L6: I/O Efficient Algorithms: Graphs

scribe: *Christian Schreiner, Vineel Yalamarthy*

Keywords: Priority Queue, Buffer Tree, Graph Structures, Topological Ordering

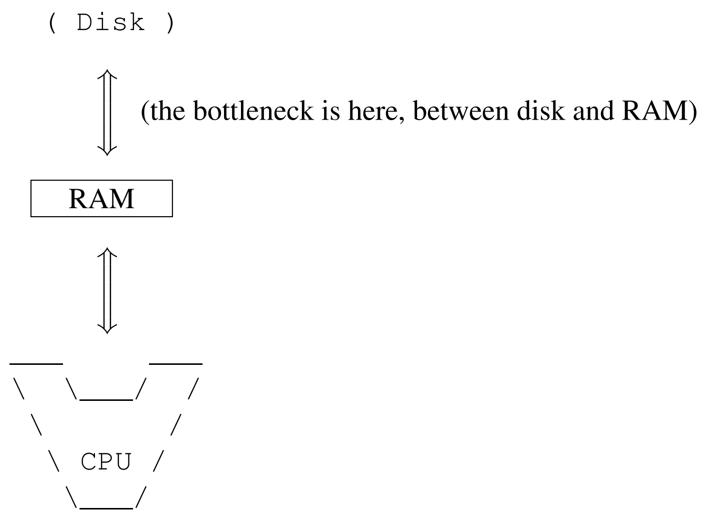
6.1 Introduction

Graph problems are ubiquitous in several domains in this age of information. From social networks, web crawling, computer networking, and computational geometry to CAD and CAM domains, several problems can be solved by modeling huge amount of data as massive graphs with millions of nodes. Extracting and visualizing necessary information from these huge graphs is not easy by using traditional BFT, DFT algorithms and RAM models. Hence, the study of external memory graph algorithms has become flourishing area of research in the recent past.

A video of this lecture is available on YouTube here:
Title: Models of Computation for Massive Data L6
URL: `http://www.youtube.com/watch?v=cy1rHu_28-c`

6.2 Review

Recall the notation we've been using from previous lectures:



D ≡ the size of the disk
B ≡ size of one disk block
M ≡ size of memory

6.3 Data Structures for Graphs

6.3.1 Priority Queue

Think of an item as having a key and a value. The priority queue defines these operations for such an item:

- insert(key, value)

Inserts an item into the queue, and resorts the items in the queue by their keys. (This usually doesn't need a full sort algorithm, as the queue is supposed to be already sorted.)

- deleteMin()

  removes from the queue the item with the lowest key, and returns it.

These operations are often done with a linear algorithm, which completes in $O(N/B)$ time.

A better algorithm can insert or delete in $O(\frac{1}{B}\log_{(M/B)}\frac{N}{B})$ time. Note that the $\frac{1}{B}$ factor here is usually much less than 1.

6.3.2 Buffer Tree

Arge in 1994 created a buffer tree, which is like a B-tree with lazy evaluation. Buffer Tree is an I/O efficient data structure, which is highly used in external memory algorithms Each node has a buffer (also stored on disk). Items being added are noted in the buffer until the buffer is full, THEN it is sorted into the lower tree structure. There are a few other details, and it gives the above access parameters. More information on buffer tree is available here in the following links:

`http://www.madalgo.au.dk/tpie`

`http://www.mpi-inf.mpg.de/~sanders/programs/spq/`

This last URL is Peter Sanders' implementation of a buffer tree. Incidentally, it also includes a priority queue, which is referred to later in this lecture. These implementations are highly sophisticated and incorporate a number of other features, so don't be alarmed if they initially appear different from this lecture.

6.3.3 Graph Structures

List Ranking

An I/O efficient list ranking algorithm can be used to obtain solutions to wide range of problems on simple graphs such as trees. Its also an example to demonstrate and understand how surprisingly difficult and complicated even a simple algorithm can become, once random memory is utilized.

Input : A linked list (which is the simplest form of graph) of size N elements

v1 $\rightarrow$ v2 $\rightarrow$ v3

Output: The distance of each node from the beginning of the list. We need to label the nodes in the linked list in the order of their distance from the head node.

Version 1: This is the list ranking algorithm in the typical RAM model.

```
LIST-RANKING(L)
Node start= L.head;
int curr= 0;
while(start.next!=null)

start.rank=curr++;
start= start.next;

return(L);
```

But in massive graphs,not all nodes at once can fit in main memory. We need to load the node into memory. This linked list comes form the randomly ordered list of blocks from the input device, though block order is not necessarily linked list order. We need to put this into the correct order.

Why is the naive list ranking algorithm inefficient?

Take B=2 and M=4,

We have each vertex in a different block. To access the head of the list, the first block has to be loaded into internal memory. The second vertex can only be accessed after loading the second block. To load the third vertex, i.e, to load the third block, one block needs to be unloaded from main memory due to space constraints. So, overall, we need to spend one I/0 per vertex since each vertex resides a separate block, loading a vertex implies loading a separate block. So we need an efficient algorithm to deal with external memory.

v1,p1 v2,p2 v3,p3

Goal: assign each node v a rank number, which is the serial number of the node from the beginning of the list, i.e. the fist node is 1, next is 2, and so forth.

Then we can sort in $O(N)$ operations.

Independent Set

This is another algorithm. It also has a linked list input.

The output is a set I of size $N/4$ (or some consistent fraction). So no two v, v' are adjacent in the independent set. So where v points to v' in the graph, it makes several scans over the data set, putting something in a priority queeue every time.

scan 1: (on un-ordered nodes)
 assign each item v a 0 or a 1 at random. Call this value $t(v)$.
 Insert the item into a priority queue, where the key is a pointer to the successor note plus $t(v)$.

scan 2: call the priority queue's deleteMin(). If $t = 0$ and $t(v) = 1$, then put v in I.

In other words, put the item in I if the node pointing to it (i.e. it's predecessor) got the random number 0 and this node got the random number 1.

If random chance means our independent set I has $\ll 25\%$ of the elements, then we rerun scan 1 with new random values and append the values thus aquired into set I.

How to use the Independent Set to do list ranking

Goal: Produce a list ranking on an input list I.

Note: list ranking (L'): $L' = L - I$

http://web.cs.dal.ca/ nzeh/Teaching/6104/Notes/zeh02.pdf, page 5.

- Set $r(v) = 1$ for all $v \in L$. Where $r(v)$ is the rank of node v, i.e. the number of nodes that this node feeds.

  I assume nodes are linked like this:

$$\begin{tikzcd}
\boxed{v}_{L-I} \arrow[r] \arrow[rr, bend left=60, "L'"] & \boxed{a}_{I} \arrow[r] & \boxed{s}_{L-I}
\end{tikzcd}$$

  Do a single pass or merge with a later part.

Initialize: Define three priority queues, PQ1, PQ2, PQ3.

Scan 1: if $v \in \{L - I\}$ then
 set $successorL'(v) = successor_L(v)$.
 Insert $PQ1$, key= $succ_L(v)$

```
Scan 2:  If $a \in I$ then
           $v \leftarrow deleteMin(PQ1)$.
           Insert $PQ2$, key=$v$, $successor(a)$.
           Insert $PQ3$, key=$a$.
Scan 3:  if $v \in L - I$, then
           if deleteMin( $PQ2$ ) == v with pointer = s
             set successor(v)= s
           if deleteMin( $PQ3$ ) == v
             set r(v) += 1 (i.e. skip over one node)
           S= deleteMin( $PQ2$ ).
           Push $v$ onto $L'(queue)$.
         Listranking(L') now all nodes in L' have corrrect rank r(v)
Scan 4:  if (successor(v in L) != sucessor(v in L') )
           put r(v) in PQ4 at successor(v in L)
Scan 5:  if $v$ in $I$
           r= deleteMin Note: from which PQ?    r(v)= r+1
```

The above steps needs to be run recursively. In depth details and analysis of List Ranking algorithm are available here `http://web.cs.dal.ca/~nzeh/Teaching/6104/Notes/zeh02.pdf`, page 5.

6.4 Topological Ordering

Say we have a tree.

```
          c
        /   \
       /     \
     c1       c2
    /  \     /  \
   g1  g2   g3  g4
```

Topological Ordering: If node v comes after node v', then v is not an ancestor of v'.

This is stored a 1 node and its links. We do this by using a similar algorithm (i.e. similar to the algorithms we just discussed). Once we have a topological order, it is easier to do some kinds of things.

This assumes we have a DAG, namely a Directed Acyclic Graph.

Topological Ordering takes several passes which we need to minimize. But once that is done, many other operations take 1 pass only. Much of this assumes a DAG, or a planar graph, or similar.

Note that in a general graph, some nodes may have HIGH rank. (Examples: a phone switch pointing to 10,000 subscribers, or several million folks following a celebrity on Twitter). Such nodes must write their edge list onto multiple disk blocks, while other nodes can write their edge list to one block and still share the block with other nodes.