April 28, 2005

Clustering the CiteSeer Citation Graph

Mini-project report for the Problem Solving in Computer Science course

by

Abishek Garg

Christophe Gensoul

Alex Şuşu

Abstract

Citation Graphs representing scientific papers contain lot of valuable information about the level of scientific activities and the evolution of research fields over a period of time. In this report, we present the results of graph partitioning applied to the citation graphs extracted from the citeseer database. A new heuristics has been proposed, which partitions the graph taking into consideration the information which is specific to the shape of Citation Graphs.

1 Introduction

In the scientific domain, papers cite other authors' work, and are themselves cited by some other articles. The relation between different technical papers can be seen as a *citation graph*, where an edge exists from paper A to paper B, if A cites the work of B.

Citation graphs representing scientific papers contain valuable information about level of scholarly activity and the evolution of the fields over the period of time. For example, by looking at the shape of citation graph over the period of time, one can have an idea of how the two different fields were interacting over that period and to some extent define the interaction between different fields in future.

In this project, we work on the citation graph constructed from the information collected from CiteSeer Database.

The reason we chose CiteSeer is that the information regarding the papers and the citation relations are easy to obtain - the information regarding a paper is stored in an XML record, and, big part of the whole CiteSeer database, was available in one big tar.gz archive downloadable from the web.

The goal of the project is to find an efficient algorithm, that partitions this graph into 16 different communities, such that the number of inter-community edges is minimized. We formulate the problem as follows.

The Multi-terminal Min-cut Problem Given an *undirected graph* G having n vertices and m edges, and k terminal nodes (also referred during the lecture as anchor or seed nodes), find a k-partitioning of the graph, such that:

1. Each terminal node belongs to one partition.
2. The number of edges crossing between different partitions is minimized.

In our case we pick $k = 16$, i.e we choose 16 highly cited papers from 16 not so directly related domains of computer science and make them the terminal nodes.

In the end, we hope that a solution for our clustering approach will yield a partitioning that indeed makes sense, i.e. the papers in one cluster are really part of that cluster - taking into consideration the content of the paper, semantics of the title, etc.

In [9], the multi-terminal min-cut problem is shown to be NP-hard on a general type of graph for a fixed k. [8] gives a polynomial time algorithm for the multi-terminal min-cut problem in the case of DAG or *planar* graphs.

To solve the problem,therefore, we have to use some approximation algorithms, using the heuristics techniques that run in at most $O(n^2)$ time complexity. This upper bound comes from the fact that the graph we are going to work has over 1 million nodes.

2 Graph Analysis

The citation information of the papers is downloaded from the Citeseer Open Access Information link. Most of the information is stored in one big tar.gz archive, currently at the following address:

http://cs1.ist.psu.edu/public/oai/oai_citeseer.tar.gz.

The archive contains the information for the first 570,000 papers of the total 718,000 papers available in the Citeseer database, in the ID order. We downloaded the XML files corresponding to the papers with IDs from 570,000 to 718,000 and centralized all the information in three text files: one containing the refers edges between the papers (the papers we represent them in the file by their IDs), one containing the referred by edges and one containing the other information related to the papers - title, authors, year of publication, etc.

To read the XML records we have used an XML parser API for O'Caml. The complete parsing of all the XML information - approximately 2 GB of data, took less than one hour.

In our paper we refer mostly to the graph that we construct from the refers edges - we discard the referred by edges. This graph has 682,669 nodes (not all IDs up to 718,000 are used) and 1,604,796 edges. The number of isolated nodes is 323,336, so the number of non-isolated nodes is 359,333.

We use adjacency lists to represent the graph. All our implementations have a space complexity of $O(m + n)$. Our programs use at most around 150 MB of memory.

Many of the papers in Citeseer database refer to papers that are not inside the database. This makes the citation graph sparse.

Another reason that makes the graph sparse is that we did not consider the referred by edges. We did not do this because we have found out they mean something different than expected - we were expecting that if we have ID2 is referred by ID1 then this represent the citation relation between two papers inside the CiteSeer database. Our colleagues have found out

that actually this means that the existing paper in the database with ID1 cites the NON-existing paper in the CiteSeer database with ID2. We call a paper that does not-exist in the database, but is referred to, in CiteSeer terminology, a context. In other words, the refers relation involves IDs of the papers in the database, while the referred by relation involves an ID of a paper inside the database and an ID of a "context" paper.

Other facts regarding the data from the XMLs from CiteSeer that we would like to mention:

- There are false refers edges: the paper with ID 340126 (Diffie Hellman - "New directions in Cryptography") is said to be referenced by paper 42727 (a PhD thesis). In the actual electronic version of paper 42727 there are no references to paper 340126. Also, as shown in Figure 1 we found another false reference edge from the paper "From Parallel Grammar Development towards Machine Translation - A Project Overview" to the paper "Semantic-Based Transfer". In this case, a possible explanation for this error would be that author Franke of the first paper wrote another paper that refers the second paper.
- We have at least one invalid XML file - the record with ID 631953
- The XML files from the TAR.GZ (that are from Jun 2004) are a little bit different compared to the XMLs that we can get from HTTP, e.g. for ID 634
- There are 6487 papers that are referencing themselves
- The publication dates of the papers inside the XML database are not reliable.

At the end of the paper, we will present the result of our partitioning algorithm obtained on the graph that contains the "context nodes" and the refers by edges.

2.1 Strongly/Weakly Connected Components Decompositions

To have more knowledge about the graph, we compute the Strongly Connected Components (SCC) and the Weakly Connected Components (WCC - the connected components in the undirected graph) in the graph. The results are shown in Tables 1 and 2.

Table 1: Number of SCCs in the Graph and their size

Number of Vertices	Number of SCCs
1	338049
2	3755
3	633
4	223
5	93
6	52
7	24
8	20
9	9
10	6
11	5
12	7
13	3
14	3
15	2
17	2
18	1
19	1
21	1
28	1
37	1
45	1
56	1
8880	1

Table 2: Number of WCCs and their size

Number of Vertices	Number of WCCs
2	4067
3	1163
4	480
5	243
6	123
7	77
8	40
9	28
10	28
11	26
12	16
13	9
14	6
15	6
16	4
17	5
18	3
19	6
20	1
21	5
23	1
24	1
25	1
26	1
27	4
29	1
31	1
33	1
34	2
35	1
41	1
48	1
49	1
52	1
53	1

For both SCC and WCC decomposition we have used two different programs, in order to remove the possibility of having erroneous outputs: one program was our (efficient) implementation in C++ and the other one was using the ocamlgraph library.

The citation graph has 343,243 SCCs. There is one SCC with 8,880 vertices and 338,049 SCCs with only 1 vertex. If we look at the WCCs, there is one WCC with 340,241 vertices and 4,067 WCCs with 2 vertices. We notice that all the seed nodes are in one big WCC, except the isolated seed - the one with ID 2643. Five seed nodes lie in the big SCC with 8,880 vertices and all the other seed nodes lie in different SCCs.

In Figure 1 we present an SCC with 10 nodes.

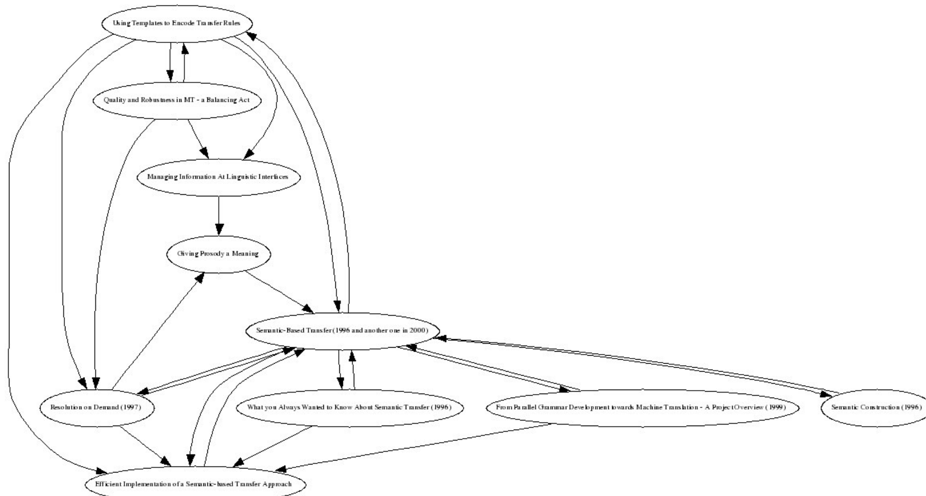

Figure 1. One SCC with 10 nodes of the citation graph. The nodes are annotated with the titles of the papers, and the year of publication. We can notice that the papers talk about very similar topics.

2.2 Planarity

The necessary condition for the graph to be planar is $m \leq 6n - 12$. We found one SCC of the graph with 6 vertices and 30 edges. This condition is violated by this SCC. Since, the planarity property is *hereditary*, i.e if the subgraph doesn't follow the planarity property, then the graph in itself is not planar, we come to the conclusion that the citation graph is not planar.

2.3 Biconnected Components Decomposition

We have used two programs once again to increase our confidence in the results. One is our (almost efficient) C++ implementation and the other one is LEDA. Our implementation of biconnected components program ran for more than 1 hour, because we used a stack data structure for edges that

is performing search inside in $O(stack_size)$. We did an improvement with a hash table that specifies if an element exists in the stack. It still is not efficient.

We obtained the same results with our applications: the citation graph, in its undirected form, has 65,464 biconnected components from which the biggest one has 1,481,065 edges, the 2nd biggest one with 138 edges, etc.

We report some problems we have encountered at this section:

- The program was returning "Segmentation fault". We found the reason, after a few hours of debugging: the stack was small - only 10 MB... When we changed it to 32MB (with the command "ulimit -HSs 32768") it worked
- On CygWin the program was crashing silently, and I could not repair the problem

3 Shortest Path Problem

In this section we describe the algorithm we have used to compute the shortest path between any two selected nodes, and the other possible algorithms we can choose from. The reasoning behind this whole exercise is to have an idea about the *diameter* of the graph. The *diameter* of the graph G is the maximum value among the shortest path between any pair of nodes of the graph.
There is no single-pair shortest-path algorithm that runs on a generic graph asymptotically faster than the best single-source algorithms in the worst case. [7] For the single-source problem, the best strongly polynomial run time algorithms are Dijkstra's and Bellman-Ford (for negative weighted edges), for finding the single pair shortest path. There are other non-strongly polynomial algorithms that perform better on specific graphs.

Dijkstra's algorithm when implemented using the binary priority heap, has a run time complexity of O((V+E)*logV).We resort to using the Dijkstra's algorithm, because we ultimately aimed at having an exact diameter of the network. To have the exact value, we need to run *all-pair shortest path* algorithm. The most efficient algorithm to solve this problem, Johnson's algorithm, make use of Dijkstra's. So, this algorithm could be used in future as a plug-in module for Johnson's algorithm.

We implement the *Dijkstra's*, in C++ using binary priority heaps. For the set of pair of nodes selected during the course, the program took 5 seconds to read the input and 1 second to compute the shortest path between the nodes.

For the all-pair shortest path problem, the most common algorithm is Floyd Warshall, which has a run time complexity of $O(n^3)$. The best algorithm for this problem is Johnson's algorithm, which runs in $O(n^2 * lgn +$

$n * m$) time. This algorithm performs very well in the case of sparse graphs.

For the problem of the shortest paths between two nodes the best time and space complexities will be attained by a BFS. The algorithm has $O(m + n)$ time and space complexities.

4 Partition

As we said before, [9] proves the multi-terminal min-cut problem is NP-hard. Therefore, to solve the problem we have to choose approximation algorithms, random algorithms or heuristics that run in at most $O(n^2)$ time complexity (taking into consideration that n will be around 1 million).

There is a significant amount of research for graph partitioning problems, since they have applicability in CAD (VLSI layout - circuit partitioning), Hw/Sw partitioning, graph embedding, identifying communities in graphs, etc.

Heuristics have been proposed for similar problems:

- the Kernighan-Lin algorithm [12] solves the k-partitioning problem that aims at clustering a graph into 2 equal-sized partitions while minimizing the number of crossing edges between the partitions. A good implementation can have a $O(n^2 log(n))$ complexity.
- Fiduccia-Mattheyses [10] is an improvement over the Kernighan-Lin 2-partitioning algorithm that has $O(m + n)$ time complexity.

General heuristics (Simulated Annealing and Genetic Algorithms) can be applied for graph partitioning problems. For example, Simulated Annealing is used in [4].

Software packages have been developed for graph partitioning. We can mention Link and Metis (http://www-users.cs.umn.edu/~karypis/metis/ and http://www-users.cs.umn.edu/~karypis/publications/Talks/multi-constraint/). An extensive list of projects and software on Graph Partitioning can be found at http://rtm.science.unitn.it/intertools/graph-partitioning/links.html.

We tried to approach the problem in an exact manner or using very good approximations. We present our results in the following subsections.

4.1 Reduction to the Max-flow Problem

We define the min-cut problem and present its correspondence with the max-flow problem. The min-cut is formulated on a directed graph G with cost (capacities) on edges. Given a source node and a target node, we have to find the cut - set of edges - with the minimal cost of edges, s.t. the source becomes disconnected from the target (that is there are no paths originated in the source that reach the target node).

A generalization of the max-flow algorithm is the multi-commodity flow algorithm, presented in [2]. Given a directed graph and a set of commodities specified as triples $(source\ node, target\ node, flow\ demand)$ the multi-commodity flow algorithm computes the maximal flow attainable simultaneously on the given graph.

[2] presents the duality between max-flow and min-cut. This result is embodied by the Max-Flow-Min-Cut Theorem that states that the value of the maximum flow in the graph between the source and the target is the value of the minimal cut for the two nodes.

We thought we can reduce the multi-terminal min-cut problem to a max-flow one. More exactly, we thought we can solve the multi-commodity flow problem on the citation graph, where the set of commodities is formed by the $k*(k-1)$ pairs of distinct multi-terminal nodes. We could not advance in this direction because there is no clear connection between the multi-terminal min-cut problem and the multi-commodity one. It is worth mentioning that the paper [13] introduces the approximate min-cut max-flow theorem for multi-commodities; in [3] a more accurate approximation is given.

We looked into one more approach. [8] presents a reduction from the multi-terminal min-cut problem on a DAG to a max-flow algorithm on a graph easily derived from the original one. Our graph, as we present in Section 2 contains cycles and a significant number of SCCs. One can attempt the following heuristic: remove the minimum number of edges from the graph s.t. the graph becomes acyclic and then apply the max-flow algorithm on the transformed graph, then reinsert the edges and maybe perturb the partition solution found by moving the nodes between the clusters in order to achieve a better cost. There are efficient algorithms for the max-flow problem, which run on a graph with unity-cost on edges with time complexity $O(n*m)$. A simple algorithm is a variation of Ford-Fulkerson algorithm that runs in $O(n*m*f)$ time on a general cost graph, where f is the value of the max-flow, taking into consideration that the max-flow in a unit-capacity graph is always upper-bounded by n - the source has at most n-1 outgoing edges; another more efficient algorithm is given in [2], Section 8.2.

Since only the biggest SCC (the one with 8,880 nodes) contains more than 35,000 edges, to remove all cycles from this SCC we have to remove probably at least 20,000 edges. Therefore, the total number of edges that need to be removed in order to make the graph a DAG is probably very big.

But actually the max-flow approach has a significant difference w.r.t. our multi-terminal min-cut problem. For example, the 2-terminal cut problem and the min-cut problem have the following subtle difference: the first one cares that there is no path in the undirected graph between the source and the target node, and the second one considers only directed paths from the source to the target. I.e., if we remove from the directed graph the edges in the cut found with the min-cut algorithm, we can still have paths from the target that reach the source node, or we can have chains of edges between the

source and the destination with edges in both directions. So, we conclude we cannot reduce our multi-terminal min-cut problem to a (multi-commodity) max-flow algorithm.

4.2 Reduction to Linear Programming

We turn our attention to the Linear programming approach for solving the multi-terminal min-cut problem. This approach is the one used by [9], [5]. We take the simple formulation given in [8], Section 2.1 and construct our linear program in the following way:

- the variables of the program are boolean variables, c_e, assigned to each edge in the graph. c_e is 1, or 0, depending if the edge e is part of the cut that is the solution to our multi-terminal min-cut problem.
- the objective function that has to be minimized in our zero-one linear program (ZOLP) is: $\sum_e c_e$
- and the constraints for the ZOLP express the fact that on each undirected path between all $(k-1)*k/2$ combinations of two different terminal nodes at least one edge has $c_e = 1$
  $\sum_{i=0}^{n-1} c_{(k_i,k_{i+1})} >= 1$, for all paths $k_0 - k_1 - k_2 - ... - k_n$ between the terminal nodes.

Unfortunately, the ZOLP is too big to be solved exactly even with state-of-the-art LP solvers. First of all, there is a very high number of constraints. For example, just between the first two terminal nods there are more than 348,410 undirected paths - we enumerated these paths, and then we decided to kill the application. There are so many paths between these nodes because the graph is highly cyclic.

Also, the number of variables for our ZOLP is very big. According to [1], 680,000 variables is the upper bound for all the implementations of ILP solvers considered. Trying to approximate our ZOLP by finding the LP solution for our ZOLP problem is probably still not feasible, once again because of the big number of variables and constraints, and also because it is not obvious how to map the LP solution back to a binary vector solution.

4.3 Using the Biconnected Components Decomposition

We presented in Section 2 the decomposition of the undirected citation graph into biconnected components (BCCs). We performed this decomposition with the idea that our graph might have several size-balanced BCCs.
Ideally, each terminal node was in one BCC. To disconnect two adjacent BCCs one may use the following properties of the BCCs: two BCCs either share only 1 edge between them, which is a bridge, or they share a node called cut (articulation) point. In the case the BCCs share a bridge, to

disconnect them we simply remove this edge - an implicit property of bridges. In the case two BCCs share a cut-point a good heuristic would be to take the cut-point in the BCC that has more adjacent nodes with it and remove the edges the cut-points has with the other BCC.

Unfortunately, we found out that our graph has one big BCC, and all relevant terminal nodes - least the isolated one, 2643 - are contained in it.

Another direction we do not pursue would be to decompose the graph into triconnected components by using an efficient algorithm, such as the one proposed in [11] and to use the result (triconnected components, bridges and cut-points) if it is helpful to build our partitioning.

We have failed in finding a good "classical" reduction (even an approximative one) of our problem to a polynomial algorithm. Therefore we have developed a heuristic algorithm specifically designed for the problem of clustering the citation graph, which we present in the following section.

4.4 The Heuristic Algorithm

In the first part of the algorithm, we are assigning the terminal nodes to one partition each.

The algorithm we are using is an iterative heuristic, that is converging to our final solution.

- The heuristic we are using is enumerating all the vertices of the graph in an arbitrary order and processes each of them:
  - To assign the current node we are checking where the papers it refers to are assigned. We assign the paper to a cluster only if all of its cited papers are already assigned to a cluster. The cluster where we distribute the current paper is one of the clusters (in our implementation, the one with the smallest ID) that contains the maximum number of cited papers.
  - We perform a similar computation as in the previous paragraph, with the difference that we take into consideration the papers that cite the current article.
- If during the current enumeration we do not distribute nodes any further, we stop.

This algorithm has a flaw. We can see it with the example from Figure 2.

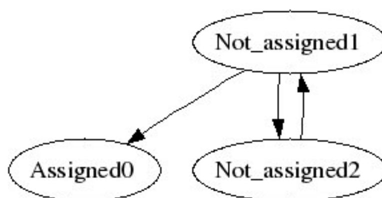

Figure 2. A graph example on which our above algorithm will not distribute the nodes Not_assigned1 and Not_assigned2 to any clusters, given any order of enumerating the vertices, considering that vertex Assigned0 was already assigned to a partition, because of the cycle formed by the two nodes.

To make the algorithm distribute all nodes to the cluster we have to relax the constraint of assigning a node only if it has all its neighbors already associated to a partition. For this we introduce the variable inv_factor that is initially set to 0. We rewrite the condition that decides if we distribute a node to any cluster as following: $the\ number\ of\ unassigned\ cited\ papers \leq (inv_factor * number\ of\ outgoing\ edges)$. If in the first enumeration we do not succeed to assign any node to a cluster, we increase with an arbitrary amount inv_factor s.t. it is between 0 and 1; in the next iteration we increase it further, and so on,. until inv_factor reaches 1. By increasing inv_factor, we are relaxing the constraint further and further until it becomes a tautology, in which case it is equal to one.

The space complexity is $O(m + n)$. The time complexity is $O(n^2)$.

We present the results for various enumerations orders of the graph nodes and schemes of increasing the inv_factor:

- Enumerating the nodes in ascending order of IDs
  - Linear increase from 0 to 1 with an increment of 1/50
    * no of vertices in the clusters: [0]=344111 [1]=85 [2]=702 [3]=63 [4]=758 [5]=80 [6]=328 [7]=689 [8]=3225 [9]=780 [10]=953 [11]=2323 [12]=3477 [13]=104 [14]=10 [15]=733
    * no of crossing edges: 92300
    * Normalized cost: 154.558168
  - Linear increase from 0 to 1 with an increment of 1/20
    * no of vertices in the clusters: [0]=145725 [1]=90 [2]=4424 [3]=145 [4]=1692 [5]=303 [6]=376 [7]=7104 [8]=159770 [9]=1388 [10]=1859 [11]=5696 [12]=28529 [13]=484 [14]=22 [15]=814
    * no of crossing edges: 305105
    * Normalized cost: 175.938715
  - "Exponential" increase from 0 to 1 with a ratio of 2
    * no of vertices in the clusters: [0]=143124 [1]=93 [2]=5106 [3]=132 [4]=1872 [5]=320 [6]=541 [7]=7456 [8]=158861 [9]=1714 [10]=1765 [11]=7117 [12]=29044 [13]=471 [14]=19 [15]=786
    * no of crossing Edges: 310712
    * Normalized cost: 170.250051
- Enumerating the nodes in "topological" order
  - "Exponential" increase from 0 to 1 with a ratio of 2

    * no of vertices in the clusters: [0]=169983 [1]=132 [2]=2450 [3]=161 [4]=1761 [5]=255 [6]=1719 [7]=5770 [8]=14850 [9]=2090 [10]=3558 [11]=12467 [12]=141819 [13]=589 [14]=21 [15]=796
    * no of crossing edges: 317165
    * Normalized cost: 157.149742

- Enumerating the nodes in the chronological order of publication. Also the vertices with degree 1 are put in the same cluster as their neighbors.
  - "Exponential" increase from 0 to 1 with a ratio of 2
    * no of vertices in the clusters: [0]=184079 [1]=104 [2]=2161 [3]=128 [4]=1400 [5]=190 [6]=574 [7]=5819 [8]=81024 [9]=2661 [10]=1998 [11]=5360 [12]=27854 [13]=293 [14]=17 [15]=684
    * no of crossing edges: 310727
    * Normalized cost: 201.396788

The different variations of the heuristic took from 4 hours to 10 hours to execute.

We present the results for one run of our program on the graph with the contexts and referred by edges with the same heuristics as in the last paragraph. The graph has 1.3 M nodes and almost 6 M edges.

- no of vertices in the clusters: [0]=659534 [1]=512 [2]=9505 [3]=22387 [4]=6832 [5]=989 [6]=6669 [7]=624 [8]=57617 [9]=8109 [10]=13805 [11]=48371 [12]=550257 [13]=81 [14]=2285 [15]=3088
- no of crossing edges: 776000
- Normalized cost: 99.148657

This last execution took more than 24 hours.

A natural extension of our algorithm is to create one more partition where to put all the papers that are assigned when $inv_factor = 1$.

5 Partitioning of graph based on Modularity

In this section we show how a different heuristics based on optimization of *Modularity*[6] of graph can be used to give a linear time algorithm.

5.1 Modularity

Modularity is the property of the network, which defines the magnitude of clearly separable communities in the network. It measures when the division is a good one, in the sense that there are many edges within communities

and only few edges in between communities.

Let A_{vw} be an element of the adjacency matrix of the network and suppose the vertices are divided into communities such that vertex v belongs to community c_v. Then the fraction of edges that lie in same community is

$$\Sigma_{vw} A_{vw} \delta(c_v, c_w)/\Sigma_{vw} A_{vw} = 1/2m \Sigma_{vw} A_{vw} \delta(c_v, c_w) \quad (1)$$

where the δ-function $\delta(i, j)$ is 1 if $i = j$ and 0 otherwise, and $m = 1/2\Sigma_{vw} A_{vw}$ is the number of edges in the graph.This quantity will be large for good divisions of the network, but it is not, on its own a good measure of community structure since it takes its largest value of 1 in the trivial case where all vertices belong to a single community. However, if we subtract from this measure, the expected value of same quantity in case of randomized graph, then we do get a meaningful result.

The *degree* k_v of a vertex v is defined to be the number of edges incident upon it:

$$k_v = \Sigma_w A_{vw} \quad (2)$$

The probability of an edge existing between any two vertices v and w,in a random network is $k_v k_w/2m$. Now, we define the *Modularity* Q to be

$$Q = 1/2m \Sigma_{vw} [A_{vw} - k_v k_w/2m] \delta(c_v, c_w). \quad (3)$$

If the fraction of edges lying within community are no different from what we would expect in a total randomized network, then this quantity will be zero. Nonzero values represent deviations from the network. In practice, the value above 0.3 is a good indicator of the significant community structure in the network.

5.2 Algorithm

If high values of Q in equation (3) corresponds to good divisions of a network into communities, then we can find a good partitioning by searching through the possible candidates with high modularity. This approach seems to be computationally expensive, but some optimizations can be used to make it tractable. The algorithm proposed in [1] uses a greedy optimization, in which, starting with each vertex being a sole member of a community of one, repeatedly join together the two communities which produces the larges increase in Q. For a network of n vertices, after $n - 1$ iterations, only one community is left and the algorithm stops. The most straightforward implementation of this algorithm have a runtime complexity of $O(n^2)$. But using the efficient data structures like AVL tree representation of the graph, brings down the complexity to $O(nlog^2n)$.

In total, three data structures are maintained:

1. A sparse matrix containing ΔQ_{ij} for each pair i, j of communities with at least one edge between them. Store each row of the matrix as a balanced binary tree(in our case AVL trees) and as a max-heap.

2. max-heap H containing the largest element of each row of the matrix ΔQ_{ij} along with the labels i, j of the corresponding pairs of the communities

3. An ordinary vector array with elements a_i, which stores the fraction of end of edges that are attached to vertices in community i.

We start off with each vertex being the sole member of community of one.So, we initially set

$$\Delta Q_{ij} = 1/2m - k_i k_j/(2m)^2 \tag{4}$$
$$a_i = k_i/2m \tag{5}$$

for each i.

The algorithm is then defined as follows:

1. Calculate initial values of ΔQ_{ij} and a_i using equations (4) and (5), and populate the max-heap with the largest element of each row of the matrix ΔQ_{ij}.

2. Select the largest ΔQ_{ij} from H, join the corresponding communities, update the matrix ΔQ, the heap H and a_i and increment Q by ΔQ_{ij}.

3. Repeat step 2 until only the required number of communities are left.

To ensure that the anchor nodes always lie in different communities, we merge only those quantities in step(2), which preserve this criteria.

The update rules in step 2 above are as follows:

If community k is connected to both i and j, then

$$\Delta Q'_{jk} = \Delta Q_{ik} + \Delta Q_{jk} \tag{6}$$

If k is connected to i but not to j, then

$$\Delta Q'_{jk} = \Delta Q_{ik} - 2a_j a_k \tag{7}$$

If k is connected to j but not to i, then

$$\Delta Q'_{jk} = \Delta Q_{jk} - 2a_i a_k \tag{8}$$

If i and j are merged, then:

$$a'_j = a_j + a_i \tag{9}$$
$$a_i = 0 \tag{10}$$

5.3 Complexity Analysis

Let us denote the degrees of i and j as $|i|$ and $|j|$ respectively. The first operation in the step 2 of the algorithm is to update the jth row. To implement equation (6), we insert row i in row j, summing the elements if they exist in both rows. Since, each row is stored as AVL trees, each of these $|i|$ insertions takes $O(log|j|) \leq O(logn)$ time. We then update other elements of jth row, which are at most $|i| + |j|$ according to equations (7) and (8).All of this thus takes $O(|i| + |j|logn)$ time.
Update in Heap can be done in $O(logn)$ time. Since, at most $|i| + |j|$ updates are made, this can be accomplished in $O(|i| + |j|logn)$ time.
Finally, the update in (9) and (10) is trivial and can be done in constant time.
Since each join takes $O(|i| + |j|logn)$ time, the total running time is at most $O(logn)$ times the sum over all nodes of the *dendogram* of the degree of the corresponding communities. The *dendogram* represents the hierarchical decomposition of the network into communities and has a tree like structure..In the worst case, if the dendogram has depth d and since the total degree of all the vertices is $2m$, we have a running time of $O(mdlogn)$. For a sparse matrix like we have in our problem of citation index m n and $d = logn$. So total runtime is $O(nlog^2n)$.

5.4 Experimental Results

The data structures involved in the above algorithm are very complex, and need careful implementation. We are still in the process of implementing the algorithm, and would provide the results very soon.

6 Conclusion

In this report we present a new heuristic for the multi-terminal partitioning problem for the CiteSeer citation graph. Further application of the heuristic is required in order to test the performance of our algorithm.

There are several possible extensions of this project:

- We can study how the citation graph evolves over the time
- Use more reliable citation information - e.g., from the HTTP version of CiteSeer, or ACM DigitalLibrary
- Use more information when clustering the citation graph. Besides the year of publication we can use the names of the author or nontrivial words from the title to build connections with other papers.
- We can assist the partitioning program by specifying to which partitions some high degree nodes are mapped. By doing so, we might

obtain a better partitioning w.r.t. the right topic distribution of papers.

References

[1] Mixed integer linear programming benchmark - ftp://plato.asu.edu/pub/milpf.txt.

[2] R. K. Ahuja, T. L. Magnanti, and J. B. Orlin. Network flows. 1989.

[3] Y. Aumann and Y. Rabani. An o(log k) approximate min-cut max-flow theorem and approximation algorithm. 1998.

[4] S Banerjee and N Dutt. Efficient search space exploration for hw-sw partitioning. *ACM/IEEE CODES+ISSS*, pages 122–127, 2004.

[5] Gruia Calinescu, Howard J. Karloff, and Yuval Rabani. An improved approximation algorithm for multiway cut. In *ACM Symposium on Theory of Computing*, pages 48–52, 1998.

[6] Aaron Clauset, M. E. J. Newman, and Cristopher Moore. Finding community structure in very large networks. 2004.

[7] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Cliff Stein. *Introduction to Algorithms, 2nd edition.* MIT Press/McGraw-Hill, 2001.

[8] Marie-Christine Costa, Lucas Locart, and Fric Roupin. Minimal multicut and maximal integer multiflow: a survey. 2005.

[9] E. Dahlhaus, D. S. Johnson, C. H. Papadimitriou, P. D. Seymour, and M. Yannakakis. The complexity of multiterminal cuts. *SIAM J. Comput.*, 23(4):864–894, 1994.

[10] C. M. Fiduccia and R. M. Mattheyses. A Linear-Time Heuristic for Improving Network Partitions. In *Proceedings of the 19th ACM/IEEE Design Automation Conference DAC 82*, pages 175–181, 1982.

[11] John E. Hopcroft and Robert Endre Tarjan. Dividing a graph into triconnected components. *SIAM J. Comput.*, 2(3):135–158, 1973.

[12] B. Kernighan and S. Lin. B. kernighan and s. lin. an efficient heuristic procedure for partitioning graphs, bell systems technical j. pages 291–307, 1970.

[13] Tom Leighton and Satish Rao. Multicommodity max-flow min-cut theorems and their use in designing approximation algorithms. *J. ACM*, 46(6):787–832, 1999.