Quickly pulling a tree out of a stream

Staal A. Vinterbo
Division of Biomedical Informatics, UCSD
sav@ucsd.edu

DBMI report SAV-November 17, 2011

Abstract

A simple streaming algorithm for transforming streams of labels into corresponding streams of edges for a directed and ordered tree structure is presented. For each received label it produces an edge in constant time. If the order of the tree structure is k, and the size of the label set is n, the algorithm can be implemented to use $O(\log(k)\log(n))$ space per stream.

1 Introduction

The reconstruction of a (directed) binary tree from its in-order and pre-order (or alternatively post-order) traversals is a well known problem. Knuth posed the problem as an exercise [4], Mu and Bird analyze four different algorithms from a functional standpoint [6], and Olariu et al. present a CREW PRAM parallel algorithm based on merging sorted sequences [7]. It is not difficult to show that a single traversal contains insufficient information for such a reconstruction. For non-ordered binary trees, the in-order traversal is needed as only considering the pre- and post-order traversals is again insufficient. However, if order is introduced (e.g, one can tell whether a branch of a binary tree is the left or right) the tree *can* be reconstructed from the pre- and post-order traversals. This also holds for directed ordered trees with arbitrary arity. The algorithm presented here is for the transformation of a stream of vertices into a stream of edges of a given tree structure. In particular, as vertices arrive in turn, every time it sees a new vertex it outputs an edge. This can be done if the vertices are presented in pre-order, and consequently the edges are output in pre-order as well. As a directed ordered tree is uniquely defined by its pre-order list of edges, the vertex to edge stream transformation is a streaming version of a tree recovery problem.

The motivation for this work comes from the problem of finding a best possible match of a particular weighted tree in a simple weighted graph. This can be done by comparing a set of trees extracted from the graph to the target tree, for example using the color coding approach of Alon et al. [1]. The set of trees is generally very large, and one way to reduce the necessary work is to progressively extend partial candidate matches in parallel such that candidates that cannot be optimal can be removed as soon as possible. A convenient way to extend a partial match is vertex wise, i.e., by processing trees in terms of a stream of vertices.

Due to the number of streams involved in our motivating problem, the algorithm should work constant time for each vertex with a minimal use of overall storage space.

2 Methods

Consider the tree in Figure 1. With the exception of the first label, the algorithm outputs a new edge

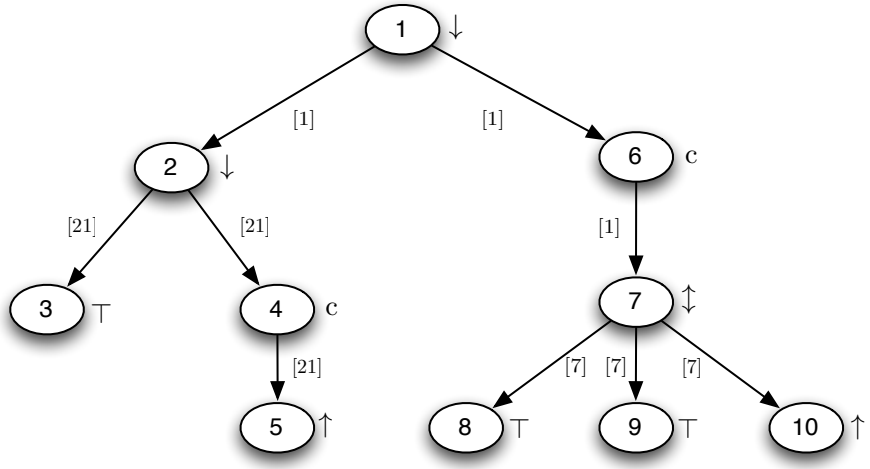

Figure 1: The example tree.

on seeing each vertex label on the stream as they appear. Without lookahead, the only options of labels in this edge are those already observed. This leads to the restriction that the vertex stream must be considered in pre-order of some ordered tree. The order of the tree must be fixed a priori since all the information received in the stream is the pre-order listing (there are $O((k-1)!)$ ways to order a directed tree, think of a star). With this in mind, it seems natural to consider the stream as being the pre-order listing of the given ordered tree structure (and not for example a rotated version of this structure). Now, when vertex v is seen, the parent u can be looked up among the vertices already seen, and the edge (u, v) can be produced. The issue now is how efficiently can this be done? Again looking at the tree in Figure 1, the parent of vertex number 4 in the pre-order sequence is the second one. Similarly, for the sixth the parent is the first. One option is to pre-compute all these positional indices, store the already seen labels, and look up the parents as vertices are received in order. Both storing and looking up labels can be done in constant time using an array. Consequently, the array must be able to store $k-1$ labels. However, only parents need to be stored, and only those needed. At any given time, there are only at most d needed parents when the depth of the tree is d. Unfortunately, the maximum depth of a tree of order k is $k-1$. However, assuming that the minimum vertex degree is 3 (minimum out degree 2) for non-leaves, the maximum depth becomes $\log_2 k$. By treating degree 2 vertices as "pass through" vertices, one can get away with storing only $\log_2 k'$ needed parents where k' is the number of vertices with degree different from 2. The following describes a 5 instruction stack

machine that allows storing and access to parents in constant time, and operates on an at most $\log_2 k'$ size stack.

The Machine In addition to a stack that can grow dynamically, as for example is the case if it is implemented in terms of a linked list, the machine has two registers, "current", and "parent". Right after performing an instruction, the machine checks if there are more instructions to perform. If there are none, the machine halts. Otherwise, if the stream still has unread elements, it reads a label from the input stream, moves it into the "current" register, and outputs the edge (u, v), where u is the label contained in the "parent" register, and v is the label contained in the "current" register. The five instructions for this machine are:

- $\downarrow$ "push": copy "current" to "parent" and push "current" on the stack
- $\top$ "top": copy the top of the stack into "parent"
- $\uparrow$ "pop": pop the stack, and copy the new top of the stack into "parent"
- $\updownarrow$ "poppush": pop the stack, copy "current" to "parent" and push "current on the stack"
- c "copy": copy "current" into "parent"

The instructions $\top$ and $\uparrow$ could be the last instructions of the program ($\top$ if the tree is a path, $\uparrow$ otherwise), in which case the stack is empty. In this case, $\top$ and $\uparrow$ do nothing. At the very start, the machine is initialized by loading the first element from the stream into the "current" register.

Again consider the tree in Figure 1. Next to each vertex there is an instruction symbol. Table 1 shows the state of the machine, the input stream, and output after each instruction in the program consisting of the list of instructions obtained by a pre-order traversal of the tree in Figure 1. Note that the

instruction	current	parent	stack	stream	edge
-	1	-	$\emptyset$	2,3,4,5,6,7,8,9,10	-
$\downarrow$	2	1	1	3,4,5,6,7,8,9,10	$(1, 2)$
$\downarrow$	3	2	2,1	4,5,6,7,8,9,10	$(2, 3)$
$\top$	4	2	2,1	5,6,7,8,9,10	$(2, 4)$
c	5	4	2,1	6,7,8,9,10	$(4, 5)$
$\uparrow$	6	1	1	7,8,9,10	$(1, 6)$
c	7	6	1	8,9,10	$(6, 7)$
$\updownarrow$	8	7	7	9,10	$(7, 8)$
$\top$	9	7	7	10	$(7, 9)$
$\top$	10	7	7	$\emptyset$	$(7, 10)$
$\uparrow$	-	-	$\emptyset$	$\emptyset$	-

Table 1: Running the program obtained from the tree in Figure 1 on its pre-order vertex label list as the input stream.

program produces the edges of the tree in pre-order. The edges in Figure 1 are also annotated with the stack contents as the program derived from the same tree proceeds along them.

Compiling a tree into a program The program for producing the pre-order edge list as demonstrated in Table 1 was obtained from the pre-order list of instructions associated with each vertex in Figure 1. Let a vertex be represented by a tuple (l, s) where l is a label and s is a list of subtree root vertices. The list of instructions associated with the tree rooted at (l, s) is produced recursively as following:

- if s is empty, i.e., the vertex is a leaf, return the list only containing $\top$,
- if $s = [t]$, the vertex is a "pass through" vertex and c followed by list corresponding to the instructions for the tree rooted at t is returned,
- if $s = [t_1, t_2, \ldots, l_m]$ for $m > 1$, and s_i is the list of instructions corresponding to the subtree rooted at t_i, first substitute the first non-c instruction in s_m with $\updownarrow$ if it is $\downarrow$, and $\uparrow$ if it is $\top$, before returning $\downarrow$ followed by the concatenation of the s_i's in order.

Time and space complexity The instructions all copy values to and from the registers, push, pop, and read the top of the stack. All these operations can be implemented to run in constant time under the standard computational model. Assuming that reading from the stream and single edge output can be done in constant time as well, the processing of each stream element takes constant time. Since degree 2 vertices are considered "pass through" and not influence the storage of elements on the stack, they can without loss of generality be disregarded in the stack space analysis. Hence assume that there are no degree 2 vertices in the tree. The maximum depth of an order k tree without degree 2 vertices is $\log_2 k$. Hence the algorithm needs to store at most $\log_2 k$ labels on the stack. If the label set size is n, it can represent these with $\log_2 n$ bits. Hence, for a program based on an order k tree with labels coming from a set of size n, it needs only store at most $\log_2(k) \log_2(n)$ bits on the stack.

If two lists can be concatenated in constant time, the number of operations we need to perform in the compilation of the tree using the outlined recursive approach is $O(k + k_2) = O(k)$ where k_2 is the number of degree 2 vertices in the tree. If concatenation of two lists can only be done in time proportional to the length of the first list, the compilation can be done in $O(k^2 + k_2) = O(k^2)$ time.

3 Discussion

The above presented streaming algorithm transforms a stream of labels, into a stream of edges for a given tree structure. The motivation for doing this was the parallel transformation of label streams into edge streams corresponding to the same given tree structure in a "Single Instruction Multiple Data" (SIMD) machine fashion. Due to the large number of streams, time and space usage is of consequence. The algorithm employs a stack to ensure constant time usage per label, and a simple trick to guarantee that the stack does not grow beyond $O(\log k)$ where $k-1$ is the number of edges in the tree structure. It can be extended to compute anything that can be computed recursively using a binary operator over edges in pre-order. This can be seen by recognizing that this type of recursive computation is a "fold" over the sequence of edges (see for example [5]). In appendix A a full OCaml implementation of both the tree compiler and the machine using folds is presented.

References

[1] N. Alon, R. Yuster, and U. Zwick. Color-coding. *Journal of the ACM (JACM)*, 42(4):844–856, 1995.

[2] J. Hughes. Why Functional Programming Matters. *Computer Journal*, 32(2):98–107, 1989.

[3] Graham Hutton. A tutorial on the universality and expressiveness of fold. *Journal of Functional Programming*, 9(4):355–372, July 1999.

[4] D.E. Knuth. *Art of Computer Programming, Volume 1: Fundamental Algorithms*. Addison-Wesley Professional, 1997.

[5] E. Meijer, M. Fokkinga, and R. Paterson. Functional programming with bananas, lenses, envelopes and barbed wire. In *Functional Programming Languages and Computer Architecture: 5th ACM conference, Cambridge, MA, USA, August 26-30, 1991: proceedings*, pages 124–144. Springer, 1991.

[6] Shin C. Mu and Richard S. Bird. Rebuilding a tree from its traversals: a case study of program inversion. In Atsushi Ohori, editor, *Programming Languages and Systems. Proceedings*, number 2895 in Lecture Notes in Computer Science, pages 265–282. Springer-Verlag, 2003.

[7] S. Olariu, M. Overstreet, and Z. Wen. Reconstructing a binary tree from its traversals in doubly logarithmic crew time. *Journal of Parallel and Distributed Computing*, 27(1):100, 1995.

A An OCaml Implementation

Below is a complete OCaml implementation of tree compiler and the machine in terms of *folds*. Specifically, the compiler operates as a fold over trees [2], and the machine operates in terms of a left fold over a list. The list fold can easily be replaced by a fold over other regular data structures such as enumerations and streams [3, 5]. Note that the implementation below stores labels directly on the stack without recoding.

The *tree* data type and the fold operation on trees as follows:

```
(* a tree is a pair consisting of a label and a list of subtrees *)
type tree = Node of (int * tree list)

(* tree fold *)
let rec treefold f g z tree =
  let rec treefolds f g z subtrees = match subtrees with
    |  [] -> z
    | (x::xs) -> g (treefold f g z x) (treefolds f g z xs) in
  match tree with Node(label, subtrees) -> f label (treefolds f g z subtrees)
```

The machine language consists of five operations for which a type is defined. Translation of the type names into functions that change the state of the machine can be done as follows:

```
(* the machine operations *)
type operations = PUSH | POP | POPPUSH | C | TOP

(* translate the operation type name into an executable function *)
let opfun x = match x with
  | PUSH -> (fun (current, parent, stack) -> (current, current, current::stack))
  | TOP -> (fun (current, parent, stack) -> match stack with
      | [] -> (current, parent, stack) (* last operation in an input path *)
      | (x::xs) -> (current, x, x::xs))
  | POP -> (fun (current, parent, stack) -> match stack with
      | [] -> (current, parent, stack) (* last operation in non-path *)
      | (x::y::ys) -> (current, y, y::ys))
  | POPPUSH -> (fun (current, parent, x::xs) -> (current, current, current::xs))
  | C -> (fun (current, parent, stack) -> (current, current, stack))
```

The compiler that takes a tree as input and outputs a list of machine instructions, i.e., a list of functions as defined above, is implemented by a fold over the tree as follows:

```
(* split a list into a prefix and a suffix such that the prefix
   contains only elements that satisfy predicate p and the first
   element in the suffix does not *)
let rec splitwhile p l = match l with
  | [] -> ([], [])
  | (x::xs) ->
    if p x then let (left, right) = splitwhile p xs in (x::left, right)
    else ([], l)

(* fix rightmost subtree: repair (using splitwhile).
   Translates first PUSH and TOP into POPPUSH and POP *)
let repair oplist =
  let (left, right) = splitwhile (fun x -> x == C) oplist in
  match right with
    | [] -> oplist
    | (x::xs) -> match x with
        | PUSH -> left @ (POPPUSH::xs)
        | TOP -> left @ (POP::xs)
        | _ -> oplist

(* fold function for generating stack operations *)
let opsf label ilist = match ilist with
  | [] -> [TOP]
  | (l::[]) -> C::l
  | _ -> let (x::xs) = List.rev ilist in
         PUSH::(List.concat (List.rev ((repair x) :: xs)))

(* generate list of symbolic operations *)
let treeopsl = treefold opsf (fun x l -> x::l) []

(* generate a list of executable machine operations *)
let treeops tree = List.map opfun (treeopsl tree)
```

The machine that runs a program is again defined in terms of a fold, this time over the stream (represented as a list):

```
(* the machine *)
let machine program labels =
  let step (outstream, state) label =
    let (current, parent, stack, program) = state in
    match program with
      | [] -> (outstream, state) (* program has ended *)
      | (operation::rest) ->
        let (_, newp, news) = operation (current, parent, stack) in
        ((newp, label)::outstream, (label, newp, news, rest)) in
  match labels with
    | [] -> []
    | (x::xs) ->
      let (l, _) = List.fold_left step ([], (x, x, [], program)) xs in
      List.rev l
```

Running the machine on the tree in Figure 1 can be done as follows:

```
let tree =
  Node (1,
        [Node (2,
               [Node (3, []);
                Node (4,
                      [Node (5, [])])]);
         Node (6,
               [Node (7,
                      [Node (8, []);
                       Node (9, []);
                       Node (10, [])])])])
```

Then, running the machine as:

```
let stream = ["one"; "two"; "three"; "four"; "five"; "six"; "seven";
              "eight"; "nine"; "ten"] and
    program = treeops tree in
machine program stream;;
```

produces

```
[("one", "two"); ("two", "three"); ("two", "four"); ("four", "five");
 ("one", "six"); ("six", "seven"); ("seven", "eight"); ("seven", "nine");
 ("seven", "ten")]
```

as expected.