A simple PageRank calculation tool

Chammika Mannakkara
National Institute of Informatics, Tokyo

December 13, 2012

1 pagerank Tool

This tool **pagerank**, is written as per the specification given in Programming Test section #2 pagerank. The program is written in generic C++ using STL. It support the two modes of operation as described in the requirements; the check mode, and run mode with appropriate parameters. There are 3 more optional parameters as follows:

```
chammika@lnotes9:[2]$ ./bin/pagerank
Usage:
pagerank <network_file> check [-l <log_level>] [-g growth_rate]
OR
pagerank <network_file> run <decay_factor> <iterations>
         [-e <epsilon>] [-l <log_level>] [-g growth_rate]
```

The option *-l log_level* is a number between 1 to 3 (default 2) which specify the verbosity of the output of the tool. The *-g growth_rate* is an integer which specify the rate at which the internal data structures are grown to accommodate the network being read. This parameter can improve the reading phase of network and should be considered to very large network computations. The option *-e epsilon* is a floating point number ϵ which represent the required accuracy of the PageRank computation. If this parameter is specified power iteration terminates if the accuracy is archived ie. $|PR(t+1) - PR(t)| < \epsilon$ or the calculation is performed specified number of iterations, which ever comes first.

2 Compiling & Running

The Makefile provided builds the tool for either release, debug or profiling. The binary is created in the
bin directory. The compilation is tested on a system *gcc version 4.4.5 (Debian 4.4.5-8)*

```
chammika@lnotes9:[5]$ make # release build
chammika@lnotes9:[6]$ make clean; make debug # to switch to debug build
```

Following test runs of the tool are using sample test cases found in the *test* directory.

check mode: To run the check on the Fig 13.(b) pp. 32 of A. Arasu et. al.

```
chammika@lnotes9:[11]$ ./bin/pagerank ./test/paper_ex_b check
Reading Network...
Network Reading complete.
Number of Nodes = 5
Number of Edges = 7
Finding rank leaks...
No PageRank leaks were found in the network
Finding rank sinks...
Sinks : there are 1 sink group(s)
Sink Group #0
[3]4
[4]5
```

The output shows the Nodes 4 and 5 are indicated as a rank sink. The output format of the node is *[unique_ID] Node*, the *[unique_ID]* is an internal ID used by the network to represent node. Node is usually a URL instead of a number on node of this example.

run mode: Following commands compute the PageRanks for the Fig 13. (a) and (b)

```
chammika@lnotes9:[19]$ ./bin/pagerank ./test/paper_ex_a run 1 20
Reading Network...
Network Reading complete.
Number of Nodes = 5
Number of Edges = 8
...
PageRank computation complete.
PageRanks :
0.285608 1
0.285962 2
0.142651 3
0.142957 4
0.142822 5

chammika@lnotes9:[20]$ ./bin/pagerank ./test/paper_ex_b run 0.8 20
Reading Network...
Network Reading complete.
Number of Nodes = 5
Number of Edges = 7
...
```

```
PageRank computation complete.
PageRanks :
0.142094 1
0.153697 2
0.101491 3
0.312295 4
0.290423 5
```

The following commands specify the accuracy to 0.001 and change the verbosity level to lower (-l 1) and higher (-l 3) respectively, to control the output [1].

```
chammika@lnotes9:[27]$ ./bin/pagerank ./test/paper_ex_a run 1 20 -e 0.001 -l 1
chammika@lnotes9:[24]$ ./bin/pagerank ./test/paper_ex_b run 0.8 50 -e 0.001 -l 3
```

It can be noted that power iteration converged to the given accuracy after 20 iterations and the loop terminated without performing to upto 50 iterations.

Duplicate links & self-loops: This tool can detect the duplicate links and self-loops at the time of reading the network from file. It simply issues a message (at -l 3) and ignore such edges.

Nodes with no incoming links: There is no special treatmet for nodes with no incoming links. Unlike rank leaks, they do not leak out the ranks. The only problem it creates is such node(s) make the reset of the network a rank sink! This incurs a performance penalty in the rank sink finding algorithm which run much faster if most of the network does not belong to a rank sink.

Handling rank leaks in PageRank computation: The rank leaks causes $\sum PageRanks < 1.0$ thus needed to be fixed to before computing PageRank. Rank leaks are fixed by adding edges to ALL the nodes pointing to it. This solution is suggested in Arvind A. et al. Searching the Web, pp 33 footnote 8 (Alternative solution). This solution ranks the leak nodes reachable through high PageRank nodes higher compared to leak nodes reachable through lower PageRank nodes.

3 Implementaion

The *pagerank* tool is build on top of a `Network` class which represents a generic network using a directed graph. The `Network` class is build using STL as underlaying containers to represent the adjacency list and other information of the graph. Here is a brief decription of class structure and key methods they provide.

[1] debug build enable some detailed output which is not controlled by the -l option. There are removed in the release build to improve performance

Network class: Represents a interconnections of Node objects in a directed graph. This graph is represented in an adjacency list (forward links from a node to neighbors) and a back links (backward link neighbors to a node). One or more attributes can be attached each edge of the network. But for the purpose of this tool only one float/double attribute is kept. Since the `Network` class is build from scratch for this PageRank computation tool it provides methods only necessary for the tool leaving out many generic functions desirable on a general purpose Network class. The main adjacency is maintained on a STL `vector` (in row's direction) and `map` (in column's direcion) as follows:

```
vector<map <node_id, attr> adj_list_;
```

Accessing in row direction in is constant in time due to `vector` (ie. $O(0)$) and in column direction is $O(log(n))$ due to `map`. Thus accessing this Network in row-¿ column order for entire elements is:

$$O(Nlog(n_{map\_avg})) \tag{1}$$

where N is the number of nodes in the network and n_{map_avg} is the average number of elements in a map given by

$$n_{map\_avg}log(n_{map\_avg}) = \frac{n_1log(n_1) + n_2log(n_2) + ... + n_Nlog(n_N)}{N} \tag{2}$$

n_1, n_2, ... n_N are number of (outbound) neighbors for each node. In other words, if all the rows have the same number of nodes, n_{map_avg} is the number of nodes which give the same computational cost for accessing.

Node class: Represents a Web page of the Network. As for this tool it only contains the URL of the page. This Node class serves as a mapping between the abstract representation of the Network class, which assigns a unique ID for each Node it encounter.

PageRank class: Derived class of a generic Network class which provides facilities to calculate PageRank of the Network. This class also provides methods to analyze a Network for rank leaks and sinks.

PageRank::find_rank_sinks method: Find rank sinks in the network and return a vector of Nodes vectors by reference. The worst case complexity is estimated to be $O(N(N+E))$ where N is number of nodes in the network and E is umber of edges in the network. The Algorithm attempts to classify each node from `UNKNOWN` state to either `SINK` or `NOT_SINK` using Breadth-First-Search (BFS). A node is a `NOT_SINK` if ALL node of the network can be reached from that node. A node is a `SINK` if ALL node of the network can NOT be reached from that node. Based on this, following is the Algorithm to find rank sinks:

```
for node 'n'in all the nodes except (leak nodes or already marked SINK nodes)
  while BFS from 'n' using visited[]
```

```
    cur_node <= next node in BFS search
    if (cur_node is NOT_SINK)
           'n' <= NOT_SINK    becasue any node can be reached here

if (BFS terminated but not ALL nodes reached)
      all visited[] nodes in BFS <= SINK

sink group <= { all visited[] nodes in BFS }
merge sink group (if possible) or create new sink group
```

PageRank::calculate_PageRanks method: Computes PageRanks on the network. Since the computation uses the transposed matrix of Network class Complexity (of core alogrithm) is $O(Nlog(n_{map\_avg}))$ according to equation 1. Algorithm : Steps

```
1. Fix leak nodes - by adding edges to ALL the nodes pointing to a leak node
2. Normalize the adjacency matrix - by transforming 1 => d* 1/#Neighbors
3. Transpose the normalized adjacency matrix
4. Initialize ranks of t0, and compute the constant term of PageRank ie. (1-d)/N
5. Perform power iteration (core algorithm)
6. Check for convergence (if given)
```

4 Scalability

In order to scale this system, matrix multiplication can be delegated to multiple processing nodes. Ranges of rows can be assigned to different processing nodes to compute in parallel within an iteration of calculation. Even on a single machine architecture this can be done in a limited way by assigning threads of computation using `pthread` with explicit thread creation or using a library like Intel Threading Building Blocks (TBB) to archive parallelism in the core computation block.

When the order of the problem is in billions of nodes, spread across multiple machines, special data structures/node managers are necessary to keep the coherence of the data among the machines. And the computation complexity depends on the architecture of those systems.

On a conceptual level calculation of PageRank may be divided sub networks (say within a country or topic) which can calcualte PageRanks within the sub network ignoring outbound links. Once a sub network finished calculation, the propagation can happen on outbound links. And another round of PageRank calculation can take place within the sub network. Since PageRank is a idea based on a relative value such method might also work in my opinion.