Backtracking in a Multiple-window Hypertext Environment

Michael Bieber and Jiangling Wan*

Institute for Integrated Systems Research
Computer and Information Science Department
New Jersey Institute of Technology
Newark, NJ 07102, U.S.A.

ABSTRACT

Multi-window interfaces allow users to work on logically independent tasks simultaneously in different sets of windows and to move among these logical tasks at will (e.g., through selecting a window in a different task). Hypertext backtracking should be able to treat each logical task separately. Combining all traversals in a single chronological history log would violate the user's mental model and cause disorientation. In this paper we introduce *task-based backtracking*, a technique for backtracking within the various logical tasks a user may be working on at any given time. We present a preliminary algorithm for its implementation. We also discuss several ramifications of multi-window backtracking including the types of events history logs must record, deleting nodes from history logs that appear in multiple logical tasks, and in general the choices hypermedia designers face in multi-window environments.

Keywords: hypertext, hypermedia, backtracking, multiple window, history log, session log, multiple pane.

1 INTRODUCTION

Characteristic features such as annotation and backtracking distinguish *hypertext* and *hypermedia* systems from other software applications that solely provide primitive "linking" abilities [2,16,21]. As software environments become more complex, designers must consider an increasing number of facets of these features to serve users adequately [5]. For example, annotation (reader-initiated comment and link authoring) in cooperative software environments must cooperate with access privileges among the individuals, work groups and "global" user community. Hypertext systems should reconnect annotations to a generated document upon its regeneration. Concerning backtracking, multi-window interfaces allow users to work on separate tasks simultaneously in different windows. Hypertext backtracking should treat each task separately and not automatically combine all traversals in a single chronological "history log." In this paper we discuss the ramifications of multi-window backtracking and present a preliminary algorithm for its implementation.

Multiple windows allow users to display related nodes on the computer screen side-by-side, such as two linked documents[1]. Users also can work on semi-independent subtasks of the same project, such as two sections of an annual report. Here a user may traverse links between the two subtasks on occasion, but primarily will work on each independently, traversing within separate clusters of nodes within each of the two subtasks. Many users work on entirely unrelated tasks in separate windows. An analyst can prepare a multi-spreadsheet budget in one set of windows, while on the same computer screen performing a portfolio analysis for a client in another set of windows. Periodically he or she will switch to the other task by selecting one of its windows. Additionally, the analyst could have an unrelated software application running in a third set of windows, such as hypertext-supported electronic mail. Occasionally the user will interrupt the other tasks to work here. Backtracking should reflect the user's logical work flow in all these cases.

To illustrate backtracking alternatives, consider Figure 1's example. Suppose we have a multi-window hypermedia system, in which each document appears in a separate window. Documents have links to other documents. A link traversal from a document to anther might open a new window. Assume the user performs the following actions:

(i) Open Document A into Window A
(ii) Traverse a link from Document A to Document B, opening it into Window B
(iii)Traverse a link from Document B to Document C,

*Both authors contributed equally to this paper. Email: bieber@cis.njit.edu, wan@irss.njit.edu; Telephone: +1 201 596-2681

[1]A single software application may support these related nodes, or separate applications may support each. Link services such as Sun Link Service [20] and architecture's such as Microcosm [9,11], Multicard [21], PROXHY [13] and SP2 [23] facilitate links among windows in multiple applications.

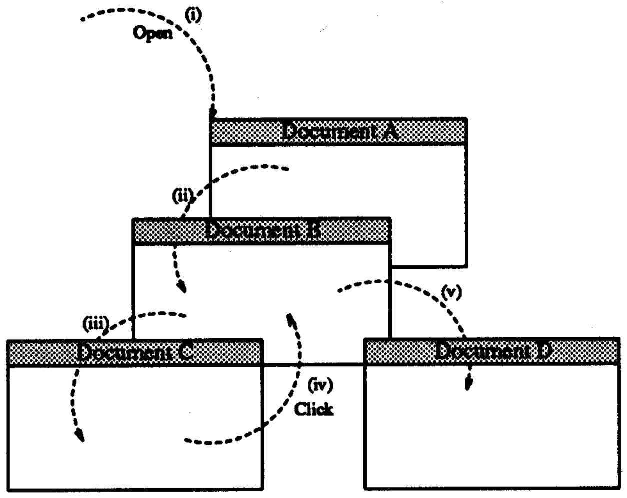


Figure 1: A Simple Multi-window Navigation Example

opening it into Window C
(iv) Click the mouse on Document B, making it the active document, and its window the active window
(v) Traverse a link from Document B to Document D, opening it into Window D

Now the user backtracks several times starting from Document D all the way to Document A. Which of the following paths will he or she follow? (Which path should he or she follow?)

(1) D → B → C → B → A
(2) D → B → A
(3) D → B → C → A
(4) D → C → B → A

Each of these paths could result, depending on a given hypertext system's design. Case (1) presents the most complete form of backtracking. It follows a chronological order and faithfully returns to every previously visited window, including a double return to Window B. Case (2) backtracks along direct links only. It ignores the click from Document C to Document B and thus the "tangential" traversal, which abruptly ends at Window C. Case (3) backtracks chronologically along each link traversal, revisiting each node only once. Case (4) backtracks chronologically to each node visited, either ignoring nodes activated through clicking, or only revisiting nodes once.

Although none of the hypermedia literature addresses multi-window backtracking, research has progressed on other important backtracking issues. For example, several systems display a chronological "history list" of recently visited nodes (or recently traversed links) [18,19]. To reduce user disorientation, Nielsen includes the time since the user last visited a node in its history list entry [19]. Users can "backjump" [18] to any node on the list. Normally the system then removes the intervening nodes from the history list, presuming that the user wished to pass them by. (Some systems such as Microcosm [9,11] and Rhythm [17] provide only history lists without any explicit backtracking features. These systems do not remove the intervening nodes.) Garzotto et al. have modeled both parametric and conditional backtracking [12]. Parametric backtracking *go-back(X)* allows the user to enter a node characteristic in the parameter *X*, which causes the system to backtrack to the most recently traversed departure node with that characteristic. Conditional backtracking *go-back(query-expression)* evaluates *query-expression* and returns to the most recently traversed departure node satisfying it. Landow discusses "arrival rhetoric"—how systems should redisplay departure nodes upon backtracking [15]. One technique is to scroll to and highlight the link marker that the user originally selected to initiate traversal. Research on these important backtracking features applies equally well in multi-window environments.

In this paper we shall address the following issues, which we have not found elsewhere in the hypertext literature.

- What options exist besides chronological backtracking?
- What kind of data structure can support alternate backtracking schemes?
- When a user opens or selects a window, should the history list record this event?
- Should backtracked nodes be deleted from the history list?
- Could users repeatedly backtrack over the same sequence of (previously backtracked over) nodes?

In §2 we introduce task-based backtracking to complement chronological backtracking for multi-window environments. In §3 we describe the event structure and our extensions to the history log to support both task-based and chronological backtracking. §4 presents and contrasts algorithms for both types of backtracking. §5 reflects on our results and describes intended extensions to this research.

Apart from a note in the final section, we shall not differentiate whether windows belong to the same or different applications. The paper's principles apply to hypertext (and hypermedia) support for both situations. To a certain degree, our conclusions apply to single-pane and multi-pane single-window systems, as we also shall explain in the final section.

2 TASK-BASED BACKTRACKING

Assume that in Figure 1, Document A contains the summary of an annual report, Document B contains its table of contents, and Documents C and D represent two separate and essentially independent sections of the report that share no common links. In his or her own mind, the user can consider working on each section as a logically separate task. Thus the user would find backtracking path D → B → A more appropriate than, e.g., D → C → B → A when working on Document D's task. Backtracking to Document C actually puts the user "out of context" temporarily, thus potentially disorienting him or her. The longer the sequence of "out of context" nodes, the greater the disorientation the user would experience.

We designed task-based backtracking to complement chronological backtracking specifically for these types of cases. To implement task-based backtracking, we employ a mechanism which separates chronological link traversals into *subtasks*. Every subtask consists of a sequence of link traversals (also called an *event-path*). Subtasks represent a period of activity during which the user traverses links among window belonging to the same overall "logical task." Opening and closing, selecting or backtracking to another window all could signal a switch to a different logical task. Thus we end the current subtask whenever the user selects a different window or backtracks, and begin a new subtask when the user next traverses a link. A series of subtasks represents an entire logical task. For example, we logically distinguish the subtasks comprising the path A → B → C from the subtasks comprising the path A→ B → D. (Logical tasks may overlap.) Logical tasks are inferred dynamically—and only implicitly—during backtracking[2].

As we discuss in §3 and §4, backtracking within a single subtask proceeds chronologically while inter-task backtracking (i.e., when backtracking spans multiple subtasks) will depend on the policy incorporated by the system designer. When the user backtracks beyond the start of a subtask Y, the system searches for an earlier subtask X containing a destination node N matching the departure node N of Y's first entry. When several subtasks contain node N, the designer's policy will determine which subtask to continue backtracking along. Designers typically will choose either the most chronologically recent subtask or the first subtask containing N as a destination node. Alternatively the system could allow the user to choose among the options, though this may disrupt the user's train of thought.

Task-based backtracking only can approximate the user's mental model of a logical task. In Figure 1's example, for instance, if Documents C and D indeed belong to the same logical task and the user simply selected Document B for convenience, then task-based backtracking will bypass document C in violation of the user's mental model. Chronological backtracking would have matched this mental model more accurately.

[2]Logical tasks also could be inferred from the subtask logs of §3.2, which extends beyond the scope of this paper's analysis.

3 AN IMPLEMENTATION FRAMEWORK

In §4 we present algorithms for implementing chronological and task-based backtracking. In this section we present the structures of the underlying events and traversal (history) logs.

3.1 Event Structure

We define an event as any user action which affects the system status. These actions usually cause some change on the user interface such as creating a new window or closing an existing one. We classify events into *forward*, *backwards* and *switching* events. Link traversal is a *forward* event. Backtracking (executing a backtrack command) is a *backwards* event. Selecting, opening and closing comprise the *switching* events as each deactivates the current window and activates a different one. (Closing a window activates the window beneath it, if any.) To support different kinds of backtracking, the system keeps a complete set of user event information, which we record in the following *event structure*.

We represent each event by a tuple $\langle I, A \rangle$. The event identifier I provides a unique reference to the event. A contains the set of attributes which characterizes the event. Event attributes include the following:

Event-type An event can be one of five types:

- 'traversal' – traverse a link to a new (or already displayed) window;
- 'open' – create a new window (or activate an already displayed window) explicitly by executing an "open window," "open new document," or "open new node" command;
- 'select' – activate an existing window directly by selecting it, not through any link traversal;
- 'close' – close an existing window directly by executing a "close window" or "close node" command; and
- 'backtrack' – backtrack along a link, or more generally, along a previous event by executing a "backtrack" command.

Departure-object This field contains the identifier of the "departure" node from which an event originates. For conciseness, in the rest of the paper we assume that each node is displayed in an entire single window and shall not distinguish between nodes and their windows.

Destination-object This field identifies the identifier of the "destination" node that the event activates.

Subtask-log-id This field indicates the subtask log (see §3.2) referencing this event.

Log-index This field contains an integer indicating the event's chronological position in the chronological log (see §3.2). This attribute applies only to traversal events.

Backtracked-index This contains the *Log-index* of the event backtracked over for a backtrack event. When backtracking over a switching event, this field will contain the *Log-index* of the initiating event's departure node.

The system stores events in a system session structure called system traversal logs, which we describe next.

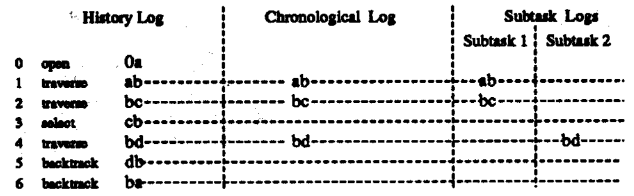
3.2 System Traversal Logs

To track user actions and enable multiple types of backtracking, we maintain a system session log structure consisting of three types of traversal logs:

- History Log
  The history log records the complete event structure for every user event, including event identifier and all attributes. (In addition to backtracking, users could employ the history log to create trails and guided tours. Experimenters could use it to trace and analyze user actions.)
- Chronological Log
  Unlike the history log, the chronological log only registers forward (traversal) events. Each entry contains an event identifier corresponding to an event in the history log.
- Subtask Logs
  Similarly, subtask logs only contain forward events. Each subtask log contains all uninterrupted forward traversals. The system starts a new subtask log whenever a forward event happens after a backwards or switching event. Each entry contains an event identifier corresponding to an event in the history log.

Figure 2 illustrates the traversal logs corresponding to Figure 1. For conciseness, we only show *logical* contents for each log. In the history log we show only event types and pairs of event endpoints. We call such pairs event "edges" following graph theory terminology. In the chronological log and subtask logs, instead of event identifiers, we show the event edges stored in the corresponding history log event structure.

Figure 2 includes two separate subtask logs. Backtracking from Window D can follow several options, depending on the traversal logs used. For chronological backtracking, the system can backtrack to each destination and departure node



History Log			Chronological Log	Subtask Logs	
				Subtask 1	Subtask 2
0	open	0a			
1	traverse	ab	ab	ab	
2	traverse	bc	bc	bc	
3	select	cb			
4	traverse	bd	bd		bd
5	backtrack	db			
6	backtrack	ba			

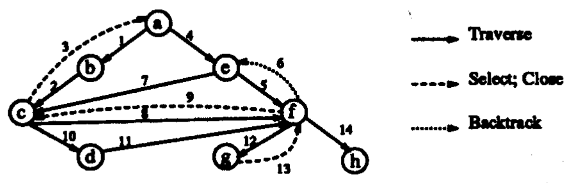
Figure 2: **Traversal Log Contents of a Simple Navigation Example**

in the chronological log, resulting in §1's case (1). Eliminating a second backtrack to Document B will result in case (3). Case (4) backtracks only along the destination nodes of each traversal event in the chronological log, concluding at the original departure node *a*. Performing task-based backtracking along subtask logs results in case (2). Starting at event *bd*, backtracking goes from the destination node *d* to its departure node *b*, finds a previous event with *b* as its destination node, and backtracks to its departure node *a*. §4 presents this algorithm in more detail. Following this latter backtracking path records the two backtracking events *db* and *ba* in the history log, as shown in Figure 2.

Although Figure 2 demonstrates the flexibility of the system session log structure, it is too simple to illustrate the advantages of task-based backtracking—this paper's the major motivation. We turn now to Figure 3's more complicated example. To simplify representation, we draw a graph of nodes instead of windows to represent user events. Figure 3(a) portrays a graph representation of a user navigation session. Each node represents a window (a displayed hypertext node). Each edge represents a user event. Different line styles indicate different event types (traversing, selecting, closing or backtracking). The numbers over edges indicate the chronological order of events. Figure 3(b) depicts the traversal logs for this series of navigation steps. (Whenever a user event takes place, some corresponding system action must be taken to update the contents of the traversal logs.)

Suppose the user begins this session by opening node *a* and traversing links to node *b* and then to node *c*. We register edges *ab* and *bc* in the history log, chronological log and subtask1 log. At node *c*, the user selects (e.g., clicks on) node *a* and traverses a link in it to node *e*. Only the history log records *ca*. We record edge *ae* in the history log, chronological log and a new subtask log (subtask 2). Then the user traverses to node *f* which we register in all three logs. Next, the user backtracks from *f* to *e*. As with selecting, we record this event only in the history log. Traversing from *e* to *c* to *f* leads to a new subtask (subtask 3), which concludes when the user again selects node *c*. Subtask 4 contains traversal events from *c* to *d* to *f* to *g*. Closing node *g* activates the prior node, node *f*, again. The final subtask (subtask 5) records the traversal from node *f* to node *h*.

Because chronological backtracking is relatively simple, in this section we only illustrate task-based backtracking to



(a) A Navigation Session Example

History Log			Chronological Log	Subtask Logs				
				Subtask 1	Subtask 2	Subtask 3	Subtask 4	Subtask 5
0	open	0a						
1	traverse	ab	ab	ab				
2	traverse	bc	bc	bc				
3	select	ca						
4	traverse	ac	ac		ac			
5	traverse	ef	ef		ef			
6	backtrack	fe						
7	traverse	ec	ec			ec		
8	traverse	cf	cf			cf		
9	select	fc						
10	traverse	cd	cd				cd	
11	traverse	df	df				df	
12	traverse	fg	fg				fg	
13	close	gf						
14	traverse	fh	fh					fh

(b) Contents of Traversal Logs

Figure 3: **Traversal Logs**

show how subtask logs are deployed. For the sake of illustration, suppose the user is still at node g in subtask 4, and decides to backtrack instead of closing that node. There is no ambiguity as the only choice is to return along subtask 4 to f. From node f there are three possible back path choices: $f \rightarrow d$ (subtask 4), $f \rightarrow c$ (subtask 3), or $f \rightarrow e$ (subtask 2). By default, paths within the same subtask have higher priority, so path $f \rightarrow d$ is chosen. Backtracking from node d to c proceeds unambiguously. Because node c originates subtask 4, inter-task backtracking now occurs. Two choices are available from node c: returning to node e in the middle of subtask 3 or to node b at the end of subtask 1. The choice depends on the system's designer. The system could pick a default or prompt the user to decide.

As we discuss in §4.3, a backtracked event should not necessarily be deleted from the subtask logs during backtracking. This is because a node might belong to multiple logical tasks (such as window B in Figure 1, and nodes c and f in Figure 3). If a multi-path node is removed upon backtracking, we might mess up other paths which pass through it, thereby disorienting the user.

4 BACKTRACKING ALGORITHMS

In this section, we illustrate the procedures of chronological backtracking and task-based backtracking using the mechanism and structures defined in the previous section. The user does not have to backtrack starting at the latest node. Instead, he or she is free to select any existing node (i.e., window) and invoke backtracking from there.

To simplify the discussion, we only consider single-step backtracking. We must distinguish the first step of a backtracking sequence from its intermediate steps. We tell this from the event-type of the history log's latest event. If the event-type is "backtrack", this is an intermediate step, otherwise it is the first step. We have to locate the starting event in the chronological log. As we shall see, this too would be a policy-dependent choice.

If the current backtracking activity is the first step (i.e., the previous event was not a backtracking event), we have to find a starting event from the starting node which is always the destination of the latest event in the history log.

4.1 Chronological Backtracking

Multiple events in the chronological log may contain the starting node. In the case of chronological backtracking, this starting node could appear as an event's departure or destination node. Which of the multiple events to choose as the starting event is a designer policy decision. (Usually the designer will implement either the first or the chronologically most recent. By default we choose the latter.) Chronological backtracking is resolved by using the event sequence recorded in the chronological log. Gaps might exist, however, between event edges. Consider the example in Figure 2. In the history log, each edge *consecutively* connects to its next edge. For any pair of adjacent edges, the destination node of the preceding edge is the same as the departure node of the succeeding edge. (This applies for individual subtask logs too.) However, for the chronological log, gaps such as that between edge *bc* and *bd* occur because it does not record switching events. When backtracking reaches such a gap, special processing is needed to ensure that no node is missed. Once a starting event (containing the starting node) is located, we can backtrack from it and add this backtrack event to the history log. Continuing to backtrack does not require us to relocate a starting node, but continues in strict chronological order straight up to the beginning of the chronological log. Therefore, locating the starting event for the first backtracking step and for any intermediate step in subsequent iterations is quite different. To find the starting event of an intermediate backtracking step, we search the chronological log to match the latest event in the history log (i.e., the previous backtracking step) against the adjacent events which created it. We operationalize this procedure in the following algorithm.

In the algorithm description, symbols such as $Event1$ and $Node1$ beginning with upper-case letters but not surrounded by quotation marks represent variables. As variable names, these symbols should not be confused with object identifiers.

```
• Algorithm 1: Chronological-backtracking()

1. Determine whether this is the first step of a backtracking
   sequence:
     • Let LatestEvent = the latest event in the HISTORY LOG.
     • Let StartNode = the destination node of LatestEvent.
     • If the 'event-type' of LatestEvent is not "backtrack", goto 3.

2. Locate the StartEvent of an intermediate backtracking step:
     • Let Node1 = the departure node of LatestEvent.
     • Let StartIndex = the 'Backtracked-index' of LatestEvent (see §3.1).
     • Search the CHRONOLOGICAL LOG for StartEvent among events with
       'Log-index' less or equal to StartIndex:
         – Case 1 (no gap):
             * An event Event1 is found which has StartNode as its
               departure and Node1 as its destination.
             * Let StartEvent = Event1, goto 5.
         – Case 2 (indicating a gap):
             * Two adjacent events Event1 and Event2 are found where
               Event1 precedes Event2.
             * Event2 has Node1 as its departure node.
             * Event1 has StartNode as its destination.
             * Let StartEvent = Event1, goto 4.

3. Locate the StartEvent for the first step of a backtracking sequence:
   Let StartEvent = the latest event in the CHRONOLOGICAL LOG which
   contains StartNode as one of its endpoints.

4. If StartNode is the destination node of StartEvent:
     • Let ToNode = departure node of StartEvent.
     • Create a new event (=NewEvent) with the following attribute values:
         – 'Event-type' = "backtrack"
         – 'Departure-object' = StartNode
         – 'Destination-object' = ToNode
         – 'Subtask-log-id' = 0
         – 'Log-index' = 0
         – 'Backtracked-index' = 'Log-index' of StartEvent
     • Add NewEvent to the HISTORY LOG.
     • Display the contents of ToNode.
     • Exit.

5. If StartNode is the departure node of StartEvent (indicating a gap):
     • If StartEvent is the first event in the CHRONOLOGICAL LOG, exit.
     • PrevEvent = the event immediately preceding StartEvent in the
       CHRONOLOGICAL LOG.
     • ToNode = destination node of PrevEvent.
     • If StartNode = ToNode, Let StartEvent = PrevEvent, goto 4.
     • Create a new event (=NewEvent) with the following attribute values:
         – 'Event-type' = "backtrack"
         – 'Departure-object' = StartNode
         – 'Destination-object' = ToNode
         – 'Subtask-log-id' = 0
         – 'Log-index' = 0
         – 'Backtracked-index' = 'Log-index' of StartEvent
     • Add NewEvent to the HISTORY LOG.
     • Display the contents of ToNode.
     • Exit.
```

In Algorithm 1, the difficulty lies in how to locate the StartEvent. It is relatively easier to find the first in a series of backtracking steps in which case we just search the chronological log and choose the latest occurrence of the currently active node. For the case of an intermediate backtracking step, we search the chronological log starting with the event whose 'Log-index' is less or equal to the 'Backtracked-index' of the LatestEvent found in the history log (step 2). Whenever an event is backtracked, we construct a new event for the history log with different contents depending on whether the StartNode is the departure (step 5) or destination (step 4) of the StartEvent. If the backtracked-over event is a traversal event, the new event will have the same link identifier as the starting event, with the departure and destination nodes reversed. If a switching event is backtracked over (a "gap" is reached), an entirely new event (as opposed to the reverse of an old event) is created for the history log. This event has no link associated with it, indicating it does not backtrack over a traversal event.

4.2 Task-based Backtracking

Because task-based backtracking does not backtrack over switching events, every backtracked event can be found directly in one of the subtask logs. There might be multiple instances of such events occurring in multiple subtask logs. We choose the latest one by default. The locating procedure of the starting event and backtracking within a single subtask log are similar to Algorithm 1. The major difference here is that whenever an intermediate backtracking reaches the beginning of a subtask log, we have to locate its destination within another subtask log. This may occur in the middle of a subtask log, as is the case in Figure 2 when we backtrack from *bd* to *ab*.

- Algorithm 2: Task-based-backtracking()

```
1. Determine whether this is the first step of a backtracking
   sequence:
     • Let LatestEvent = the latest event in the HISTORY
       LOG.
     • If the 'event-type' of LatestEvent is not "back-
       track", goto 3.
2. Locate the StartEvent of an intermediate backtracking
   step:
     • Let Node1 = the departure node of LatestEvent.
     • Let StartIndex = the 'Backtracked-index' of Lat-
       estEvent.
     • Search the CHRONOLOGICAL LOG for S-
       tartEvent among events with 'Log-index' less or
       equal to StartIndex, where the departure node of
       StartEvent is StartNode and the destination node
       of StartEvent is Node1.
     • Let CurrentLog = 'Subtask-log-id' of StartEvent.
     • Goto 4.
3. Locate the StartEvent for the first step of a backtracking
   sequence:
   Let StartEvent = the latest event in the CHRONOLOG-
   ICAL LOG which contains StartNode as one of its end-
   points.
4. If StartNode is the destination node of StartEvent:
     • Let ToNode = departure node of StartEvent.
     • Create a new event (=NewEvent) with the follow-
       ing attribute values:
         – 'Event-type' = "backtrack"
         – 'Departure-object' = StartNode
         – 'Destination-object' = ToNode
         – 'Subtask-log-id' = 0
         – 'Log-index' = 0
         – 'Backtracked-index' = 'Log-index' of S-
           tartEvent
     • Add NewEvent to the HISTORY LOG.
     • Display the content of ToNode.
     • Exit.
5. If StartNode is the departure node of StartEvent and
   StartEvent is not the first event in the current SUBTASK
   LOG:
     • PrevEvent = the previous event of StartEvent.
     • ToNode = destination node of PrevEvent.
     • Let StartEvent = PrevEvent.
     • Goto 4.
6. If StartNode is the departure node of StartEvent and
   StartEvent is the first event in the current SUBTASK
   LOG, discard the current StartEvent, goto 2.
```

Algorithm 2 is similar to Algorithm 1 except in the handling of intermediate backtracking steps. If the StartEvent of such a step is not the beginning of a subtask log, we simply backtrack to the previous event without worrying about any "gaps" (step 4). If StartEvent is the first event in the current subtask log, we relocate the StartEvent as if this were a first step of a backtracking sequence (step 6). The resulting event is not necessarily at the end of a subtask log. We often jump from the beginning of the current subtask log to the middle of another subtask log.

4.3 Deleting Backtracked Events

In the above two algorithms, we purposely left out deleting backtracked entries from the chronological or subtask logs. It is up to the system designer to choose a deletion strategy based on the needs of his or her users.

In the case of chronological backtracking, if the user selects some node which is not chronologically the latest, we can classify such an event as either "backjumping" or "repositioning". If we treat it as backjumping, either the first or the most recent (our default) of the events containing this current node is chosen as the starting event. All event traversal (chronologically) later than this event would be deleted from the chronological log and their windows closed. We also delete each event from the chronological log after it is backtracked, and close its window. On the other hand, if we treat such a selecting only as a repositioning, we do not delete events from the traversal logs.

Consider the possible deletions in the case of task-based backtracking. A backtracking sequence could start at any entry of any subtask log and also could jump from one subtask log to any entry of another. We should not delete any entry in a subtask log which we do not enter it from its end; otherwise we will remove backtracking access for the succeeding events in that subtask log. If we do enter a subtask log at its end, we could delete subsequent backtracked events until this subtask log is empty and we backtrack up to another. But this approach, too, can cause inaccessible events. Consider Figure 3. Suppose the user starts backtracking from node h in subtask 5 and then we jump to event ef of subtask 2 by choosing the first event containing node f. Backtracking and deletion continue until we reach node a. All of the events in subtask 2 have been deleted at this point. Since other nodes are not affected, the user may select node f later again and start another backtracking sequence. This will lead us to subtask 3. However, this sequence will be blocked at event ec of subtask 3 because no further events from subtask 2 remain. We lost event ae by deleting it from subtask 2 and this event turned out to be needed in another backtracking sequence.

Deletion strategy is another of the many designer's issues in backtracking. Efforts are needed to address this problem to find a mechanism that can satisfy the majority of applications. Designers can update §4.2's algorithm fairly easily to incorporate the deletion strategy chosen.

5 CONCLUSION AND FUTURE RESEARCH

We kept the paper's algorithm simple to emphasize our strong belief that hypertext systems should support alternative forms of backtracking. Relatively easy extensions include incorporating conditional backtracking and parametric backtracking, as well as backjumping (see §1).

Task switching has relevance for both single-pane and multi-pane single-window hypermedia systems. Their designers should consider treating both opening and backjumping from a history list as "switch events" for users to change logical task domains. Task-based backtracking then could supplement these systems' functionality. Of course, either event could reflect its more typical interpretation—going to or returning to a node in the same logical workgroup. Future research must develop ways of presenting backtracking options without disorienting the user.

For multi-pane systems, "switch events" and backtracking options depend on each pane's domain and purpose. In KMS [1], for example, each of the two panes may hold the same type of node. In Janus [10] and gIBIS [8], for example, each pane serves a different purpose. We leave to future study the extent to which pane context affects switching logical tasks. In some systems, users open multiple windows simultaneously. In this case, we also need to extend the algorithms accordingly.

Multi-window backtracking is but one of many backtracking related issues that have arisen in our current research on designing hypermedia engines [2,3]. We view it as an obligation to couple backtracking with all forms of forward navigation. To this goal, backtracking issues reserved for our future research include:

- Backtracking within guided tours:
  If one considers a guided tour as an enforced series of link traversals, may users backtrack along these either during or after the tour? Can tours explicitly incorporate backtracked steps as part of the "stops" users must visit (e.g., if the tour should reflect the exact stages an analyst or trainee should follow in an analysis [4])?
- Backtracking along "destructive" link traversals:
  In complex hypertext-supported application domains, traversing links can trigger application-based commands in addition to simply displaying a document destination [2,6,7]. For example, in a document management domain [24], traversing an action link may result in merging two directory folders. What happens when a user backtracks across such a link? Should backtracking trigger an "undo" operation or simply reflect the current state of the departure nodes? This reflects the analogous question of backtracking to nodes which have been deleted. Backtracking along computation links, such as in Schnase and Leggett's biological modeling application [22] or Bieber's Max system [4,14] would face similar issues.

We also shall investigate different ways of presenting task-based backtracking options in conjunction with chronological backtracking without confusing the user.

Like so many other hypertext features, backtracking seems simple and intuitive. The desire to exploit the wealth of features and opportunities that hypertext provides application systems and their users, however, calls for sophisticated backtracking options. Judging from the recent hypermedia literature, backtracking research appears to be experiencing a temporary hiatus. This paper demonstrates the wealth of backtracking issues still "waiting in the wings" for researchers and system designers to tackle.

ACKNOWLEDGMENTS

The authors thank Rob Akscyn, Hugh Davis, Dennis Egan, Chuck Kacmar, John Leggett, Marc Nanard, Amy Pearl, Helge Schütt, Norbert Streitz and Fabio Vitali for personal communications about backtracking in their systems. Tomás Isakowitz provided fundamental assistance on our new model of hypertext (currently under development) that will incorporate task-based backtracking. Thanks also to Troy Ames and Walt Truszkowski of NASA's Goddard Space Flight Center for their support of this research. This work was supported, in part, by the New Jersey Institute of Technology under Grant Number 421200, by a grant from the AT&T Foundation and through the NASA JOVE faculty fellowship program.

References

[1] R. Akscyn, D. McCracken, and E. Yoder. KMS: A Distributed Hypermedia System for Managing Knowledge in Organizations. *Communications of the ACM*, 37(7):820–835, 1988.

[2] M. Bieber. On Integrating Hypermedia into Decision Support and Other Information Systems. Forthcoming in *Decision Support Systems*.

[3] M. Bieber. Issues in Modeling a "Dynamic" Hypertext Interface for Non-Hypertext Information Systems. In *Hypertext'91 Proceedings*, pages 203–218, San Antonio, Dec. 1991.

[4] M. Bieber. Automating Hypermedia for Decision Support. *Hypermedia*, 4(2):83–110, 1992.

[5] M. Bieber and C. Kacmar. Designing Hypertext Support for Computational Applications. (Technical Report), 1994.

[6] M. Bieber and S.O. Kimbrough. On the Logic of Generalized Hypertext. Forthcoming in *Decision Support Systems*.

[7] M.P. Bieber and S.O. Kimbrough. On Generalizing the Concept of Hypertext. *Management Information System Quarterly*, 16(1):77–93, 1992.

[8] J. Conklin and M.L. Begeman. gIBIS: A Hypertext Tool for Exploratory Policy Discussion. *ACM Transactions on Information Systems*, 6(4):303–331, 1988.

[9] H. Davis, W. Hall, I. Heath, G. Hill, and R. Wilkins. Towards an Integrated Information Environment with Open Hypermedia Systems. In *Proceeding of the ACM Conference on Hypertext*, pages 181–190, Milan, Italy, November 1992.

[10] G. Fischer, R. McCall, and A. Morch. JANUS: Integrating Hypertext with a Knowledge-based Design Environment. In *Hypertext '89 Proceedings*, pages 105–117, Pittsburgh, Nov. 1989.

[11] A.M. Fountain, W. Hall, I. Heath, and H.C. Davis. MICROCOSM: An Open Model for Hypertext With Dynamic Linking. In *Hypertext'89 Proceedings*, pages 298–311, Pittsburgh, Nov. 1989.

[12] F. Garzotto, L. Mainetti, and P. Paolini. Navigation Patterns in Hypermedia Data Bases. In *Proceedings of the Twenty-Sixth Hawaii International Conference on System Sciences (HICSS)*, Maui, Jan. 1993.

[13] C.J. Kacmar and J.J. Leggett. PROXHY: A Process-Oriented Extensible Hypertext Architecture. *ACM Transactions on Information Systems*, 9(4):399–419, 1991.

[14] S.O. Kimbrough, C. Pritchett, M. Bieber, and H. Bhargava. The Coast Guard's KSS Project. *Interfaces*, 20(6):5–16, 1990.

[15] G. Landow. Relationally Encoded Links and the Rhetoric of Hypertext. In *Hypertext'87 Proceedings*, 1987.

[16] A. Littleford. Artificial Intelligence and Hypermedia. In E. Berk and J. Devlin, editors, *Hypertext/Hypermedia Handbook*, pages 357–378. McGraw-Hill Publishing Co., Inc., New York, 1991.

[17] C. Maioli, W. Penzo, S. Sola, and F. Vitali. Using a Reference Model for Information Systems Compatibility. In *Proceedings of the Twenty-Seventh Annual Hawaii International Conference on System Sciences (HICSS), Vol. III*, pages 376–385, Maui, Jan. 1994.

[18] J. Nanard and M. Nanard. Using Structured Types to Incorporate Knowledge into Hypertext. In *Hypertext'91 Proceedings*, pages 329–343, San Antonio, Dec. 1991.

[19] J. Nielsen. The Art of Navigating Through Hypertext. *Communications of the ACM*, 33(3):296–310, 1990.

[20] A. Pearl. Sun's Link Service: A Protocol for Open Linking. In *Hypertext'89 Proceedings*, pages 137–146, Pittsburgh, Nov. 1989.

[21] A. Rizk. Multicard: An Open Hypermedia System. In *Proceeding of the ACM Conference on Hypertext*, pages 4–10, Milan, Italy, November 1992.

[22] J.L. Schnase and J.J. Leggett. Computational Hypertext in Biological Modelling. In *Hypertext'89 Proceedings*, pages 181–197, Pittsburgh, Nov. 1989.

[23] J.L. Schnase, J.J. Leggett, D.L. Hicks, P.J. Nürnberg, and J. Alfredo Sánchez. Open Architectures for Integrated Hypermedia-based Information Systems. In *Proceedings of the Twenty-Seventh Annual Hawaii International Conference on System Sciences (HICSS), Vol. III*, pages 386–395, Maui, Jan. 1994.

[24] J. Wan, M. Bieber, J.T.L. Wang, and P.A. Ng. Document Management Through Hypertext: A Logic Modeling Approach. In *Proceedings of the Twenty-Seventh Annual Hawaii International Conference on System Sciences (HICSS), Vol. III*, pages 558–568, Maui, Jan. 1994.