Navigating with a Browser*

Michał Bielecki,[1]** Jan Hidders,[2] Jan Paredaens,[2]
Jerzy Tyszkiewicz, [1] Jan Van den Bussche[3]

[1]Warsaw University, Poland
[2]University of Antwerp, Belgium
[3]University of Limburg, Belgium

NAVIGARE NECESSE EST, VIVERE NON EST NECESSE. POMPEIUS

Abstract. We consider the navigation power of Web browsers, such as Netscape Navigator, Internet Explorer or Opera. To this end, we formally introduce the notion of a navigational problem. We investigate various characteristics of such problems which make them hard to visit with small number of clicks.

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

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.

In this paper, we study such a macro mechanism for Web browsers. Thereto, we introduce the *browser stack machine*. This is a finite-memory automaton as introduced by Kaminski and Francez [4], i.e., an automaton with finite control and a finite number of registers which can store Web nodes, which is extended with the basic features offered by a Web browser and already summarized above: clicking on a link; going "back"; and going "forward". The browser stack machine is a restriction of the *browser machine* introduced by Abiteboul and Vianu [2], which has an unlimited Turing tape for storing Web nodes, rather than just the finite memory plus the stack which we have here.

* Research supported in parts by Polish KBN grant 7T11C 007 21 (M.B. and J.T.) and by FWO grant G.0246.99 (J.H.)

** Contact author: Institute of Informatics, Warsaw University, Banacha 2, PL-02-097 Warszawa, Poland, e-mail `mab@mimuw.edu.pl`.

Just like Alan Turing was interested in understanding the problems solvable by a clerk following a formal algorithm, using only pencil and sufficient supply of paper, we are here interested in the problems solvable by such a browser stack machine. However, while Turing could easily define a "problem" as function on the natural numbers, what kind of "problems" over the Web can we consider in our setting? Abiteboul and Vianu considered the Web as a database and studied the power of browser machines in answering *queries* to this database. Browser stack machines from such a database querying perspective were studied in another paper [6]. We will take a different angle here, and want to focus on *navigational problems*. A navigational problem asks the browser to visit a certain specified set of Web nodes, and no others. It thus corresponds to avoiding "getting lost in hyperspace" and getting your job done. Specifically, we focus on *structural* navigational problems only, where the browser has to solve the problem purely on structural information of the Web graph alone. More advanced models could also introduce various predicates on Web nodes so that the browser can detect various properties of the nodes, but we feel the basic "uncolored" model remains fundamental.
Concretely, we offer the following contributions:

1. We show that browser stack machines, simple as they may appear, can simulate arbitrary Turing machines.
2. We introduce a notion of "data transfer-optimal" browser programs which never download a node more than once. We show that this is a real restriction, by exhibiting various 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.
3. We provide concrete lower bounds on the number of data transfers a browser program has to make to solve certain simple problems. Interestingly, our proof employs a basic result from communication complexity.
4. Finally, we propose a new feature for Web browsers: switching the contents of the "back" and "forward" stacks. We show that this feature allows problems to be solvable with provably less data transfer, and that it allows solving navigational problems unsolvable without it.

1 Web instances and browser stack machines

1.1 Web instances

Definition 1. *A* Web graph *is a finite, locally ordered directed graph* $\mathbb{V} = \langle V, l, < \rangle$*.* V *is the finite set of vertices of* $\mathbb{V}$ *(we always use the matching Roman letter for the set of vertices of any Web graph denoted by a blackboard-font letter),* l *is the edge relation (we call it also the link relation), and* $<$ *is a ternary relation giving the local ordering of the vertices reachable by edges outgoing from the current node.*

Definition 2. *A* page *of a Web graph* $\mathbb{V}$ *is the following structure: it is a node* $v \in V$ *together with the ordered list* $t_1, \ldots, t_k$ *of all the vertices* $t_i \in V$ *such that*

$l(v, t_i)$, the list being given in the local order of all the vertices reachable by one edge from v. We will often depict such a page as follows: $\begin{array}{|c|}\hline v \\ \hline t_1, \ldots, t_k \\ \hline\end{array}$. *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 with a distinguished node s, and such that all vertices 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]

1.2 Browser stack machine

We next define *browser stack machines*, which we abbreviate later on as BSM, for the automatic navigation of Web instances.

Definition 4. *A* browser stack machine *is a finite state computing device B equipped with the following ingredients:*

1. *The components of B are:*
   (a) *A finite state control.*
   (b) *A read-only tape, which stores a sequence of vertices of a Web graph. The tape can be accessed by a head, which can move backwards and forwards, and can sense the beginning and the end of the tape.*
   (c) *A finite number of registers $r_1, \ldots, r_k$, each one capable of storing a single node of a Web graph.*
   (d) *Two stacks, called $\Rrightarrow$ and $\Lleftarrow$, on which B can store always the entire content of its tape (as a single stack item), together with the current location of the head.*
2. *The following actions can be undertaken by B, as ordered by its finite control:*
   (a) *B can change its control state.*
   (b) *B can move the head forward or backward on the tape.*
   (c) *B can store in any of the registers the identity of the node currently seen by the head on the tape.*
   (d) *B can make a forward move, which consists of storing the current content of the tape, together with the head position, on the $\Lleftarrow$ stack, removing the top of the $\Rrightarrow$ stack and restoring it as the current tape and setting the head in the recorded position. This move is impossible if the $\Rrightarrow$ stack is empty.*
   (e) *B can make a backward move, which consists of storing the current content of the tape, together with the head position, on the $\Rrightarrow$ stack, removing the top of the $\Lleftarrow$ stack and restoring it as the current tape and setting the head in the recorded position. This move is impossible if the $\Lleftarrow$ stack is empty.*

(f) B *can* click, *which causes the current content of the tape together with the current head position to be stored at the top of the* $\Leftarrow$ *stack, removes the* entire content *of the* $\Rrightarrow$ *stack, and fills the tape with the list of the vertices, which are accessed by the edges outgoing from the node seen by the head on the tape at the moment of clicking. The head is set at the first cell of the tape. If the link points to the current page itself, nothing happens.*[1]
(g) B *can 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 between values of registers and/or the node currently under the head.*
   (c) *Information whether the head scans the leftmost or rightmost cell of the tape. (Recall that, according to the mechanism of changing the tape content, the length of the tape can vary.)*
   (d) *Information whether any of the stacks is empty.*
4. *The initial configuration of the machine in a Web instance* $(\mathbb{V}, s)$ *is as follows:*
   (a) s *is the value of all the registers of* B.
   (b) s *is the only node on the tape.*
   (c) *Both stacks are empty.*

A formal definition of the *computation* of a machine on an instance is easily produced and omitted from this extended abstract.

2 Navigational problems

Our central idea is to study the power of BSMs in solving *navigational problems*:

Definition 5. *A navigational problem is a partial computable function* P *from Web instances to finite sets of nodes, such that*

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

The second condition is a common "consistency criterion" found in database querying [3, 1] and corresponds to the conceptual practice not to distinguish between isomorphic logical structures.

Definition 6. *Recall that the set of nodes on which a BSM* B *clicks during its computation on* $(\mathbb{V}, s)$ *is denoted* $B(\mathbb{V}, s)$. B *solves a navigational problem* P *if for every instance* $(\mathbb{V}, s)$ *on which* P *is defined* $B(\mathbb{V}, s) = P(\mathbb{V}, s)$.

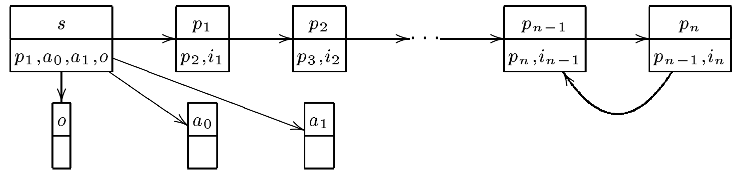
[1] The same behavior is shown by real life browsers.

Definition 7. *A navigational problem P is called "visitable" if it can be solved by a BSM.*

We begin with the most crude approximation of the computational power of BSMs.

Theorem 1. *There is a translation of Turing machines M into BSMs B_M and of input words w into Web instances $(\mathbb{V}_w, s)$ such that the computation of B_M in $(\mathbb{V}_w, s)$ simulates the computation of M on w.*

Proof. We give a direct encoding of the computations of any single-tape Turing machine in the model of BSM.
Turning to the construction, let M be a Turing machine with a a single input-work tape with tape alphabet consisting of symbols $\{0, 1\}$. Blank is allowed too, but cannot be written by the machine on the tape. Let $w = w_1 \dots w_n \in \{0, 1\}^*$ be an input word for M. We convert w into a Web instance $(\mathbb{V}_w, s)$ as follows:



and where additionally each link i_j points on the node a_l iff $w_j = l$. Note the loop from p_n to p_{n-1}.
Now we construct a BSM B_M able to mimic in $(\mathbb{V}_w, s)$ the computation of M on w as follows:

1. Starting from s, B stores the addresses of a_0 and a_1 in its registers. (Here and in the following, B identifies links by their order of appearance on the pages. There are always at most 4 of them, so all of the necessary information can be encoded in the finite control of B.)
2. Next it follows the first link on each page it arrives at, until it finds a two-page-large loop, which can be easily detected by comparing the link with the address of the previous page (stored on the $\Lleftarrow$ stack). When this loop is found, B must be in p_n.
3. B memorizes p_n in a register.
4. On page p_i B repeats the actions
   (a) Compare the second link on the page with the stored addresses a_0, a_1.
   (b) If it is a_j, then move the head to the j-th link on that page.
   (c) Go back one step.
   until it comes back to s.
5. Presently, the position of the head on each of the pages $p_1, \dots, p_n$ which are on the $\Rrightarrow$ stack indicates the corresponding symbol of M's input word w.
6. B starts simulating M, which is done by walking backward and forward on the stacks exactly as the head of M does, updating always the head position on the actually visited page to be over the j-th link, with j the symbol

M writes to the current tape cell. In this simulation, the page currently visited by B corresponds to the tape cell with the head of M, the $\Leftarrow$ stack corresponds to the portion of M's tape to the left of the head, and the $\Rrightarrow$ stack corresponds to the portion of M's tape to the right of the head, except that the not-yet-used portion of the tape will be only created by B when it is necessary.

7. If B senses that the $\Rrightarrow$ stack is empty and M wants to go right, then B memorizes in a register the current location of the head on the present page (which is always either p_n or p_{n-1} in such cases), clicks on the first link (so that it goes to either p_{n-1} or p_n, respectively), makes one move backward, restores the head position to the memorized position, makes a move forward, and prints there the intended symbol, assuming that what is saw there was a blank symbol.
8. If M accepts and halts, B goes all the way back to s and clicks on link o.

Note that in longer computations the vast majority of the pages stored on the stacks are copies of p_{n-1} and p_n. However, the position of the head is always the page currently visited[2], so the BSM never gets lost in the simulation.
As it is readily seen, M accepts w iff $o \in B(\mathbb{V}_w, s)$. □

Remark 1. Now, if we take a universal Turing machine as M, a standard argument shows that the navigational problem solved by B is RE-complete.

It is worth noting that the above construction depends crucially on the Web instance possessing a cycle. Hence, it is a natural question to ask what happens if we restrict attention to acyclic Web graphs. The following result answers this question.

Theorem 2. *Every visitable navigational problem, when restricted to acyclic instances, is in PSPACE, and there exist such problems that are PSPACE-complete.*

Proof. The first part follows by a straightforward simulation of B by a Turing machine, which stores the stacks of B on its work tape. In the absence of cycles, these stacks never exceed height equal to the cardinality of the input Web instance, and the pages themselves are of size linearly dependent on the number of pages of the input. This gives altogether a polynomial amount of space, necessary to perform the simulation.
After minor technical modifications, the proof method of Theorem 1 yields the PSPACE-hardness part. □

Let $Reach_k$ be the navigational problem $(\mathbb{V}, s) \mapsto$ "the set of vertices reachable from s in at most k steps" (we permit $k \in \{1, 2, \dots\} \cup \{\infty\}$).

Theorem 3. *For each $k > 0$ $Reach_k$ is visitable.*

[2] With the small and only temporary exception when the BSM clicks to extend the stacks by a new page.

Proof. The machine does depth-first-search, limited to k steps. To avoid loops and to assure that the BSM always halts, it clicks only pages that are not already on the $\Leftarrow$ stack. Before clicking a page it stores it in a register then goes back to the source comparing the page to all pages on the $\Leftarrow$ stack. □

2.1 Navigational problems and data transfer

The above theorems do not exhaust all the interesting questions connected with the navigational abilities of BSMs.
Clicking on a link causes data transfer to happen, even if the page has been already seen by the browser. The pages stored on the stacks are cached locally, so visiting them does not cause any data transfer. Thus, by measuring how many times a BSM clicks during its computation, we indeed measure the data transfer it generates.

Definition 8. *A BSM 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" if it is solvable by a 1-visitor.

1-visitability is an important feature of problems. Such problems can be solved without making any unnecessary data transfer. In certain sense, for 1-visitable problem there exist BSM which achieve the optimal possible communication cost.
We next present an intrinsic necessary condition on navigational problems to be 1-visitable:

Definition 9. *Given a Web instance $(\mathbb{V}, s)$ and a subset V' of the vertices of $\mathbb{V}$, a node n of $\mathbb{V}$ is said to be* distant from V' *if $n \notin V'$ and all paths in $\mathbb{V}$ from a node in V' to n contain more than one edge.*

Definition 10. *Given a Web instance $(\mathbb{V}, s)$, a subset V' of V is called a "PD[k] set" (Path with k Distant nodes) if there is a subset $S \subseteq V'$ such that*

- *$s \in S$,*
- *if $|S| > 1$ then there is a simple path in $\mathbb{V}$ that starts in s and contains exactly the vertices in S, and*
- *there are at most k vertices in V' that are distant from S.*

Theorem 4. *A navigational problem P can be 1-visitable only if there is a natural number k such that for every Web instance $(\mathbb{V}, s)$, $P(\mathbb{V}, s)$ is a PD[k] set.*

Proof. (Sketch) Let B be the BSM solving P. If B has no registers then it cannot compare pages and can therefore only visit the source s without the risk of visiting a page twice. If, on the other hand, B has registers then it can compare pages and check if a certain page that is to be visited was not visited before.
Let us call the path that is formed by the $\Leftarrow$ stack plus the current page the Current path. It either holds that B has directly after every click all the visited pages that are distant from the Current path in its registers, or it does not. Let

us first assume that it does. Since this will then also hold after the last click, the visited pages will be an $PD[k]$ set with S the Current path just after the last click. Let us assume that directly after a click to a certain page the BSM holds no longer all the visited pages that are distant from the Current path in its registers. Let p be the first page for which this holds, and S the corresponding Current path. Let p' be the page that B clicked to just before p and let S' be the corresponding Current path. After the click to page p the BSM can only visit pages that are children of pages in S' or were in the registers of B just after the click to p'. Since after this click all visited pages distant from S' are also in the registers it follows that at that moment all the pages distant from S' that were and will be visited by B are then in its k registers. It follows that all the pages visited by B form an $PD[k]$ set. □

It is obvious that the above condition is not sufficient for a problem to be visitable at all, let alone 1-visitable. The reason is that, according to Theorem 2, the problem must obey certain complexity requirements to have at least a chance to be visitable.
A counterpart of Theorem 3 is the following one.

Theorem 5. *$Reach_1$ is 1-visitable. For $k > 1$ the problem $Reach_k$ is not 1-visitable.*

Proof. The proof follows from Theorem 4 or can be done in a similar way to the Proof 2 of Theorem 7. □

An obvious approach for showing certain navigational problems to be 1-visitable is to start from a BSM B, not necessarily a 1-visitor, 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. Putting this idea in practice in full generality is impossible (see Theorem 7), but we can at least show that there are cases where such an approach is successful.

Theorem 6. *If a navigational problem P is visitable by a BSM B and, for every Web instance $(\mathbb{V}, s)$, the trace of the computation of B on $(\mathbb{V}, s)$ is a simple path (perhaps with a link from the last node to some earlier one on the path), then P is indeed 1-visitable.*

Proof. Omitted. □

3 Click complexity

We now know that many navigational problems are not 1-visitable: they cannot be solved by a BSM without clicking more often than the actual number of nodes that are visited by the BSM. Hence, it is natural to wonder about the actual "click complexity" of a visitable navigational problem: how many clicks must any BSM make to solve them? We next prove a quite negative result, to the effect that the number of needed clicks in general *cannot be bounded* as a function of the number of visited nodes.

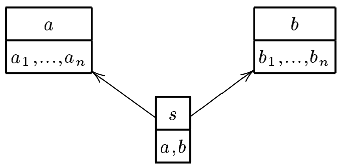
Definition 11. *For a BSM B and a Web instance $(\mathbb{V}, s)$ let $C_B(\mathbb{V}, s)$ be the number of clicks during computation of B on $(\mathbb{V}, x)$.*

Theorem 7. *There exists no fixed recursive function $f : \mathbb{N} \to \mathbb{N}$ such that for every visitable navigational problem P on Web instances, there exists a BSM B solving P for all Web instances $(\mathbb{V}, s)$ with $C_B(\mathbb{V}, x) \leq f(|B(\mathbb{V}, x)|)$.*

Proof. We give two fundamentally different proofs of the theorem. Each of them highlights a quite different reason for which a BSM might have to click many times on the same page.

Proof 1. Suppose such a function f exists.
We proceed as in the proof of Theorem 1. By appropriately choosing a Turing machine M to be simulated by a BSM B, we can construct a visitable navigational problem such that the decision problem $o \overset{?}{\in} B(\mathbb{V}, s)$ is not in DSPACE$(n^3 \cdot f(n))$. This follows from the well-known hierarchy theorem for deterministic space complexity.
Now suppose there exists another BSM B' such that $B'(\mathbb{V}, s) = B(\mathbb{V}, s)$ for all Web instances $(\mathbb{V}, s)$ and $C_{B'}(\mathbb{V}, s) \leq f(|B(\mathbb{V}, s)|)$. By a straightforward modification of the argument given in the proof of Theorem 2, we can simulate B' by a Turing machine using at most $(n^3 \cdot f(n))$ space for input Web instances of size n. By this simulation, $o \overset{?}{\in} B(\mathbb{V}, s)$ is in DSPACE$(n^3 \cdot f(n))$, a contradiction.

Proof 2. Consider the following Web instance $(\mathbb{V}, s)$.



where the pages of $a_1, \ldots, a_n, b_1, \ldots, b_n$ contain no further links. Moreover, some of the a_i'a may equal to some of the b_j's. The task of the BSM is to verify if the sequences $a_1, \ldots, a_n$ and $b_1, \ldots, b_n$ are identical, and if so, to click on b_n.
The following BSM B does that, clicking altogether on a number of vertices linear in n.
After clicking on s, B clicks on a, then memorizes a_1 in a register x_a, goes back to s, clicks on b, memorizes b_1 in a register x_b and compares x_a with x_b. If they are distinct then B immediately halts.
Otherwise B repeats the following action until x_a and x_b are the last links on the pages a and b: click on a, then move the head to the link next to the value of x_a, memorize its value in x_a, go back to s, click on b, move the head to the link next to the value of x_b, memorize its value in x_b, and compare x_a with x_b. If they are distinct then immediately halt.
If B finds a_n and b_n equal (i.e., the above loops terminates without discovering any difference), click on b_n.

Now we prove that any BSM B solving the same navigational problem must indeed use a large number of clicks, even though $|B(\mathbb{V}, s)| \leq 3$.

We use an argument from communication complexity. Let there be two players, Alice and Bob, each of whom has an ordered subset (unknown to the other person) of a (known to both Bob and Alice) set $\Sigma = \{\sigma_1, \ldots, \sigma_t\}$. Their task is to decide if the sequences are equal, by sending each other messages: words over the same alphabet Σ. They do so according to a predefined communication protocol. This protocol specifies whose turn 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 the words are equal. A theorem of communication complexity asserts that for any such protocol, which 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 $\log_t(\binom{t}{n} n!)$, see [5, Section 1.3].

We show that if there exists a BSM B with k registers and $r < |\Sigma|$ control states, solving our initial navigational problem with c clicks, then Alice and Bob can use it to construct a protocol to decide the equality of any two ordered subsets of Σ of size n, with communication of $(c-1)/(k+1)$ elements. This protocol is valid for all Σ of sufficiently high cardinality.

Indeed, suppose such a B exists and that it has k registers and r control states. Let the ordered subsets (i.e., words) Alice and Bob have be $a_1 \ldots a_n$ and $b_1 \ldots b_n$, respectively. Each of the players converts his/her sequence into a Web page $\begin{array}{|c|}\hline a \\ \hline a_1,\ldots,a_n \\ \hline\end{array}$ and $\begin{array}{|c|}\hline b \\ \hline b_1,\ldots,b_n \\ \hline\end{array}$, respectively. Now each of them simulates the BSM B in the so formed instance of the above figure. 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 this moment that player sends to the other one a message "B clicks on the link to your page with $x_1, \ldots, x_k$ as the values of the registers and q as the control state". Note that indeed it is enough to send $x_1, \ldots, x_k, \sigma_q$, where σ_q is used to encode the control state. Then the other player takes over the simulation, and so on. Note that he/she can do it, since at this moment the content of the $\Lleftarrow$ stack is $\begin{array}{|c|}\hline s \\ \hline a,b \\ \hline\end{array}$ with head pointing at the entered page, the $\Rrightarrow$ stack is empty, and all the remaining information B has are the values of the registers and the control state. If B either halts or clicks on b_n, Bob notifies Alice about the decision of B. Now, besides the last message of length 1, each message exchanged has length $r+1$ and the number of such messages is equal to the number c of clicks B has done minus one (the initial click on s). The total amount of communication is then $(c-1)(r+1) \geq \log_t(\binom{t}{n} n!) \geq \log_t((t-n)^n)$, by the communication complexity theorem. This estimation is valid for any sufficiently large t. Hence the number c of clicks satisfies

$$c \geq 1 + \frac{\lim_{t\to\infty} \log_t((t-n)^n)}{r+1} = 1 + \frac{n}{r+1}.$$

□

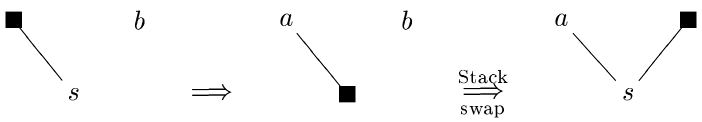
Remark 2. We haven't made one specific feature of real-life browsers a part of our BSM. Namely, they are capable of remembering which pages they visited, and represent this information by displaying the already visited links in a specific color. This feature could be, without much difficulty, incorporated into our model. Now the remark is that even in presence of this feature, the last theorem remains true, and, indeed, both proofs remain valid. In particular, there are visitable navigational problems which are not 1-visitable, even by BSM with memory.

4 Enhanced Browsing

If we have a look at the navigation mechanism of browsers, it appears that they behave very much like Theseus in the maze, in the old Greek myth. Theseus had been equipped with a roll of veil 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 Minotaur. Modern 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. If they decide to do so, they can 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 about the way how 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 should be reset to empty upon a click: $\Rrightarrow$ or $\Lleftarrow$. 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.
The new style of navigation is indeed provably more efficient than the old one.

Theorem 8. *The navigational problem defined in the second proof of Theorem 7 requires* 4 *clicks altogether to compute by an enhanced BSM.*

Proof. First we link a and b by a stack, as shown below. Next, we can walk between a and b on the stack, comparing their sons, without any more clicks.



□

The class of 1-visitable problems increases even more substantially, when we move from normal BSMs to the enhanced ones. Formally, we show that Theorem 4 is not valid for enhanced BSMs. Note that the problem we have shown above to

be 1-visitable is not 1-visitable for standard BSMs, even with memory of visited pages, according to the Remark 2.
Let Q be the following navigational problem: $Q(\mathbb{V}, s)$ = the set of all nodes reachable from s by following always the leftmost link union the set of all nodes reachable from s by following always the rightmost link to a page not yet visited. Q has been shown in Theorem 4 not to be 1-visitable.

Theorem 9. *The necessary condition on 1-visitability expressed by Theorem 4 fails for enhanced BSMs.*

Proof. Since $Q(\mathbb{V}, s)$ doesn't satisfy the condition it suffices to show that $Q(\mathbb{V}, s)$ is 1-visitable by an enhanced BSM. The BSM follows the path formed by choosing always the first link on each page whenever this leads to a new page. This can be checked by memorizing the page the new link points to in a register and going down the $\Leftarrow$ stack to see if this page is a new one. When this path ends (because either there are no more links to follow or the leftmost link points to a page already visited), the BSM goes back to s, swaps the stacks (so that now the $\Leftarrow$ stack contains the whole so far visited path), finds the rightmost link on s which leads to a new node, and follows this procedure as long as on the current page there is at least one link not yet on the $\Leftarrow$ stack, choosing always the rightmost one. □

However, the first proof of Theorem 7 is easily seen to carry over to enhanced browser stack machines, so

Theorem 10. *There exists no recursive function $f : \mathbb{N} \to \mathbb{N}$ such that for every visitable navigational problem P on Web instances, there exists an enhanced BSM B solving P for all Web instances $(\mathbb{V}, s)$ with $C_B(\mathbb{V}, x) \leq f(|B(\mathbb{V}, x)|)$.*

Enhanced browsers are not only more efficient but also more powerful than the standard ones.

Theorem 11. *There exists a navigational problem which is solvable by an enhanced BSM, but not by a standard one.*

Proof. Omitted due to space limitations. □

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. A. Chandra and D. Harel. Computable queries for relational data bases. *Journal of Computer and System Sciences*, 21(2):156–178, 1980.
4. M. Kaminski and N. Francez. Finite-memory automata. *Theoretical Computer Science*, 134(2):329–363, 1994.
5. E. Kushilevitz and N. Nisan. *Communication complexity.* Cambridge University Press, 1997.
6. 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.