UNIVERSITÀ DEGLI STUDI DI TRENTO

Department of Information Engineering and Computer Science

Bachelor's Degree in
Computer Science

Final report

All-sources personalized pagerank

Supervisor
Prof. Alberto Montresor

Co-supervisor
Dr. Alessio Guerrieri

Graduating
Jacopo Gobbi

Academic year 2016/2017

Acknowledgements

I thank my family for their patience in everyday life, Dr. Alessio Guerrieri for the opportunity to work on such an interesting subject, and Prof. Alberto Montresor for his guidance and for enduring my grammar.

Contents

1 Summary

The Pagerank algorithm from Lawrence Page and Sergey Brin [7] [8] had its first intended use as a way to rank web pages in the web graph by simulating the behaviour of the user, which follows links from one page to another by clicking on them, with a small chance to *teleport* back to the starting page.
This teleporting behaviour is modeled using a *teleport set*, that, as the name suggests, contains a set of web pages that are potential destinations of teleports, in its classic form this set is the same as the set of vertices of the graph.
The use of this model is not just tied to graphs representing web pages, and it is reasonable to say that it can be used to gain insight on any kind of data that can be mapped to a graph.
Personalized Pagerank or Topic-Sensitive Pagerank [12] in its non approximated form is a particular case of Pagerank where the *random restart* of the stationary distribution is arbitrary; in the teleport set there might be a single node or a subset of the graph nodes.
Personalized examination of data can lead to insights of great value; it usually finds a lot of use in social networks [18] [11] or in web ranking [14] [6], but it has been used for very different problems as well, such as graph partitioning [1], name solving [20] or clustering [4]. Very often, these kind of queries are on-line due to the initial cost that computing results for each node would bear.
Computing the exact Personalized Pagerank for every node in large graphs can be prohibitive and it is often not that useful if we are only interested in the top-K scoring nodes.
We have scrutinized different algorithms that provide Personalized Pagerank for single nodes, analyzing in depth the two that were the most promising as a base for a solution giving results for all nodes in an efficient way.
Two algorithms are proposed, the first is named GRank and was provided by Dr. Guerrieri, while a second version of the algorithm described in Chapter 4 was elaborated by the candidate. The other proposed algorithm was put forward by the candidate, it is based on random walks done in the like of one of the analyzed algorithms - Monte Carlo Complete Path [5] -, and because of that it is named MCCompletePathV2.
Benchmarking the algorithms has been done by taking three parameters in consideration: the *Jaccard similarity* [13], *the Kendall correlation coefficient* [15] and the running time; the data sets used are from the Stanford Large Network Dataset Collection [26] and the The Koblenz Network Collection [24] and are described in Chapter 5.
The Jaccard similarity was used to compare how similar the results provided by two algorithms - Pagerank and the benchmarked algorithm - were, the Kendall correlation was instead used to check how similarly the two algorithms would order the nodes in the results in terms of importance; a more detailed explanation of these two parameters can be found in Chapter 5.
All the work was done while in what the Department of Information Engineering and Computer Science calls a "internal internship", a "period of training designed to integrate knowledge acquired during the university studies and to help students with their future career choices, but, unlike the external internship (stage), it is carried out within the Department or within an entity cooperating with the Department". In this case, the cooperating entity was "SpazioDati" [27], a company specialized in leveraging data, which is where Dr. Guerrieri currently works.

2 Background theory and the problem

To better understand why algorithms approximating what would be the exact results of Personalized Pagerank are needed it may be worthwhile to spend a few words on Pagerank.
The user behaviour simulated by this algorithm translates to a random walk on the graph where being at node A we have a chance to follow one of its outlinks, given by the *damping factor*, or to instantly move to a node part of the teleport set $(1 - dampingfactor)$; nodes which are traversed frequently will have a high Pagerank score, while the ones rarely visited will obviously be left behind.
Assuming the teleport set is composed of the entirety of the graph nodes, the Pagerank of a node A can be formally described as:

$$PR(A) = \beta(\frac{PR(N_1)}{outlinks(N_1)} + \cdots + \frac{PR(N_n)}{outlinks(N_n)}) + \frac{(1-\beta)}{|V|} \tag{2.1}$$

where:

- $PR(node)$ is the Pagerank of a node
- $N_1 \dots N_n$ are the n nodes having an edge towards A
- $outlinks(node)$ is the number of outgoing links of a node
- β is the damping factor that can be between 0 and 1 both inclusive
- $|V|$ is the number of nodes in the graph

The teleport set is altered for Personalized Pagerank, making it so that only the nodes for which we are calculating the Personalized Pagerank are part of it. The rank of each node can then be computed with the following equation:

$$PR(A) = \beta(\frac{PR(N_1)}{outlinks(N_1)} + \cdots + \frac{PR(N_n)}{outlinks(N_n)}) + \begin{cases} \frac{(1-\beta)}{|teleportset|} \; if \; A \; \in teleport \; set \\ 0 \; otherwise \end{cases} \tag{2.2}$$

If the Pagerank score roughly translates into how important a node is from a global perspective, the Personalized Pagerank score offers the point of view of a node or a set of nodes, and can be seen as either an importance or a similarity.
Analytically solving the Pagerank of each node would mean solving a system of $|V|$ equations which for large graphs, like the web graph, would end up being too computationally expensive.
This leads to an iterative, and approximative, implementation of the algorithm where after initializing the Pagerank of every element of the teleport set to 1, for I iterations the Pagerank of each node is synchronously computed as in Equation 2.1, with $O(|E|)$ complexity for each iteration.
In conclusion, to have the Personalized *Pagerank vector* of each node we would have to run Pagerank $|V|$ times, each time with a single node as the teleportation set, resulting in a $O(|V^3|)$ complexity, which is not feasible for even modestly large graphs.
It must also be pointed out that most real use cases for Personalized Pagerank are not interested in the whole Personalized Pagerank vector but only on a subset of the highest scoring nodes; this is due to the fact that storing the resulting vector for each node would incur in a $O(|V|^2)$ space complexity which might not be justifiable from a pragmatic point of view, since the lowest scoring nodes will not hold much significance.

3 Analyzed algorithms

We analyze two in-memory algorithms for top-K Personalized Pagerank that require no beforehand offline computations.
Both [5] and [10] have been benchmarked to see if there was any notable gain in exploiting information between nodes instead of having each Personalized Pagerank result computed separately, as expected there is as it can be seen in Chapter 5.
We have used these two algorithms, along with others ([21], [19], [9]) which have been considered but have not been made part of the project, to try and build an all-sources algorithm on top of them.
For our second proposal this is the case, where the random walks have been done in the same way as *Monte Carlo Complete Path* (Section 3.1); the first algorithm is instead built entirely considering an all-sources perspective.
These two analyzed algorithms have been choosen not only because they had promising results: they also spanned very different approaches, going from probabilistic placement schemes to tolerance based iterations.

3.1 Monte Carlo Complete Path

Monte Carlo Complete Path from "Quick Detection of Top-k Personalized Pagerank Lists" [5] by K. Avrachenkov, N. Litvak, D. Nemirovsky, E. Smirnova and M. Sokol, is a probabilistic algorithm based on *Monte Carlo simulations* of random walks traversing the graph; the approach of their research is mostly mathematical and they provide proof that their placement scheme is a correct estimator of the Personalized Pagerank vector of a node.
The process is quite simple: to compute the top-K basket of nodes for a starting node S, we simulate m random walks starting from S, incrementing the value of each visited node by one. A random walk has a β chance of continuing onto a randomly choosen direct successor of the current node and a $(1 - \beta)$ probability to end, thus modeling the random walk with damping factor from the original Pagerank.
The K most important nodes for S are then easily obtained by partially ordering the visited nodes by value descending.
The resulting complexity for a single node is $O(m + |V|)$, where the first addend is due to the random walks and the second is given by the need to retrieve the top-K nodes from all visited nodes.
It is also worth noting that this algorithm performs mostly increments, besides it being a very fast operation, it helps in avoiding rounding errors that might be encountered when instead dealing with scores that can reach very low levels after numerous iterations involving products or divisions.
The authors of this algorithm prefer to consider its result as a top-K basket, but our experiments show that the Jaccard value between the approximated top-K basket and the real top-K basket goes almost hand in hand with the Kendall coefficient; usually differing by a few percentile points, as it can be seen in Figure 3.1 and Figure 3.2. This remains true for all the algorithms we have analyzed or proposed.
Intuitively, this can be explained by the fact that a top-K basket of nodes can be seen as a descending partial order around the K-th element; if the Jaccard value is low it means that we are failing at partially sorting the elements, which implicates that we can not expect to correctly sort them and to obtain a good Kendall value.

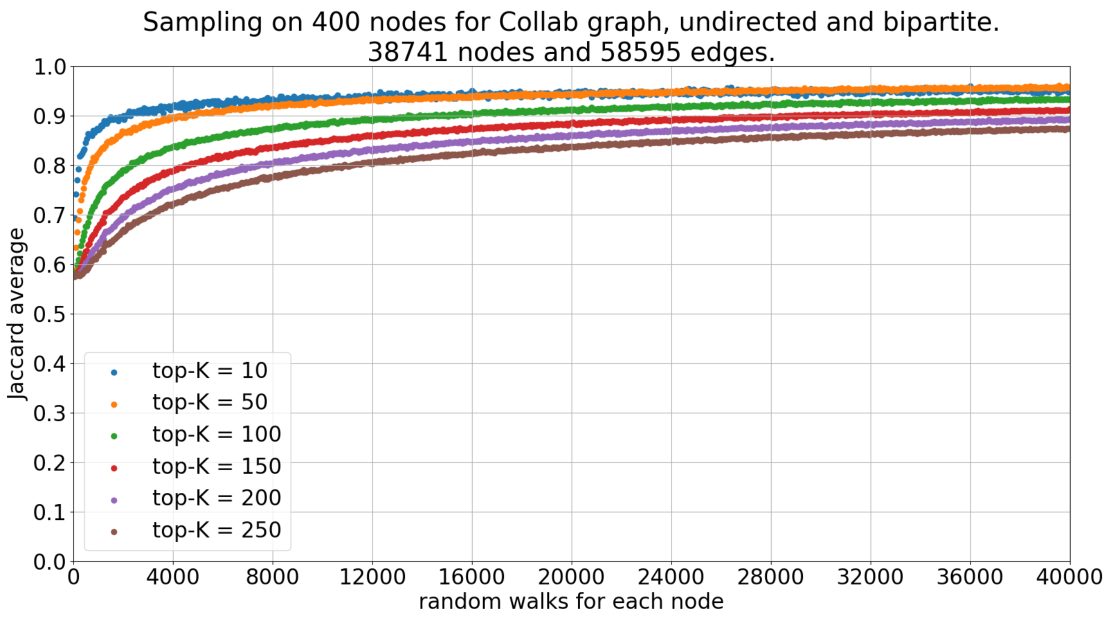

Figure 3.1: Diminishing return on the number of walks for Jaccard.

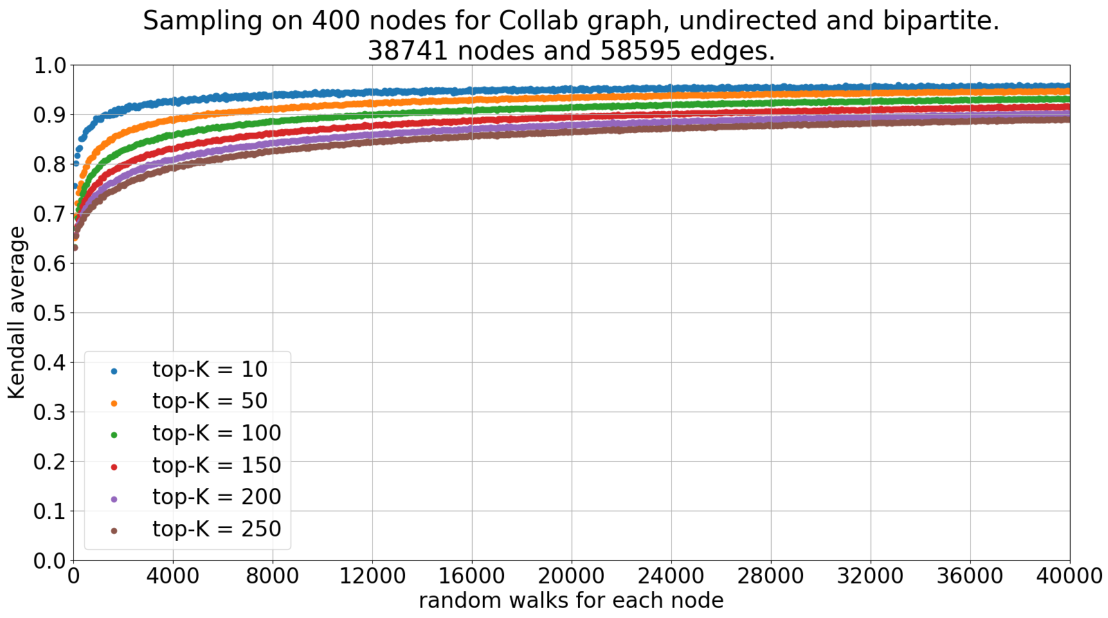

Figure 3.2: Diminishing return on the number of walks for Kendall.

Monte Carlo Complete path is simple to implement and can get to good results fairly fast, but we have met a steep diminishing return on the effectiveness of the random walks when trying to reach for high Jaccard values; this can be seen graphically from Figure 3.1 and Figure 3.2 by noting how the plot swiftly tends to become flat.
Since the amount of random walks directly translates into running time, it might be worth it to consider m as a score that recursively splits among children, and then to run the simulations starting from the nodes where the residual of m goes beyond a certain threshold, as suggested in [17].
We have also found that using a rotating index to choose the next node of the walk, instead of randomly picking a direct successor, can improve the precision of the algorithm by up to 5%. This works especially well when the number of simulations run is low and inequalities might arise due to random noise.

3.2 Boundary restricted personal Pagerank

Boundary restricted personal Pagerank is the second analyzed algorithm, its authors are David Gleich and Marzia Polito and they propose it in “Approximating Personalized Pagerank with Minimal Use of Web Graph Data” [10].
It is based on two sets of nodes, named “*active*” and “*frontier*”; nodes part of the former are taken in consideration when performing an iteration of Pagerank, and their score is distributed among their children, while the outgoing edges of nodes not part of the active set are not considered, as if not existing.
Nodes that have a score but are not part of the active set concur to form the frontier and can be *promoted* from the frontier to the active set thanks to a policy based on a *threshold* value, better explained in the following pseudocode.
The algorithm computes its results one node at a time and terminates after convergence based on the norm of the difference between the current vector and the one from the previous iteration.

Boundary restricted personal Pagerank

```
$x(s) = 1$        // init the Pagerank vector by giving a score to the starting node
$A.add(s)$        // s starts as the only element of A
**repeat**
    $y \leftarrow pagerank\_iteration(A,\ x)$        // returns a Pagerank vector
    $\omega \leftarrow 1 - \|y\|_1$
    $y(s) = y(s) + \omega$        // add back score lost due to damping
    **while** $y(F) > \kappa$ **do**
        $v \leftarrow n \in F \mid y(n) >= y(n') \ \forall \ n' \in F$
        $F.remove(v)$
        $A.add(v)$
    **end while**
    $\delta = \|y - x\|_1$
    $x = y$
**until** $\delta <$ *stopping tolerance*
```

Where:

- s is the starting node for which the Personalized Pagerank vector is being computed
- A is the active set and F the frontier
- $y(F)$ is the total score of F, the sum of the score of all nodes part of F
- κ is the *frontier threshold*

Due to the presence of a Pagerank iteration, even if in a limited form, the complexity for computing a single Personalized Pagerank is $O(|E| + |V| \log |V|)$, where the $|V| \log |V|$ arises from the need to sort the nodes of the frontier to move the highest scoring nodes to the active set.
Despite the complexity being suspiciously similar to Pagerank, in a real case scenario this algorithm performs way better thanks to its two parameters.
The frontier threshold has also the quality to be almost linear in its relation with the Jaccard and Kendall coefficients, as it can be seen in Figure 3.3 and Figure 3.4; this allows for easier predictability of the results and the running time, compared to Monte Carlo Complete Path.

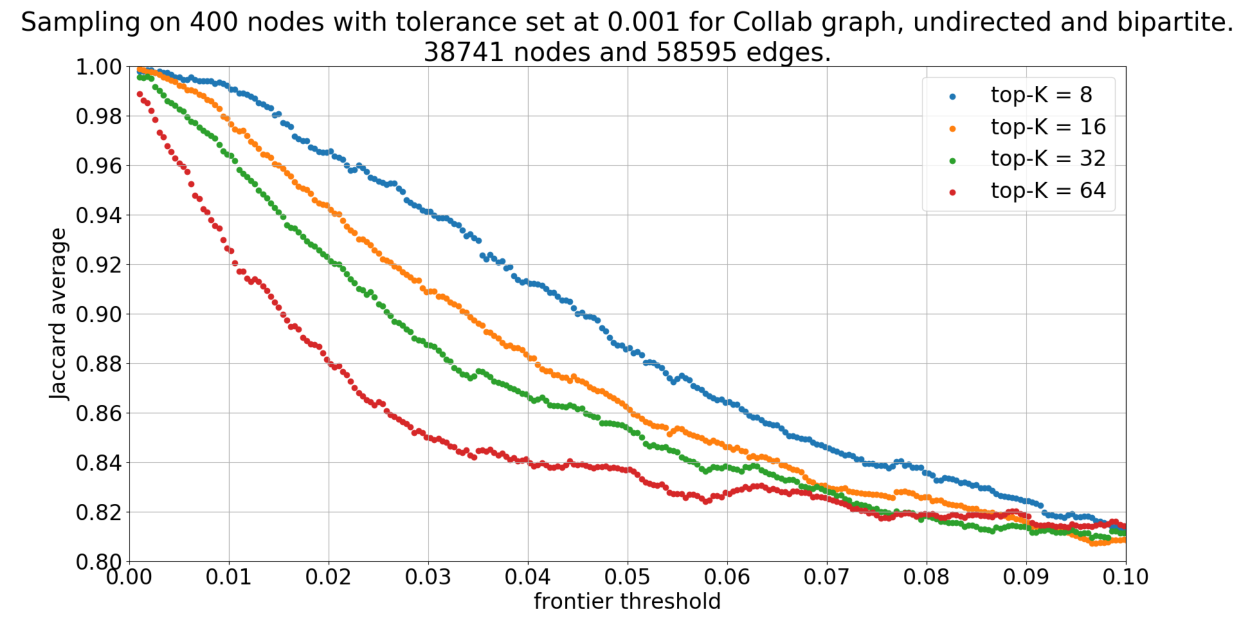

Figure 3.3: Relation between the frontier threshold and the Jaccard coefficient.

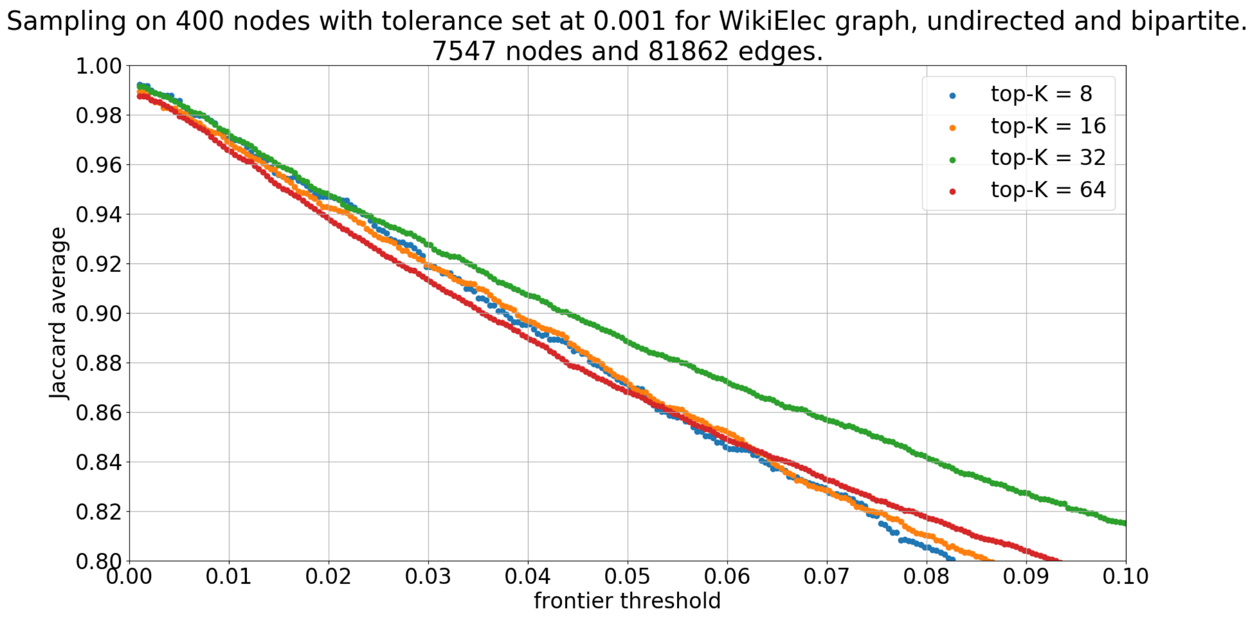

Figure 3.4: Relation between the frontier threshold and the Jaccard coefficient.

4 Our proposal

We propose two different solutions which have shown good performance in different cases, these two algorithms aim to provide top-K scoring lists for all nodes in the graph.
We have tried to mantain both algorithms easy to parallelize like [5] and [10] , where each node is run as a separate instance of the algorithm.

4.1 GRank

The standard solution for computing the Personalized Pagerank vector of a node is based on Equation 4.1:

$$PR_s(i) = \beta \sum_{j \in P_i} \frac{PR_s(j)}{|S_j|} + \begin{cases} (1-\beta) \; if \; s \; = \; i \\ 0 \; otherwise \end{cases} \tag{4.1}$$

$PR_s(i)$ stands for the Personalized Pagerank score of i for source node s; P_i is the set of direct predecessors of i, while S_i would be the set of direct successors.

An equivalent equation that makes $PR_s(i)$ dependant on $PR_j(i)$, which translates as "the Pagerank of i for source node s depends on the Pagerank of i for source node j" and that inverts the score flow from "father to children" to "children to father" is instead Equation 4.2:

$$PR_s(i) = \beta \sum_{j \in S_s} \frac{PR_j(i)}{|S_s|} + \begin{cases} (1-\beta) \; if \; s \; = \; i \\ 0 \; otherwise \end{cases} \tag{4.2}$$

GRank iteratively uses Equation 4.2 to compute the Personalized Pagerank vectors of all nodes, but with one condition: nodes i for which $PR_s(i)$ is not part of the *top-L* scoring nodes for source node s are removed from the vector of s; this gives great benefits in terms of memory and speed.
Formally, the algorithm is based on 2 parameters:

- L, or top-L, which is the number of highest scoring nodes to keep in the Personalized Pagerank vector of each node. Distinction between K and L has been made because having L greater than K has proved to be very effective in leading to good approximations of the top-K.
- A *tolerance* which dictates when to stop iterating.
- Furthermore, a *limit* on the number of iterations can be set to have more control on the running time.

Intuitively, at every iteration each node combines the top-L of its children and then keeps the top-L scoring nodes resulting from this merge; convergence is decided based on the difference between the Personalized Pagerank vector (containing L elements) of the current iteration and the previous one.
More specifically we have experimented with two forms of convergence:

- Not considering the algorithm converged until the difference between the new and the old Pagerank vector of every node is below the tolerance.
- Deciding convergence on a per node basis: a node stops reading from its children and computing a new vector if the difference between its new and old vector is below the tolerance, and the algorithm is considered converged when all nodes are. The idea behind this is that a graph might have areas where approximating Pagerank is easier, thus requiring less iterations, and more demanding areas needing more calculations.

While traversing the space of parameters to benchmark the algorithm we noticed that the minimum time to reach the same Jaccard values is slightly higher for the second policy, while it is lower when the two versions are given the same, somewhat overestimated parameters.
Since traversing the space of parameters and pinning them perfectly for the input is not feasible in a real case scenario we believe that using the second policy while marginally overestimating the parameters be a reasonable approach.

GRank

```
$PR \leftarrow \{\}$        // PR is a dictionary of dictionaries
for all $i \in V$ do
    $PR[i][i] \leftarrow 1.0$        // init every source as the highest score for its own vector
end for
repeat
    $PR' \leftarrow \{\}$
    for all $i \in V$ do
        $PR'[i][i] = 1.0 - \beta$
        for all $j \in S_i$ do
            for all $k \in PR[j]$ do
                $PR'[i][k] = PR'[i][k] + \beta * \frac{PR[j][k]}{|S_i|}$        // $PR'[i][k] = 0$ if it is not already mapped
            end for
        end for
        $PR'[i] \leftarrow keepTop(L, PR'[i])$
    end for
    $PR \leftarrow PR'$
until convergence
```

The resulting complexity is $O(|E| * L * I)$ where I is the number of iterations; the algorithm can easily be parallelized thanks to the fact that the top-L maps of the nodes are built synchronously between iterations.
Additionally a minor modification allows to halve the running time, as explained in Section 4.1.1.

4.1.1 GRank with partitions

First, assume that the input graph is bipartite; due to how the algorithm works each *partition* will only read from the other one, creating two flows of informations that never meet, as seen in Figure 4.1. Circles represent the two partitions and iterations proceed vertically: an arrow from one partition to another means that the nodes in the first partition are reading and combining the top-L vectors of their direct successors that are part of the pointed partition.
By doing calculations for only one partition during each iteration, as shown in the right part of Figure 4.1, it is possible to save time.

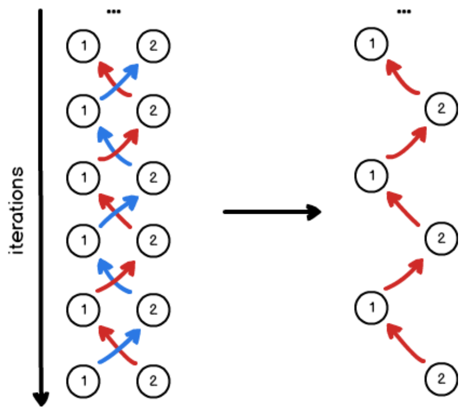

Figure 4.1: GRank flow of information for bipartite graphs.

It must be noted that the obtained results would be one iteration behind for one partition: if we were to retrieve them as in Figure 4.1 we would have partition 2 results be from iteration t, while the ones from 1 to be from $t-1$. This does not change much in terms of outcome, especially when the iterations to be done are many.
If the graph is not bipartite the two flows mix and we would encounter the following problem: during iteration t doing calculations for partition 1 we would be missing data from 1 that would be resulting from iteration $t-1$. This is due to the fact that each partition does not read from the other exclusively,

but needs to read from itself aswell.
This can be solved by reading data resulting from iteration $t-2$ when a partition needs to read from itself, and from $t-1$ while reading from nodes of the other partition as shown in Figure 4.2.
Pragmatically this does not change the algorithm at all compared to the bipartite case: we just need to carry over to iteration $t+1$ the newly obtained vectors, related to the current partition, and the ones computed in $t-1$, belonging to the partition that was ignored during this round.

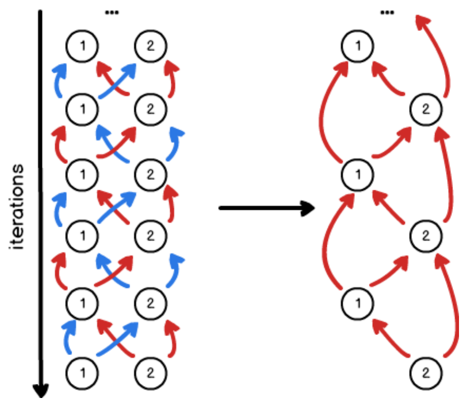

Figure 4.2: GRank flow of information for non-bipartite graphs.

This second version of the algorithm introduces the need to individuate two partitions , a slight change to the initialization phase and the fact that only one partition executes per iteration; resulting in a change in the pseudocode of GRank.
Of course in the occurrence of a non-bipartite graph dividing the nodes in two partitions is an approximation; we have found *2-coloring* to work, different methods are possibile: the lower the intra-edges for each partition the better.

GRank with partitions

```
$A, B \leftarrow find\ two\ partitions$
$PR \leftarrow \{\}$    // PR is a dictionary of dictionaries
**for all** $i \in V$ **do**
    $PR[i][i] \leftarrow 1.0 - \beta$    // init phase
    **for all** $j \in S_i$ **do**
        $PR[i][j] \ = PR[i][j] \ + \ \frac{\beta}{|S_i|}$
    **end for**
**end for**
**repeat**
    $PR' \leftarrow \{\}$
    **for all** $i \in A$ **do**
        $PR'[i][i] = 1 - \beta$
        **for all** $j \in S_i$ **do**
            **for all** $k \in PR[j]$ **do**
                $PR'[i][k] \ = PR'[i][k] \ + \ \beta * \frac{PR[j][k]}{|S_i|}$
            **end for**
        **end for**
        $PR'[i] \leftarrow keepTop(L,\ PR'[i])$
    **end for**
    $swap(A,\ B)$
    **for all** $i \in A$ **do**    // carry on results from previous iteration
        $PR'[i] \leftarrow PR[i]$
    **end for**
    $PR \leftarrow PR'$
**until** convergence
```

As stated above, this change allows to cut the running time almost in half, as shown in Figure 4.3. Calculation speed is not strictly doubled because we traversed the space of parameters to find the minimum time required to reach for a certain Jaccard or Kendall value, often resulting in a niche combination of tolerance, top-L and max iterations that would make the original version have a running time less than twice the one of the partitioned version, but not faster than it.
The upfront cost of finding the partitions should also be considered, especially when aiming for lowish Jaccard or Kendall values that would normally require very few iterations.

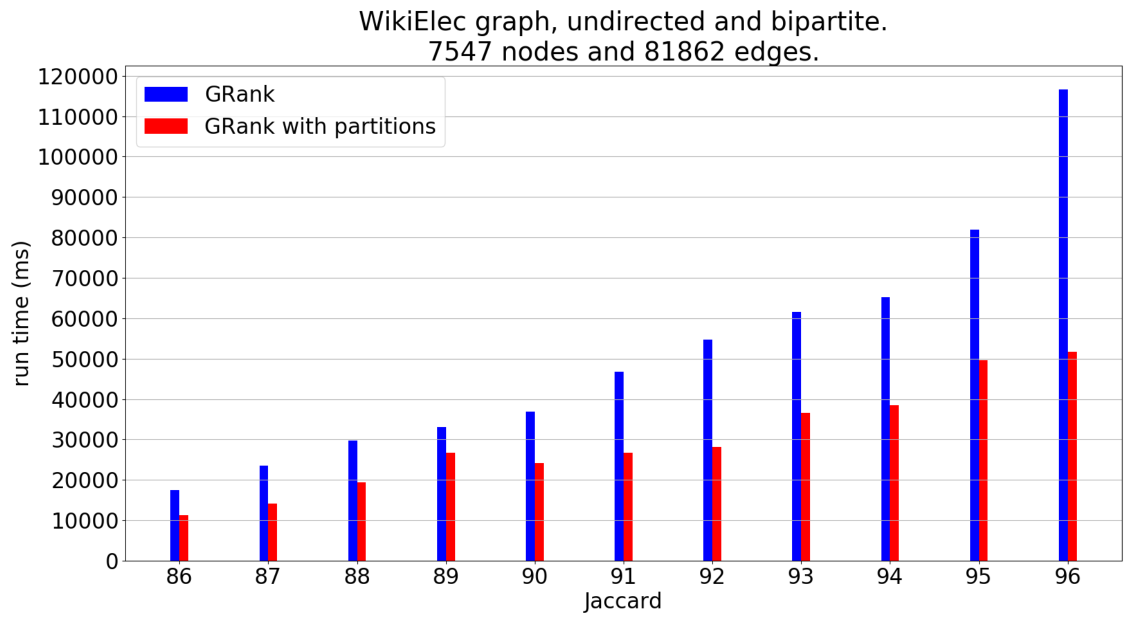

Figure 4.3: Lowered running time using partitions in GRank.

4.1.2 Further improvements

As a possible future improvement we highlight an issue that we could not solve, emerged after analyzing the results in a node per node manner, and that appeared in both versions of GRank.
The issue is that nodes with high degree would be excluded from a top-K they belonged to more frequently than other nodes, making it so that the algorithm tends to require more “heavy” parameters, thus more running time, in graphs with high degree.
This is likely because in a context with many edges a Personalized Pagerank vector might have many nodes for which their score is the result of the summation of scores that have traversed different paths to get to the source node.
Since in GRank each node only keeps track of L scores at a time, the more a node depends on its value being the sum of many addends the more it is likely that some of those addends might be discarded by a node being traversed before they reach another node, for which the combined result would instead be part of its top-L.

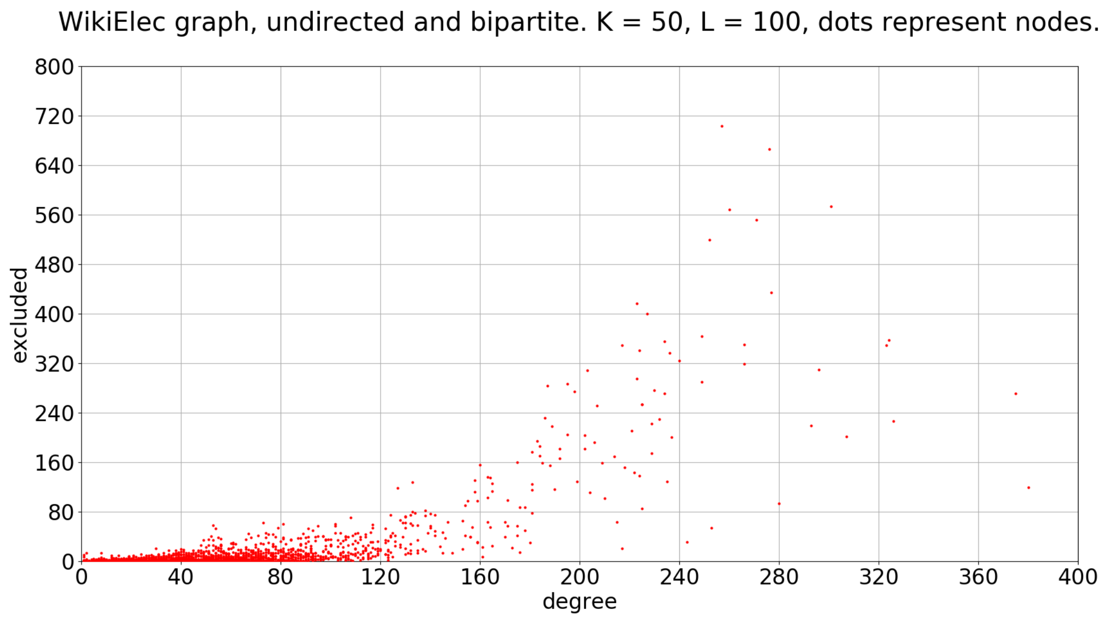

Figure 4.4: Nodes with high degree being more frequently excluded from top-L they belonged to. The "excluded" axis stands for how many times a node should have been part of the top-K of another node but it was not.

After attempting different changes to the algorithm we found that considering the difference between the top-L and top-K (remember that $L >= K$) a *budget* to be distributed instead of a fixed value for each node to work in flattening the curve in Figure 4.4.
We ultimately choose not to include this change in the algorithm for the following reason: distributing a budget of $|V| * (L - K)$ proportionally between nodes based on their degree, the higher the better, would naturally mean that nodes with higher degree would contain more than L elements.
This implies that a majority of nodes, the direct predecessors of the high budgeted ones, would now read more than L elements, thus increasing the running time.
The increase in running time plus the small overhead for budgeting made adding this change not worth it.

4.2 MCCompletePathV2

The second proposal is a probabilistic algorithm that aims to minimize the number of *Monte Carlo simulations* needed by sharing their results between neighbours and propagating those results in the graph.
It is based on the following concept: doing R *random walks* for each node might not be enough to properly estimate their Personalized Pagerank, but we can see each node as the union of its direct successors, and combine their results.
We would still be doing R walks for each node but their Personalized Pagerank would be based on $S * R$ random walks, where S is the average number of direct successors.
Additionally, to keep the complexity in check we limit the Personalized Pagerank vector length of each node, making this algorithm based on two parameters:

- L, or top-L, as in GRank the Pagerank vector is actually a map of limited size, where elements not contained are considered to have a value of 0. In this algorithm each node has two different maps, one resulting from random walks and the other resulting from the combination of its successors, both affected by this parameter. The walk map can be seen as a temporary result used by the direct predecessors of the node, while the result map is actually the algorithm output for the node. As in GRank, a distinction between L and K has been made because having L greater than K has proved to be very effective in leading to good approximations of the top-K.
- R, number of random walks done for each node in the worst case. The algorithm tries to limit

the number of walking nodes by exploiting the fact that when all successors of a node are done walking or have their results ready those can be combined, saving time that would otherwise be spent on simulations to calculate the Personalized Pagerank vector of the node.

The algorithm is characterized by two phases: first an *order of execution* of the nodes is established to avoid doing walks as much as possible, then the order is traversed and each node combines the maps of its successors to make its own.
To order the nodes a *heuristic* is employed: the aim is to minimize simulations while maximizing the times that a node, having to combine its direct successors maps, will be able to use their *result map*, which was a combination itself, instead of using the one obtained by doing random walks starting from that successor.
The heuristic is quite simple: first, nodes are ordered by indegree descending and outdegree ascending, then this order is traversed and nodes are added to a list constituting the final order.
When a node is added, all its direct predecessors for which all direct successors have already been inserted in the order are considered visited and added themselves, if they were not inserted already. A node added in this way can itself cause its direct predecessors to be added, recursively.
The second and final phase elaborates every node in the order provided by the heuristic, combining the node direct successors maps, eventually simulating R random walks for a successor if it has not been elaborated already and it has no data resulting from random walks.
Random walks are done in almost the same fashion as the ones in the Monte Carlo Complete Path algorithm described in Chaper 3, where the value of each node encountered during the walk is incremented, with a difference: when visiting a new node that would cause the map size to exceed L the node is ignored and the walk goes on; this is key in limiting the map size, hence keeping the algorithm complexity in check.
Furthermore, since the number of walks required tends to not be large, to diminish disparities that might appear due to the random nature of the selection of the successor, we have associated to each node an index, a simple increment with modulo, which dictates the next successor to move on to during the walk.

MCCompletePathV2

```
$walks \leftarrow \{\}$    // dictionary of dictionaries, will contain results from walks
$scores \leftarrow \{\}$    // dictionary of dictionaries, will contain the final results
$order \leftarrow heuristic()$    // Use heuristic to order the nodes
for all $i$ in $order$ do    // traverse the order and calculate results for each node
    $result \leftarrow \{\}$
    for all $s \in S_i$ do    // combine the maps of the direct successors
        if $scores[s] \neq NULL$ then    // if the node results are available use those
            $result = result + scores[s];$
        else    // else use data from walks
            if $walks[s] == NULL$ then
                $walks[s] \leftarrow doRandomWalks(s, R)$
                $walks[s] = walks[s] * \frac{1.0}{R}$    // average values by number of walks
            end if
            $result = result + walks[s]$
        end if
    end for
    if $|S_i| > 0$ then
        $result = result * \frac{\beta}{|S_i|}$    // average values by number of direct successors
    end if
    $result[i] = result[i] + 1.0;$    // each node visits itself at least once per walk
    $result \leftarrow keepTop(L, result)$
    $scores[i] \leftarrow result$
end for
```

Note that:

- $result = result * \frac{\beta}{|S_i|}$ means that the score of every mapped element of that map will be multiplied by $\frac{\beta}{|S_i|}$.
- $result = result + walks[s]$ means that every key-value pair in the walks[s] map will concur to increment the value of that key in result by an amount equal to its value in walks[s]. A key that is not already mapped has a default value of 0.

The resulting complexity is $O(|V| \log |V| + |V| * R + |E| * L)$, where $|V| \log |V|$ is due to the ordering, $|V| * R$ is due to the worst case where every node will have to do R random walks, and $|E| * L$ is given by the fact that every node will read a map of up to L size from each one of its direct successors.
This algorithm lends itself to parallelization less than GRank since there is an order of execution to be respected, however the bulk of the work can still be parallelized since the nodes that will have to walk are known after the first phase is done.

5 Experimental results and discussion

This chapter serves as a confrontation between the discussed algorithms, when GRank is referred here it is the partitioned version with convergence policy based on all nodes synchronously converging, unless stated otherwise.
We propose two different algorithms because of them performing differently in diverse scenarios, and thus the inability to tell if one is strictly better than another, however we provide guidelines to help in making a more informed choice.

5.1 Experimental setup

To test the goodness of the results of each algorithm, given a number of samples nodes from a graph, both the outcome from the algorithm and from Pagerank were being computed for each of those nodes, and the two vectors containing the scores of the top-K scoring nodes were then compared using the *Jaccard similarity* and *the Kendall correlation.*
The former to express the difference between the real top-K scoring nodes of a source node and the top-K provided by the approximating algorithm, the latter to measure how similar the ordering between the two is.
The Jaccard index [13] is a statistic that compares the similarity of two given finite sets, and is formally described as the size of the intersection divided by the size of the union:

$$Jaccard(A, B) = \frac{|A \cap B|}{|A \cup B|} = \frac{|A \cap B|}{|A| + |B| - |A \cap B|} \tag{5.1}$$

The Kendall rank correlation coefficient [15], also named *Kendall's tau coefficient*, is instead used to describe how correlated two rankings are; given a set of people characterized by height and weight we could order them by the former or the latter, if the two orders are the same we would obtain a Kendall of 1.0, if they are reversed we would end up with -1.0, and we would get a value of 0.0 if weight and height were independent.
In our case this statistic was used to evaluate how similar the approximation algorithms were ranking the nodes by importance compared to Pagerank, so the two compared variables were the score given by the tested algorithm and the score given by Pagerank.
The coefficient is simply computed as:

$$\tau = \frac{|concordant\ pairs| - |discordant\ pairs|}{|total\ pairs|} = \frac{|concordant\ pairs| - |discordant\ pairs|}{n(n-1)/2} \tag{5.2}$$

This formula does not take into account ties, which are possible for Pagerank scores; to adjust for those we must use a slightly different statistic, called Kendall's τ_b.

$$\tau_b = \frac{|concordant\ pairs|\ -\ |discordant\ pairs|}{\sqrt{(|total\ pairs|\ -\ |pairs\ tied\ in\ first\ variable|)\ (|total\ pairs|\ -\ |pairs\ tied\ in\ second\ variable|)}} \quad (5.3)$$

We found the value of Jaccard and Kendall to move almost conjointly for all algorithms during our experiments, therefore to avoid redundancy there will not be two charts differentiated only by what coefficient they are about.
To compare the algorithms running time each algorithm had to compute results for all nodes in a graph, for those algorithms that compute results one node at a time they were simply applied $|V|$ times, once for each node.
Pagerank has been run with the following parameters:

- 100 iterations
- 0.85 damping factor
- 0.0001 tolerance

Java has been used to implement the algorithms, and the graphs to test them have been provided by the *Stanford Large Network Dataset Collection* [26] and the *The Koblenz Network Collection* [24]; data for different charts might have been generated on different computers, but for the same chart all data has been generated on the same computer.
Many graphs have been used to test the algorithms, here we describe only the ones referred in this document.
The various *Gnutella* graphs represent the peer-to-peer gnutella network in different time periods, nodes are hosts and edges are existing connections between those hosts.
WikiElec is a bipartite graph about Wikipedia adminship election data. Nodes in the first partition represent users that can vote for the adminship of another one, while the second set is made of these potentially future admins; an edge between an elector and a potentially future admin exists if the user gives a positive vote for giving the adminship of some part of Wikipedia to that potentially future admin. Note that the graph was originally directed, but we considered it undirected because SpazioDati mostly use undirected graphs for their Pagerank related services.
Slashdot is a graph representing friend/foe (not distinguished) links between users of the Slashdot forum, a website about technology-related news.
The *Collab* graph was taken from the The Koblenz Network Collection, where it is called "arXiv cond-mat" and described as a "bipartite network that contains authorship links between authors and publications in the arXiv condensed matter section (cond-mat) from 1995 to 1999. An edge represents an authorship connecting an author and a paper.". Internally we renamed it Collab for easier recalling since it is reasonable to say that it is a graph about collaborations on papers.
Another entry taken from The Koblenz collection is the *Edinburgh Associative Thesaurus* ("Eat"), each node represent a word, an edge exists from word A to B if the word B was the response to input A during the interview of a user.

name	source	nodes	edges	directed	bipartite	diameter
WikiElec	Stanford	7547	81862	false	true	9 edges
Slashdot	Stanford	77360	905468	true	false	10 edges
Gnutella30	Stanford	36682	88238	true	false	10 edges
Gnutella24	Stanford	26518	65369	true	false	10 edges
Collab	Koblenz	38741	58595	false	true	36 edges
Eat	Koblenz	23132	312310	true	false	6 edges

5.2 The results

Given the fact that GRank and MCCompletePathV2 share information between nodes, while computing results in Boundary restricted Pagerank and Monte Carlo Complete Path means running a different instance of the algorithm for each node, we naturally expected the former to perform better. In all our experiments we have found both Monte Carlo Complete Path and Boundary restricted Personalized Pagerank to be much slower than the two proposed solutions, and Monte Carlo Complete Path to be usually faster than Boundary.
Due to the difference in run times between the discussed algorithms not all charts will contain information about every one of them.
As for running parameters, we found that a tolerance between 0.0001 and 0.04, a top-L ranging between 2 to 10 times the desired top-K and a cap to the number of iterations set between 20 and 50 to be reasonable for reaching Jaccards spanning from about 0.90 to 0.98 with GRank.
Parameters for MCCompletePathV2 are quite more volatile, while having a top-L 5 or 10 times greater than the top-K and random walks R for each node set between 200 and 1000 can generally reach good results, some graphs might require a top-L 20 times the top-K, and 10000 walks or more.
For both algorithms we strongly discourage the use of a top-L equal to top-K in any case, since as it can be seen in Figure 5.1 it significantly impairs results.

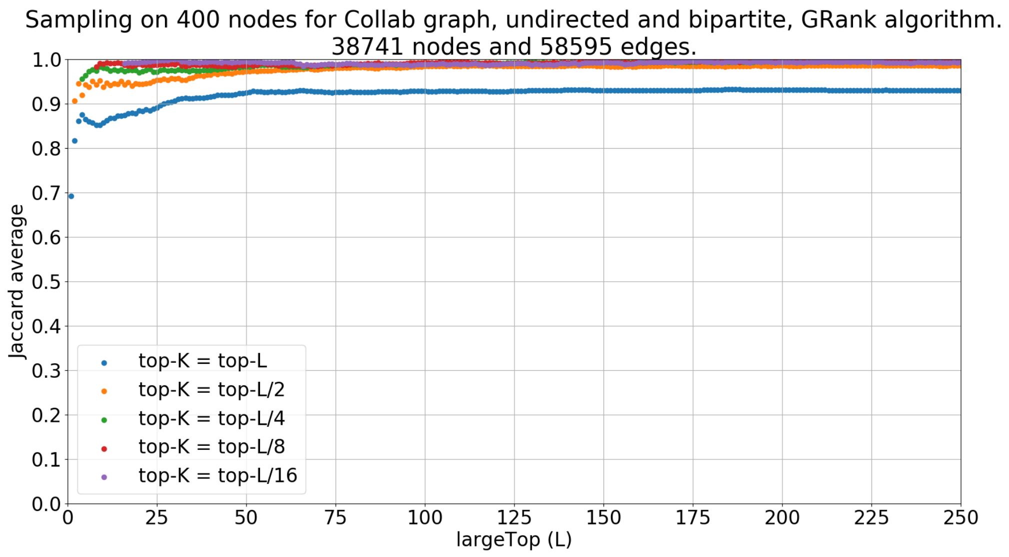

Figure 5.1: Quality of results being bottlenecked by having K equal to L.

Figure 5.2 displays the great advantage of exploiting and sharing information between nodes instead of splitting every top-K query in a single instance.

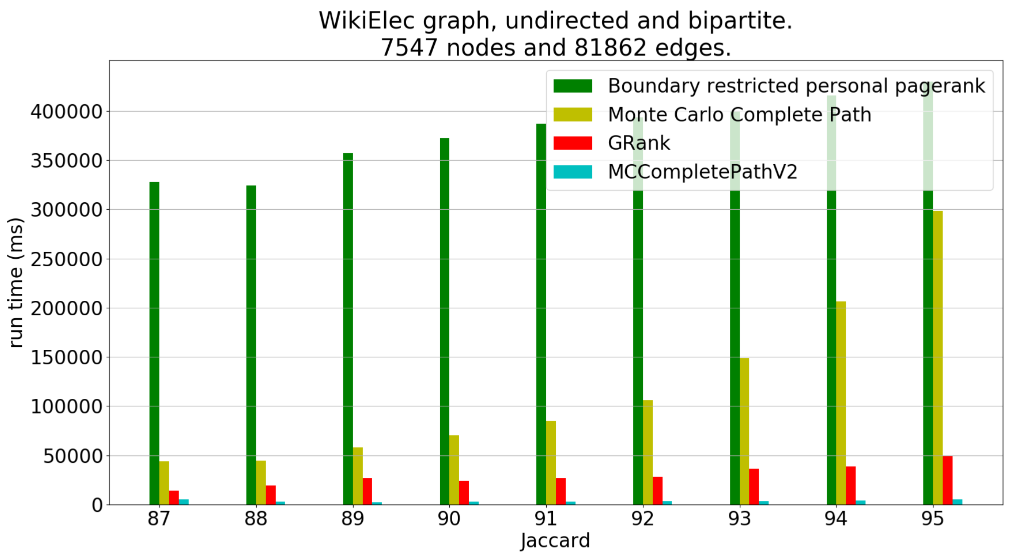

Figure 5.2: Running times for the analyzed and proposed algorithms.

Figure 5.3 offers another example where MCCompletePathV2 is performing very strongly, the running time of the non-partitioned version of GRank is also included, to show the advantage of partitioning in a non-bipartite case.

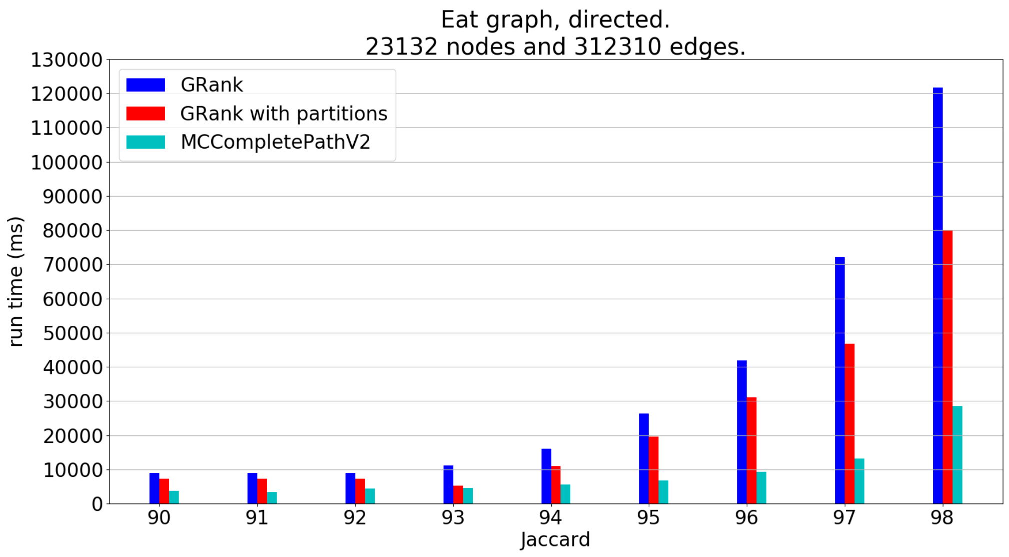

Figure 5.3: Comparison of GRank with and without partitions, MCCompletePathV2.

These last 3 charts (Figure 5.5, Figure 5.4, and Figure 5.6) show instead GRank performing way better than MCCompletePathV2.
Despite the fact that when MCCompletePathV2 outpeformed GRank it usually did it very strongly it must be pointed out that it also ends up using more memory, usually about two times to reach the same results.
Since the ram needed for large graphs may amount to very large figures it would be perfectly reasonable to use GRank to be conservative about memory.

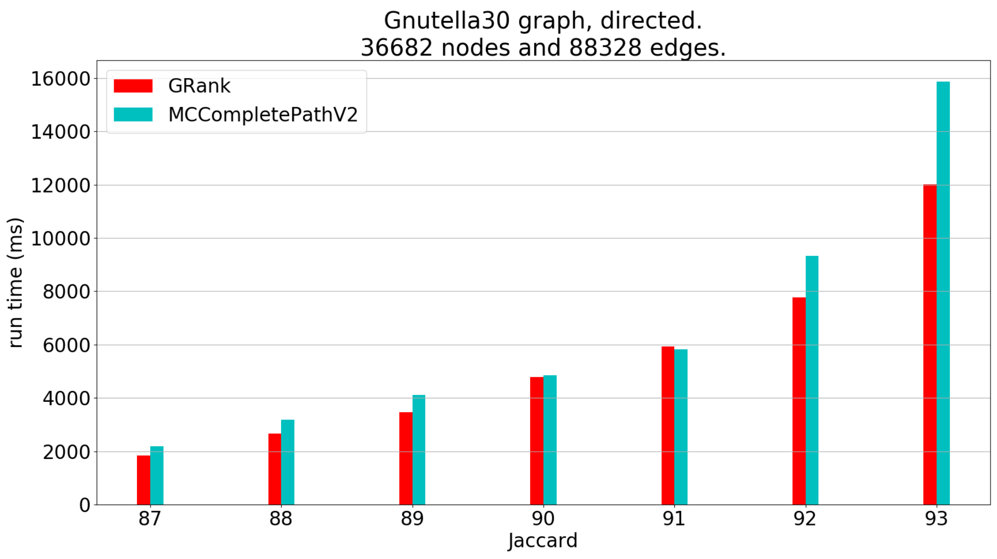

Figure 5.4: An example of GRank outperforming MCCompletePathV2; Boundary restricted Personalized Pagerank needed about 400000 ms for a Jaccard of 90.

There is not much else to say about Boundary restricted personal Pagerank and Monte Carlo Complete Path, those are fine algorithms for avoiding the use of the entire graph when computing the Personalized Pagerank for a few nodes or for online queries, but they naturally fall behind when computing results for all sources due to not sharing information used for their calculations among different instances.

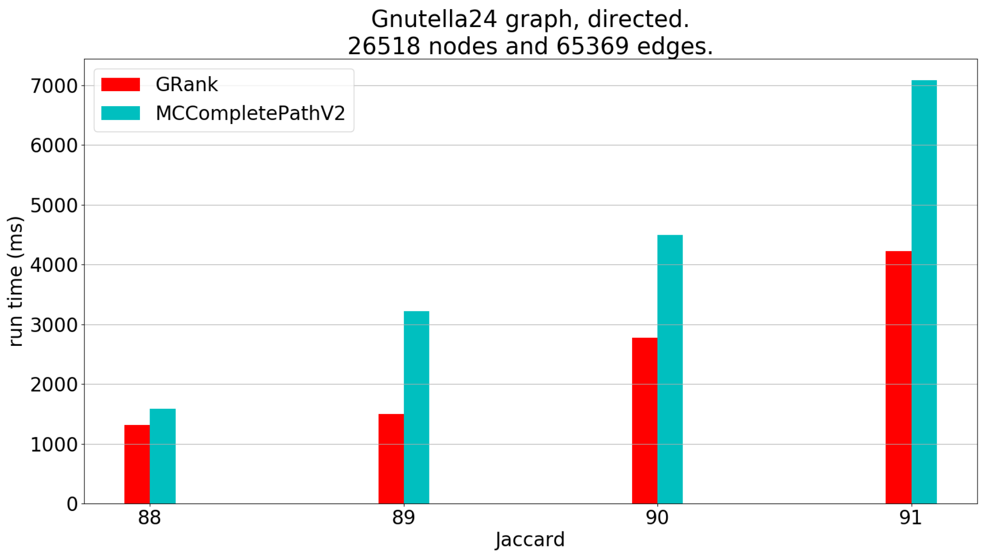

Figure 5.5: An example of GRank outperforming MCCompletePathV2.

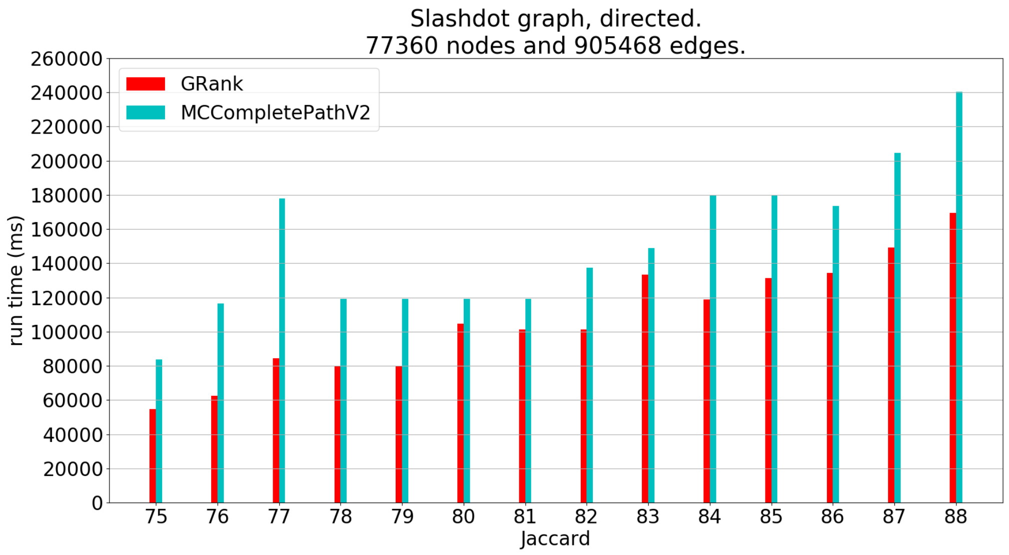

Figure 5.6: An example of GRank outperforming MCCompletePathV2.

When deciding between using GRank or MCCompletePathV2 we believe the former to be the safest choice when there is no knowledge about the graph, because it tends to not waste or overdo given a sensible tolerance.
As an example, consider a directed graph of $|V|$ nodes and $|V|$ edges, where nodes are paired making it so that the member of a pair has an edge towards the other and vice versa, running GRank here with a very high top-L would bear no inconveniences because each node would have in its top-L only itself and the node it is paired with, no matter how many iterations or how low the tolerance.
On the contrary, executing MCCompletePathV2 with a uselessly high number of random walks would be a huge waste of time here because not many simulations would be needed to assess that each node only has itself and its partner as scorers in its Personalized Pagerank vector, and which is the most important between the two.
This example highlights the fact that GRank can self regulate while MCCompletePathV2 cannot because it lacks a mechanism to detect when having done a certain number of simulations for a node is enough.
Numerous attempts have been made but without success, mostly due to the fact that usually the number of needed walks is so low that deciding how many are needed a priori or checking if it is worth to keep going would cost more than the walks themselves.
In the general case this problem shows itself when a graph has very complex areas where many walks are needed and more simple zones that might entail a waste of walks.
We attempted to minimize the cost of the walks by using different strategies, like the one named "Weighted Particle Filtering" from [17], which recursively splits the number of walks among direct successors, adding this residual to their score and actually walking only when it reaches a certain threshold. This proved to be useful only when the number of walks to do was fairly high, like in Monte Carlo Complete Path, but not in MCCompletePathV2.
Different types of walks have also been put to a test, such as the ones from [3], but to no avail: incrementing the value of every node met, a method formally called "complete path" in [3] and [5] proved to be the better working.

6 Conclusion

Extracting information and gaining insight from data is an ever increasing business opportunity, data gathering and machine learning can prove to be very complex processes, using graph based solutions and algorithms like Pagerank can provide valid alternatives.
Their strength is in simplicity: classic machine learning data might be very high dimensional and obtaining it can be an obstacle itself whereas graphs are simple and can be easily built with inferred or public information.
Personalized Pagerank for all nodes can be computationally demanding and we hope that the proposed algorithms aid in mitigating the problem or serve as material for the development of new solutions.
The two presented alternatives have different strengths: GRank tends to avoid wasting time when convergence is reached and it is a safe bet when dealing with unknown graphs, meanwhile MCCompletePathV2 can perform very fast when the misspending is at a minimum. Sadly the ordering phase of MCCompletePathV2 is still a heuristic and the lack of a tolerance or convergence mechanism is what weakens it the most.
Both algorithms lend themselves to parallelization: the update of the top-L vectors between two iterations of GRank is synchronous, while when using MCCompletePathV2 each node that will require random walks is known once the heuristic phase is complete.

6.1 An online - offline hybrid

All-sources Personalized Pagerank algorithms fit the use case of having one or more graphs with an amount of nodes that would make running classic Pagerank not feasible, but sometimes the graph can be so large that even the upfront cost of running approximation algorithms is impractical, either due to time or primary memory costraints.
At the same time services that rely on online algorithms for their functioning can lead to high long-term expenses, especially if constantly under heavy load.
MCCompletePathV2 can be adapted into being an online algorithm that slowly transitions into being offline, allowing the answering of online queries, while gradually storing information to save on long-term cpu time; of course data is assumed to now be saved on secondary memory if primary memory was the restriction that made us not able to compute all results upfront.
Nodes for which we have enough information can have their top-K query answered in an offline fashion if all their direct successors have already been computed or have walked, exploiting the concept behind MCCompletePathV2 where a node can be seen as the combination of its children.
Each node that has been queried or which has a direct predecessor that has been will require $O(L)$ space, due to having to either store the result map or the map obtained by doing random walks starting from that node. Nodes that will never be queried will not waste any space, which is something that instead happens in the offline MCCompletePathV2.
This online-offline hybrid is identical to the offline version at its core, while the sorting heuristic and the resulting execution order are removed, resulting in a simpler algorithm.

MCCompletePathV2 online-offline hybrid, algorithm for a queried node

```
$i \leftarrow queried\ node$
$result \leftarrow \{\}$
**for all** $s \in S_i$ **do**    // combine the maps of the direct successors
    **if** $scores[s] \neq NULL$ **then**    // if the node results are available use those
        $result = result + scores[s]$;
    **else**    // else use data from walks
        **if** $walks[s] == NULL$ **then**
            $walks[s] \leftarrow doRandomWalks(s,\ R)$
            $walks[s] = walks[s] * \frac{1.0}{R}$    // average the values with respect to the number of walks
        **end if**
        $result = result + walks[s]$
    **end if**
**end for**
**if** $|S_i| > 0$ **then**
    $result = result * \frac{\beta}{|S_i|}$    // average the values with respect to the number of direct successors
**end if**
$result[i] = result[i] + 1.0$;    // each node visits itself at least once per walk
$result \leftarrow keepTop(L,\ result)$
$scores[i] \leftarrow result$
```

Note that *scores* and *walks* can still be (abstractally) seen as two dictionaries of dictionaries, but they will probably be on secondary memory. As in the original version of MCCompletePathV2 please note that:

- $result = result * \frac{\beta}{|S_i|}$ means that the score of every mapped element of that map will be multiplied by $\frac{\beta}{|S_i|}$.
- $result = result + walks[s]$ means that every key-value pair in the walks[s] map will concur to increment the value of that key in result by an amount equal to its value in walks[s].

Bibliography

[1] Reid Andersen, Fan Chung, and Kevin Lang. Local graph partitioning using pagerank vectors. In *Proceedings of the 47th Annual IEEE Symposium on Foundations of Computer Science*, FOCS '06, pages 475–486, Washington, DC, USA, 2006. IEEE Computer Society.

[2] F. J. Anscombe. Graphs in statistical analysis. *The American Statistician*, 27(1):17–21, 1973.

[3] K. Avrachenkov, N. Litvak, D. Nemirovsky, and N. Osipova. Monte carlo methods in pagerank computation: When one iteration is sufficient. *SIAM Journal on Numerical Analysis*, 2007.

[4] Konstantin Avrachenkov, Vladimir Dobrynin, Danil Nemirovsky, Son Kim Pham, and Elena Smirnova. Pagerank based clustering of hypertext document collections. In *Proceedings of the 31st Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, SIGIR '08, pages 873–874, New York, NY, USA, 2008. ACM.

[5] Konstantin Avrachenkov, Nelly Litvak, Danil Nemirovsky, Elena Smirnova, and Marina" Sokol. Quick detection of top-k personalized pagerank lists. In Alan Frieze, Paul Horn, and Paweł Prałat, editors, *Algorithms and Models for the Web Graph: 8th International Workshop, WAW 2011, Atlanta, GA, USA, May 27-29, 2011. Proceedings*, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[6] Bahman Bahmani, Abdur Chowdhury, and Ashish Goel. Fast incremental and personalized pagerank. *Proc. VLDB Endow.*, 4(3):173–184, December 2010.

[7] S. Brin and L. Page. The anatomy of a large-scale hypertextual web search engine. *Computer Networks and ISDN Systems*, 1998.

[8] S. Brin, L. Page, R. Motwami, and T. Winograd. The pagerank citation ranking: bringing order to the web. *Stanford University Technical Report*, 1998.

[9] Dániel Fogaras, Balázs Rácz, Károly Csalogány, and Tamás Sarlós. Towards scaling fully personalized pagerank: Algorithms, lower bounds, and experiments. *Internet Mathematics*, 2(3):333–358, 2005.

[10] David Gleich and Marzia Polito. Approximating personalized pagerank with minimal use of web graph data. *Internet Mathematics*, 3(3):257–294, 2006.

[11] P. Gupta, A. Goel, J. Lin, A. Sharma, D. Wang, and R. Zadeh. Wtf: The who to follow service at twitter. In *In Proceedings of the 22nd international conference on World Wide Web*, 2013.

[12] Taher H. Haveliwala. Www2002 conference proceedings. In *Topic-Sensitive PageRank*, 2002.

[13] Paul Jaccard. The distribution of the flora in the alpine zone. *New Phytologist*, 11(2):37–50, February 1912.

[14] Glen Jeh and Jennifer Widom. Scaling personalized web search. In *Proceedings of the 12th International Conference on World Wide Web*, WWW '03, pages 271–279, New York, NY, USA, 2003. ACM.

[15] M. G. Kendall. A new measure of rank correlation. *Biometrika*, 30(1/2):81–93, 1938.

[16] William R. Knight. A computer method for calculating kendall's tau with ungrouped data. *Journal of the American Statistical Association*, 61(314):436–439, 1966.

[17] Ni Lao. Efficient random walk inference with knowledge bases. *Carnegie Mellon University*, 2012.

[18] David Liben-Nowell and Jon Kleinberg. The link-prediction problem for social networks. *Journal of the Association for Information Science and Technology*, 58(7):1019–1031, May 2007.

[19] Atish Das Sarma, Sreenivas Gollapudi, and Rina Panigrahy. Estimating pagerank on graph streams. *J. ACM*, 58(3):13:1–13:19, June 2011.

[20] Elena Smirnova, Konstantin Avrachenkov, and Brigitte Trousse. Using web graph structure for person name disambiguation. In *CLEF*, 2010.

[21] Chao Zhang, Shan Jiang, Yucheng Chen, Yidan Sun, and Jiawei Han. Fast inbound top-k query for random walk with restart. In *Proceedings, Part II, of the European Conference on Machine Learning and Knowledge Discovery in Databases - Volume 9285*, ECML PKDD 2015, pages 608–624, New York, NY, USA, 2015. Springer-Verlag New York, Inc.

Sitography

[22] Fastutil, a collection of type-specific Java classes. `http://fastutil.di.unimi.it/`. Last access: 2017-05-20.

[23] JGraphT, Java graph library. `http://jgrapht.org/`. Last access: 2017-06-20.

[24] The koblenz network collection. `http://konect.uni-koblenz.de/`. Last access: 2017-05-30.

[25] Big graph data sets. `http://lgylym.github.io/big-graph/dataset.html`. Last access: 2017-05-30.

[26] Stanford large network dataset collection. `http://snap.stanford.edu/data/index.html`. Last access: 2017-05-30.

[27] Spaziodati. `http://www.spaziodati.eu`.