius

Abstract

We investigate the computational capabilities of Web browsers, when equipped with a standard finite automaton. We observe that Web browsers are Turing-complete. We introduce the notion of a navigational problem, and investigate the complexity of solving Web queries and navigational problems by Web browsers, where complexity is measured by the number of clicks.

1 Introduction

The Web browser is an indispensable piece of application software for the computer user. All the popular browsers essentially implement a very basic machinery for navigating the Web: a user can enter a specific Web address as some kind of "source node" to start his navigation; he can then further click on links to visit other Web nodes; and he use Back and Forward buttons to go along a stack of already visited nodes.

*A preliminary report of part of this research was presented at ICALP 2002. Research supported in parts by Polish KBN grants 7T11C 007 21 and 4T11 042 25 (M.B. and J.T.) and by FWO grant G.0246.99 (J.H.) Contact author: Jerzy Tyszkiewicz, Institute of Informatics, Warsaw University, Banacha 2, PL-02-097 Warszawa, Poland, e-mail jty@mimuw.edu.pl.

The question we want to address is the description of the power of the browser as a tool. We (the authors) still remember the early browsers and appreciate the new features that have been introduced since, like history in the form of marking already visited links with a distinctive color, or that the page accessed by the "back" button is reopened at the same location when we left it. Definitely, they help the human users of browsers. This immediately raises the question if these improvements can be given mathematical, precise meaning. This paper offers a precise formulation of this problem and provides a number of results.

First, we must give the browser a formal meaning.

For computer scientists it is natural to equip a Web browser with a standard finite automaton, and wonder about the computational capabilities of these automata. We believe, that these capabilities reflect the power of browser as a tool. After all, a lot of application software, such as word processors or spreadsheets, allows in addition to the standard "manual" use of the software, also some kind of "programmed" use, by allowing the user to write macros which are then executed by the software tool. Such macros are typically simple programs, which offer the standard test and jump control constructs; some variables to store temporary information; and for the rest are based on the basic features offered by the application. This reflects the extremely common method of creating macros, by recording the actions taken by a human user and then repeating them, perhaps in a loop. They can also been written in a graphical programming environment (such as Apple's Automator) where the user drags actions into some flowchart.

In this spirit, we introduce in this paper the *browser automaton* (or just "browser" for short). This automaton model is simply obtained by equipping a finite-memory automaton with the basic features offered by a Web browser and already summarized above: clicking on a link; going "back"; and going "forward". Here, a finite-memory automaton [13] is simply an automaton with finite control and a finite number of registers (which we use to store Web addresses). The model thus obtained can also be regarded as a restriction of the *browser machine* model introduced earlier by Abiteboul and Vianu [2], which has an unlimited Turing tape for storing Web nodes, rather than just the finite memory plus the back and forward stacks which we have here.

Just like Alan Turing was interested in understanding the problems solvable by a computer following a formal algorithm, using only pencil and sufficient supply of paper, we are here interested in the problems solvable by a mechanical browser. We believe that what is impossible for browser automata is also impossible for human users of browsers, and that complexity limitations of browser automata apply to humans as well.

However, while Turing could easily define a "problem" as a function on the natural numbers, what kind of computational problems over the Web can we consider in our setting?

A first, basic, approach to this question would be just to consider these classical computable functions, by simple analogy to Turing. From this perspective, we will show that browser automata are actually Turing-complete: they can simulate any computable function. While this result initially surprised us, such an approach is clearly not the only one.

Indeed, a more natural approach was taken by Abiteboul and Vianu, as well as Mendelzon and Milo [17], who considered the Web as a database, and studied the power of browser machines in answering *queries* to this database. We will consider this approach in this paper, and show here that browser automata can answer any Web query that is computable in logarithmic space.

In this paper, however, we also propose a third approach, which is to focus on *navigational problems*. A navigational problem asks the browser to visit a certain specified set of Web nodes, and no others. Thus, navigational problems formalize browsing tasks, avoiding "getting lost in hyperspace" [7, 9]. Specifically, we focus on *structural* navigational problems only, where the browser has to solve the problem dependent purely on structural information of the Web graph alone. More advanced models could also introduce predicates on Web nodes so that the browser can detect various properties of the nodes, but we feel the basic "uncolored" model remains fundamental, and resembles quite well what human users of browsers do.

Toward investigating the navigational power of browsers, we introduce a notion of "data-transfer optimal" browser, in which a node is never downloaded more than once. We show that this is a real restriction, by exhibiting several natural navigational problems that cannot be solved in such an optimal manner, and by providing a rather general necessary condition on the structure of such problems. We also provide concrete lower bounds on the number of data transfers a browser has to make to solve certain simple problems. Our proof employs basic communication complexity theory. Finally, we investigate a new feature for Web browsers: switching the back and forward stacks. We show that this feature strictly increases the navigational power of browsers, and that it also allows problems to be solvable with provably less data transfer.

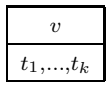
As discussed in the conclusion, our results also provide a first direction in which the complexity of user interfaces (in our case user interfaces for browsing the Web) can be studied.

2 Web instances and browser automata

2.1 Web instances

Definition 1. A *Web graph* is a finite, locally ordered directed graph $\mathbb{V} = \langle V, l, O \rangle$. Here, V is the finite set of nodes of $\mathbb{V}$ (we always use the matching Roman letter for the set of nodes of any Web graph denoted by a blackboard-font letter), l is the edge relation (we also call it the link relation), and O is a ternary relation such that for each node x, the binary relation $O^x := \{(y, z) \mid (x, y, z) \in O\}$ is a total order of the nodes linked from x.

Definition 2. A *page* of a Web graph $\mathbb{V}$ is pair (v, S), where $v \in V$ and $S = t_1, \ldots, t_k$ is the ordered sequence of nodes linked from v. We will refer to S as the *list of children of* v. We will often depict such a page in the following way:



A Web graph can be equivalently represented by the set of its pages.

Definition 3. A *Web instance* $(\mathbb{V}, s)$ is a Web graph $\mathbb{V}$ with a distinguished node s, and such that all nodes of $\mathbb{V}$ are reachable by links from s, which is henceforth called the *source*.

The source node is where browsing starts in the Web graph. Obviously, nodes not reachable from the source are irrelevant to browsing, hence the reachability requirement. This formalization of Web instance is similar to earlier formal models of the Web, e.g., that by Abiteboul and Vianu [2].

A very simple example of a Web instance is shown in Figure 1. Node 1 is the source.

2.2 Browser automata

The general idea of the following definitions is to create a model of an automated Web browser, as a finite state automaton equipped with the principal navigation facilities offered in common Web browsers such as Internet Explorer, Mozilla Firefox, Opera, or Safari. A crucial feature of these browsers, which our formalization includes, is that the back and forward stacks not only remember the addresses of previously visited web pages, but also the scrolling position on a page at the time of leaving it by clicking a link or the Back or Forward button. We model the scrolling position by a tape that contains the list of children (outgoing links) of the current Web page.

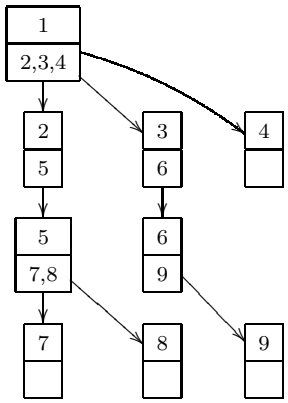


Figure 1: Example of a Web instance.

Definition 4. A *browser automaton* (or just "browser" for short) is a finite state computing device equipped with the following ingredients:

1. Components:
   (a) A finite state control.
   (b) A read-only tape, which stores a sequence of nodes of a Web graph. The tape is accessed by a two-way head, and can sense the beginning and the end of the tape. Note that since Web graphs can have arbitrarily many nodes, there is no fixed finite tape alphabet as in classical finite automata.
   (c) A finite number of registers, each one capable of storing a single node of a Web graph.
   (d) Two stacks, denoted $\Rrightarrow$ and $\Lleftarrow$. Each stack will hold pairs (v, i), where v is a node, and i is a natural number.

   The automaton is run on Web instances $(\mathbb{V}, s)$. During the run there is always a *current node*; the initial current node is s. The tape will invariantly hold the list of children of the current node. The initial content of the two stacks is empty.

2. The following actions can be undertaken, as ordered by its finite control:
   (a) Change the control state.
   (b) Move the head left or right on the tape.

(c) Store in a register the identity of the current node, or the node currently seen by the head on the tape.

(d) Go *forward*, which consists of pushing the current node and head position on the $\Leftarrow$ stack, and popping the $\Rightarrow$ stack, yielding a new current node and head position. This action is impossible if the $\Rightarrow$ stack is empty.

(e) Go *backward*, which consists of pushing the current node and head position on the $\Rightarrow$ stack, and popping the $\Leftarrow$ stack, yielding a new current node and head position. This action is impossible if the $\Leftarrow$ stack is empty.

(f) *Click*, which causes the following to happen. Let v be the node seen by the tape head. Then the current node and head position are pushed on the $\Leftarrow$ stack; the new current node is v (thus producing a new tape content as well); and the new head position is 1. The $\Rightarrow$ stack is erased.

(g) Halt.

3. The following information determines the next state and the next move of the machine:

   (a) The current control state.

   (b) Equalities and non-equalities among the values of registers, the current node, and the node seen by the tape head.

   (c) Information on whether the tape head is at the beginning or the end of the tape (i.e., the first or the last child of the current node), or whether the current node has no children at all.

   (d) Information on whether any of the stacks is empty.

A more formal definition of a browser automaton, and its computation on an instance, is easily produced and left to the reader.

As an illustration, Table 2 shows an example of a sequence of possible actions performed on the Web instance of Figure 1.

Definition 5. A *browser automaton with history* is like an ordinary browser automaton, with the additional feature that the next state and action of an automaton can depend additionally on whether the automaton, in its current run, has already visited the node seen by the tape head.

Table 1: A sequence of actions a browser automaton might perform on the Web instance of Figure 1. Note that registers are not used.

action	current node	tape	$\Leftarrow$	$\Rightarrow$
initial	1	$\underline{2}34$	empty	empty
click	2	$\underline{5}$	(1,1)	empty
click	5	$\underline{7}8$	(1,1)(2,1)	empty
right	5	$7\underline{8}$	(1,1)(2,1)	empty
back	2	$\underline{5}$	(1,1)	(5,2)
forward	5	$7\underline{8}$	(1,1)(2,1)	empty
back	2	$\underline{5}$	(1,1)	(5,2)
back	1	$\underline{2}34$	empty	(5,2)(2,1)
right	1	$2\underline{3}4$	empty	(5,2)(2,1)
forward	2	$\underline{5}$	(1,1)	(5,2)
back	1	$2\underline{3}4$	empty	(5,2)(2,1)
click	3	$\underline{6}$	(1,2)	empty

Equivalently, browsers with history are equipped with a perfect recall memory, which equals the set of all the nodes already visited, and is allowed to test whether the node under the tape head is in that set.

Under the latter definition each browser can be understood as a browser with history that never makes use of its memory.

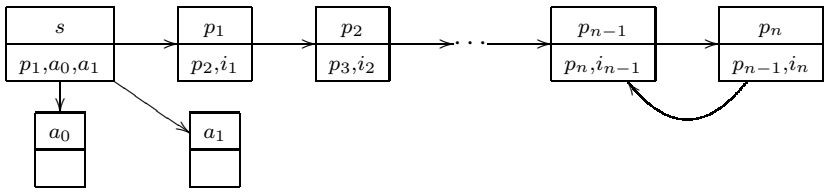
A history mechanism is indeed present in most of the Web browsers used in practice. We will still study browsers without history as well. Indeed, their theory is very appealing: much of the computational power of browsers is already exhibited without using the history. Yet, we will also show that history does help to solve more navigational problems.

We conclude this section with a remark on the use of registers. In the original finite-memory automata of Kaminski and Francez [13], different registers cannot store the same node, but transitions can be based only on equalities, not nonequalities. In such a transition model, allowing the same node in different registers leads to a weaker model in which classical decision problems such as universality or inclusion become decidable [14]. In our work, we are not focused on such decision problems, and in our browser automaton model, transitions can be based on nonequalities among registers as well as equalities, so it becomes inconsequential whether or not different registers can store the same node.

3 Turing-completeness of browsers

In this section, we show that browsers can simulate arbitrary Turing machines. Without loss of generality, we restrict attention to single-tape Turing machines with the binary tape alphabet $\{0, 1\}$.

For the simulation, we need a way to represent binary words as Web instances. So, let $w = w_1 \dots w_n \in \{0, 1\}^*$ be a binary word. We represent w by the Web instance $(\mathbb{V}_w, s)$, where $\mathbb{V}_w$ is as follows:



Here, additionally (not depicted), each link i_j points to the node a_l iff $w_j = l$. Note the loop between p_n and p_{n-1}.

By designating two special output states for 0 and 1, we can easily agree that a browser B *outputs* 0 (1) on a given input word w if B, when run on the instance $(\mathbb{V}_w, s)$, halts in the output state 0 (1). We can then further agree that B *computes* a boolean function f (i.e., a partial function $f : \{0, 1\}^* \to \{0, 1\}$), if for every binary word w, the browser B, when run on the instance $(\mathbb{V}_w, s)$, halts in one of the two special output states iff $f(w)$ is defined, and in that case, the output indeed equals $f(w)$.

Theorem 1. *Every Turing-computable boolean function can be computed by a browser.*

Proof. Let f be a boolean function computed by the Turing machine M. We have assumed that M's tape alphabet is $\{0, 1\}$, extended by the blank symbol which, however, cannot be written by M. Initially the tape contains the input word, with tape cells after that holding blanks. Each time M enters a blank tape cell for the first time, it will overwrite it with 0 or 1. We can again assume that M outputs 0 or 1 by halting in one of two designated output states.

It now suffices to show how the working of M on an input word w can be simulated by a browser B_M working on the input instance $(\mathbb{V}_w, s)$. The browser B_M works according to the following instructions:

1. Starting from s, store the addresses of a_0 and a_1 in two registers. (Here and in the following, B_M identifies links by their order of appearance

on the pages. There are always at most four of them, so all of the necessary information can be encoded in the finite state control.)

2. Next, follow repeatedly the first link on each page, until a two-page loop is found. Such loop can be easily detected by comparing the link with the address of the previous page (which can be kept in a register). When this loop is found, B_M must be in p_n, and the $\Leftarrow$ stack content is $(s,1),(p_1,1),...,(p_n,1)$.

3. Memorize p_n in a register.

4. Now repeatedly perform the following actions, until we are back at s:

   (a) Compare the second link on the page with the stored addresses a_0, a_1.

   (b) If it is a_0 (a_1), then place the head on the first (second) link on the page.

   (c) Go backward once.

5. Presently, the position of the head on each of the pages $p_1, \ldots, p_n$, as they are stored on the $\Rightarrow$ stack, indicates the corresponding symbol of M's input word w.

6. Now start simulating M, by going backward and forward exactly like the head of M walks left and right on its tape. When M writes 0 (or 1) on the current tape cell, place the head on the current page on the first (second) link. In this simulation, the current page of B_M corresponds to the tape cell currently seen by M; the position of the head on the current page corresponds to the symbol written on the tape cell; the $\Leftarrow$ stack corresponds to the portion of M's tape to the left of the head; and the $\Rightarrow$ stack corresponds to the portion of M's tape to the right of the head (as far as it has already been visited by M). In particular, the distinction between a_0 and a_1 is at this point no longer relevant.

7. If M wants to go right but the $\Rightarrow$ stack is empty, then simulate the creation of a new tape cell as follows: *(i)* memorize in a register the link in the current page that is currently under the head; *(ii)* click on the first link of the current page (which is always p_{n-1} or p_n in this case); *(iii)* make one backward step; *(iv)* restore the head position to the memorized position; *(v)* make one forward step. We are then ready to continue the simulation.

8. Halt whenever M does.

Note that when M uses a lot of space, the vast majority of the pages stored on the stacks during the simulation are copies of p_{n-1} and p_n. However, for different copies of these pages, the stacks hold different head positions, corresponding to the symbols written by M on its different tape cells. □

An obvious corollary of the Turing machine simulation exhibited in the proof of the above theorem is that the halting problem for browsers is undecidable.

We observe that the proof depends crucially on the loop that is present in the representation of a word as a Web instance. Hence, it is a natural question to ask what happens if we restrict attention to acyclic Web graphs:

Theorem 2. *Restricted to acyclic instances, browsers (even with history) can do only linear-space computations, and every boolean function computable in space n by a Turing machine with two-element tape alphabet can be simulated by a browser working on acyclic instances only.*

Proof. The linear-space upper bound is clear, given that the number of registers is fixed, and, in the absence of cycles, the stacks and history never become larger than the input Web instance. The simulation of space n Turing machines with two-element tape alphabet is a straightforward adaptation of the proof of Theorem 1: we can omit the loop between p_{n-1} and p_n in the Web graph $\mathbb{V}_w$ simulating a word w. □

Many PSPACE-complete problems are solvable in linear space, [18, Chapter 19]. Using simple padding to allow for encoding of larger tape alphabet, a corollary of the above theorem is that the data complexity [20, 16] of browsers on acyclic instances is PSPACE-complete.

4 Web queries

So far, we have obtained a first impression of the computational power of browsers, by having looked at their ability to simulate Turing machines. In a natural next step, we now give arbitrary Web instances as input to browsers (rather than just special instances that represent binary words), and look at the ability of browsers to answer Web queries. Intuitively, a Web query asks to retrieve a certain set of "interesting" pages. Formally, we define:

Definition 6. A *Web query* is a mapping Q from Web instances to finite sets of nodes, such that

- $Q(\mathbb{V}, s)$, whenever defined, is a subset of V; and
- if $\alpha : (\mathbb{V}, s) \to (\mathbb{V}', s')$ is an isomorphism, then $Q(\mathbb{V}', s') = \alpha(Q(\mathbb{V}, s))$.

The second condition is a consistency criterion well established in database theory [4] (often called "genericity" [1]). In our case it essentially says that the precise identities of the nodes in a web graph are unimportant; only the way they are linked together, and the order in which the links appear on a page, is important.

By designating a number of states as output states, a browser can be used to compute Web queries. Indeed, whenever, during its run on a given instance, the browser switches to an output state, we consider the current node to be output. (In practice, the id of the current node could be written to an output file.) After halting we can then consider the set of nodes that were output as the output set of the run. We then agree that *a browser B computes a Web query Q* if B halts on every Web instance I, with precisely the set $Q(I)$ as output.

Note that we can regard decision problems about Web instances as a special kind of Web queries, which could be called "yes/no queries"; such a query either outputs the empty set (interpreted as "no") or outputs just the source node (interpreted as "yes"). Note also that the simulation of boolean functions on strings, as discussed in the previous section, can be viewed as yes/no Web queries on a special kind of Web instances. As a consequence, in view of Theorem 1, there is no upper bound (other than computability) on the computational complexity of Web queries computable by a browser.

In sharp contrast to Theorem 1, however, certainly not all computable Web queries are computable by a browser, even with history. Indeed, by the linear-space upper bound on acyclic instances observed in Theorem 2, any Web query that is not computable in linear space on acyclic instances is not computable by a browser with history. One can actually do better: there are also linear-space computable Web queries that are not computable by a browser with history. Consider so-called "simple" Web instances: instances where the Web graph is a simple path starting in the source. Simple instances can be identified with unary strings (i.e., strings over a one-letter alphabet). On simple instances, all a browser can remember are a fixed number of nodes in the registers, plus the history, but that history is determined by the farthest node visited so far, and thus can be eliminated at the cost of two additional registers. So simulating such a browser with a Turing

machine we really need only a constant number of counters for storing node positions, a logarithmic number of bits in total. Consequently, any property of unary strings decidable in linear space but not in logarithmic space yields an example of a Web query computable in linear space but not computable by a browser with history.

Our main goal now in this section is to prove the following positive result.

Theorem 3. *Every Web query that is computable in logarithmic space by a Turing machine is also computable by a browser, without history.*

Here, by "a Web query computable in logarithmic space", we mean this in the conventional sense of computability by Turing machines; we assume some standard encoding of Web instances as inputs for Turing machines, and we represent the nodes of the Web graph by the numbers 1 to n, with n the number of nodes.

The proof of Theorem 3 is based on two lemma's, the second a strengthening of the first.

Lemma 1. *The Web query that simply returns all nodes of the given Web instance is computable by a browser without history.*

Proof. Recall that all nodes in a Web instance are reachable from the source. The desired browser performs a depth-first left-to-right traversal of the graph, starting in the source, by running the following program. Start by repeatedly clicking on the first link on the current page, until we reach a page without outgoing links. When this happens, backtrack by going backward one step, and move the head one position to the right. If the head was already placed at the end of the page (so that we cannot move the head to the right), continue to backtrack until we find a tape configuration where we can move the head one position to the right. Now continue the depth-first traversal by clicking the link currently under the head, and continue. according to the previous instructions. When we can no longer backtrack, we are finished. We still owe an explanation on how to avoid running in a cycle. We do this by avoiding to click on a link to a node that occurs already on the $\Leftarrow$ stack; instead, we then immediately move the head one position to the right. Thereto, before actually clicking on a link to a node v, we examine all tape configurations stored in the $\Leftarrow$ stack, by repeatedly going backward, and make sure that in each such configuration the head is not placed on v. To resume the traversal after performing this test, go forward again until we have returned to the link to v. □

As a corollary to the above proof, we note:

Corollary 1. *A browser without history can check for the existence of a cycle.*

The browser described in the above proof makes multiple visits to every node that is reachable from the source by multiple simple paths. We next show that we can adapt the browser to detect that we have already visited a node earlier, so that we can make sure every node is output only once.

Lemma 2. *There exists a browser (without history) that, on any given instance, outputs all nodes without repetitions, in depth-first left-to-right order, and then halts.*

Proof. The idea is to refine the basic depth-first traversing browser from the proof of Lemma 1. So, we refer to that as the *basic traversal*, and we are now going to define what we call the *refined traversal*. The idea is the following. Suppose we are ready to click on a link to a node v. Before we do this, we test whether we have already visited v before; if so, v is not visited and we immediately move the head one position to the right.

We perform this crucial test as follows. Let p be the current page (thus containing a link to v). We remember p and v in two registers. We now go backward, all the way back to the source, and restart a basic traversal; we refer to this inner traversal as the "subtraversal initiated at (p, v)". Now when, during this subtraversal, we click on v, we remember this fact. Moreover, the subtraversal is aborted the first time we find ourselves back at p and ready to click on v. If, at this point, the result is that we have indeed clicked on v during the subtraversal, the test succeeds (i.e., the refined traversal will skip v).

To show that this program has the desired behavior, we must show two properties.

1. When an inner traversal is aborted, the $\Leftarrow$ stack has the same contents as at the point when the refined traversal initiated that inner traversal. This property is fundamental, as it guarantees that the refined traversal, after it was interrupted for a subtraversal, can resume in the same state where it was left.

2. The refined traversal visits each node exactly once, in depth-first left-to-right order.

To show these two properties, we start from the observation that the basic traversal traces all possibly simple paths that start in the source node. Each time a node v is visited, the content of the $\Leftarrow$ stack equals one of the

possibly many simple paths from the source to v, and every simple path from the source to every possible node is followed exactly once. (We are talking here about the *basic* traversal.) We can actually order the set of all possible simple paths starting in the source by the depth-first left-to-right order; this total ordering coincides with the order in which the basic traversal will follow the paths. In particular, the sequence of final nodes of all those paths, listed in order, spells out the sequence of visits the basic traversal will make.

We now narrow down property 2 as follows:

2. The refined traversal visits each node v precisely when coming from the *first* path to v.

The depth-first left-to-right order of output is then obvious.

Now to the actual proof of the two properties. We will prove them simultaneously by induction on the number of nodes already visited by the refined traversal (the base case, where we visit the source, is trivial).

1. Suppose the refined traversal is currently at node p and is going to test whether it has already visited node v (which is thus a child of p). By induction, we know that the current content of the $\Leftarrow$ stack equals the first path to p. When the inner traversal is aborted, it visits p for the first time, and hence the $\Leftarrow$ stack will at this point again contain the first path to p.

2. Suppose the refined traversal decides to click on v, coming from p. Then we must show that the first path to v equals πv, where π is the first path to p. So, suppose to the contrary that a path $\pi' v$ to v smaller than πv would exist. Let q be the last node of π'. Note that $q \neq p$, so q is visited before p by the basic traversal. But then, since v must be a child of q, the subtraversal initiated at (p, v) would visit v, and hence the refined traversal would not click on v, a contradiction.

   Conversely, suppose the refined traversal is at a node p, deciding whether or not to click on v, and assume that πv is the first path to p, with π the current content of the $\Leftarrow$ stack, or equivalently, the first path to v. Then we must show that the refined traversal will indeed click on v. So, suppose to the contrary that the subtraversal initiated at (p, v) would already visit v. Let $\pi' v$ be the path to v followed for that visit; clearly, $\pi' v$ comes before πv in the dept-first left-to-right order. But πv was assumed to be the first path to v, a contradiction.

□

We are now equipped for the

Proof of Theorem 3. We will use a theorem of descriptive complexity theory and finite model theory [11, 8, 16], due to Immerman [12], stating that every logspace query about finite ordered relational structures is expressible in deterministic transitive-closure logic; this logic is denoted by FO(DTC) and its definition will not be repeated here. Note that by their very definition, Web instances $(\mathbb{V}, s) = (V, l, O, s)$ are indeed relational structures, over the vocabulary (l, O, s), where l appears as a binary relation symbol, O as a ternary relation symbol, and s as a constant symbol. Ordered Web instances then are relational structures $(\mathbb{V}, s, <)$, with $<$ an additional total order relation on V.

Of course a Web query, as we defined it, is defined on ordinary Web instances, not ordered ones, but we can easily agree that a Web query can be applied also to ordered Web instances simply by ignoring the given order relation. We then obtain that for any logspace Web query Q, there exists a FO(DTC) formula $\varphi(x)$ such that for any Web instance $(\mathbb{V}, s)$, any total ordering $<$ of V, and any node $v \in V$, we have $v \in Q(\mathbb{V}, s)$ iff $(\mathbb{V}, s, <) \models \varphi(v)$.

It therefore suffices to show that browser automata can evaluate FO(DTC) formulas on ordered Web instances, where the order relation is always the depth-first left-to-right order, which we denote by $<^{\text{dflr}}$.

We start with the following.

Lemma 3. *There exists a browser (without history) $B_{\text{succ}}(x)$ with a register x that, on any given instance $(\mathbb{V}, s)$ and with any initial values of registers and any initial contents of $\Leftarrow$ and $\Rightarrow$, outputs the immediate successor of the value of register x according to the depth-first left-to-right order on $(\mathbb{V}, s)$, and then halts. If no such successor exists, it halts and returns the initial value of x.*

Proof. Let v be the value of x. We run the browser from Lemma 2 from scratch, comparing the nodes it outputs with v and do so until it outputs v, and then continue until it outputs the next node v' (i.e., one immediately following v in the enumeration). If it does so, we output v', and if it terminates without producing such v', we output v and terminate, too. □

Now, it suffices to show that for every FO(DTC)-formula $\varphi(x)$ there is a browser B_φ having a register x that, when started on a Web instance $(\mathbb{V}, s)$

in a configuration where register x contains a node v, irrespectively of the content of $\Leftarrow$ and $\Rightarrow$, will determine the truth of $\varphi(v)$ in $(\mathbb{V}, s, <^{\text{dflr}})$.

Indeed, we initialize v with s. Each time, we store the present v in a register, run B_φ to verify whether $\varphi(v)$ is satisfied, and if so, output v. Then we run B_{succ} on v to get the successor v' of v. If the output is equal to v we terminate, and otherwise v' takes over the role of v and the procedure is repeated.

Since the construction of B_φ is by induction on the structure of φ, and subformulas of φ may have multiple free variables, we will actually consider general formulas $\varphi(x_1, \ldots, x_m)$ with multiple free variables, and show that there always exists a corresponding browser B_φ with registers x_1, ..., x_m, that evaluates φ on the tuple of the nodes stored in those registers. For atomic formulas of one of the three forms $x_i = x_j$; $l(x_i, x_j)$; or $O(x_i, x_j, x_k)$, this is easy. For atomic formulas of the form $x_i < x_j$ we can use the browser from Lemma 2.

If φ is of the form $\psi \vee \chi$ or $\neg\psi$, we can use the browsers B_ψ and B_χ in the obvious manner.

If φ is of the form $\exists x_{m+1}\, \psi(x_1, \ldots, x_m, x_{m+1})$, we use the browser from Lemma 3 to try all possible nodes x_{m+1} one by one, and for each of them test using the browser B_ψ whether $\psi(x_1, \ldots, x_m, x_{m+1})$ is satisfied.

Finally, if φ is of the form $[\text{DTC}\, \psi](x_1, \ldots, x_k, y_1, \ldots, y_k)$, we work as follows. Using the browser from Lemma 3, we can run through all possible k-tuples of nodes. In this way we can also simulate a counter that counts up to n^k, with n the number of nodes of the Web instance. If $(y_1, \ldots, y_k)$ is reachable from $(x_1, \ldots, x_k)$ in the relation defined by ψ, this is possible in n^k steps. So, in a first step, we initialize registers z_1, ..., z_k with the contents of x_1, ..., x_k, and look for a tuple $\bar{u} = (u_1, \ldots, u_k)$ such that $\psi(z_1, \ldots, z_k, u_1, \ldots, u_k)$ is satisfied (using B_ψ to test this). If we find no such $\bar{u}$, we report that φ is not satisfied. If we do find a suitable $\bar{u}$, and $\bar{u}$ is actually equal to $\bar{y} = (y_1, \ldots, y_k)$, we report that φ is satisfied. Otherwise, we assign u_i to z_i and repeat. If after n^k repetitions $\bar{y}$ has still not been encountered, we report that φ is not satisfied. □

We conclude this section with two remarks. First, we notice that on any class of acyclic instances where there is a fixed upper bound on the depth of the longest path starting in the source, browsers without history are confined to logarithmic space. Hence, on these classes, browser automata without history capture precisely logarithmic space.

Second, one may naturally wonder about an example of a Web query computable in polynomial time, but not computable by a browser automaton

without history. By Theorem 3, however, finding such a query would entail separating polynomial time from logarithmic space, which is a well-known open question from complexity theory [18].

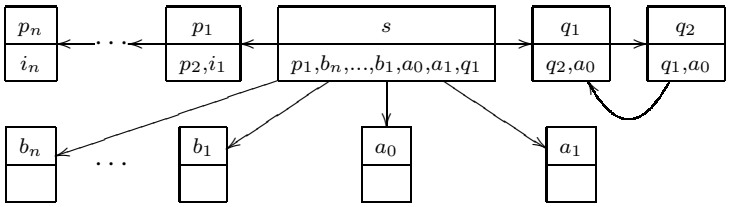
5 The power of history

It is a natural question to ask whether the history mechanism actually makes browsers more powerful. We answer this affirmatively in this section.

Theorem 4. *There exists a yes/no Web query that can be computed by a browser automaton with history, but not by any browser automaton without history.*

The proof, given next, is combinatorial in nature and does not yield a natural example of a Web query satisfying the statement of the theorem. Giving such an example is an interesting open problem.

Proof. We will adapt the representation of binary strings by Web instances that we already used for Theorem 1. Let $w = w_1 \dots w_n \in \{0,1\}^*$ be a binary word. Before, we represented the letters of w using a linear path $p_1 \dots p_n$ of n nodes from the source. Additionally, there was a cycle between p_n and p_{n-1} for the purpose of simulating arbitrary computations. Now, we separate the representation of the input word from the computing cycle; moreover, we add n additional nodes $b_1, \dots, b_n$ in between. Concretely, we now represent w by the Web instance $(\mathbb{V}_w, s)$, where $\mathbb{V}_w$ is now as follows:



Here, as earlier, each link i_j points to the node a_l iff $w_j = l$.

To every boolean function $f : \{0,1\}^* \to \{0,1\}$ we can associate a yes/no Web query Q_f as follows:

$$Q_f(\mathbb{V}_w, s) := \begin{cases} \emptyset & \text{if } f(w) = 0 \\ \{s\} & \text{if } f(w) = 1 \end{cases}$$

For any instance I which is not $(\mathbb{V}_w, s)$ for any $w \in \{0,1\}^*$, we put $Q_f(I) := \emptyset$. (Notice that a browser, even without history, can check whether its input instance is of the form $(\mathbb{V}_w, s)$.)

We are going to show:

1. For every computable f, the Web query Q_f is computable by a browser automaton with history.

2. There exists a computable f such that Q_f is not computable by any browser automaton without history.

The first claim is a variation on the Turing machine simulation from the proof of Theorem 1. The added difficulty is in the beginning, where we need to transfer the input bits, represented using the p-nodes, to the tape, simulated using the (q_1, q_2)-cycle. This transfer happens in two phases: in the first phase, for every $i = 1, \ldots, n$, we are going to click the link from s to b_i iff bit $w_i = 1$. This is easily accomplished, using registers to remember you are in the p's and in the b's. In the second phase, we then write the bits $w_1 \ldots w_n$ on the Turing machine tape. Here, we read off the bits from the links to the b-nodes, using the history mechanism, and we construct and write on the Turing machine tape as in the proof of Theorem 1. Indeed, since we no longer need to visit the p-nodes in this phase, the $\Rrightarrow$ stack over the nodes q_1 and q_2, used to simulate the tape, is left intact when inspecting the b-nodes.

Now to the second claim. Fix an arbitrary natural number n, and consider a browser automaton B without history working on instances $(\mathbb{V}_w, s)$ where the length of w is n. We refer to such instances as "instances of length n". When, during its run, B clicks for the first time on q_1, we say that B enters "computational mode". It leaves this mode when it clicks on p_1. When, later, B clicks on q_1 again, it enters computational mode again, and so on. Thus, during the run, B will be in computational mode in some periods, and not in the other periods.

Proposition 1. *Each time upon either entering or leaving computational mode, the $\Lleftarrow$ and $\Rrightarrow$ stacks are empty.*

Upon entering computational mode, by the above proposition, B's configuration consists of the current state, and the contents of the registers. Every register can hold either (i) the source; (ii) a b-node; (iii) q_1 or q_2; or (iv) a p-node.

We thus must distinguish among $1 + n + 2 + n = 2n + 3$ possible contents per register. For n such that $2n+3$ is at least the number of different states, we can thus represent a configuration upon entering computational mode as a vector in $\{1, \ldots, 2n + 3\}^{r+1}$, with r the number of different registers.

Also upon leaving computational mode, by the proposition stacks are empty. Some registers will have a new value (which can then only be a node of the first three kinds), and the new state assumed by B is also important. Thus, the configuration after leaving computational mode can be represented as a vector in $\{1, \ldots, 2n+3\}^{r+1}$.

In summary, the global behavior of B on instances of length n, as far as computational mode is concerned, can be fully described by a function

$$G_n : \{1, \ldots, 2n+3\}^{r+1} \to \{1, \ldots, 2n+3\}^{r+1}$$

that describes how the configuration changes by entering, and leaving again, computational mode.

Without loss of generality, we may restrict attention to browsers that do their output and halt outside of computational mode. This allows us to regard B, on instances of length n, as a combination (G_n, B'), where B' is a "reduced" browser B' with access to G_n as an oracle function. Browser B' works not on the full instance $(\mathbb{V}_w, s)$, but on the instance $(\mathbb{V}'_w, s)$, where $\mathbb{V}'_w$ is $\mathbb{V}_w$ without the nodes q_1 and q_2. When B would enter computational mode, B' directly invokes the oracle on its current configuration.

Going one step further, we can now consider a reduced browser R by itself, and combine it with different oracle functions $G : \{1, \ldots, 2n+3\}^{r+1} \to \{1, \ldots, 2n+3\}^{r+1}$, with r the number of registers of R. So, consider such a pair (R, G), and consider a boolean function $f : \{0,1\}^n \to \{0,1\}$ on words of length n only. We can then naturally agree that (R, G) computes f if, on any instance $(\mathbb{V}'_w, s)$, with $w \in \{0,1\}^n$, the combined browser (R, G) outputs "yes" ("no") if $f(w) = 1$ ($f(w) = 0$). Now observe that there are 2^{2^n} different boolean functions on words of length n, whereas there are only $(2n+3)^{(r+1)(2n+3)^{r+1}} = 2^{n^{O(r)}}$ different functions $G : \{1, \ldots, 2n+3\}^{r+1} \to \{1, \ldots, 2n+3\}^{r+1}$. Hence, for each reduced browser R, and n sufficiently large when compared to r, there exists a boolean function on words of length n that cannot be computed by R combined with any oracle function G. We call f "uncomputable for R".

The idea now is to combine all those uncomputable boolean functions, for all possible reduced browsers, and for different input lengths. This combination will produce the desired computable $f : \{0,1\}^* \to \{0,1\}$ such that no browser automaton without history can compute the Web query Q_f. The formal definition of f follows next.

List all possible reduced browsers in some arbitrary but fixed, computable, enumeration: R_1, R_2, ... We now define a sequence $(n_i)_i$ of natural numbers and a sequence $(f_i)_i$ of boolean functions, where each f_i is

defined on bitstrings of length n_i. For $i = 0$, we put $n_0 := 0$. For $i > 0$, we define n_i as the first natural number larger than n_{i-1} and large enough so that there exists a function on length-n_i bitstrings uncomputable for R_i. We then define f_i as the first such function (in, say, the lexicographic order). We finally define $f(w)$, for an arbitrary $w \in \{0,1\}^*$, as follows. Let n be the length of w. If n occurs in the sequence $(n_i)_i$ just defined, say $n = n_j$, then we define $f(w) := f_j(w)$. If n does not occur in the sequence, we put $f(w) = 0$. It is now evident that f is effectively computable and that Q_f is not computable by any browser automaton without history. □

6 Click complexity

In Web browsers in practice, clicking on a link causes data transfer to happen, even if the page was already visited earlier. The pages stored on the stacks, however, are generally cached on the client side, so visiting them again may not cause additional data transfer. Thus, by measuring how many times a browser clicks during its computation, we can measure the data transfer it generates.

It is an important measure both for the client and the server owner. Using Internet access over mobile phone networks, the users are often charged for transfer, or there is a limit on the total transfer quantity in billing periods. So it is desirable for the user to avoid clicking more than necessary.

On the other hand, for the server owner, each transfer consumes the resources: bandwidth and server workload. It is therefore desirable to organize his/her site structure to minimize the need of unnecessary clicks by the users.

Formally, we define:

Definition 7. For a browser B and a Web instance I, we denote by $C_B(I)$ the number of clicks during the run of B on I, and we denote by $C_B(n)$ the maximum number of clicks during the run of B on an instance of size n.

We observe the following relationship between click complexity and space complexity:

Proposition 2. *Each browser B with history can be implemented in space $O(n + C_B(n) \log n)$; without history, in space $O(C_B(n) \log n)$.*

Proof. Since the total length of the $\Leftarrow$ and $\Rightarrow$ stacks can increase only with every click, this length can be at most $C_B(n)$. Each element on the stack consists of a current node and a head position for that node; each of these

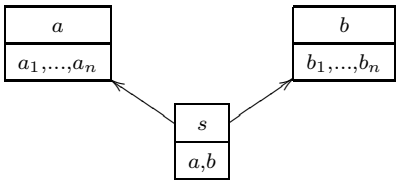
items can be implemented using a counter up to n requiring $\log n$ bits. The fixed number of registers require $O(\log n)$ size. The history requires linear size. □

There is also a relationship between click complexity and communication complexity [15]. A classical setting in communication complexity is the problem EQUAL, which we recall next.

Let there be two players, Alice and Bob, each of whom possesses a sequence of n distinct elements from a set $\Sigma = \{\sigma_1, \ldots, \sigma_t\}$. Alice does not know Bob's sequence, and vice versa, but Σ is known to both players. We can view the sequences as words (without repeated letters) over the alphabet Σ. For the problem EQUAL, the players' task is to decide if their words are equal, by sending each other *messages*: these are again words over Σ. They do so according to a predefined communication protocol. This protocol specifies whose turn it is to send the next message, based on the full history of prior messages. At the end of the protocol, the players must be able to decide if their subsets are equal.

The following lower bound on the communication complexity of EQUAL is known [15, Section 1.3]: for any protocol that gives the correct answer for every pair of words of length n, there exists a pair of words for which the total length of exchanged messages is at least $\log_t(\binom{t}{n}n!)$.

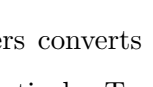
We can simulate EQUAL in the Web browser setting as follows. Consider Web instances of the following form, which, for any natural number n, we refer to as an "EQUAL Web-instance of size n":

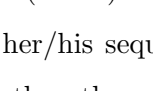


Importantly, some of the a_i's may equal some of the b_j's; the pages a_1, ..., a_n, b_1, ..., b_n themselves do not contain any further links. We define the yes/no Web query Q_{EQUAL} as follows. Let I be an EQUAL Web instance as above. Then $Q_{\text{EQUAL}}(I)$ equals yes if and only if $\{a_1, \ldots, a_n\} = \{b_1, \ldots, b_n\}$. For Web instances I that are not EQUAL instances, $Q_{\text{EQUAL}}(I)$ is empty.

The Web query Q_{EQUAL} is easily computable by a browser, even without history, as follows: go to a; remember a_1 in a register; go back and go to b; compare b_1 to a_1; remember b_2 in a register; go back and go to a again; compare a_2 (this is the link following a_1 on a's tape) to b_2; and so on. The number of clicks here is linear in n, and it seems impossible to do much better. Indeed, we observe:

Proposition 3. *Let B be a browser automaton with history. Let Σ be a finite alphabet of size at least the number of states of B. If B computes Q_{EQUAL}, then the communication complexity of EQUAL over Σ is $O(C_B(n))$.*

Proof. Let Alice's (Bob's) sequence be $a_1, \ldots, a_n$ ($b_1, \ldots, b_n$). Each of the players converts her/his sequence into a Web page 

and 

, respectively. Together, they now simulate B on the instance of size n formed by these two pages (and a source node). Whenever B enters a (b, respectively) then only Alice (Bob, respectively) can continue the simulation, and does so until B enters the other node from among a and b. At that moment, the current player sends to the other a message consisting of the current values of the registers and a letter encoding the current state. Then the other player takes over the simulation, and so on. Note that the players indeed have all needed information to run B, since at the moment of a switch, the content of the $\Leftarrow$ stack is invariantly 

, with the head positioned at the entered page; the $\Rightarrow$ stack is invariantly empty; and all the remaining information B has are precisely the values of the registers and the control state, and also the history. The history now is dealt with as follows. When B is at a and clicks on a link a_i, then Alice sends a_i to Bob to communicate this. Similarly, when B is at b and clicks on a link b_j, then Bob communicates this to Alice. In this way both players remain up to date about the history. It is now clear that the total length of the messages exchanged during this protocol is linearly proportional to the number of clicks done by B. □

7 Navigational problems

The concept of Web query does not exhaust the kinds of problems that can be naturally solved using browsers. A central idea of this paper is the concept of *navigational problem* as a kind of problem specifically suited to solution by Web browsers. Formally, a navigational problem P is a Web query, but what it takes for a browser B to solve P is totally different from what it would take for B to compute P as a Web query:

Definition 8. Let P be a Web query, and let B be a browser automaton. We say that B *solves the navigational problem* P if for each Web instance I, the set of nodes clicked on during the run of B on I equals $P(I)$.

If navigational problem P is solvable by a browser automaton (with or without history), we call P *visitable* (with or without history).

As a simple example of visitable navigational problems, we can mention the following:

Proposition 4. *The problem Reach: the set of all nodes reachable from the source, is visitable without history, as is, for each natural number k, the problem $Reach_k$: the set of all nodes reachable from the source in at most k steps.*

Proof. Visiting all reachable nodes is immediate from the proof of Lemma 1. The modification to visit only those at distance k is easy: using different states we can count up to k, incrementing the count with each click and decrementing it when going backwards. □

We make two further remarks. First, clearly, a navigational problem P can only be visitable if $P(I)$ is a union of paths from the source.

Second, we note that every yes/no Web query Q can be modeled as a special kind of navigational problem. It suffices to add to every instance I an extra node o and insert it in the beginning of the list of links from the source. The node o itself has no outgoing links. If we denote the thus modified instance by I', we can model Q by the navigational problem Q' defined by

$$Q'(I') = \begin{cases} \text{all nodes of } I' & \text{if } Q(I) = \text{yes} \\ \text{all nodes of } I' \text{ except } o & \text{if } Q(I) = \text{no} \end{cases}$$

Clearly, if Q is computed by a browser B, then an easy modification B' of B (cf. Proposition 4) solves the navigational problem Q'. And conversely, if Q' is visitable, then Q is computable by a browser.

As a consequence, results about Web queries carry over to the setting of navigational problems. In particular, by Theorem 1 and the remarks made after Definition 6, there is no upper bound (other than computability) on the computational complexity of visitable problems, and by Theorem 4, there exist navigational problems that are visitable with history but not without.

Click complexity can be applied to navigational problems as well as to Web queries. We can present a rather general theorem in this respect. The theorem is negative in the following sense. A browser solving a navigational problem might have to visit the same page several times. One might hope that the number of times that a page needs to be revisited can be somehow bounded as a function of the total number of pages that need to be visited. This hope is in vain, however:

Theorem 5. *There exists no function f on the natural numbers such that for every visitable navigational problem P there exists a browser (even with history) B that solves P with $C_B(I) \leq f(|P(I)|)$.*

Proof. Recall the Web query Q_{EQUAL} from Proposition 3. We can define a related navigational problem P_{EQUAL} as follows. Given an EQUAL Web instance I as was shown in Section 6, we define

$$P(I) := \begin{cases} \{a, b, b_n\} & \text{if } Q_{\text{EQUAL}}(I) = \text{yes} \\ \{a, b\} & \text{if } Q_{\text{EQUAL}}(I) = \text{no} \end{cases}$$

A browser solving P_{EQUAL} is readily modified into a browser computing Q_{EQUAL} without any additional clicks. Note also that $|P_{\text{EQUAL}}(I)| \leq 3$. Hence, $C_B(I) \leq f(|P_{\text{EQUAL}}(I)|)$ would imply, by Proposition 3, that the communication complexity of EQUAL is bounded by a constant, which is in contradiction with the lower bound that was mentioned in Section 6. □

8 One click per page

The optimal click complexity that a browser can have in solving a navigational problem is captured by the following definition:

Definition 9. A browser B is called a 1-*visitor* if in its computation on any Web instance, B clicks every node at most once.

A navigational problem is called 1-*visitable* (with or without history) if it is solvable by a 1-visitor (with or without history).

So, navigational problems that are 1-visitable can be solved without making any unnecessary data transfer. In order to understand better what it takes to be 1-visitable, we will present a characterization of 1-visitability in the next section. In the present section, we present some general properties.

A simple example of a navigational problem that is 1-visitable without history is the following: "repeatedly click the first link of every visited page, starting from the source, but stop when we run in a cycle, and stop also when the link already occurred as a link (not necessarily the first one) of page already visited." Indeed, this problem can be solved by using the $\Leftarrow$ stack to check for earlier occurrences (using registers to compare them), not requiring any page to be clicked twice. Precisely, the browser stores the the first link on the current page in a register, and then goes back, on each page checking if it is the same as the one stored in the register, or contains a link to it. It does so until the $\Leftarrow$ stack is empty, and then go forward until the

forward stack is empty. If the link from then register was found, it stops, and if not, then the browser clicks on that link when the $\Rightarrow$ stack is empty.

Since a 1-visitor can click, on a Web instance of size n, at most n times, we have the following corollaries to Proposition 2:

Proposition 5. *Every 1-visitor with history can be implemented in space* $O(n \log n)$.

Corollary 2. *Not all visitable navigational problems are 1-visitable, even with history.*

Still, an obvious approach for showing certain navigational problems to be 1-visitable is to start from a browser B, not necessarily a 1-visitor yet, that solves the problem, and then to try to "optimize" B's program, making it more "careful" not to click on a page that has been clicked before. By Corollary 2, putting this idea in practice in full generality is impossible, the following proposition at least shows that there are cases where such an approach is successful.

Proposition 6. *Let P be a navigational problem solved by a browser A with history, such that on every Web instance the trace of clicks of A is a simple path, possibly with a link from the last node to some earlier one on the path. Then P is* 1*-visitable, even without history.*

Proof. The idea is to simulate A by a browser B that mimics A, except that B avoids clicking when A clicks on a node that is already in the forward stack; in that case, B simply moves forward and repositions the tape head at the beginning. If A clicks on a node that is already in the backward stack (which can be tested by going backward and using a register), B can halt because, by the assumption about A, this means that all required nodes have already been visited. In order to do this simulation, B keeps track of the most distant page from the source visited so far, as well as the current bottom of A's forward stack. The latter is necessary because A may already be at it's forward stack while this is not the case for B. In that case, a forward move by A cannot be simulated by a forward move by B, but B must then do what A would do if the forward stack is empty. □

To conclude this section we note that the above proof would be much easier if B would be allowed to use history. It is even tempting to believe that if a navigational problem is visitable at all, then it is 1-visitable with history. Corollary 2 dashes that temptation, however.

9 Structural conditions for 1-visitability

In this section, we give a structural necessary condition for 1-visitability without history, which is helpful for showing that certain problems are not 1-visitable (without history), as we will demonstrate. We also complement this necessary condition with a sufficient one, albeit on the instance level only.

In order to state the result, we first introduce some graph-theoretic definitions.

Definition 10. For G a directed graph and p and q two nodes in G, we call:

- q a *close successor of p in G* (abbreviated G-cs) if $p = q$ or there is a directed edge from p to q in G;
- q a *far successor of p in G* (abbreviated G-fs) if q is not a close successor of p and there is a directed path from p to q in G.

Moreover, for S a set of nodes in G, we call:

- q a G-cs of S if q is a G-cs of some $p \in S$;
- q a G-fs of S if q is not a G-cs of S, and q is a G-fs of some $p \in S$.

Definition 11. Let T be a tree. A simple path in T is called a *trunk* if it starts in the root and ends in a node, all whose children are leafs of T.

The structural notion is now the following.

Definition 12. Let $(\mathbb{W}, s)$ be a Web instance, and let $r > 0$ be a natural number. We say that $(\mathbb{W}, s)$ is *r-coverable* if there exists a spanning tree ST of $\mathbb{W}$ with root s, containing a trunk TR such that the number of ST-fs's of TR in $\mathbb{W}$ is at most r.

We now establish r-coverability as a necessary condition for 1-visitability without history:

Theorem 6. *Let B be a 1-visitor with r registers, and without history; let $I = (\mathbb{V}, s)$ be a Web instance; let $B(I)$ be the set of nodes visited by B on I; and let $\mathbb{W}$ be the subgraph of $\mathbb{V}$ induced by $B(I)$. Then $(\mathbb{W}, s)$ is r-coverable.*

Proof. Let $p_0, p_1, \ldots, p_n$ be all the nodes of $B(\mathbb{V}, s)$, visited (clicked) in that order. For each $m \in \{0, \ldots, n\}$ we define the following sets:

$$R_m = \{p \mid \text{node } p \text{ is in a register just after the click on } p_m\}$$
$$S_m = \text{the nodes on the } \Leftarrow \text{ stack just after the click on } p_m$$
$$S_m^{\mathbb{V}\text{-cs}} = \text{the } \mathbb{V}\text{-cs's of } S_m$$

Remark that $S_m^{\mathbb{V}\text{-cs}} \cup R_m$ contains exactly all nodes whose identities can influence the next step of B after the click on p_m. Informally, they are the nodes B can "see" at this moment. Hence, we denote that set by See_m. Note that $p_m \in See_m$.

Let p_j be the the last visited node such that See_j contains all p_i with $i \leq j$. I.e., j is the largest index such that $See_j = \{p_0, \ldots, p_j\}$. That maximal j exists, because $j = 0$ satisfies the condition trivially. Then obviously either $j = n$ or $See_{j+1} \subsetneq \{p_0, \ldots, p_{j+1}\}$. *We claim that See_j then must contain all p_i, for $i = 0, 1, \ldots, n$, so indeed $j = n$.*

To prove this claim, suppose for the sake of contradiction that for some $k > j$ we have $p_k \notin See_j$. By definition of p_j, there exists $i < j$ such that $p_i \notin See_{j+1}$. Between the visits of p_j and p_{j+1}, B can only put in its registers nodes that are in See_j, so $p_k \notin R_{j+1}$. Since p_k is not in See_j but p_{j+1} is, we also know $k > j + 1$. Furthermore, p_k can only be in $S_{j+1}^{\mathbb{V}\text{-cs}}$ if there is an edge from p_{j+1} to p_k (since $p_k \notin S_j^{\mathbb{V}\text{-cs}}$). We summarize the following:

- $i < j < j + 1 < k$;
- $p_i \notin See_{j+1}$;
- $p_k \notin See_j$;
- $p_k \notin R_{j+1}$;
- $p_k \in See_{j+1}$ only if p_k is a child of p_{j+1}.

In order now to prove the claim, we prove that B cannot be a 1-visitor, thus obtaining the desired contradiction. Specifically, we construct an instance $(\mathbb{V}', s)$ on which B will click node p_i twice. We obtain $\mathbb{V}'$ from $\mathbb{V}$ by replacing, for each $q = j + 1, \ldots, n$, a possible edge (p_q, p_k) by the edge (p_q, p_i) in $\mathbb{V}'$, and a possible edge (p_q, p_i) by the edge (p_q, p_k).

B cannot distinguish between $(\mathbb{V}, s)$ and $(\mathbb{V}', s)$ until just before its visit to p_{j+1}, because the only changes we have made are to nodes p_q with $q \geq j + 1$. But also after B has reached p_{j+1}, we have the following property (*):

> B sees p_k in $\mathbb{V}'$ in the same way as it sees p_i in $\mathbb{V}$, and vice versa:
> B sees p_i in $\mathbb{V}'$ in the same way as it sees p_k in $\mathbb{V}$.

Indeed, right after clicking on p_{j+1}, in $\mathbb{V}$, we know that B cannot see p_i, and can see p_k only, if at all, as a link; in particular, $p_k \notin R_{j+1}$. Hence, in $\mathbb{V}'$, B cannot see p_k, and can see p_i only, if at all, as a link (obtained from the replacement of the link to p_k). Now if property (*) holds right after clicking

on p_{j+1}, then it will continue to hold after that, by construction of $\mathbb{V}'$. As a consequence, B will click p_i on $\mathbb{V}'$ when it would click p_k on $\mathbb{V}$, but this is the second time because B had already clicked on p_i before arriving in p_j. We thus come to the contradiction that B is not a 1-visitor, as desired.

We are now ready to construct TR and ST. Recall that B is a 1-visitor. Hence, during the run of B on I, the clicks of B trace out a tree that is a subgraph of $\mathbb{V}$, the nodes of which constitute precisely $B(I)$. This tree is the original design for ST, but we will need to modify ST a bit in order for $TR := S_j$ to be a trunk of ST. Indeed, suppose there would be edges $p_j \to p \to q$ in ST. In particular, $q \in B(I)$, and $B(I) \subseteq S_j^{\mathbb{V}\text{-cs}} \cup R_j$, as we know from the above. If $q \in S_j^{\mathbb{V}\text{-cs}}$, then we can remove the edge $p \to q$ from ST and attach q as a child directly from some node of S_j. If $q \in R_j$, then we have seen q before the first click on p_j, and we can again instead attach q as a child directly from another page where we saw q earlier. In this way, ST is a spanning tree of $\mathbb{W}$ and TR is a trunk of ST. Moreover, any ST-fs of TR in ST is, by definition, not an ST-cs of TR, i.e., not in $S_j^{\mathbb{V}\text{-cs}}$, and hence in R_j. Since B has only r registers, the number of ST-fs's of TR in ST is thus at most r, as desired for r-coverability. □

As a corollary, we obtain the following counterpart to Proposition 4:

Corollary 3. *The problems Reach and Reach$_k$, with $k \geq 2$, which we saw in Proposition 4 to be visitable without history, are not 1-visitable without history.*

Proof. It is clear that Reach and Reach$_k$ with $k \geq 2$ do not satisfy the necessary condition for 1-visitability expressed in Theorem 6. □

Note that the problems mentioned in the above corollary are 1-visitable with history; also, Reach$_1$ is 1-visitable even without history.

It is a natural question whether r-coverability is also a sufficient condition for 1-visitability by a browser automaton with r registers. On the instance level, and using a slightly technical stronger notion of coverability, we can indeed show that it is.

Definition 13. Let $(\mathbb{W}, s)$ be a Web instance that is r-coverable, as witnessed by spanning tree ST and trunk TR. In particular, the number m of ST-fs's of TR in $\mathbb{W}$ is at most r. Let p be the last node of TR.

We now say that $(\mathbb{W}, s)$ is r-coverable *in the strong sense* if either not all siblings of p in ST are leafs, or if this is the case, then m is at most $r-1$ rather than r.

The intuition behind this definition will be provided in the proof of the following Theorem.

Theorem 7. *Let $(\mathbb{V}, s)$ be a Web instance, and let $\mathbb{W}$ be a subgraph of $\mathbb{V}$ such that $(\mathbb{W}, s)$ is r-coverable in the strong sense. Then there exists a 1-visitor B with r registers and without history, such that B, on $(\mathbb{V}, s)$, visits precisely all nodes in $\mathbb{W}$.*

Proof. Let ST and TR be the spanning tree and trunk that witness r-coverability (in the strong sense) of $(\mathbb{W}, s)$. The basic idea is to make a browser B that simply does a depth-first traversal of ST, taking care to continue down a node on the trunk only after all its siblings have been traversed. This is possible because B is allowed to depend on $\mathbb{V}$, ST, and TR. We use many different states to make sure to click on the right nodes, identified by their sequence number in the list of links on their parent page. Also all different backward moves are identified by different states. If B is run on an instance other than $(\mathbb{V}, s)$ and encounters a page that has not enough links as expected, then B halts.

The problem with this basic idea, however, is that, although B has the desired behavior on $(\mathbb{V}, s)$ and clicks every node only once on that instance, there is no guarantee that B will actually be a 1-visitor on all other possible instances, as required by the definition of 1-visitor. We can augment B, however, by using registers to remember all ST-fs's of TR that have been visited so far. (Of course, on an instance other than $(\mathbb{V}, s)$, we do not take ST and TR literally, but work with isomorphic copies.) Moreover, before we click on a link, we verify whether the node is not already on the stack; or was not already clicked from a node on the stack (whether a link from a node on the stack was already clicked can be remembered, using sequence numbers, by the finite state control); or is not equal to one of the nodes already stored in the registers. It is then clear that B, thus modified, is a 1-visitor on all possible instances.

The only problem remaining, however, is that, in order to compare the node tested for clicking with other nodes, as described above, we need to store the test node in a register as well. In the worst case, we might have already used all r registers to store ST-fs's of TR. By r-coverability, this implies that we have already visited *all* ST-fs's of TR. Now if this would already happen before arriving at the last node of TR, then all siblings of the last node would be leafs in ST, and by the strong sense of r-coverability, there would be at most $r - 1$ such ST-fs's of TR, a contradiction. Hence, the worst case is only possible at the last node of TR. To cover this case we modify B still a bit further, to the effect that B will first compare *all*

links for equality with one of the registers and remember in its finite state control which ones need not be visited; at that point the registers can be freed and one of them used to compare the links that survived the first test with earlier clicked links from nodes on the stack, as before. □

To conclude this section, we note without proof:

Theorem 8. *For every fixed $r \geq 1$, deciding r-coverability is NP-complete. On acyclic instances, the problem is solvable in polynomial time.*

10 Enhanced browsers

If we have a look at the navigation mechanism of the common Web browsers, it appears that they behave very much like Theseus in the maze, in the ancient Greek myth. Theseus was equipped with a roll of string by Ariadne. He set one end of the veil at the entrance to the maze, and following it, he could find the way back from the maze after killing the Minotaur. Likewise, Web browsers set one end of the veil at the entrance to the WWW and, at any time, they can leave the roll at any place and walk back and forth along the veil. They can also take the roll again and relocate it to any other place in the maze.

We propose another, more powerful navigation mechanism. We suggest that, in addition to what they already can do, browsers should be able to relocate the *beginning* of the veil, too. This is similar to the way professional climbers use their ropes, reusing them over and over again on their way up. On the level of user interface, it would amount to giving the user the choice, which of the two stacks $\Rrightarrow$ or $\Lleftarrow$ should be reset to empty upon a click. This can be achieved quite easily, by adding a button to exchange the contents of the two stacks, leaving unchanged the rule that the forward stack is always discarded. It is a conservative enhancement, i.e., those not interested can still use their old way of navigating.

We refer to browsers equipped with this additional action to exchange the contents of the $\Rrightarrow$ and $\Lleftarrow$ stacks, as *enhanced* browsers. We can show that enhanced browsers can be more efficient in terms of click complexity. Recall from Proposition 3 and the lower bound on the communication complexity of EQUAL, that there is a linear lower bound on the click complexity of ordinary browsers that compute the Web query Q_{EQUAL}. In contrast, we have:

Proposition 7. *Q_{EQUAL} can be computed by an enhanced browser, even without history, with only 4 clicks.*

Proof. First, we link a and b by a stack by clicking on a, going back, switching the stacks, and clicking on b; this is illustrated below. Next, the browser can walk between a and b on the stack, comparing their children, without any more clicks.

Table 2: The sequence of actions a browser automaton performs to link a and b by stack.

action	current node	tape	$\Leftarrow$	$\Rightarrow$
initial	s	$\underline{a}b$	empty	empty
click	a	$\underline{a_1}a_2 \dots a_n$	$(s,1)$	empty
back	s	$\underline{a}b$	empty	$(a,1)$
swap	s	$\underline{a}b$	$(a,1)$	empty
right	s	$a\underline{b}$	$(a,1)$	empty
click	b	$\underline{b_1}b_2 \dots b_n$	$(a,1)(s,2)$	empty

□

The general negative result about the click complexity of visitable navigational problems (Theorem 5) was proven using the linear lower bound on the click complexity of ordinary browsers that compute Q_{EQUAL}, which no longer holds for enhanced browsers, as we have just seen. Nevertheless, an alternative proof still leads to similar negative result that holds for enhanced browsers, as we will show next:

Proposition 8. *Proposition 2 still holds for enhanced browsers.*

Indeed, the proof is the same. As a consequence, we now have the following theorem similar to Theorem 5:

Theorem 9. *There exists no space-constructible [3] function f on the natural numbers such that for every visitable navigational problem P there exists an enhanced browser (even with history) B that solves P with $C_B(I) \leq f(|P(I)|)$.*

Proof. We have seen that the computation of arbitrary computable boolean functions, from Theorem 1, is a special case of Web queries, and that Web queries are a special case of navigational problems. Consequently, there are arbitrarily complex visitable navigational problems. In contrast, if a function f as in the statement of the theorem would exist, since $|P(I)|$ is always at most the size of the instance, every visitable navigational problem would

be solvable in at most $f(n)$ clicks and thus within $O(n + f(n) \log n)$ space. This is in contradiction with the space hierarchy theorem from computational complexity theory [3, 18]. □

We conclude by showing that enhanced browsers are not only more efficient, but also more powerful than ordinary ones:

Theorem 10. *There exists a yes/no Web query that can be computed by an enhanced browser automaton without history, but not by any ordinary browser automaton, even with history.*

Proof. The proof is a variation of that of Theorem 4. Recall the representation of binary words by Web instances used there, where the input word was represented by a chain at the left side of the source, and a cycle gadget (to accommodate the simulation of arbitrary computations) at the right side of the source. There, we also had n nodes in the middle to transfer, using the history, the input word from the left side to the right side. Now, using an enhanced browser, we do not need these middle nodes anymore. Instead, we can simply go all the way down the chain on the left, then switch the stacks, and begin the simulation of an arbitrary computation as in the proof of Theorem 1.

In contrast, on instances of this new form, i.e., without the n middle nodes, ordinary browsers are limited to linear space. Indeed, recall from the proof of Theorem 4 the notion of "computational mode" within a run. Reasoning as in that proof, but now without the n middle nodes, we see that the global behavior of the browser, as far as computational mode is concerned, can now be fully described by a fixed finite function, independent of n. We can incorporate this constant function in the finite state control of an equivalent reduced browser which works, as in the proof of Theorem 4, on the linear chain only, without the cycle gadget on the right. We are thus left with an acyclic instance on which browsers are limited to linear space, as we already observed in Theorem 2. □

11 Conclusion

Since Web browsers are among the most common contemporary software tools, a good theoretical understanding of their navigational and computational capabilities seems desirable. In this paper we have initiated such an analysis. We admit that some of the results we present are a bit diminished by the fact that our proofs rely on the computational capabilities of browsers which are unlikely to really be used. However, they are still an

intrinsic property of the model of browser automaton. In other cases, however, we have a strong feeling that the proofs are based on difficulties which are experienced by human users, too, and thus the theorems point out real problems of navigating the Web.

We have also proposed an extension of the repertoire of navigation tools used by the browsers, aiming at more efficient and powerful navigation. Interestingly, also within the Human-Computer Interaction community, a variety of similar enhancements of Web browsers have been proposed [10, 5, 6]. We hope our paper can lead the way towards a principled study of the formal capabilities of such enhancements, complementing empirical user studies, which remain of course essential.

Acknowledgment

We thank the anonymous referees for their critical comments which helped us to improve the paper.

References

[1] S. Abiteboul and V. Vianu. *Foundations of Databases*. Addison-Wesley, 1995.

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

[3] D. Bovet and P. Crescenzi. *Introduction to the Theory of Complexity*. Prentice Hall, 1993.
Freely available, 2006, http://www.algoritmica.org/piluc/

[4] A. Chandra and D. Harel. Computable queries for relational data bases. *Journal of Computer and System Sciences*, 21(2):156–178, 1980.

[5] A. Cockburn, B. McKenzie and M. JasonSmith. Pushing Back: Evaluating a New Behaviour for the Back and Forward Buttons in Web Browsers. *International Journal of Human-Computer Studies*. 57(5):397–414, 2002.

[6] A. Cockburn et al. *Web Navigation*.
http://www.cosc.canterbury.ac.nz/andrew.cockburn/web_navigation.html

[7] J. Conklin. Hypertext: An introduction and survey. *Computer*, 20(9):17–41, 1987.

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

[9] D.M. Edwards and L. Hardman. Lost in hyperspace: Cognitive mapping and navigation in a hypertext environment. In R. McAleese, editor, *Hypertext: Theory into Practice*, pages 90–150. Intellect, 1999.

[10] S. Greenberg and A. Cockburn. Getting Back to Back: Alternate Behaviors for a Web Browser's Back Button. *5th Conference on Human Factors and the Web*, Gaithersburg, Maryland. 3 June, 1999.

[11] N. Immerman. *Descriptive Complexity*. Springer, 1999.

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

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

[14] M. Kaminski and T. Tan. Regular expressions for languages over infinite alphabets. *Fundamenta Informaticae*, 69:301–318, 2006.

[15] E. Kushilevitz and N. Nisan. *Communication complexity*. Cambridge University Press, 1997.

[16] L. Libkin. *Elements of Finite Model Theory*. Springer, 2004.

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

[18] C.H. Papadimitriou. *Computational Complexity*. Addison-Wesley, 1994.

[19] M. Spielmann, J. Tyszkiewicz, and J. Van den Bussche. Distributed computation of Web queries using automata. In *Proceedings 21st ACM Symposium on Principles of Database Systems*. ACM Press, 2002.

[20] M.Y. Vardi. The complexity of relational query languages. In *Proceedings 14th ACM Symposium on the Theory of Computing*, Page 137–146, 1982.