Algorithms for External Memory

Graph Algorithms - Time Forward Processing & Connected Components

Lecturer: Dr. Nodari Sitchinava
Scribe: Andi Hellmund, S. Lange

December, 5th 2012

1 Introduction & Motivation

In this lecture, we complete the external memory algorithms by considering graph algorithms again, specifically the time forward processing problem and the problem of finding connected components. Note, this lecture also contained a section about applications of weighted list ranking – this section is shifted to the notes of the previous lecture.

2 Time Forward Processing

The problem statement of time forward processing (TFP) is very similar to the one of weighted list ranking. The base data structure of TFP is a directed acyclic graph (DAG) $G = (V, E)$, as exemplified in figure 2, with each node (or edge) being weighed by a numeric score, $w(v)$.

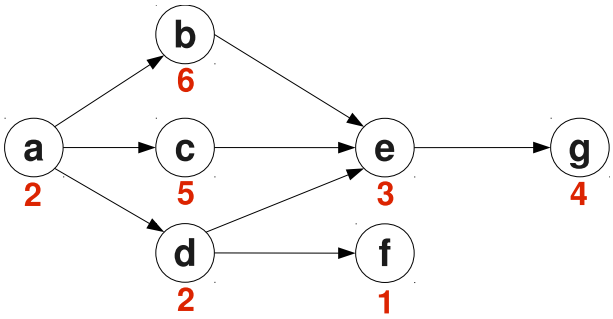

Figure 1: Directed Acyclic Graph with Node Scores

A second prerequisite is that the DAG is stored in topological order in external memory, that is, for each two nodes u, v with $(u, v) \in E$, node u is stored in memory before node v. As an example, one topological order for the DAG in figure 2 is **a-b-c-d-e-g-f**. Based on this, the aim of TFP is to recursively compute for each node $v \in V$:

$$f(v) = f(w(v), f(u_1), \ldots, f(u_k))$$

where $u_1, \ldots u_k$ are the predecessors of node v. Note, in case of weighted list ranking with G being a linked list, function $f()$ is defined as

$$f(v) = f(w(v), f(u_1), \ldots, f(u_k)) = f(w(v), f(u)) = w(v) + f(u)$$

In the next sections, we will (a) present the algorithm to compute the time forward processing score I/O-efficiently, (b) give a detailed example oriented to figure 2 and (c) finally prove that the I/O complexity of the algorithm is $\mathcal{O}(sort(|G|))$.

2.1 Algorithm for Time Forward Processing

As analyzed below, algorithm 1 computes the per-node score for each node in a DAG in an I/O-efficient manner.

Algorithm 1 Time Forward Processing - Recursively Compute Per-Node Score

```
1: function COMPUTE_TFP(D : DAG)
2:     Q = ∅        //Priority Queue
3:     for each v ∈ D in topological order do
4:         // let u_1, ..., u_k be in-neighbors of v
5:         Extract f(u_1), ..., f(u_k): call k-times extractMin(Q)      ▷ O((1/B) log_{M/B} (N/B))
6:         Compute x(v) = f(w(v), f(u_1), ..., f(u_k))                  ▷ O((1/B) log_{M/B} (N/B))
7:         // let w_1, ..., w_k be out-neighbors of v
8:         Insert m copies of x(v) into Q with priority of w_i.         ▷ O((1/B) log_{M/B} (N/B))
9:     end for
10: end function
```

2.1.1 Example

Here we will given an exemplary run of algorithm 1 when applied to the DAG in figure 2. Thereby we use the following topological order including the given ordinal scale:

a | 1 b | 2 c | 3 d | 4 e | 5 g | 6 f | 7

Table 2.1.1 shows the results for each node after running the algorithm in the above order. Function $f()$ is defined as the sum of all inputs. For each node, the table contains the layout of the priority queue at for-loop body entry, the value $x(v)$ and the layout of the priority queue at for-loop body exit. The superscript letters for priority queue entries indicate the priority of each entry. The resulting DAG with cumulative per-node scores is shown in figure 2.

Node	Q_{begin}	$x(v)$	Q_{end}
a	$\emptyset$	2	$2^{(p=b)} <_p 2^{(p=c)} <_p 2^{(p=d)}$
b	$2^{(p=b)} <_p 2^{(p=c)} <_p 2^{(p=d)}$	8	$2^{(p=c)} <_p 2^{(p=d)} <_p 8^{(p=e)}$
c	$2^{(p=c)} <_p 2^{(p=d)} <_p 8^{(p=e)}$	7	$2^{(p=d)} <_p 8^{(p=e)} <_p 7^{(p=e)}$
d	$2^{(p=d)} <_p 8^{(p=e)} <_p 7^{(p=e)}$	4	$8^{(p=e)} <_p 7^{(p=e)} <_p 4^{(p=e)} <_p 4^{(p=f)}$
e	$8^{(p=e)} <_p 7^{(p=e)} <_p 4^{(p=e)} <_p 4^{(p=f)}$	22	$22^{(p=g)} <_p 4^{(p=f)}$
g	$22^{(p=g)} <_p 4^{(p=f)}$	26	$4^{(p=f)}$
f	$4^{(p=f)}$	5	$\emptyset$

Table 1: Time Forward Processing Example for DAG in Figure 2

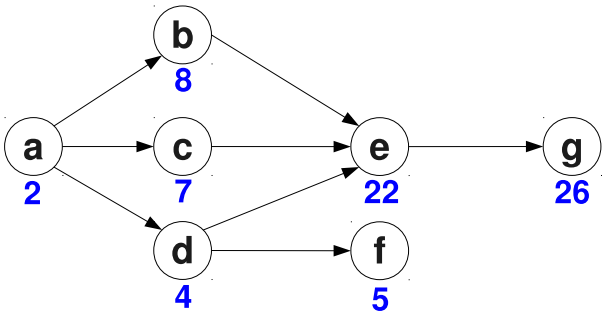

Figure 2: Directed Acyclic Graph with Cumulative Node Scores

2.2 I/O Complexity

As already known from previous lectures, the I/O complexity for inserting an entry into or deleting an entry from the priority queue takes $\mathcal{O}(\frac{1}{B}log_{\frac{M}{B}}\frac{N}{B})$ I/O operations. For a DAG with $|V| = n$ nodes and $|E| = m$ edges, there are in total $\mathcal{O}(2m)$ priority queue operations, one per edge for extracting the score from the predecessor nodes (line 5 of algorithm 1) and one per edge for inserting the cumulative score for the successor nodes (line 8). In addition to these operations, the DAG is scanned once in topological order (line 3) so that in total the algorithm takes

$$\mathcal{O}(2m) \cdot \mathcal{O}(\frac{1}{B}log_{\frac{M}{B}}\frac{m}{B}) + \mathcal{O}(scan(n)) = \mathcal{O}(\frac{m}{B}log_{\frac{M}{B}}\frac{m}{B}) + \mathcal{O}(\frac{n}{B})$$

I/O operations. For a generic DAG, the number of edges m has an upper limit $m \leq \mathcal{O}(n^2)$, so partially using this upper limit in the above formula yields:

$$\mathcal{O}(\frac{m}{B}log_{\frac{M}{B}}\frac{m}{B}) + \mathcal{O}(\frac{n}{B}) = \mathcal{O}(\frac{m}{B}log_{\frac{M}{B}}\frac{n}{B})$$

The final step in proving the initially claimed I/O complexity of $\mathcal{O}(sort(|G|))$ then is:

$$\mathcal{O}(\frac{m}{B}log_{\frac{M}{B}}\frac{n}{B}) \leq \mathcal{O}(\frac{m+n}{B}log_{\frac{M}{B}}\frac{m+n}{B}) = \mathcal{O}(sort(|V| + |E|)) = \mathcal{O}(sort(|G|))$$

□