Efficient pebbling for list traversal synopses

Yossi Matias* Ely Porat†

Tel Aviv University Bar-Ilan University

& Tel Aviv University

Abstract

We show how to support efficient back traversal in a unidirectional list, using small memory and with essentially no slowdown in forward steps. Using $O(\log n)$ memory for a list of size n, the i'th back-step from the farthest point reached so far takes $O(\log i)$ time worst case, while the overhead per forward step is at most ϵ for arbitrary small constant $\epsilon > 0$. An arbitrary sequence of forward and back steps is allowed. A full trade-off between memory usage and time per back-step is presented: k vs. $kn^{1/k}$ and vice versa. Our algorithms are based on a novel pebbling technique which moves pebbles on a virtual binary, or k-ary, tree that can only be traversed in a pre-order fashion.

The compact data structures used by the pebbling algorithms, called list traversal synopses, extend to general directed graphs, and have other interesting applications, including memory efficient hash-chain implementation. Perhaps the most surprising application is in showing that for any program, arbitrary rollback steps can be efficiently supported with small overhead in memory, and marginal overhead in its ordinary execution. More concretely: Let P be a program that runs for at most T steps, using memory of size M. Then, at the cost of recording the input used by the program, and increasing the memory by a factor of $O(\log T)$ to $O(M \log T)$, the program P can be extended to support an arbitrary sequence of forward execution and rollback steps: the i'th rollback step takes $O(\log i)$ time in the worst case, while forward steps take $O(1)$ time in the worst case, and $1 + \epsilon$ amortized time per step.

*School of Computer Science, Tel Aviv University; `matias@cs.tau.ac.il`. Research supported in part by the Israel Science Foundation.

†Department of Mathematics and Computer Science, Bar-Ilan University, 52900 Ramat-Gan, Israel, (972-3)531-8407; `porately@cs.biu.ac.il`.

1 Introduction

A unidirectional list enables easy forward traversal in constant time per step. However, getting from a certain object to its preceding object cannot be done effectively. It requires forward traversal from the beginning of the list and takes time proportional to the distance to the current object, using $O(1)$ additional memory. In order to support more effective back-steps on a unidirectional list, it is required to add auxiliary data structures.

The goal of this work is to support memory-efficient and time-efficient back traversal in unidirectional lists, without essentially increasing the time per forward traversal. In particular, under the constraint that forward steps should remain constant, we would like to minimize the number of pointers kept for the lists, the memory used by the algorithm, and the time per back-step, supporting an arbitrary sequence of forward and back steps.

Of particular interest are situations in which the unidirectional list is already given, and we have access to the list but no control over its implementation. The list may represent a data structure implemented in computer memory or in a database, it may reside on a separate computer system. The list may also represent a computational process, where the objects in the list are configurations in the computation and the next pointer represents a computational step.

1.1 Main results

The main result of this paper is an algorithm that supports efficient back traversal in a unidirectional list, using small memory and with essentially no slowdown in forward steps: $1+\epsilon$ amortized time per forward step for arbitrary small constant $\epsilon > 0$, and $O(1)$ time in the worst case. Using $O(\log n)$ memory, back traversals can be supported in $O(\log n)$ time per back-step, where n is the distance from the beginning of the list to farthest point reached so far. In fact, we show that a back traversal of limited scope can be executed more effectively: $O(\log i)$ time for the i'th back-step, for any $i \leq n$, using $O(\log n)$ memory.

More generally, the following trade-offs are obtained: $O(kn^{1/k})$ time per back-step, using k additional pointers, or $O(k)$ time per back-step, using $O(kn^{1/k})$ additional pointers; in both cases supporting $O(1)$ time per forward step (independent of k). Our results extend to general directed graphs, with additional memory of $\log d_v$ bits for each node v along the backtrack path, where d_v is the outdegree of node v. These extensions are omitted from this extended abstract and appear in the full paper [7].

We will refer to pointers as *pebbles*. The crux of the list traversal algorithm is an efficient pebbling technique which moves pebbles on a virtual binary or k-ary trees that can only be traversed in a pre-order fashion. We introduce the *virtual pre-order tree data structure* which enables managing the pebbles positions in a concise and simple manner, and the recycling bin data structure that manages pebbles allocation.

1.2 Applications

Consider a program P running in time T. Then, using our list pebbling algorithm, the program can be extended to a program P' that supports rollback steps, where a rollback after step i means that the program returns to the configuration it had after step $i-1$. Arbitrary rollback steps can be added to the execution of the program P' at a cost of increasing the memory requirement by a factor of $O(\log T)$, and having the i'th rollback step supported in $O(\log i)$ time. The overhead for the forward execution of the program can be kept an arbitrary small constant.

Allowing effective rollback steps may have interesting applications. For instance, a desired functionality for debuggers is to allow pause and rollback during execution. Another implication is the ability to take programs that simulate processes and allow running them backward in arbitrary positions. Thus a program can be run with ϵ overhead and allow pausing at arbitrary points, and running backward an arbitrary number of steps with logarithmic time overhead. The memory required is keeping state configuration of $\log T$ points, and additional $O(\log T)$ memory. Often, debuggers and related applications avoid keeping full program states by keeping only differences between the program states. If this is allowed, then a more appropriate representation of the program would be a linked list in which every node represents a sequence of program states, such that the accumulated size of the differences is in the order of a single program state.

Our pebbling technique can be used to support backward computation of a hash-chain in time $O(kn^{1/k})$ using k hash values, or in time $O(k)$ using $O(kn^{1/k})$ hash values. A *hash-chain* is obtained by repeatedly applying a one way-hash function, starting with a secret seed. There are many cryptographic applications, including micro-payment, authentication, and session-key maintenance, which are based on rolling back a hash-chain. Our results enable effective implementation with arbitrary memory size.

The list pebbling algorithm extends to directed trees and general directed graphs. Applications include the effective implementation of the parent function ("..") for XML trees, and effective graph traversals with applications to "light-weight" Web crawling and garbage collection.

1.3 Related work

The Schorr-Waite algorithm [9] has numerous applications; see e.g., [10, 11, 3]. It would be interesting to explore to what extent these applications could benefit from the non-intrusive nature of our algorithm. There is an extensive literature on graph traversal with bounded memory but for other problems than the one addressed in this paper; see, e.g., [5, 2]. Pebbling models were extensively used for bounded space upper and lower bounds. See e.g., the seminal paper by Pippenger [8] and more recent papers such as [2].

The closest work to ours is the recent paper by Ben-Amram and Petersen [1]. They present a clever algorithm that, using memory of size $k \leq \log n$, supports back-step in $O(kn^{1/k})$ time. However, in their algorithm forward steps take $O(k)$ time. Thus, their algorithm supports $O(\log n)$ time per back-step, using $O(\log n)$ memory but with $O(\log n)$ time per forward step, which is unsatisfactory in our context. Ben-Amram and Petersen also prove a near-matching lower bound, implying that to support back traversal in $O(n^{1/k})$ time per back-step it is required to have $\Omega(k)$ pebbles. Our algorithm supports similar trade-off for back-steps as the Ben-Amram Petersen algorithm, while supporting simultaneously constant time per forward step. In addition, our algorithm extends to support $O(k)$ time per back-step, using memory of size $O(kn^{1/k})$.

Recently, and independently to our work, Jakobsson and Coppersmith [6, 4] proposed a so-called fractal-hashing technique that enables backtracking hash-chains in $O(\log n)$ amortized time using $O(\log n)$ memory. Thus, by keeping $O(\log n)$ hash values along the hash-chain, their algorithms enables, starting at the end of the chain, to get repeatedly the preceding hash value in $O(\log n)$ amortized time. Note that our pebbling algorithm enables a full memory-time trade-off for hash-chain execution, and can guarantee that the time per execution is bounded in the worst case.

The most interesting aspect of our algorithm is the proper management of the pointers positions under the restriction that forward steps have very little effect on their movement, to achieve ϵ-overhead per forward step. This is obtained by using the virtual pre-order tree data structure in conjunction with a so-called recycling-bin data structure and other techniques, to manage the positions of the back-pointers in a concise and simple manner.

2 The list pebbling algorithm

In this section we describe the list pebbling algorithm, which supports an arbitrary sequence of forward and back steps. Each forward step takes $O(1)$ time, where each back-step takes $O(\log n)$ amortized time, using $O(\log n)$ pebbles. We first introduce in Section 2.1 the virtual pre-order tree data structure. In Section 2.2 we present the basic pebbling algorithm which, based on the virtual pre-order tree data structure, supports back steps and forward steps as required, using $O(\log^2 n)$ pebbles. Section 2.3 describe the pebbling algorithm which uses $O(\log n)$ pebbles without considerations such as pebble maintenance. Finally, Section 3 describes a full implementation of the algorithm that handles pebbles allocation as well, using a so-called recycling bin data structure.

2.1 The virtual pre-order tree data structure

The reader is reminded that in a pre-order traversal, the successor of an internal node in the tree is always its left child; the successor of a leaf that is a left child is its right sibling; and the successor of a leaf that is a right child is defined as the right sibling of the nearest ancestor that is a left child. An alternative description is as follows: consider the largest sub-tree of which this leaf is the right-most leaf, and let u be the root of that sub-tree. Then the successor is the right-sibling of u. Consequently, the backward traversal on the tree will be defined as follows. The successor of a node that is a left child is its parent. The successor of a node v that is a right child is the rightmost leaf of the left sub-tree of v's parent.

The *virtual pre-order tree* data structure consists of (1) an implicit binary tree, whose nodes correspond to the nodes of the linked list, in a pre-order fashion, and (2) an explicit sub-tree of the implicit tree, whose nodes are pebbled. The main part of the sub-tree is the path from the root of the tree to the current position. This path is pebbled using *blue pebbles.* The purpose of the additional pebbles in the sub-tree is to support effective relocation of the blue-pebbles as the current position moves.

2.2 The basic list pebbling algorithm

The list pebbling algorithm uses two sets of pebbles: *blue pebbles*, and *green pebbles*. The path from the root to the current position is pebbled using the blue pebbles. These pebbles alone would be sufficient to support $O(\log n)$ time per back step, if we were to execute only a sequence of back traversal on the list. The purpose of the green pebbles is to be kept as placeholders behind the blue pebbles, as those are moved to new nodes in forward traversal. Thus, getting back into a position for which a green pebble is still in place takes $O(1)$ time.

Define a *left-subpath (right-subpath)* as a path consisting of nodes that are all left children (right children). Consider the (blue-pebbled) path p from the root to node i. We say that v is a left-child of p if it has a right sibling that is in p (that is, v is not in p, it is a left child, and its parent is in p but not the node i). As we move forward, green pebbles are placed on right-subpaths that begin at left children of p. Since p consists of at most $\log n$ nodes, the number of green pebbles is at most $\log^2 n$.

When moving backward, green pebbles will become blue, and as a result, their left subpaths will not be pebbled. Re-pebbling these sub-paths will be done when needed. When moving forward, if current position is an internal node, then p is extended with a new node, and a new blue pebble is created. No change occurs with the green pebbles. If the current position is a leaf, then the pebbles at the entire right-subpath ending with that leaf are converted from blue to green. Consequently, all the green sub-paths that are connected to this right-subpath are un-pebbled. That is, their pebbles are released and can be used for new blue pebbles.

We consider three types of back-steps:
(i) Current position is a left child: The predecessor is the parent, which is on p, and hence pebbled. Moving takes $O(1)$ time; current position is to be un-pebbled.
(ii) Current position is a right child, and a green sub-path is connected to its parent: Move to the leaf of the green sub-path in $O(1)$ time, convert the pebbles on this sub-path to blue, and un-pebble the current position.
(iii) Current position is a right child, and its parent's sub-path is not pebbled: Reconstruct the green pebbles on the right sub-path connected to its parent v, and act as in the second case. This reconstruction is obtained by executing forward traversal of the left sub-tree of v. We amortize this cost against the sequence of back-steps starting at the right sibling of v and ending at the current position. This sequence includes all nodes in the right sub-tree of v. Hence, each

back-step is charged with one reconstruction step in this sub-tree.

Consider a back step from a node u. Since such back step can only be charged once for each complete sub-tree that u belongs to, we have:

Claim 2.1 *Each back step can be charged at most* $\log n$ *times.*

We can conclude that the basic list pebbling algorithm supports $O(\log n)$ amortized list-steps per back-step, one list-step per forward step, using $O(\log^2 n)$ pebbles.

2.3 The list pebbling algorithm with $O(\log n)$ pebbles

The basic list pebbling algorithm is improved by the following modification. Let v be a left child of p and let v' be the right sibling of v. Denote v to be the *last left child* of p if the left subpath starting at v' ends at the current position; let the right subpath starting at the last left child be the *last right subpath.* Then, if v is not the last left child of p, the number of pebbled nodes in the right subpath starting at v is at all time at most the length of the left subpath in p, starting at v'. If v is the last left child of p, the entire right subpath starting at v can be pebbled. We denote the (green) right subpath starting at v as the *mirror subpath* of the (blue) left subpath starting at v'. Nodes in the mirror subpath and the corresponding left subpath are said to be *mirrored* according to their order in the subpaths. The following clearly holds:

Claim 2.2 *The number of green pebbles is at most* $\log n$.

When moving forward, there are two cases:
(1) *Current position is an internal node:* as before, p is extended with a new node, and a new blue pebble is created. No change occurs with the green pebbles (note the mirror subpath begins at the last left child of p).
(2) *Current position i is a leaf that is on a right subpath starting at v (which could be i, if i is a left child):* we pebble (blue) the new position, which is the right sibling of v, and the pebbles at the entire right subpath ending at i are converted from blue to green. Consequently, (1) all the green sub-paths that are connected to the right subpath starting at v are un-pebbled; and (2) the left subpath in p which ended at v now ends at the parent of v, so the mirror (green) node to v should now be un-pebbled. The released pebbles can be reused for new blue pebbles.

Moving backward is similar to the basic algorithm. There are three types of back-steps.
(1) *Current position is a left child:* predecessor is the parent, which is on p, and hence pebbled. Moving takes $O(1)$ time; current position is to be un-pebbled. No change occurs with green pebbles, since the last right subpath is unchanged.
(2) *Current position is a right child, and the (green) subpath connected to its parent is entirely pebbled:* Move to the leaf of the green subpath in $O(1)$ time, convert the pebbles on this subpath to blue, and un-pebble the current position. Since the new blue subpath is a left subpath, it does not have a mirror green subpath. However, if the subpath begins at v, then the left subpath in p ending at v is not extended, and its mirror green right subpath should be extended as well. This extension is deferred to the time the current position will become the end of this right subpath, addressed next.
(3) *Current position is a right child, and the (green) subpath connected to its parent is only partially pebbled:* Reconstruct the green pebbles on the right subpath connected to its parent v, and act as in the second case. This reconstruction is obtained by executing forward traversal of the sub-tree T_1 starting at v, where v is the last pebbled node on the last right subpath. We amortize this cost against the back traversal starting at the right child of the mirror node of v and ending at the current position. This sequence includes back-steps to all nodes in the left sub-tree T_2 of the mirror of v. This amortization is valid since the right child of v was un-pebbled in a forward step in which the new position was the right child of the mirror of v. Since the size of T_1 is twice the size of T_2, each back-step is charged with at most two reconstruction steps.

As in Claim 2.1, we have that each back step can be charged at most $\log n$ times, resulting with:

Theorem 2.3 *The list pebbling algorithm supports full traversal in at most* $\log n$ *amortized list-steps per back-step, one list-step per forward step, using* $2 \log n$ *pebbles.*

3 The Recycling Bin data structure

A full algorithm implementation requires management of pebbles allocation. The allocation of pebbles is handled by an auxiliary data structure, denoted as the *recycling bin data structure*, or *RB*. The RB data structure supports the following operations:
Put pebble: put a released pebble in the RB for future use; this occurs in the

simple back-step, in which the current position is a left child, and therefore its predecessor is its parent. (Back-step Case 1.)
Get pebble: get a pebble from the RB; this occurs in a simple forward step, in which the successor of the node of the current position is its left child. (Forward-step Case 1.)
Put list: put a released list of pebbles - given by a pair of pointers to its head and to its tail – in the RB for future use; this occurs in the non-simple forward step, in which the pebbles placed on a full right path should be released. (Forward-step Case 2.)
Get list: get the most recent list of pebbles that was put in the RB and is still there (i.e., it was not yet requested by a get list operation); this occurs in a non-simple back-step, in which the current position is a right child, and therefore its predecessor is a rightmost leaf, and it is necessary to reconstruct the right path from the left sibling of the current position to its rightmost leaf. It is easy to verify that the list that is to be reconstructed is indeed the last list to be released and put in the RB. (Back-step Cases 2 or 3.)

The RB data structure consists of a bag of pebbles, and a set of lists consisting of pebbles and organized in a double-ended queue of lists. The bag can be implemented as, e.g., a stack. For each list, we keep a pointer to its header and a pointer to its tail, and the pairs of pointers are kept in doubly linked list, sorted by the order in which they were inserted to RB. Initially, the bag includes $2\log n$ pebbles and the lists queue is empty. Based on the claim, the $2\log n$ pebbles will suffice for all operations.

In situations in which we have a get pebble operation and an empty bag of pebbles, we take pebbles from one of the lists. For each list ℓ we keep a counter M_ℓ for the number of pebbles removed from the list.

The operations are implemented as follows:
Put pebble: Adding a pebble to the bag of pebbles (e.g., stack) is trivial; it takes $O(1)$ time.
Put list: a new list is added to the tail of the queue of lists in RB, to become the last list in the queue, and M_ℓ is set to 0. This takes $O(1)$ time.
Get pebble: If the bag of pebbles includes at least one pebble, return a pebble from the bag and remove it from there. If the bag is empty, then return and remove the last pebble from the list ℓ, which is the oldest among those having the minimum M, and increment its counter M_ℓ. This requires a priority queue according to the pairs $\langle M_\ell, R_\ell\rangle$ in lexicographic order, where R_ℓ is the rank of list ℓ in RB according to when it was put in it. We show below that such PQ can be supported in $O(1)$ time.

Get list: return the last list in the queue and remove it from RB. If pebbles were removed from this list (i.e., $M_\ell > 0$), then it should be reconstructed in $O(2^{M_\ell})$ time prior to returning it, as follows. Starting with the node v of the last pebble currently in the list, take 2^{M_ℓ} forward steps, and whenever reaching a node on the right path starting at node v place there a pebble obtained from RB using the get pebble operation. Note that this is Back-step Case 3, and according to the analysis and claim the amortized cost per back-step is $O(\log n)$ time.

Claim 3.1 *The priority queue can be implemented to support delmin operation in $O(1)$ time per retrieval.*

We can conclude:

Theorem 3.2 *The list pebbling algorithm using the recycling bin data structure supports $O(\log n)$ amortize time per back-step, $O(1)$ time per forward step, using $O(\log n)$ pebbles.*

4 The advanced list pebbling algorithm

The list pebbling algorithm of Section 2 can be improved in several aspects: bounding the time per back-step in the worst-case (rather than amortized); making it sensitive to the number of back-steps; limiting the number of pebbles to $\log n$ (rather than $O(\log n)$; and reducing the overhead per forward step to arbitrary small ϵ. The advanced list pebbling algorithm that obtains these improvements is given in the full paper [7].

Ensuring $O(\log n)$ list steps per back-step in the worst case is obtained by processing the rebuilt of the missing green paths along the sequence of back traversal, using a new set of red pebbles. For each green path, there is one red pebble whose function is to progressively move forward from the deepest pebbled node in the path, to reach the next node to be pebbled. By synchronizing the progression of the red pebbles with the back-steps, we can guarantee that green paths will be appropriately pebbled whenever needed.

The time per back step is reduced to $O(\log i)$, where i is the distance from the current position to the farthest point reached so far. This is obtained by delaying the update of all green paths that are not "essential" for immediate execution. Specifically, we only update green paths that are in the two minimal virtual sub-trees that include the path from the current position to the farthest point reached so far. The number of pebbles used by the advanced algorithm is bounded by $\log n$ (rather than $O(\log n)$), by having a careful implementation and a sequence of refinements. The overhead per forward step is reduced to arbitrary small ϵ by applying the algorithm for a list of "super-nodes", each consisting of a group of $1/\epsilon$ nodes from the given list.

Theorem 4.1 *The list pebbling algorithm can be implemented on a RAM to support $O(\log i)$ time in the worst-case per back-step, where i is the distance from the current position to the farthest point traversed so far. Forward steps are supported in $O(1)$ time in the worst case, $1+\epsilon$ amortized time per forward step, and no additional list-steps, using $\log n$ pebbles. The memory used is at most $1.5(\log n)$ words of $\log n + O(\log \log n)$ bits each.*

5 Reversal of program execution

Uni-directional linked list can represent the execution of programs. A program state can be thought of as nodes of a list, and a program step is represented by a directed link between the nodes representing the appropriate program states. Since typically program states cannot be easily reversed, the list is in general uni-directional list.

Moving from a node in a linked list that represents a particular program back to its preceding node is equivalent to reversing the step represented by the link. Executing a back traversal on the linked list is hence equivalent to rolling back the program. Let the sequence of program states in a forward execution be $s_0, s_1, \ldots, s_T$. A *rollback* of a program of some state s_j is changing its state to the preceding state s_{j-1}. A rollback step from state s_j is said to be the *i'th rollback step* if state s_{j+i-1} is the farthest state that the program has reached so far.

We show how to efficiently support back traversal with negligible overhead to forward steps.

Theorem 5.1 *Let P be a program, using memory of size M and time T. Then, at the cost of recording the input used by the program, and increasing the memory by a factor of $O(\log T)$ to $O(M \log T)$, the program can be extended to support arbitrary rollback steps as follows. The i'th rollback step takes $O(\log i)$ time in the worst case, while forward steps take $O(1)$ time in the worst case, and $1+\epsilon$ amortized time per step.*

The rollback method of Theorem 5.1 can be effectively combined with delta-encoding, which enables quick access to the last sequence of program states by encoding only the differences between them.

References

[1] A. M. Ben-Amram and H. Petersen. Backing up in singly linked lists. In *ACM Symposium on Theory of Computing*, pages 780–786, 1999.

[2] M. A. Bender, A. Fernandez, D. Ron, A. Sahai, and S. P. Vadhan. The power of a pebble: Exploring and mapping directed graphs. In *ACM Symposium on Theory of Computing*, pages 269–278, 1998.

[3] Y. C. Chung, S.-M. Moon, K. Ebcioglu, and D. Sahlin. Reducing sweep time for a nearly empty heap. In *27th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM Press, 2000.

[4] D. Coppersmith and M. Jakobsson. Almost optimal hash sequence traversal. In *Fifth Conference on Financial Cryptography*, 2002.

[5] D. S. Hirschberg and S. S. Seiden. A bounded-space tree traversal algorithm. *Information Processing Letters*, 47(4):215–219, 1993.

[6] M. Jakobsson. Fractal hash sequence representation and traversal. In *IEEE International Symposium on Information Theory*, 2002.

[7] Y. Matias and E. Porat. Efficient pebbling for list traversal synopses. Technical report, Tel Aviv University, 2002.

[8] N. Pippenger. Advances in pebbling. In *International Colloquium on Automata, Languages and Programming*, pages 407–417, 1982.

[9] H. Schorr and W. M. Waite. An efficient machineindependent procedure for garbage collection in various list structures. *Communications of the ACM*, 10(8):501–506, Aug. 1967.

[10] J. Sobel and D. P. Friedman. Recycling continuations. In *ACM SIGPLAN International Conference on Functional Programming*, volume 34(1), pages 251–260, 1999.

[11] D. Walker and J. G. Morrisett. Alias types for recursive data structures. In *Types in Compilation*, pages 177–206, 2000.