Algorithms for External Memory

Lecture 6

Graph Algorithms - Weighted List Ranking

Lecturer: Nodari Sitchinava
Scribe: Andi Hellmund, Simon Ochsenreither

1 Introduction & Motivation

After talking about I/O-efficient tree algorithms in the previous lectures, we started the last topic of the external memory section, that is I/O-efficient graph algorithms. The problem addressed in this lecture is to compute the weighted list rank of a linked list. Before motivating the need for I/O-efficient algorithms for linked lists, we will shortly recap some of the basics from graph theory lectures. A graph is a tuple $G = (V, E)$ with V being the set of nodes of the graph and E being the set of edges $(u, v), u, v \in V$ between nodes u and v. The degree $deg(v)$ of a node represents the total number of incoming and outgoing edges; for directed graphs we may further distinguish between $deg_{in}(v)$ as the number of incoming edges and $deg_{out}(v)$ as the number of outgoing edges of a node. A singly, non-circular linked list is a linear graph with each node v having $deg_{in}(v) = deg_{out}(v) = 1$ assuming markers for the start and end of the list as shown figure 2. The rank $rank(v)$ of a node in a linked list is defined as the number of edges that need to be traversed from the beginning of the list to reach node v. Finally, the weighted list ranking problem is described as the process to compute a per-node (or per-edge), cumulative score based on node (or edge) weights as shown in figure 1. In the trivial case of all weights being 1, the per-node score represents the rank of a node.

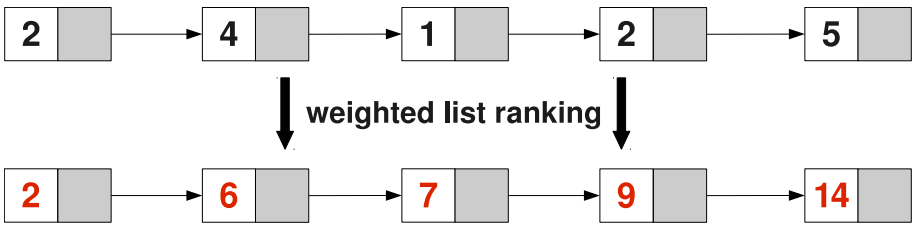

Figure 1: Example for Weighted List Ranking

Considering linked lists and the list ranking problem from the I/O perspective, the problematic aspects of its simplicity are the *next* pointers targeting arbitrary memory locations as shown in figure 2.

Using our commonly used external memory model as introduced in lecture 1 with M being the size of the internal memory and B being the size of blocks transferred between internal and external memory, traversing this linked list requires a single I/O per visited node for $B = 2$ and $M = 2B$. This shows that the upper bound of simple list traversal

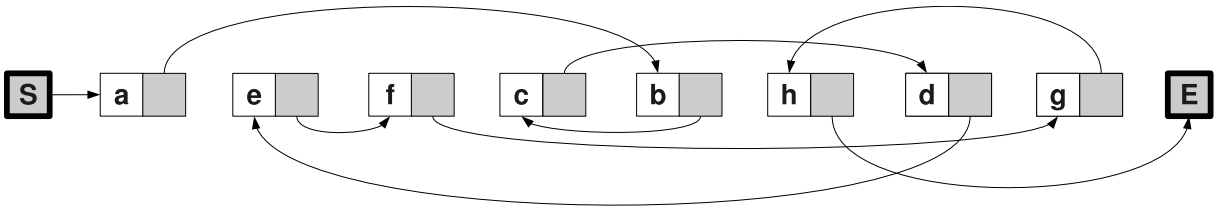

Figure 2: Generic Linked List with Distinct Non-Node Markers for the Start and End of the List

for solving the list ranking problem is $\mathcal{O}(N)$ I/O operations for a list with N nodes. In this context, remember the complexity notation used throughout this lecture:

$$\mathcal{O}(scan(N)) = \mathcal{O}(\frac{N}{B}) < \mathcal{O}(sort(N)) = \mathcal{O}(\frac{N}{B} \cdot \log_{\frac{M}{B}} \frac{N}{B}) \ll \mathcal{O}(N)$$

2 Weighted List Ranking

In this section, we present an I/O-efficient algorithm to solve the weighted list ranking problem with an I/O complexity of $\mathcal{O}(sort(N))$[1]. Before presenting the final list ranking algorithm and its I/O complexity analysis, we will firstly introduce - as a prerequisite - two algorithms to (a) compute a pairwise sum function (PSF) for each node of the linked list and (b) to find an independent set of the linked list with a specific size. We will also show that both algorithms require $\mathcal{O}(sort(N))$ I/O complexity.

2.1 Computing Pairwise Sum Function (PSF)

In the context of solving the weighted list ranking problem, we will only consider pairwise sums as cumulative score, but the problem statement is generalized as follows. Assuming a linked-list $L = (V, E)$ with each node having assigned a weight score $val(v)$: we want to compute the function $res(v) = f(u, v)$ for each $v \in V$ with $(u, v) \in E$, i.e. function $f()$ is applied to node v and its predecessor u. For PSF, the function $f()$ is defined as $f(u, v) = val(u) + val(v)$[2].

To compute the PSF for each node of a list, the following algorithm is applied (we assume the nodes of the original list are sorted by the IDs (e.g. addresses) of the nodes):

1. Create a copy of the linked list and sort it by the ID (e.g. address) of each node's *successor*. (I/O complexity: $\mathcal{O}(sort(N))$).

2. Scan both lists, the original and the copied list, simultaneously and compute $f(u, v)$ for each node v (I/O complexity: $\mathcal{O}(scan(N))$).

Figure 3 shows the original linked list from figure 2 and its sorted copy. The links between the nodes of boths lists are omitted for clarity. Using this algorithm, it is obvious from the figure that the node and its predecessor are immediately available during simultaneous, linear scan of both lists. The total I/O complexity of this algorithm is:

$$\mathcal{O}(scan(N)) + \mathcal{O}(sort(N)) = \mathcal{O}(sort(N))$$

[1] It can be shown that the lower bound of the list ranking problem is $\Omega(sort(N))$.

[2] The first node of the linked-list requires special handling, but this corner case is omitted for simplicity.

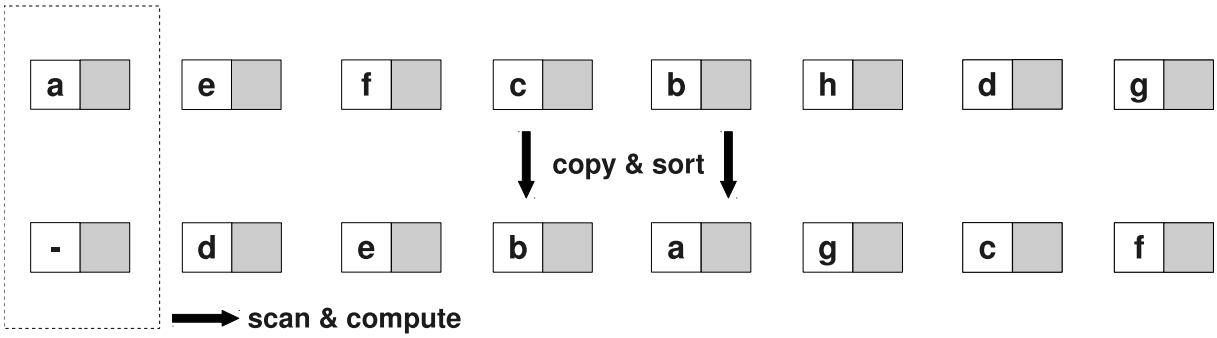

Figure 3: I/O-Efficient Computation of Pairwise-Sum-Function (PSF)

2.2 Computing Independent Set of Graph

An independent set I in a graph $G = (V, E)$ is a set of nodes $v \in V$ such that if $v \in I$, then $w \notin I$ if $(v, w) \in E$ or $(w, v) \in E$, i.e. the independent set only contains non-neighboring nodes. We will show that we may compute an independent set of a linked list of size $\frac{N}{3}$ with $\mathcal{O}(sort(N))$ I/O operations by reducing this problem to the problem of 3-coloring a linked list. Once a 3-coloring of the linked list is found, the nodes having the color with highest quantity represent an independent set.

More formally: k-coloring of a graph is defined as coloring the nodes of a graph with k distinct colors such that no two neighboring nodes are colored with the same color. Note, a linked list may generally be colored with two distinct colors by alternating the colors for each node, however this requires $\mathcal{O}(N)$ I/O complexity due to complete list traversal. For the purpose of illustrating the single steps of the algorithm, we will use the linked list in figure 4 as a base example.

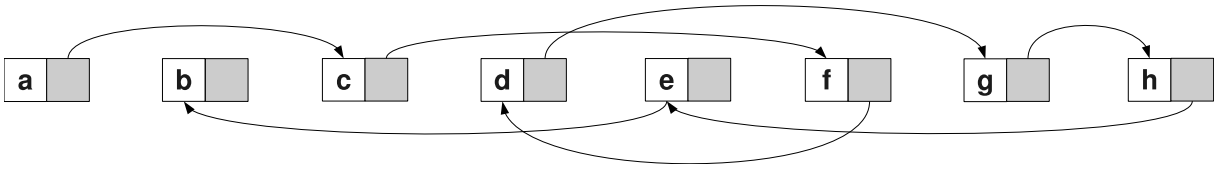

Figure 4: Example: 3-coloring of Linked List

2.2.1 3-Coloring of Linked List

The first step in determining the 3-coloring of the linked list is to compute contiguous forward and backward lists of the original linked list. A contiguous forward list is defined as a set of neighboring nodes with increasing memory address, e.g. the nodes **a-c-f** and **d-g-h** in figure 4 are examples for two forward lists. Contiguous backward lists are defined likewise with decreasing memory address as exemplified with nodes **h-e-b** and **f-d** in figure 4.

The function to process the forward lists is shown in algorithm 1. It iterates over the elements of the linked list by increasing memory addresses, i.e. it does not follow the *next* of each visited node, and builds up multiple forward lists each colored with alternating colors red and blue – the first node of each forward list is colored red. A priority queue (PQ) - prioritized by the address of the nodes - is used to manage multiple forward lists

simultaneously, e.g. the forward list **d-g-h** is started before the forward list **a-c-f** is finished. The function to process the backward lists (algorithm 2) works likewise. The main differences are that nodes are processed by decreasing memory addresses and the nodes of each backward list are colored with alternating colors green and blue – the first node of each forward list is colored green. The function opposite() referenced in both algorithms returns the opposite color of the input argument, e.g. 'blue' for the input argument 'red' in the case of forward lists.

Algorithm 1 Find Forward Lists in Linked List

```
 1: function PROCESS_FORWARD_LIST(L : linked_list)
 2:     for each v ∈ L in increasing order of addresses do
 3:         if v.addr == PQ.minKey() then                          ▷ middle of list
 4:             x = PQ.deleteMin()
 5:             if v.nextptr == NULL then                          ▷ end element of L
 6:                 color v with x.color
 7:             else if v.nextptr > v.addr then          ▷ not last element of forward list
 8:                 color v with x.color
 9:                 PQ.insert(new object(key = v.nextptr, color = opposite(x.color)))
10:             end if
11:         else if v.nextptr > v.addr then                  ▷ start of new forward list
12:             color(v, 'red')
13:             PQ.insert(new object(key = v.nextptr, color = 'blue'))
14:         end if
15:     end for
16: end function
```

Algorithm 2 Find Backward Lists in Linked List

```
 1: function PROCESS_BACKWARD_LIST(L : linked_list)
 2:     for each v ∈ L in decreasing order of addresses do
 3:         if v.addr == PQ.maxKey() then                          ▷ middle of list
 4:             x = PQ.deleteMax()
 5:             if v.nextptr == NULL then                          ▷ end element of L
 6:                 color v with x.color
 7:             else if v.nextptr < v.addr then          ▷ not last element of forward list
 8:                 color v with x.color
 9:                 PQ.insert(new object(key = v.nextptr, color = opposite(x.color)))
10:             end if
11:         else if v.nextptr < v.addr then                  ▷ start of new forward list
12:             color(v, 'green')
13:             PQ.insert(new object(key = v.nextptr, color = 'blue'))
14:         end if
15:     end for
16: end function
```

Applying algorithms 1 and 2 to the linked list in figure 4 yields two forward lists and two backward lists colored as shown in figure 5.

As obviously visible in this figure, coloring conflicts may occur for some nodes in the forward and backward list, as for example for node **f** that is colored 'red' in the forward

Figure 5: Example: 3-coloring of Linked List - Forward and Backward Lists

list and 'green' in the backward list. Note that these conflicts only occur for start or ending nodes of forward/backward lists, because an inner node cannot be part of both lists, a forward and a backward list, – otherwise the in- and out-directions would point into different directions. To resolve the coloring conflicts, the ending node in conflict is colored like starting node in conflict, e.g. the node **h** in the second forward list is colored 'green' instead of 'red'. Since only the colors of ending nodes of forward/backward lists are swapped, the graph coloring property as defined above is not violated[3].

I/O complexity analysis

The algorithm uses a priority queue (PQ) whereof each operation, i.e. query, insertion and deletion, takes $\mathcal{O}(\frac{1}{B}\log_{\frac{M}{B}}\frac{N}{B})$ I/O operations. The forward and backward algorithms each iterate over all nodes in the linked list, and perform at most 3 priority queue operations for each node. So the total I/O complexity of each algorithm is:

$$3 \cdot N \cdot \mathcal{O}(\frac{1}{B}\log_{\frac{M}{B}}\frac{N}{B}) = \mathcal{O}(sort(N))$$

2.3 Weighted List Ranking Algorithm

To solve the weighted list ranking problem, we apply the following recursive algorithm to achieve storing the cumulative per-node scores within the nodes rather than in arbitrary memory regions increasing the I/O complexity of the computation.

1. Find independent set I of size $\frac{N}{c}$ with predefined constant c. (I/O complexity: $\mathcal{O}(sort(N))$, see section 2.2)
2. Compute pairwise sum function (PSF) on non-members of I. (I/O complexity: $\mathcal{O}(sort(N))$, see section 2.1)
3. Bridge-out $\forall v \in I$. (I/O complexity: $\mathcal{O}(sort(N))$)[4]
4. Recursively solve weighted list ranking on the new list.
5. Bridge-in $\forall v \in I$. (I/O complexity: $\mathcal{O}(sort(N))$)
6. Compute PSF on $\forall v \in I$. (I/O complexity: $\mathcal{O}(sort(N))$)

Figure 6 illustrates the working principle of the above algorithm with $c = 2$. The numbered circles on the left side indicate the step number of the algorithm with step 0 representing the original linked list.

[3]For an informal proof, validate the graph coloring property for different cases, e.g. forward/backward lists with even/odd lengths.

[4]The bridge out operation may be reduced to the complexity of PSF computation due to changing *next* pointers in respective nodes.

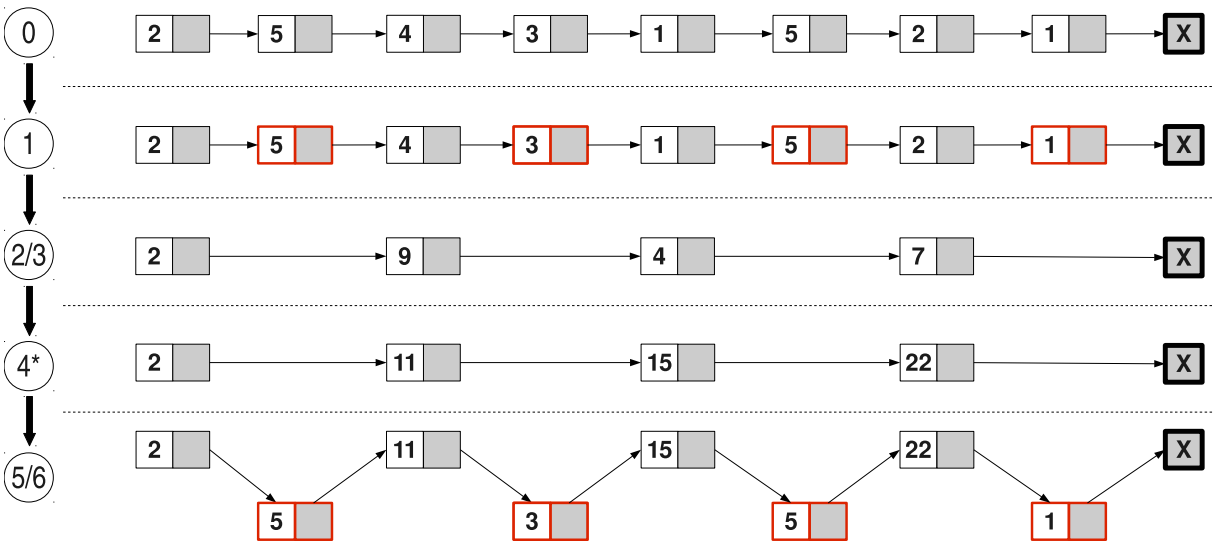

Figure 6: Example for I/O-Efficient List Ranking Algorithm

2.3.1 Analysis of I/O complexity

If the independent set at each recursive stage is of size at least N/c for some constant $c \geq 2$, then the recursive call is performed on a list of size at most $N - \frac{N}{c} = N \cdot \frac{c-1}{c}$. Then the I/O complexity is computed as:

$$\begin{aligned} Q(N) &= \begin{cases} Q(N \cdot (\frac{c-1}{c})) + \mathcal{O}(sort(N)) & \text{if } N > M \\ \mathcal{O}(\frac{N}{B}) & \text{if } N < M \end{cases} \\ &= \sum_{i=0}^{\log_2 \frac{N}{B}} \left(\frac{c-1}{c}\right)^i \cdot \mathcal{O}(sort(N)) \end{aligned}$$

Since $c \geq 2$, $\frac{c-1}{c} < 1$ and we can apply the calculus for geometric series to the summation, resulting in the final I/O complexity of:

$$Q(N) = \mathcal{O}(sort(N))$$

In Section 2.2 we showed how to find an independent set of size at least $N/3$, i.e. in our case $c = 3$.

2.4 Applications of the Weighted List Ranking Algorithm

The applications of weighted list ranking were mostly covered in the lecture 7, but we will discuss them here for the sake of completeness of this lecture. All applications shown here are considered in the context of analyzing or computing tree properties. As a prerequisite for this, we firstly need to transform the tree such that each undirected edge is represented by two directed edges, one going down and the other going up, as shown in figure 7.

Visibly, connecting the down-going and up-going edges yields a linked list with the nodes of the tree occurring multiple times. To effectively connect all the down-going and up-going edges in memory for purpose of running through the nodes in a single run, we construct the Euler Tour of a tree. The Euler Tour of a graph is a cycle through the graph such that each node/edge is visited once. Considering a single node in the tree,

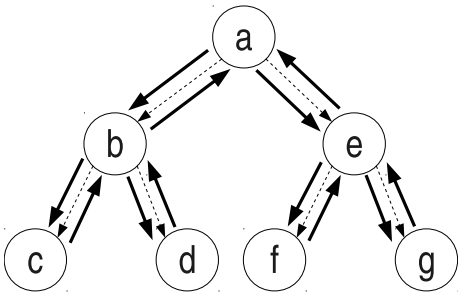

Figure 7: Transformation of Tree

as in figure 8, we may compute the Euler Tour by creating edges $(i_j, o_{j+1} \bmod k)$ where k is the number of child nodes plus the current node, i.e. $k = 3$ in our example. For correctness of this specification, $x \bmod 1 := 0$. The I/O complexity for constructing the Euler Tour is $\mathcal{O}(sort(N))$ due to sorting the indices of edges.

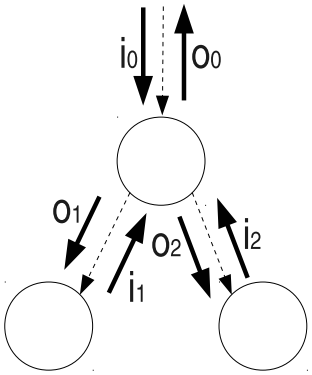

Figure 8: Construction Principle of Euler Tour

In the following paragraphs, we will introduce three examples for weighted list ranking applications. While the Euler Tour is important for finding a path for visiting the nodes, we will stick to the graphical representation shown figure 7 for clarity.

2.4.1 Rooting a tree: Parent/Child Relationships for Un-Rooted Trees

An un-rooted tree is a tree without an explicitly marked root node. An example of an un-rooted tree is shown in figure 9.

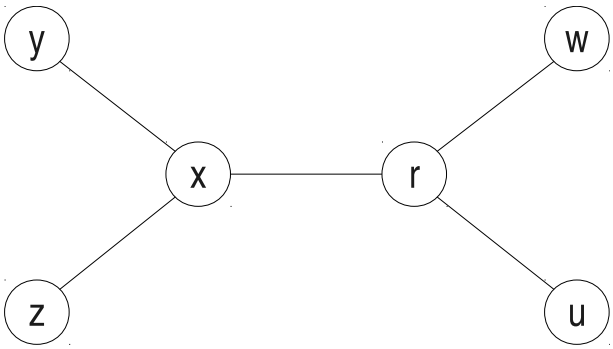

Figure 9: Example for an Un-Rooted Tree

As an example, assume that node **r** is selected as root node. Applying the weighted list ranking (of the Euler Tour) with each down-going and up-going edge weighted with 1, yields the graph in figure 10.

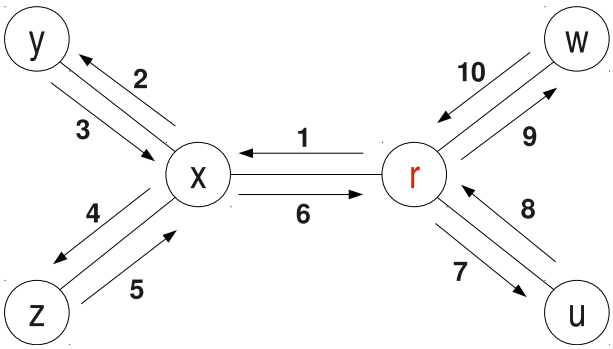

Figure 10: Example for an Un-Rooted Tree

Finally, an arbitrary node **x** is parent of its neighboring node **z**, if and only if, $wlr\_score((x,z)) < wlr\_score((z,x))$.

2.4.2 Finding the Depth of Each Node in a Rooted Tree

Finding the depth, i.e. distance from the root, of each node in the tree, the edges in figure 7 are weighted with 1 for down-going edges and -1 for up-going edges. Afterwards, the weighted list ranking algorithm is applied for each node (of the Euler Tour).

2.4.3 Computing the Size of Sub-Trees

To compute the size of a sub-tree, the edges in figure 7 are weighted with 1 for down-going edges and 0 for up-going edges. Afterwards, the weighted list ranking algorithm is applied for each edge (of the Euler Tour). Finally, the size of a sub-tree rooted at node **x** is computed by $wlr\_score((x, \text{parent}(x))) - wlr\_score((\text{parent}(x), x)) + 1$.