Parallel PathFinder Algorithms for Mining Structures from Graphs

Samson Hauguel, ChengXiang Zhai, Jiawei Han
Department of Computer Science
University of Illinois at Urbana-Champaign
Urbana-Champaign, USA
hauguel1@illinois.edu, {czhai, hanj}@cs.uiuc.edu

Abstract— **PathFinder networks are increasingly used in Data Mining for different purposes, like network visualization or knowledge extraction. This novel way of representing graphical data has been proven to give better results than other link reduction algorithms, like minimum spanning networks. However, this increase in quality comes with a high computation cost, typically of the order of n^3 or higher, where n is the number of nodes in the graph. While this problem has previously been tackled by using mathematical properties to speed up the algorithm, in this paper, we propose two new algorithms to speed up PathFinder computation based on parallelization techniques to take advantage of the increasingly available multi-core hardware platform. Experiments show that both new algorithms are more efficient than the state of the art algorithms; one of them can achieve speed-ups of up to x127 with an average of x23 on recent hardware (2007).**

Graph Mining, Parallel Computing

I. INTRODUCTION

Pathfinder networks were proposed in 1990 [1] as a new way to represent information in graphs. Given a weighted network, the basic idea of the algorithm (referred to as PFNET) is to repeatedly delete edges if there exists a shorter path between the two nodes connected by that edge. The intuition behind PFNET is to keep only the "essential" edges which can best explain the similarity between two nodes, thus achieving a compact representation of the original network which can not only reveal interesting latent structures in a graph, but also facilitate visualization of large graphs.

Pathfinder networks were initially used in various domains, such as Social Sciences and Graph Theory [2], in order to enlighten the most important features in a given network. Recently, Pathfinder networks have been used in Data Mining for different purposes, such as network visualization [8] or knowledge extraction [6] with promising results. For example, PathFinder has been shown to be quite effective for visualization of evolving networks [7] and author co-citation analysis [9]. Compared with the main-stream graph mining algorithms such as frequent subgraph mining [15], PathFinder offers a principled complementary way of mining latent structures of graphs and generating a semantic summary of graphs.

Although the applications of Pathfinder networks are numerous, a main drawback is its high computation cost which is in order of n^4, where n is the number of nodes in the input graph. While previous works focused on using mathematical properties to speed up the algorithm, we propose to leverage multi-cores and design a partition-based parallel PathFinder algorithm (called PB-PathFinder) which would take full advantage of the increasingly available multi-core hardware. The main idea of PB-PathFinder is to use a divide and conquer approach to reduce the original problem into subproblems that can be solved in parallel.

We evaluate PB-PathFinder by comparing it with both a state of the art non-parallel algorithm and a straightforward parallel implementation of PathFinder. Experiment results show that PB-PathFinder outperforms the baseline parallel implementation and can achieve speed-ups of up to x127 with an average of x23 on recent hardware (2007) over a state of the art non-parallel algorithm.

The rest of the paper is organized as follows. In section 2, we introduce PathFinder. In section 3, we explain the ideas of our two parallel algorithms, namely the Multi-Threaded Pathfinder and the Partition-Based Pathfinder algorithm. We discuss the performance of these two parallel algorithms in section 4. Finally, we conclude in Section 5.

that follow.

II. PATHFINDER NETWORKS

Informally, Pathfinder networks are the output of the PFNET algorithm, which takes a weighted graph as input and prunes its edges to obtain a more compact graph. The rule for pruning an edge is described in the next paragraphs.

Given an undirected weighted graph *G=(V,E)* and two parameters, *r* real and *q* integer, we denote *A* and *B* as two nodes of the input graph. We also denote the edge between *A* and *B* by $e_{A,B}$, and its associated weight by $w_{A,B}$.

We define a path $P_{A,B}$ between *A* and *B* as a non-repeating sequence of nodes $\{A = X_1, X_2, \dots, B = X_l\}, X_i \in V$.

The length $|P_{A,B}|$ of the path is defined as the number of edges in that path. From our definition, this length $|P_{A,B}|$ is always less than *n*.

The weight $w_{P_{A,B}}$ of the path is defined by the following formula:

$$w_{P_{A,B}} = \left|\left(w_{X_1,X_2}\right)^r + \left(w_{X_2X_3}\right)^r + \cdots + \left(w_{X_{l-1}X_l}\right)^r\right|^{\frac{1}{r}}$$

Now we can define the pruning condition. With the same notations, the edge $e_{A,B}$ will be pruned if and only if there exists a path $P_{A,B}$ from A to B such that:

$$\left|P_{A,B}\right| \leq q \ \ AND \ \ w_{P_{A,B}} < w_{A,B}$$

As an example of application of PFNET, we show in Figure 1 part of the extracted structures of words that are semantically related to "disruption" from applying PFNET to a word network constructed based on co-occurrences in text database. The tree structure extracted is intuitively quite meaningful.

Previous optimizations of PFNET have led to two new algorithms. The first is the binary pathfinder proposed by V.P. Guerrero-Bote and al. in [3], which performs in $O(n^3.\log n)$. The second, proposed by A. Quirin et al. in [4] performs in $O(n^3)$, at the cost of fixing the parameter q to n – 1. The interested reader will find more details about the various pathfinder algorithms in [1] [3][4].

|-- disruption
    |-- inconvenience
        |-- distraction
            |-- constant
            |-- diversion
        |-- hassle
            |-- bother
            |-- grind
        |-- Nuisance
    |-- outages
        |-- brownout
        |-- blackout
    |-- delay
        |-- postponement
            |-- rescheduling
                |-- deferment
            |-- cancellation
                |-- recall
    |-- breakdown
    |-- interruption

Figure 1. Sample PFNET results

III. PARALLEL COMPUTATION OF PATHFINDER

In this section, we introduce two new versions of pathfinder algorithm, namely MT-PFN, a direct parallelization of binary pathfinder, and PB-PFN, a new parallel pathfinder algorithm based on a partition of the input to solve the problem when q=n-1.

A. MT-PFN: a Multi-Threaded Pathfinder

The algorithm MT-PFN is mainly a parallelization of the binary pathfinder algorithm, or more precisely of the computation of each intermediate D^i matrix. The idea is that each element of a matrix D^i can be computed independently of the other elements of the matrix.

For a single matrix D^i, we spawn n threads (n being the number of nodes) each of them computing one row of matrix. We do not spawn n^2 threads, one for each element, because the overhead of spawning a thread is too big compared to the computation of one element. Besides, current parallel computer architectures do not support very high numbers of simultaneous threads, and spawning n^2 threads would quickly exceed the maximum number of threads which can be executed in parallel.

In term of synchronization, barriers are needed to ensure that each matrix is entirely computed before any threads can start computing the next matrix to ensure correctness.

The performance results are shown in section 5.

B. PB-PFN: a Partition-Based Pathfinder

Although MT-PFN is much faster than the binary pathfinder, it still scales in $O(n^3.\log n)$. At the cost of flexibility, we fix $q = n-1$, as in the cubic time PFNET, and we propose another parallel algorithm PB-PFN based on partitioning the graph into multiple subgraphs.

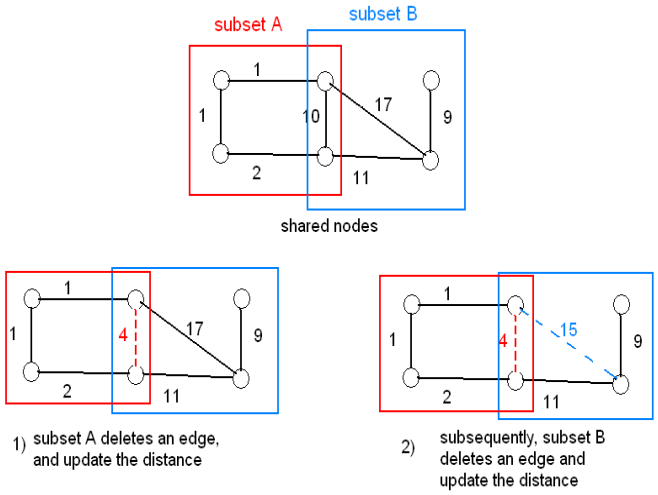
Informal explanation:

Intuitively, if we could apply PFNET to a smaller subset of the original network, and then aggregate the results of all the subsets, the resulting computing time will be much smaller. Indeed, if we partition the input graph into M subgraphs and denote by k_i the size of subgraph s_i, we could reduce the computation cost from $O(n^4) = O((\sum k_i)^4)$ to $O(\sum(k_i)^4 + aggregation\_cost(s_1, s_2, \ldots, s_M))$. Moreover, we could gain substantial speed-ups by running all the PFNET tasks in parallel.

However one major problem arises: how do we aggregate the results? Put in another way, how can we ensure that the triangular inequality is enforced for the whole graph (= global property), while we have just computed PFNET for subsets of the graph (= local property)?

The proposed solution is to propagate the results of the computation from one subset to another. More precisely, once PFNET has been computed for a subset, every adjacent subset (i.e. that shares one or more nodes) has access to the updated distance between every pair of nodes in the first subset. With this new information, the adjacent subsets can update their own distance information and delete edges if needed. These updates will in turn start other updates in the adjacent subsets of these subsets. The algorithm is guaranteed to stop because at some point, the (last) update will not start any action from the adjacent subsets.

To visualize this, let us take a simplified example in figure 1.



The top graph is the initial input graph, partitioned into two subsets A and B. In our context, a partition is such that every edge is assigned at least to one subset. We allow edges and nodes to be shared by different subsets. We also allow nodes not to be connected within a subset. However, we will see that the algorithm achieves better performance when subsets share as few nodes as possible. Back to our example, we notice that running PFNET on B will result in no edge deleted. We start running PFNET on A, and as a consequence, one edge is deleted (10 > 1+1+2) and the matrix of distances is created. We can see on the bottom left graph the changes induced by PFNET on A. Since A and B shares two nodes and the distance between these two nodes has been updated, the computation unit in charge of B must update its subset. This update is followed by the deletion of an edge (17 > 4+11), and the update of the distance matrix. Since no subset is concerned by this change, the algorithm stops. The final result is shown in the bottom right graph.

It is worth mentioning that a subset treats each incoming update one at the time. That is, if multiple updates are transmitted from one subset to another, the second subset will serialize these updates.

Pseudo-Code for PB-PFN:

Input:

matrix G; //the input graph as a matrix
double r; // parameter for PFN
int M; // the number of subsets

Main Thread:

$for\ each\ matrix\ m_i\ in\ partition(G, M) do$:
  $S_i = spawnASlave(m_i)$
end
  $while(!\ allInqueuesEmpty())\ do$:
  // wait for all slaves to finish
end
$return\ mergeAllDistanceMatrices(S_1, \dots, S_M)$

Subset Thread:

$local\ matrix\ dist_i$ *// current distance between each node in subset*
$local\ queue\ inqueue_i$ *// queue of incoming updates*
$local\ List\ outqueues_i$ *// list of adjacent subsets' inqueues*

// --- phase 1: PFNET cubic time ---
$tmp = shortestDistMatrix(dist_i, r)$
$putUpdates(outqueues_i, diff(dist_i, tmp, G))$
$dist_i = tmp$

// --- phase 2: updates ---
$while(1) do$
  $for\ each\ update\ updt\ in\ inqueue_i\ do$
    $tmp = updateDist(dist_i, updt)$
    $putUpdate\ (outqueues_i, diff(dist_i, tmp, G))$
    $dist_i = tmp$
  end
end

Update routine:

$input\ matrix\ dist_i$
$input\ update\ updt$

$matrix\ res = copy(dist_i)$
$for\ each\ edge\ currEdge\ in\ res\ do$ *//test each pair of nodes in subset*
  $node\ A = currEdge.node1$
  $node\ B = currEdge.node2$
  $node\ U = updt.node1$
  $node\ V = updt.node2$

  //can we find a shorter path involving the new update?
  $AUVB = weightPath(A, U, V, B)$
  $AVUB = weightPath(A, V, U, B)$
  $w_{p_{A,B}} = \min(AUVB, AVUB)$

  // the new path is shorter, hence update distance
  $if\ w_{p_{A,B}} < currEdge.weight\ then$
    $res[A][B] = w_{p_{A,B}}$
  end
end
$return\ res$

As shown in the pseudo code, every subset is in charge of computing its distance matrix by first running a cubic time pathfinder and then processing incoming updates. In the pathfinder phase, each subset computes the shortest distance matrix and compares the result to the initial matrix, in order to know which edge must be deleted (purpose of method diff()). Every difference detected also generates an update which is transmitted to all neighboring subsets. In the second phase, each subset waits for incoming updates and process them by testing all the potential paths which might be shorter, due to the distance update. These potentially shorter paths must contain the newly updated edge, because we ensured in the previous phase / updates that no path is shorter than the remaining edge. Hence, for each pair of node, there are only two such paths involving the update. The end of the update

phase is just a matter of computing the diff() and transmitting the new update (if there is one).

Partitioning and Correctness of the algorithm:

Partitioning of the input graph plays a major role in the algorithm, in term of performance and also in terms of correctness. Ideally, the subsets should be of the same size in order to keep a balanced workload for each thread and they should share as few nodes as possible, so that the interaction between different subsets is minimized. Fortunately, the problem of finding such a balanced partition has been previously studied in the area of parallel computation [12, 13, 14]. State of the art software such as [11] are freely available online, which we use in our experiments.

From the correctness perspective, we will state the condition under which the algorithm is correct, in the form of two lemmas. For the sake of brevity, the demonstrations are not included.

Lemma 1: (Correctness) Given an input graph G, for any partition of G, every edge deleted by PB-PFN is also deleted by PFNET.

Before stating lemma 2, we introduce the notion of partition graph. Once a partition is chosen, every edge is assigned to one or more subsets and consequently, every node is assigned to one or more subsets. Now we define the partition graph as the hypergraph where the nodes represent a subset and two nodes are connected in the hypergraph if and only if the subsets share a node in the original graph. Therefore, in the partition graph, one edge can connect multiple nodes and there can be multiple edges between to nodes.

Lemma 2: (Completeness) Given an input graph G and a partition S of G, if the partition graph relative to S is acyclic then every edge deleted by PFNET is also deleted by PB-PFN.

Although this condition might seem difficult to enforce, we will present a strategy to overcome this constraint in the next section.

IV. Experiments and Results

After the text edit has been completed, the paper is ready for the template. Duplicate the template file by using the Save As command, and

In this section, we present our experimental set up and the results obtained with our new algorithms. We also present our strategy for correctness in a concrete example. We will explore the following questions:

1. How do our algorithms perform with regard to their respective baseline?
2. Is it possible to leverage the sparseness of the input graph to achieve even greater speed ups?

For the sake of clarity, we restate the denomination of each algorithm in the following table:

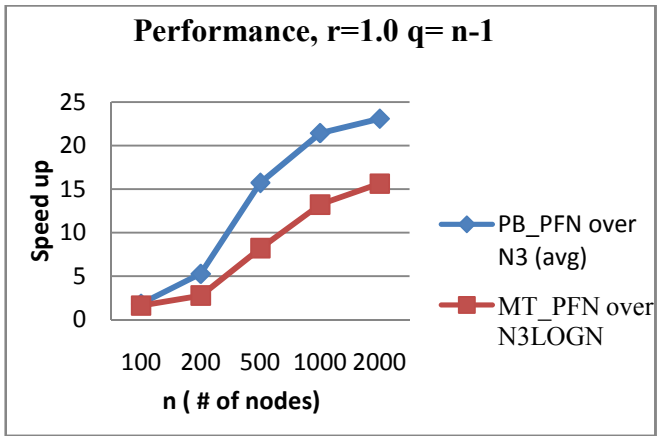
N^3LOGN	The binary pathfinder
N^3	the $O(n^3)$ implementation of pathfinder, which accepts only maximal value for parameter q.
MT-PFN	the Multi-Threaded implementation of N^3LOGN, using common parallelization techniques.
PB-PFN	the Partition Based pathfinder, which also accepts only maximal value for parameter q.

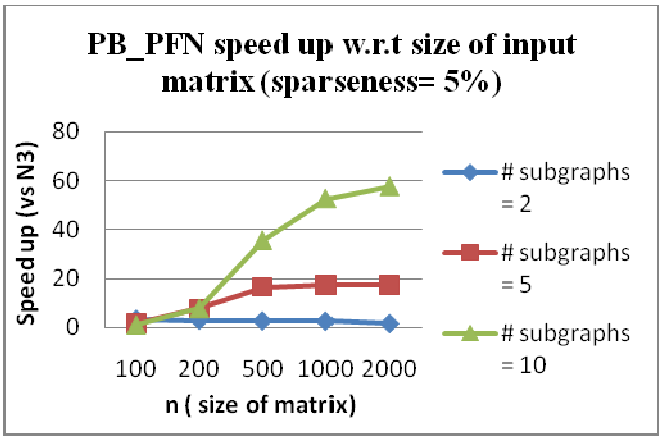
Unless stated otherwise, all experiments are designed in Java and performed on a simultaneous multi-threaded machine, namely a Sun Niagara II. The specification of the machine is the following: 8 cores, each of which can run 8 threads simultaneously, for a total of 64 simultaneous threads.

The input graphs have been generated using a real graph generation software, described in [10]. The partition of the input graph needed by PB-PFN have been performed by a state of the art partitioning software, namely Metis, which is freely available at [11]. The cost of partitioning has not been included, since they are negligible (1.1 seconds for the largest input compared to 423 seconds for the computation).

A. Experiment 1: Scalability

In this experiment we run both algorithms against their respective baseline, for different sizes of the input, all other parameters fixed. Specifically, we fixed the r parameter to 1.0 and q parameter to the number of nodes minus one. This setting is the most commonly used. Then we compare the speed ups obtained by both algorithms over their baseline. The top graph shows the overall performance of both algorithms, while the bottom graph shows the performance of PB_PFN for different granularities of the input partitions.

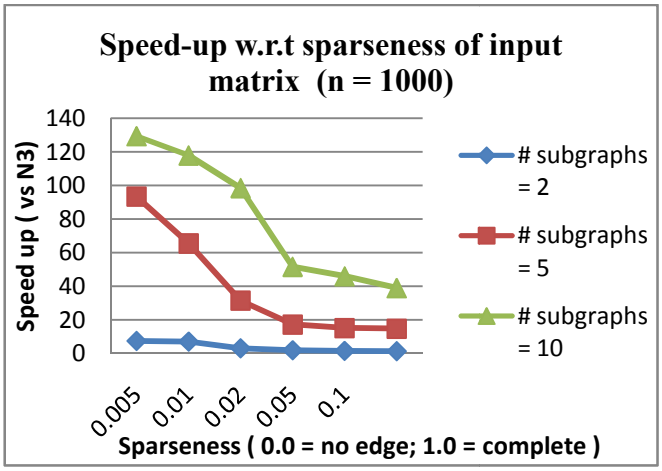




Interpretation:
On the top graph, we can observe that both algorithms scale well. Indeed, we achieved up to 15x speed up with MT-PFN and 23x speed up with PB-PFN, on average. We can explain such speed ups by the fact that both algorithms spawn a lot of threads that are executed in parallel. Specifically, MT-PFN generates one thread per line of the input matrix, that is, n threads. For PB-PFN, the number of generated threads directly depends on the partition of the input graph. As we can observe on the bottom graph, finer partitions (i.e. break the graph into a lot of sub-graphs) generate more threads (one per sub-graph) and achieve better speed ups. Hence, fine grain partition should be preferred, as long as the correctness is preserved (see section 3: there must be no cycle amongst the sub-graphs).

B. Experiment 2: Speed ups on sparse input

In this experiment, we run PB-PFN on graphs of size n = 1000 and of different sparseness. The sparseness is measured as the ratio of the number of edges over the maximum number of edges (if the input graph was complete). The results are shown for different granularities of the input partitions.



Interpretation:
As the input graph gets more edges, the speed ups obtained are less important: at 20% of sparseness(rightmost point) the speed-up is almost negligible (1.02x) for coarse partitions. Indeed, when the graphs are dense, more edges can be pruned, which yields to more communication between the threads. This extra communication increases the computation in two ways. First, it triggers an update routine which is quite costly (o(k^2), where k is the average size of a sub-graph). Second, the outcome of an update routine is potentially a new update to transmit, that is, new communications between threads are triggered.

Overall when the graph grows denser, the performance of PB-PFN decreases a lot. However, the reverse is also true: for very sparse graphs _which are commonly found in real world applications, it is possible to achieve great speed ups (127x with 10 sub-graphs and 0.5% of sparseness).

This observation yields to a strategy for enforcing correctness with PB-PFN. As we have stated earlier, PB-PFN is correct only if the partition of the input is such that there is no cycle amongst the sub-graphs. It turns out that state of the art partitioning algorithms rarely produce 'acyclic' fine grain partitions. However, all edges deleted with an incorrect partition will also be deleted by a correct partition (but some edges will not be deleted with an incorrect partition). Hence the idea of applying several times PB-PFN, with coarser partitions (i.e. with less sub-graphs) each time. Indeed, as the partition gets coarser, the probability of finding an 'acyclic' partition increases (worst case: partition in 2 sub-graphs is always acyclic). In the same time, each processing step prunes the graphs, producing a sparser graph for the next step. As a result, the sparseness of the graph compensates the fact we choose coarser partitions. Sparser graphs allow faster computation, balancing the slowdown induced by coarser partitions.

As an illustration, we applied this strategy on a real world example. The input graph is a list of 1000 common words from newspapers; each edge is weighted by the distance between words. We ran successively PB-PFN with a partition of size 10, 5, and 2. The result is compared to the N^3 computation, and we obtained a speed up of 3.7. As expected, the output graphs are identical.

C. Summary: Recommendations for the practitioners.

As we can see from the experiments, PB-PFN performs extremely well on sparse graphs. As long as there is no need for having *q* different from *n-1*, PB-PFN, with multiple runs as explained in the strategy for correctness, is the best choice.

However, if graphs are growing denser (>30% complete) or there is a need for different value for *q*, MT-PFN is recommended.

For small scale and exploratory experiments (less than 100 nodes), the cubic time algorithm is recommended for its really simple implementation.

V. CONCLUSIONS

PathFinder is a new principled algorithm for mining latent structures from graphs with many potential applications in multiple domains. A main barrier in applying this method to large graphs is its computational complexity. In this paper, we present two new ways to speed up PathFinder through parallelization, leading to two new PathFinder algorithms (MT-PFN and PB-PFN). Experiment results show that both can significantly speed up PathFinder compared with the corresponding baseline implementation.

REFERENCES

[1] Schvaneveldt, R. W. (Ed.) (1990) Pathfinder Associative Networks: Studies in Knowledge Organization. Norwood, NJ: Ablex

[2] Schvaneveldt, R. W., Durso, F. T., & Dearholt, D. W. (1989). Network structures in proximity data. In G. Bower (Ed.), The psychology of learning and motivation: Advances in research and theory, Vol. 24 (pp. 249-284). New York: Academic Press

[3] V.P. Guerrero-Bote, F. Zapico-Alonso, M.E. Espinosa-Calvo, R.G. Crisostomo, F. de Moya-Anegon (2006) Binary Pathfinder : An imporvement to the Pathfinder algorithm, Information processing and Management, 42, 1484-1490

[4] A. Quirin, O. Cordon, J. Santamaria, B. Vargas-Quesada, F. Moya-Anegon (2007), A New Variant of the pathfinder algorithm to generate large visual science maps in cubic time, Information processing and Management.

[5] Ojmason, Fast PFNETs in Erlang, http://omlog.wordpress.com/2009/02/14/fast-pfnets-in-erlang/

[6] C. Chen et al 1999, Generalised similarity analysis and Pathfinder network scaling, Interacting with Computers, 10 (2): pp. 107-128.

[7] C. Chen et al 2003, Visualizing evolving networks: minimum spanning trees versus pathfinder networks, Information Visualization. INFOVIS2003. IEEE Symposium on

[8] C.Chen et al 1998, Bridging the gap: The use of pathfinder networks in visual navigation, Journal of Visual Languages and Computing, 9 (3): pp. 267-286.

[9] H. White 2003, Pathfinder networks and author co-citation analysis: a remapping of paradigmatic information scientists, Journal of the American Society for Information Science and Technology, 54 (5): pp.423-434.

[10] F. Viger, M. Latapy (2004) Random generation of large connected simple graphs with prescribed degree distribution, Lecture Notes in Computer Science 2005: pp.440 – 449. http://www-rp.lip6.fr/~latapy/FV/generation.html

[11] Metis, graph partitioning algorithm.
http://glaros.dtc.umn.edu/gkhome/metis/metis/download

[12] Multilevel Algorithms for Partitioning Power-Law Graphs. Amine Abou-Rjeili and George Karypis. IEEE International Parallel & Distributed Processing Symposium (IPDPS), 2006.

[13] Multi-Constraint Mesh Partitioning for Contact/Impact Computations. George Karypis. Supercomputing, 2003.

[14] A fast and high quality multilevel scheme for partitioning irregular graphs. George Karypis and Vipin Kumar. SIAM Journal on Scientific Computing, Vol. 20, No. 1, pp. 359 - 392, 1999.

[15] J. Han, M. Kamber, Data Mining: Concepts and Techniques, Morgan Kaufmann, 2005.