Abstract

Graphs are increasingly important and useful in our life as they represent social media, web blogs and networks. Graph statistics are used to obtain several properties of a graph. The graphs in real life are usually with billions of edges and nodes. This makes naive calculation of graph statistics expensive or impossible. Many algorithms are proposed to reduce the cost of the calculation.

All distance sketches (ADS) an important data structure to estimate a class of graph statistics. This was shown in a theoretical paper recently. In this thesis, we implement three flavors of ADS (bottom-k, k-mins, and k-partition respectively) and the naive method of calculating two statistics (distance distribution and closeness centrality). Based on experiments using a real graph dataset, we found that ADS are at least at two times faster than the naive method. Besides that, the accuracy of ADS is increasing with k.

Keywords: All distance sketches, MinHash, Graph statistics, Bottom-k, K-mins, K-partition

Acknowledgement

I would like to express my gratitude to my supervisor Dr. Grigorios Loukides, for his expert guidance, patience, and support throughout this project, without his help the project would not complete.

I would also like to thank my parents and friends who helped me a lot during the project. Furthermore, special thanks are also given to all teachers of the Department of Informatics, King's College London.

Table of contents

List of Table

List of Figure

Nomenclature

ADS	=	All Distance Sketches
BFS	=	Breadth First Search
DFS	=	Depth First Search
FIFO	=	First-In-First-Out
LIFO	=	Last-In-First-Out

1. Introduction

Graphs are an increasingly important in real life. Many data that is important to analyze comes in the form of graphs. In real life, graphs are used to connect with friends on social media like Facebook, Twitter, or search for a web page in search engine, where pages are linked with each other by hyperlink; each web page is a node of the graph, and the link between two pages is an edge. Besides that, graphs also can be used in biological and process modeling. They include construction of bonds in chemistry, medical research, disease pathologies and so on [24]. Moreover, graphs are also used in biology and conservation efforts, and nodes represent regions, and migration path or movement between the regions can represent as the edges.

Graph analytics are always used to analyze these graphs then we can understand the graph better. There are many types of graph analytics. It can be applied to find the centrality of the graph, or find groups of interacting people in a social network. However, these graphs are large, and computations on them will take much time, if they are done in naively, the cost of it will quite high especially for business use. How to make graph analytics more efficiency will be a problem. For instance, we want to find the most influential person among communities; the centrality analysis can be applied in this situation [23]. We can look the example as a network and then the influence of a node in the network can be calculated as the sum of the length of each shortest path from the node to all other nodes. As the formula (1) shows:

$$C = (n - 1) / \sum_{u \in V} d_{vu} \qquad (1)$$

In the centrality [21], n represents the number of nodes in the graph and d_{vu} the length of shortest path from node v to node u [5].

The task is simple if there are only a few individuals that need to be considered. However, in practice, there are always have many users. For example, Facebook had 2 billion monthly active users [3]. Similar concerns apply to the computation of other statistics, such as degree distribution. We will address the problem of efficient computation of closeness centrality and degree distribution. For these problems, we find some questions that reveal the motivation of the work:

1. Can sketches be used to solve the problem? If so, what sketches can be used?
2. How do they perform in terms of efficiency?
3. How do they perform in terms of accuracy?

Sketches are data structures based on hashing, and useful to summarize the compact data structures. The main algorithms we will use to estimate the statistic is all distance sketches (ADS). The ADS extends the simpler MinHash sketches [1] [9], and the ADS of a node v can be represented the union of MinHash sketches for all neighbors of v. There are three common flavors of MinHash, bottom-k [18], k-mins, and k- partition [6], which give three flavors of ADS, bottom-k, k-mins, and k-partition.

The ADS paper [6] is theoretical and shows the computation of generic statistics and tests the effectiveness and efficiency using simulation. We show how distance distribution and closeness centrality can specifically be calculated with ADS and what is the efficiency and effectiveness on a real social network dataset. Furthermore, we show an implementation which utilizes an efficient and easy-to-use a library called igraph to answer question 1. Moreover, questions two and three are based on the experiment and analysis. In the experiment, we found that k-partition is faster than bottom- k and k-mins. Moreover, bottom-k shows the highest accuracy among all sketches.

In Section 2, we will present background theories in the paper, such as shortest path, Dijkstra's algorithm, distance distribution, closeness centrality and MinHash sketches. Then we will detail about how ADS works in Section 3. The most important contributions of the thesis are the implementation of ADS using igraph and its use to compute closeness centrality and distance distribution (Section 4). In Section 5, we present experiments on the computation of closeness centrality and distance distribution (Section 5). The observation and discussion will follow in the next section. The last part is the conclusion and the future plan.

2. Background

2.1 Overview

The previous chapter described the basic background of the work and what kind of problem we need to solve and an organization of this paper. In this chapter, we will introduce the theory of graph, followed by definition the graph statistics we will use in the experiment. Then it will give some algorithms which associate with the ADS that can lead to a better understanding of ADS.

2.2 Graph

Graphs consist of vertices and edges. A graph represented many of nodes connected by edges. In mathematics, the graph can be represented in a data structure [26]. We consider $G = (V, E)$ repersents a graph, where V uses to store the vertices in the graph and E store the set of edges. (x, y) represents the edge from vertex x to vertex y. In the following graph (Figure 1) $V = \{ a, b, c, d, e \}$ and $E = \{ (a, b), (a, c), (b, d), (c, d), (d, e) \}$. A path in graph meaning a sequence of vertices connected by edges. The path of vertex a to vertex d in the following graph will be $path = \{ a, b, d \}$ or $\{ a, c, d \}$.

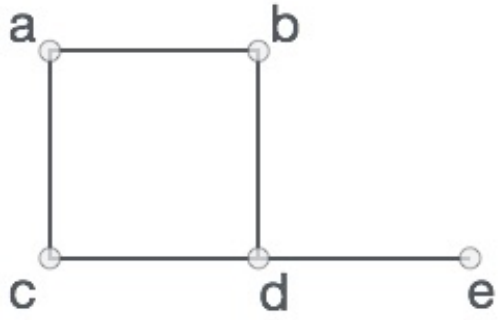


Figure 1 An undirected graph

There are many kinds of graphs in the data structure, the direction of edges may be directed or undirected, and the edges may be weighted or unweighted. The graph in Figure 1 is an undirected and unweighted graph. A directed graph means there is the distinction between two vertices linked with each edge. In a directed graph, there exist arrows showing the

direction of edges. In-degree and out-degree will be considered in a directed graph, the number of the vertex directed to another vertex is in out-degree, otherwise, is in-degree. The sum of out-degree or in-degree for all vertices in the graph will equal to the number of edges. The directed graph is quite useful in real life, for instance, there has a railway from city A to city B then the route from A to B will be represented by a directed edge.

As we have mentioned before, the edges also can be assigned a weight. The notation of $w(V1, V2) = 10$ means the weight of edge $(V1, V2)$ is 10. The weight may be represented as a cost, distance, or capacity between two vertices. It depends on the problem which one wants to be solved. We can use the example we have given above and added weight on the route from A to B represent the distance from A to B. Besides that, it also arises in many situations, such as shortest path problem, traveling salesman problem.

2.3 Naive Method

2.3.1 The Shortest Path

The shortest path problem is the problem of searching the smallest distance between two vertices which means minimizing the sum of each weight of the edges in this path [10]. The shortest path problem can be defined both in an undirected graph and a directed graph. The path from vertex v_1 to vertex v_n is $P = (v_1, v_2, \dots, v_n)$, then let $w(i, j)$ be the weight of the edge (i, j). The shortest path from v_1 to v_n is the path $P = (v_1, v_2, \dots, v_n)$ that over all the possible n minimizes the sum $\sum_{j=1}^{n-1} w(i, j)$. If each edge is equal in the graph, this is same as finding the path with fewest edges.

Many algorithms have been proposed to calculate the shortest path problem. Bellman-Ford, this algorithm was first proposed by Alfonso Shimbel in 1955 [11]. Also, Bellman – Ford is suitable for a graph which includes negative cycle. Edsger W. Dijkstra conceived Dijkstra's algorithm in 1956 and then published it in 1959 [12].

2.3.2 Bread First Search and Depth First Search

Bread First Search (*BFS*) is an algorithm for the traversing a tree or graph data structures [25] [27]. In a tree, it starts at the root and goes to the neighbor nodes (the child) first, before moving to the next level neighbors. It uses a queue (First-In-First-Out) to remember to get the next node to start a search. In a graph, we choose a node V and then find all neighbors of V considered as $Adj(v)$ after that we move to all neighbors of $Adj(v)$ that have not been visited yet.

Depth first search (DFS) is different from BFS that DFS search as far as possible from source vertex [27]. In a tree DFS from the root, while in a graph DFS is starting from an arbitrary vertex and using the stack (Last-In-First-Out) to record the node we have visited previously, or whether the node exists outgoing edges, retreat and try another path. If there are still unvisited vertices, repeat.

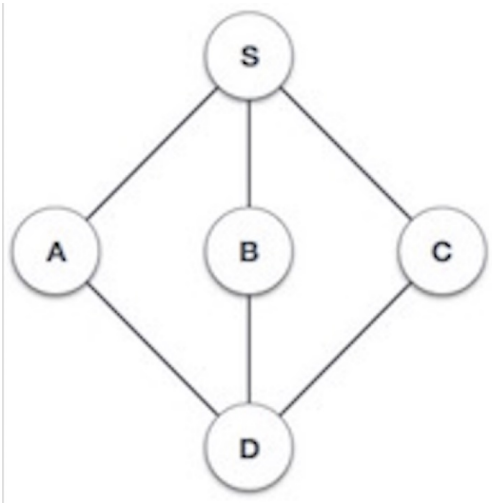


Figure 2. undirected graph

Example 1

There is an example to show the difference between *DFS* and *BFS*.

Consider the graph of Figure 2 to determine the traversal of this graph use in both BFS and DFS.

BFS: We starting from node S and mark it as visited. We then see an unvisited adjacent node from S is A, B, C. We choose A, mark it as visited and then repeat it until all the nodes have been visited. The order of BFS traversal is $\{S, A, B, C, D\}$.

DFS: In DFS we still starting from nodes S and mark it as visited and explore any unvisited adjacent node from S. We have three nodes in this example. We take node A in an alphabetical order and mark A as visited. Next, we explore any unvisited adjacent node from A. Both S and D are adjacent to A but we need to find unvisited node only, we choose node D and repeat it until all nodes has been visited. The result is $\{S, A, D, C\}$.

Comparing with BFS, DFS goes as far as possible while BFS traversal by level. In shortest path problem BFS and DFS both can use to solve this issue. The difference between these two algorithms is that DFS require less memory than BFS and useful in cycle detection. Moreover, BFS can only be used to find the shortest path in a graph if there are no loops and all edges have the same weight or no weight (just start from the source and perform a breadth first search and stop when we find the destination vertex). If the graph is more complex, containing weighted edges and loops, then we need a more sophisticated version of BFS, i.e., algorithm, Bellman-Ford.

2.3.3 Dijkstra's Algorithm

Dijkstra's Algorithm is an algorithm only used for searching the shortest path in a graph with non-negative weight [28]. This algorithm is based on BFS and can be used in a weighted and directed graph. We assume *S* as a set of known vertices and $I(u)$ means the cost of reaching this vertex from the source node. V is set of the node in the graph. Firstly, we select a source node and put it in $S = \{U\}$ and for all node in $(V - S)$ do $I(v) = w(u, v)$. Choosing a node from $(V - S)$ with lowest $I(x)$ to *S* and for all node $y \epsilon (V - S)$ do $I(y) = min(I(y), I(x) + w(x, y))$ until all nodes are added in *S*.

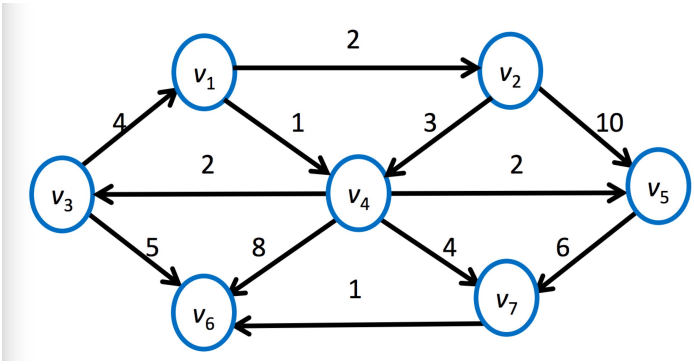


Figure 3 A directed and weighted graph

Example 2

We will give an example to explain how Dijkstra works. The detail of each step will be shown in next table.

Step	Set S	V-S
1	Select v1, S={v1} Start from V1	{V-S} = {v2, v3, v4, v5, v6, v7} v1->v2=2; v1->v4=1 v1->all other node = ∞ the shortest one is (v1, v4)
2	Select v4, S = {v1, v4} The shortest path is (v1, v4)	{V-S} = {v2, v3, v5, v6, v7} v1->v2=2; v1->v3=1+2=3 v1->v5=1+2=3; v1->v6=1+8=9 v1->v7=1+4=5 shortest path is (v1, v2)
3	Select v2, S = {v1, v4, v2} The shortest path is (v1, v4) (v1, v2)	{V-S} = {v3, v5, v6, v7} v1->v3=3; v1->v5=3 v1->v6=9; v1->v7=5 shortest path is (v4, v5)
4	Select v5, S = {v1, v4, v2, v5} The shortest path is (v1, v4), (v1, v2), (v4, v5)	{V-S} = {v3, v6, v7} v1->v3=3; v1->v6=9 v1->v7=5 shortest path is (v4, v3)
5	Select v3, S = {v1, v4, v2, v5, v3} The shortest path is (v1, v4), (v1, v2), (v4, v5), (v4, v3)	{V-S} = {v6, v7} v1->v6=3+5=8; v1->v7=5 shortest path is (v4, v7)
6	Select v3, S = {v1, v4, v2, v5, v3, v7} The shortest path is (v1, v4), (v1, v2), (v4, v5), (v4, v3), (v4, v7)	{V-S} = {v6} v1->v6=5+1=6 shortest path is (v7, v6)

Table 1 The result of Dijkstra's algorithms

The shortest path from node v1 to all other nodes of this graph is (v1, v2), (v4, v5), (v4, v3), (v4, v7), (v7, v6).

2.4 Graph Statistics

The statistics about the graph is important to study the properties of the graphs. There have been many kinds of statistics, like distance-based and clustering statistics [29]. They are usually used to estimate the most important of the nodes in the graph or some social media like Facebook; it always uses to recommend a friend to the user. The statistics we will use in work are distance distribution and closeness centrality.

2.4.1 Distance Distribution

Distance distribution is a tool used in data mining and useful to get the properties on social networks or sketches [6]. In a graph, the distance between a pair of nodes means the length of the shortest path between two nodes. The distance distribution in a graph G represents how many pairs of nodes are at distance d [2] [6]. We assume that there is a node v and the distance is d. $| N_d(v) |$ is the number of nodes is at most at distance d from node v. We set d_{vu} as the length of the shortest path from node v to node u, then we get the formula (2):

$$| N_d(v) | = \{ u | d_{vu} \leq d \} \quad (2)$$

2.4.2 Closeness Centrality

In networks, centrality shows how important the nodes in the graph. There are many measurements of centrality in networks, degree centrality, betweenness centrality and closeness centrality [4]. Degree centrality means a node has many neighbors is most important, it depends on the in-degree and out-degree of the node. The betweenness centrality of a graph means the node has the most number of pairs nodes need to go through this node to reach another node. The closeness centrality calculated the length of the average shortest path from the nodes to all other nodes [20], as we mentioned in formula (1). In our experiment, we will use this statistic as an application to show the efficiency and accuracy of ADS.

2.5 MinHash

MinHash is a kind of technic used for check the similarity of two sets. This technic was proposed by Andrei Broder [14]. MinHash can also be used for detecting how similar between two web pages. MinHash sketches is a locality sensitive hashing (LSH) scheme [1]. LSH is differing from other hash function; it maximizes the probability of collisions for similar individuals. There are three common flavors of MinHash are k-partition, k-mins, and bottom-k.

• A *k-mins* sketch [15], assigns random ranks in each elements k times and picks the smallest rank of each permutation. Considering one random rank assignment to ($T_1, T_2, \dots, T_n$), we get the elements with the smallest rank and we add them into $MinHash_1(i)$. Then, we create another random rank assignment and we add them into $MinHash_2(i)$. We repeat this *k* times and obtain $MinHash_1(i), MinHash_2(i), \dots, MinHash_k(i)$. Then, $MinHash(i) = MinHash_1(i) \cup MinHash_2(i) \cup \dots \cup MinHash_k(i)$.

• A *k- partition* sketch, assign a single random rank for each element and mapping of the element to *k* buckets, then select the smallest element in each bucket [24]. We assign a single random rank to ($T_1, T_2, \dots, T_n$), and separate all elements to *k* bucket by random. In each bucket, select an element with the smallest rank, $Bucket_i[i]$. Then, we get MinHash(i) $=\{ Bucket_i[i] \mid i \in (1 \sim k)\}$.

• A *bottom-k* sketch: assign a single random rank to each node and $MinHash(i)$ includes node j if the rank of j is one of the kth smallest ranks amongst nodes that are at least as close to i [6]. In other word, $j \epsilon MinHash(i)$ if rank of node *j*, r_j, is smaller than the kth smallest of nodes that are closer to *i* .than *j*.

All three flavors are the same when k=1.

3. All distance sketches

3.1 Overview

We have given a detail about the background knowledge of the work in the previous section in order to have a better understanding of ADS. In this section, we will give an example to explain how does bottom-k works in a real graph. We will implement the algorithm in a directed graph (Figure. 4) and explain it step by step.

3.2 Example

The notation we need to know in the example below example is: $\boldsymbol{N_d(v)}$ is the set of node are at most at distance *d* from node *v*.

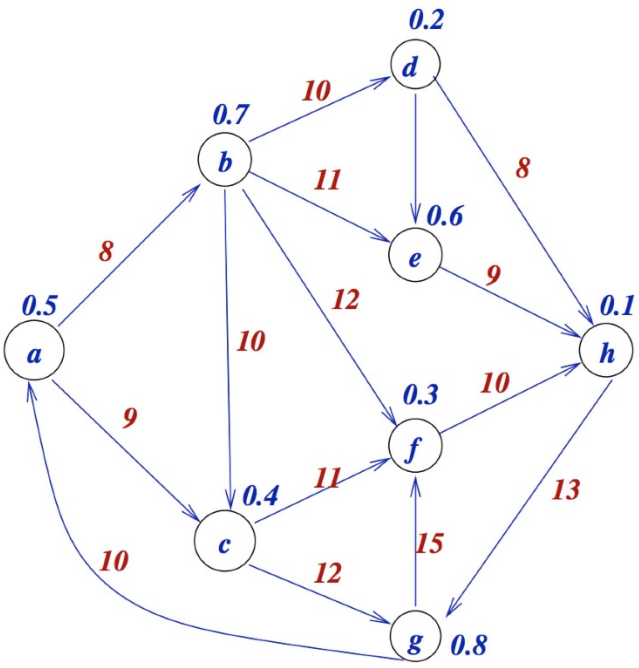


Figure 4 A directed and weighted graph

***Example* 2** For the Figure 4. The first step we need to do is sort shortest path distances from node a to all other nodes. The order will be $\boldsymbol{a, b, c, d, e, f, g}$ and the distance is

$(\mathbf{0}, \mathbf{8}, \mathbf{9}, \mathbf{18}, \mathbf{19}, \mathbf{20}, \mathbf{21}, \mathbf{26})$ respectively. Then we need to consider the random rank of each node to determine which node is included in the ADS of node a.

For k=1 and the node $\boldsymbol{a}$, we will show how to construct the All distance sketch. This sketch is the union of the MinHash sketches, where each MinHash sketch is constructed for a different possible distance d. So, we need to construct a MinHash sketch for d=0, another for d=1, …, another for the maximum possible distance. Each MinHash sketch will contain some nodes. Then, we will take the union of these nodes to construct the All distance sketch. Let's see how to do this step by step.

We start with d=0 and show how to construct the MinHash sketch. We first need to find the set of nodes $\boldsymbol{N_0(a)}$. This set contains all nodes that are within distance 0 from $\boldsymbol{a}$, $\boldsymbol{N_0(a) = \{a\}}$, since only $\boldsymbol{a}$ is at distance at most 0 from $\boldsymbol{a}$. Since k=1, $\boldsymbol{a}$ is contained in $\boldsymbol{N_0(a)}$ and there is no other node with smaller rank than the rank of $\boldsymbol{a}$, which is $\boldsymbol{r(a) = 0.5}$, we include $\boldsymbol{a}$ in the MinHash sketch.

We then go to d=1. $\boldsymbol{N_1(a) = \{a\}}$, since $\boldsymbol{a}$ is at distance at most 1 from $\boldsymbol{a}$ and there is no other node at distance at most 1 from $\boldsymbol{a}$ (beacuse the next closest node is at distance 9 from $\boldsymbol{a}$). Similarly, $\boldsymbol{N_2(a) = \cdots = N_7(a) = \{a\}}$. So, we need to construct one MinHash sketch containing $\boldsymbol{a}$ from d=2, another for d=3, …, another for d=7.

We then consider d=8. $\boldsymbol{N_8(a) = \{a, b\}}$. Since k=1, the MinHash sketch for d=8 will contain 1 node, which will be the node with the minimum rank. The rank of $\boldsymbol{a}$ is $\boldsymbol{r(a) = 0.5}$ and the rank of $\boldsymbol{b}$ is $\boldsymbol{r(b) = 0.7}$. Since $\boldsymbol{r(a) < r(b)}$, the MinHash sketch will contain $\boldsymbol{a}$.

We then consider d=9. $\boldsymbol{N_9(a) = \{a, b, c\}}$. The MinHash sketch for d=9 will contain $\boldsymbol{c}$, because the rank of c is $\boldsymbol{r(c) = 0.4}$, which is smaller than the ranks of the other nodes in $\boldsymbol{N_9(a)}$.

We then do the same for distance d=10,…,d=19, until d=20 (the farthest distance), $\boldsymbol{N_{20}(a) = \{a, b, c, d, e, f\}}$ and again the MinHash sketch for d=20 will contain the node $\boldsymbol{d}$ because the node with the minimum rank in $\boldsymbol{N_{20}(a) = \{a, b, c, d, e, f\}}$ is $\boldsymbol{d}$.

Finally, we get the result is ADS (a) = {(a, 0.5), (c, 0.4), (d, 0.2), (h, 0.1)}. If *k=2* in this graph, each time we need to choose the 2 smallest in the sorted order and the result includes {(a, 0.5), (b, 0.7), (c, 0.4), (d, 0.2), (f, 0.3), (h, 0.1)}.

4. Implementation

This section describes the development process of the components mentioned above, the problems encountered during the development and steps taken to overcome them. Firstly, the main library we have used in the program will be introduced and also the platform choice. After that, we will elaborate how the k-mins and bottom-k are used to estimate distance distribution and closeness centrality.

4.1 Igraph and Xcode

Igraph is open source and free for users. *Igraph* is a tool includes a lot of network analysis such as BFS, DFS, Dijkstra, and Bellman-Ford [17]. The important feature of *igraph* is high efficiency, strong portability, and ease of use. It can be used in R, Python and C/C++; we choose C++ for programming.

Mac OS X operating systems is selected as the implementation platform to implement algorithms. Since C++ was chosen as the programming language, *Xcode* becomes the best choice. For installation *igraph*, we used Homebrew (package management software which is a free and open source), the *homebrew/science/igraph* formula. The next step is to import *igraph* to the project in *Xcode*. We provided the link to the directory of *igraph.h* and *igraph library* in *Header Search Paths* and *Library Search Paths* respectively. Also then put three *lib-igraph files* in build phases. Preparations have been done; we can use *igraph* to start programming.

4.2 Naive Method

The naive method we mentioned in the paper is the shortest path (Algorithm 1). In order to estimate distance distribution and closeness centrality, we need to calculate the length of the shortest path from the source node to all other nodes. In this paper, we will use igraph as a tool to achieve the naive method and all graphs we input in work will non-negative weighted graph.

In line 5, ***igraph_get_shortest_path_dijkstra*** is used for calculating the length of shortest path between two nodes. The first argument of the function is an input graph and the second and third arguments can return either the Id of the vertices or the Id of the edges of the path along the shortest path. Then, we need offer the id of two nodes in fourth and fifth arguments. The last argument makes this function more flexible, it uses to specify the direction of edges in a directed graph. When we set is as IGRAPH_IN means calculate the reverse of the graph, while when the argument is set as IGRAPH_ALL, then the direction of the graph will be ignored and IGRAPH_OUT represents following the edges direction.

After we were getting the Id of edges, we use VECTOR () to index the edge and obtain the weight of the edge. The result is stored in an array, and then we can use the formula we mentioned in section2 to estimate statistics.

Algorithm 1 Naive Method

```
1:  function Naive_method (igraph_t *g, igraph_vector_t *weight)
2:      int array [|V|]; counter←0;
3:      for i←0, Length (|V|) do
4:          dis ← 0;
5:          igraph_get_shortest_path_dijkstra (&g, &vertices_of_path,
6:                          &edges_of_path, 0, i, &weight, IGRAPH_OUT);
7:          for j←0, Length (&edges_of_path) do
8:              int edges ← VECTOR(edges_of_path) [i];
9:              dis ← dis+VECTOR(weight)[edges];
10:             counter ++;
11:         array[counter] ← dis;
12: end function
```

4.3 Basic functions

In the program, the user needs to input the graph, distance d (the distance you want to estimate from the source node) and k. We can get the output from the file is the distance distribution of the graph and the closeness centrality. This part aims to decompose how to

achieve the sketch. First thing is read the graph from the file and then assign a random rank to each vertex. The core idea of bottom-k is to find the kth smallest rank in the MinHash sketches. Besides that, the most difficult function of this algorithm is getting all nodes within the distance d scaled from the source node.

Now we will start it from the first: reading the graph from the file. There exist some functions (<fstream>) in C++ to achieve this function, but they take too much time if the graph has millions of edge. Then, we found out Igraph has a function to read a graph from a file (Algorithm 2) which can simplify the code.

Algorithm 2 Read a graph from file

```
1:  function Read_Graph_From_File ()
2:      igraph_t g; //create a graph
3:      FILE ← *ifille; // a file object
4:      ifile ← fopen ("…/conpolblogs.ncol", "r"); //open file
5:      if (ifile==0) do
6:          printf (“File not found\n”);
7:          return 0;
8:      igraph_read_graph_ncol (&g, ifile, NULL, 0, IGRAPH_ADD_WEIGHTS_YES,
9:      IGRAPH_DIRECTED); // read file and set the graph as a weighted directed graph
10:  end function
```

In algorithm 2, the main function of reading file is igraph_read_graph_ncol, use for reading a nocl file by large graph layout. The argument in the function &g is a pointer to a graph which we will use in the function; the second one ifile is a pointer to a stream. The function can also set name to each vertex and assign weights to the edges. The last argument is used to set whether to create a directed graph. The time complexity of igraph_read_graph_ncol is O (|E| log(|V|) +|V|) (|V| is the number of vertices , |E| is the number of edges).

Rank assignment (Algorithm 3) is an important step in bottom-k. Each vertex of the graph needs a random weight that can be used in the later operation. We create a variable called “vrank” to store the rank. Srand () is a seed used to generate a random value by rand (), since every different seed will generate a different rank, we use time as the seed. We use

SETEANV to set a number to all vertices, and VECTOR () is used to index each vertex. In SETEANV, the first argument is the graph, the second one is the name of the attribute, and the last one contains the new value of the new attribute.

Algorithm 3 Assign a random rank for each vertex

```
1:  function Random_Rank_Assignment (igraph_t &g)
2:      igraph_vector_t vrank; // an attribute for rank
3:      igraph_vector_intit (&vrank, igraph_ecount(&g)); //initial the attribute
4:      srand(time(NULL));
5:      SETEANV (&g, "vrank", &vrank);
6:      for i← 0, Length(|V|) do
7:          VECTOR(vrank)[i] ← (double) rand () /RAND_MAX;
8:  end function
```

After setting a single random rank to each vertex, the program needs to get nodes by distance d which is entered by the user. ADS is using for graphs with large-scale, distance d uses for selecting a part of the graph. Now, we need to talk about how to calculate the shortest path first which is important to carry on algorithm 5.

In igraph there exist a *igraph_get_shortest_path* function that can get the shortest path between two nodes. The argument *&edge* in this function is a vector that return edges Id along this path. The *VECTOR ()* is used to index the weight of this path and calculate the sum of weights to obtain the length of this shortest path.

Algorithm 4 Get Shortest Path of Two Nodes

```
1:  function get_shortest_path (igraph_t *g, igraph_integer_t node,
2:                  igraph_vector_t * weight, int d)
3:      igraph_vector_t vertices, edges; // attributes for storing path properties
4:      int distance ←0;
5:      igraph_vector_init(&vertices, 0); igraph_vector_init(&edges, 0); //initial
6:      igraph_get_shortest_path (g, &vertices, &edges, 0, node, IGRAPH_OUT);
7:      for j←0, Length(&edges) do
8:          distance += VECTOR(edge)[j];
```

```
9:        if distance <d do
10:           return distance;
11:       else return -1;
12:   end function
```

In Algorithm 5, we create a recursive function called get_nodes_at_distance_d to selection the partial of the graph. Many methods have been tried to achieve this function. At first, BFS was used to traversal the graph, but in igraph, BFS does not work in a weighted graph. Then the callback function of igraph_bfs () was considered and combined with get_shortest_path() (Algorithm 4), but the structure of the whole algorithm is too complicated, and the callback function has some restriction, this method was abandoned. Finally, DFS was selected. In the function, &father is an argument store the parent of all nodes. Starting from the source node, finding the child of this node and calculate the shortest path between two nodes. If the result is smaller than distance d, record the node and go further to get the child of the node, otherwise go back to find the node's sibling and repeat it.

Algorithm 5 Get nodes at distance d

```
1:   function get_nodes_at_distance_d (igraph_t *g, igraph_integer_t node, int distance []
2:                   igraph_vector_t * weight, int d, igraph_vector_t &father)
3:       int dis ←0; counter ←0;
4:       for i←0, Length(&father) do
5:           if VECTOR(father)[i] == node do
6:               dis ← get_shortest_path (g, i, weight, d);
7:               if (dis!=-1) do
8:                   counter++;
9:                   distance[counter] ← dis;
10:                  get_nodes_at_distance_d (g, i, distance, weight, d, father); //call self
11:  end function
```

4.3.1 bottom-k

After finishing with all the steps above, we can start to estimate the sketches. Bottom-k flavor finds the k-th smallest rank in the sketch and compares it with the rank of the node which is being considered, then it determines whether the sketch should contain this node. In Algorithm 6, the array (which store the sketches), random rank (vrank) and k as parameters in the function. If k is smaller than the size of the array, we just contained the new node in the sketches, otherwise sort the array in the ascending order and return kth element.

Algorithm 6 Find kth smallest rank

```
1:   function find_kth_Smallest (int A [], int k, igraph_vector_t * vrank)
2:         if k> Length (A []) do
3:               rerutn -1;
4:         for i←0, Length (A []) do
5:               for j←0, Length (A []- i) do
6:                     if VECTOR (*vrank) [A[j]] >= VECTOR(*vrank) [A[j+1]]
7:                           swap (A [j], A[j+1]);
8:               return A[k-1];
9:    end function
```

4.3.2 K-mins flavor

K-mins flavors assign k times random rank for each node and each time picks a node with the smallest rank and then combine the result. The structure of most functions in k-mins are same with bottom-k, we can use *get_nodes_at_distance_d* and *get_shortest_path* to get all nodes at distance d and then execute k times rank assignments, each time use *find_kth_Smallest* (Algorithm 6) with k=1 to get the smallest node and store the sketches in an array.

Algorithm 7 K-mins

```
1:   function k-mins (int A [], igraph_vector_t * vrank)
2:         int array[igraph_vcount(&g)]; counter ← 0;
```

```
3:        srand (time(NULL));
4:        for i ← 0, Length (k), do
5:            for j ← 0, Length (igraph_vcount(&g)) do
6:                VECTOR(vrank)[j] = (long double) rand () / RAND_MAX;
7:            int n ← find_kth_Smallest (A [], 1, * vrank)
8:            array[counter] ← A [n];
9:    end function
```

4.3.3 k- partition

K-partition does a single rank assignment and separate elements into k bucket and picks the smallest rank in each bucket. To implement the algorithm, first, we need to allocate all items in k bucket. Then, finding the smallest rank in each bucket. We would use rand() function to assign a random number to each node to determine which bucket the node would be added. We also need to consider the value of k, if k greater than the number of nodes all nodes will be added to the sketches. Finally, we can get the result of the sketches.

Algorithm 8 K-partition

```
1:  function k_partition (int A [], igraph_vector_t *vrank)
2:      float section ← 1.0/k; //determine the node's bucket
3:      float bucket[k]=1; // store the smallest node in each bucket
4:      if k>Length(A[]) do
5:          break; //The result is A[]
6:      else do
7:          for i ← 0, Length (A []), do
8:              float r← rand() /RAND_MAX; // assign a random value
9:              if (VECTOR(vrank) [A[i]] < bucket[(int)(r/ section)]) do
10:                 bucket [(int)(r/ section)]← VECTOR(vrank) [A[i]];
11:   end function
```

4.4 Statistics estimated with ADS

Distance distribution is a statistic to estimate the total number of nodes at distance d, as we mentioned formula(2), $| \boldsymbol{N_d(v)} |$ = { u | $\boldsymbol{d_{vu} \leq d}$ }. Now we have an array which stores the result of the sketches with an ascending order. We use two loop to calculate it, the first represents distance d and growing one by one, the second loop traversal the array one by one and record the result by a counter.

When estimating the closeness centrality, we use the sum of node quantity at each distance, instead of the sum of the length of shortest path, the formula will change to

$$\boldsymbol{C = (N-1)/\sum_{i=0}^{d} | N_i(v) |},$$

moreover, the closeness of the source node will be easier to estimate.

Algorithm 9 Distance Distribution

```
1:  function distance_distribution (int A [])
2:      int counter ← 1; position ← 1;
3:      for i ← 0, Length (A []), do
4:          for j ← position, Length (A []) do
5:              if i > A [j] do
6:                  counter++;
7:                  pisition←j+1
8:              else break;
9:          print ("distance d=", i, "|Nd(v)|=", counter);
10: end function
```

In this chapter, we have introduced the tool we used and the platform we choose in the program and the functions about how to implement the sketches.

5. Experimental work

In this chapter, we will present an experiment to show the efficiency of ADS in two different flavors (K-mins, Bottom-k). Moreover, two statistics will be applied in the experiment distance distribution and closeness centrality respectively. We will use the value of k, distance d (the distance from source node) and the random rank of all nodes as parameters to consider their effect to time and relative error. We will answer the following questions for each of the two flavors and each of the two tasks (distance distribution and closeness centrality):

1. Are ADS faster than the naive method?
2. Are ADS sensitive to k and d?
3. What the accuracy of ADS compared to the naive method that always is 100% accurate?

5.1 Framework

5.1.1 Application

We use distance distribution and closeness centrality as the application in the experiment. For distance distribution, we calculate the number of nodes at most at distance d from the source node ($|N_{dv}|$ as the notation, $d\epsilon[0, d_{max}]$, where d_{max} means the distance maximum possible between *v* and another node in the graph). For closeness centrality, we use the sum of the number of nodes at the different distance to estimate the centrality of the source node (formula 1).

$$C(x) = (N-1)/\sum_{d=0}^{d=d_{max}} |N_{dv}| \qquad (3)$$

The naive method we used in this experiment is the Dijkstra's algorithm, this algorithm is a popular algorithm and uses in a non-negative weighted graph. In the experiment, we used *igraph_get_shortest_path_dijkstra* to compute it, and the function return edges Id of the path. Finally, we can calculate the value of the shortest path by Id. In order to estimate the distance distribution of the node, we need to sort the node by distance and count it one

by one. For closeness centrality, we can estimate it by formula (3). The time complexity of this algorithm is $O(|V| * (|E| + |V| \log|V|))$.

5.1.2 Experimental Dataset

Pol blogs dataset will be used in this test. This dataset was posted by Adamic and Glance. We can download it from http://www-personal.umich.edu/~mejn/netdata/. This graph is a directed graph and it represents the hyperlinks between weblogs on US politics. The number of nodes in this graph is 1213 ($|V| = 1213$) and the number of edges is 14408 ($|E| = 14408$).

5.1.3 Sampling

The graph we used in the test is an unweighted, directed graph. We will set each edge a random weight and random rank on each node. There are also two important parameters will be considered when we want to check the efficiency and accuracy of ADS. First, the value of k is the main parameter of ADS; it decides how many times to assign the random rank on each node in K-mins flavor and the kth smallest we need to find in bottom-k function. The value of k does not matter to the naive method. Second, the distance d will decide how many nodes will include in the estimator.

5.1.4 Relative error

The accuracy of ADS is a question in the experiment. We will use relative error to measure it. The formula of relative error is:

$$RE = \frac{|N(d) - N(d')|}{N(d)}$$

In the error function, $N(d)$ is the result of the naive method and $N(d')$ is the value estimate by ADS. Note: if $N(d)$ = 0 then we use another formula:

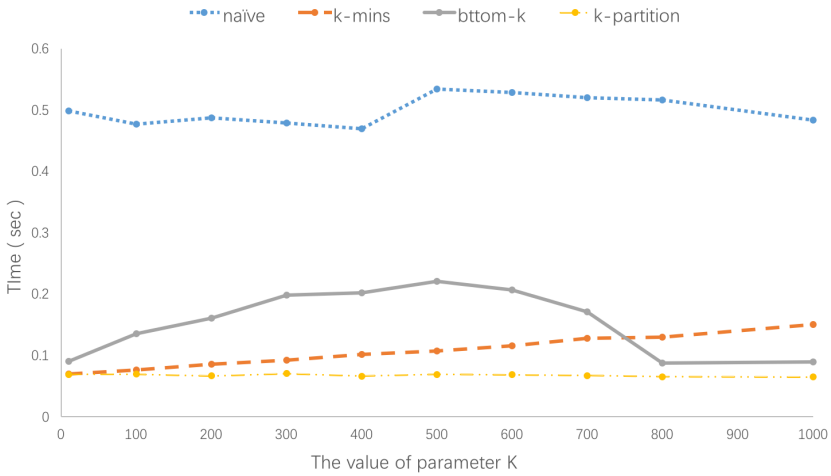
$$RE = \frac{|N(d) - N(d')| + 1}{N(d) + 1}$$

However, this case does not appear in out experiments.

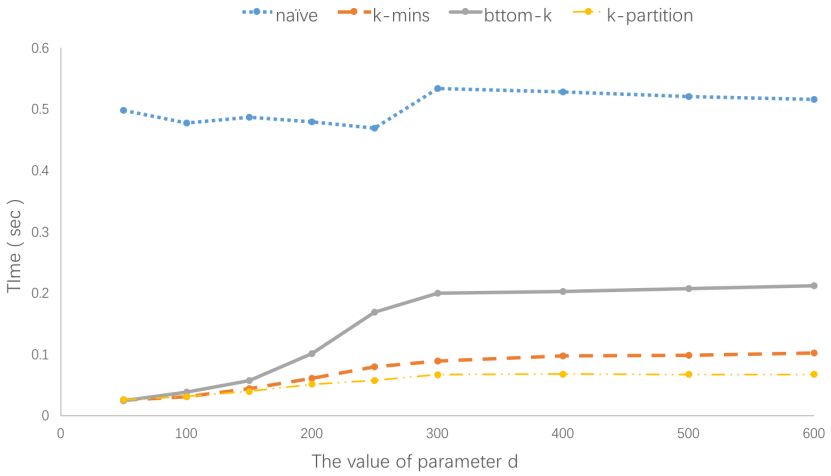
5.2 Result

5.2.1 Time

In this part, we test the running time of three algorithms (the naive method, bottom-k, k-mins, and k- partition respectively) in two tasks (distance distribution and closeness centrality). Moreover, considering whether the running time is sensitive to parameter d and k. Figure 5 shows the running time of distance distribution and illustrates the effect of k and d on time, respectively. The result of the experiment for closeness centrality is provided in Figure 6.



(a) Varying k for d=500, distance distribution



(b) Varying d for k=500, distance distribution

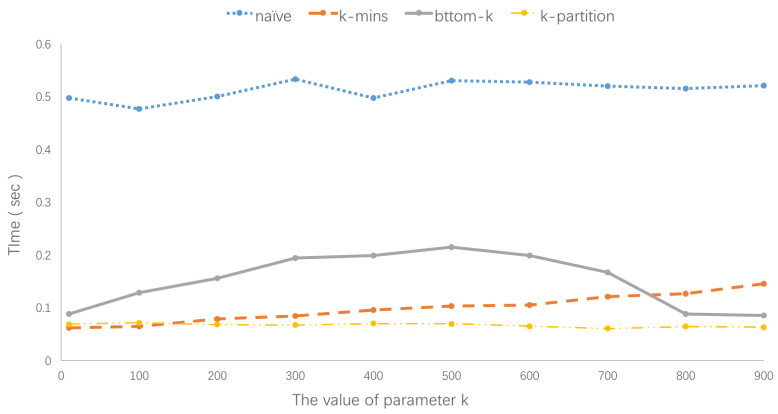
Figure 5 The result of varying k and d on the running time of distance distribution

What the plot shows above?

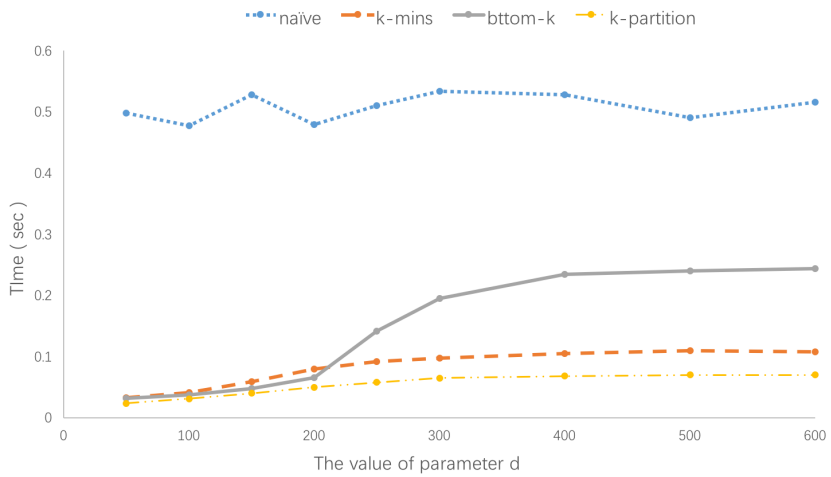
Two charts illustrate that the effect of distance d and k on the running time of distance distribution task. In the chart a, all flavors at least at two times and up to five times faster than the naive method. K-partition is the fastest flavor among all algorithms. Besides that, k-mins is faster than bottom-k at the begin. Both k-partition and k-mins becomes slower with k, whereas bottom-k is constant if k is big enough. For the chart b, both sketches become constant with d. The running time of bottom-k flavor increasing sharply with d and k-partition is faster than bottom-k and k-mins.

The reason for this result:

In a chart a, the naive method does the computation for all nodes, whereas the sketches for a selection of k nodes. K-mins becomes slower because k times we need to apply k-mins, each time different random assign. Larger k for k-mins means more random assignment. Larger k for bottom-k means less comparison. If the value of k is larger than the total number of the reachable node from the source node the algorithm does not need to find the kth smallest that's why bottom-k becomes constant. In k-partition, k means the number of buckets, if k is larger than the number of nodes all nodes will be contained in sketches. For plot b, the running time of sketches increases with d until all nodes are picked then both flavors become constant.



(a) Varying k for d=500, closeness centrality



(b) Varying d for k=500, closeness centrality

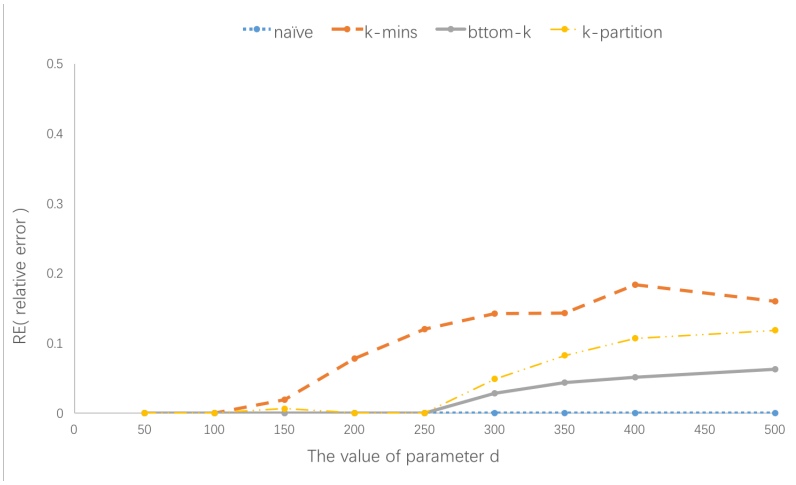
Figure 6 The result of varying k and d on the running time of closeness centrality

The plot leads us to the conclusion that the effect of parameter d and k on time are similar in both tasks. Bottom-k is slower than k-mins and k-partition is faster than all other algorithms the naive method is the slowest one.

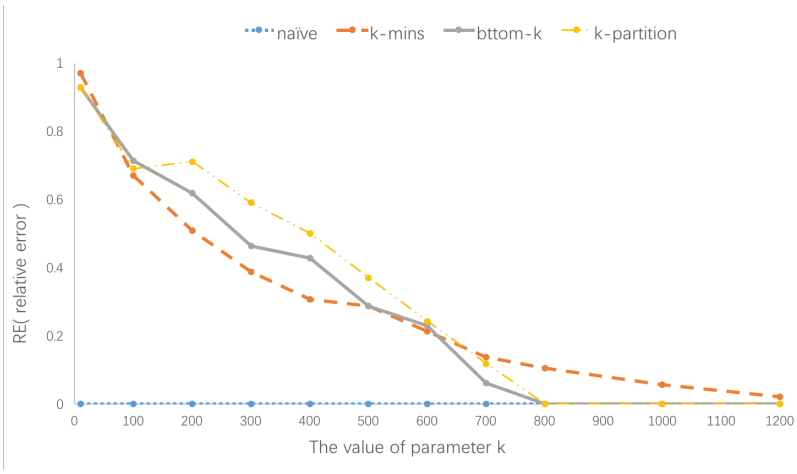
From above experiment, we can get the result that for larger k we will take more time to implement k-mins in comparison the running time of bottom-k becomes constant. If k is small, k-mins spends less time than bottom-k in this graph. In all situations, k-partition always faster than k-mins and bottom-k. Moreover, the effect of d and k in time are similar in both tasks.

5.2.2 Relative error

The relative error shows the accuracy of ADS in this section. The parameter d and k are considered to be the main variables to effect on relative error. We separate the experiment in two tasks, Figure 7 for distance distribution and Figure 8 for closeness centrality.



(a) Varying d for k=700, distance



(b) Varying k for d=500, distance distribution

Figure 7 The result of varying k and d on relative error of distance distribution

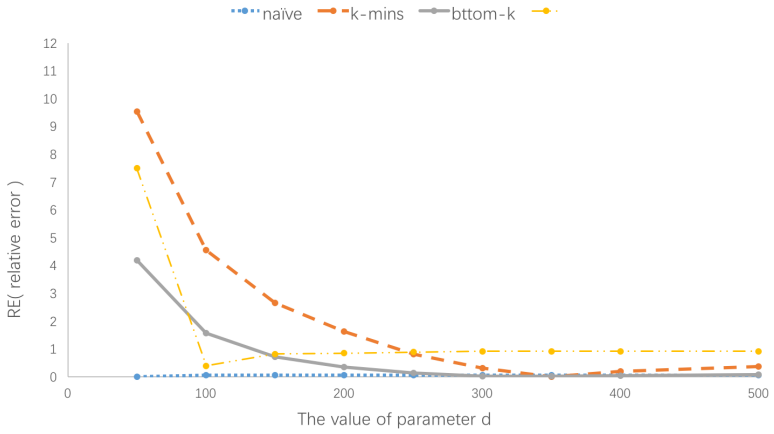
What the Figure 7 shows?

According to the chart a, the relative error of all flavors are lower than 20%, and the naive method always is 0. In the beginning, the relative error of sketches is 0 and then increases sharply with d. Bottom-k is more accurate than k-mins and k-partition. In the chart b, the relative error of both algorithms decreases sharply with k. Bottom-k and k-partition are more accurate than k-mins when k larger than 600 and the relative error of bottom-k and k-partition becomes 0 when k equals 800.

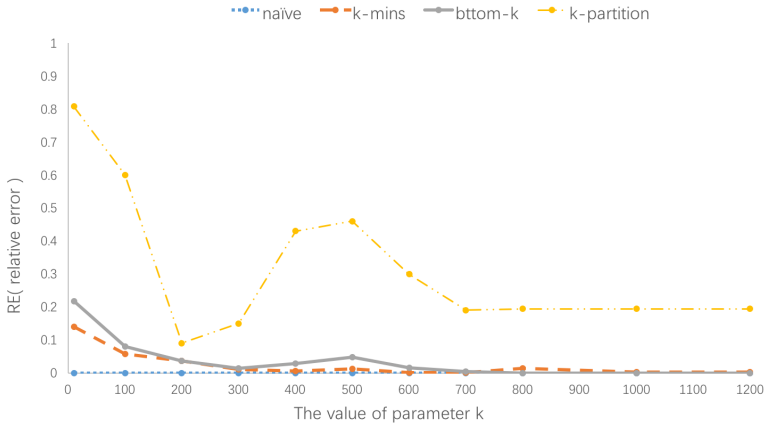
The reason for this result:

In the beginning, d with small value means the graph contain fewer nodes. In bottom-k, it picks up k smallest nodes, and all nodes would contain in sketches if the total number

of nodes in the graph is smaller than k. For k-partition, the sketches will also include all nodes if the number of the bucket (k) is greater than the total number of nodes. However, in k-mins, all nodes will assign k times random rank and then picks nodes by the rank, so the result is random. After many tests, bottom-k shows a higher accuracy in chart *a*. Comparing with the chart *b*, if we fixed the distance the relative error decreases with k. If k smaller than 600, bottom-k and k-partition show a higher relative error than k-mins, the accuracy of bottom-k increases with k, even to 100%. In conclusion, the relative error is sensitive to parameter d and k. Larger k leads to a higher accuracy in the same graph.



(a) Varying d for k=500, closeness centrality



(b) Varying k for d=500, closeness centrality

Figure 8 The result of varying k and d on relative error of closeness centrality

What the Figure 8 shows?

In the first plot, the relative error drops rapidly with d. When d equal 50, the relative error of k-mins is two times as much as bottom-k, and the value of the relative error is very high (almost 10). If d greater than 350, the relative error of bottom-k becomes 0 and k-mins fluctuate around 0 and k-partition shows the highest relative error. In chart b, k-

mins and bottom-k are decreases with k and the relative error of these are up to 20%. K-mins shows a slightly higher accuracy than bottom-k, both sketches become constant 0 when k greater than 800. However, the relative error of k-partition is fluctuating when k with a small value, and becomes constant when k greater than 700.

The reason for this result:

In plot *a*, we consider *d* as the main parameter and set *k=500*. In all sketches, we choose nodes at a distance at most *d* from the source node; more nodes will contain in the graph when d with a larger value, then the closeness of the source node is more accurate. For chart b, we set *d=500* and then pick *k* nodes each time, as we mentioned before if k greater than the number of nodes bottom-k will pick all nodes and result in no error in this task. However, in k-partition, each node is allocated to k bucket randomly then pick the smallest rank from each bucket are unstable especially when k is a small. In another word, the relative error of k-partition is high when k with small value.

In combination with time and the relative error of this experiments, all flavors are faster than the naive method in both statistics and k-partition is faster than k-mins and bottom-k. The accuracy of all flavors is increasing with k in a certain distance. However, the accuracy of k-partition is unstable when k is relatively small.

6. Conclusion

6.1 Overview

Graphs are very important in our lives and are getting larger. Graph statistics are used to find out useful properties about graphs. Example of graph statistics is degree distribution, which shows the ratio of nodes that have a certain degree, and closeness centrality of a node, which shows the central (or importance for some applications) of the node. However, as the graphs are getting larger, these statistics are taking a significant amount of time to compute. Thus, recently, a theoretical work ADS [6] suggested the use of ADS sketches to calculate a class of graph statistics.
In this thesis, we examined how distance distribution and closeness centrality can be computed with ADS sketches. We considered three flavors of sketches, k-mins, bottom-k, k-partition. We also applied these flavors to a larger real graph dataset which represents political blogs and their connections. Our results show that all flavors are at least two times and up to five times faster than the naive method for distance distribution and achieve very high accuracy. K- partition is the fastest flavor among all sketches. The relative of ADS is decreasing with k in a certain distance. It means more nodes are picked to estimate the fewer errors would occur.

In conclusion, the ADS algorithm is good as it can be applied to various networks, or some large graph to obtain the properties of the graph. The ADS can both save time and reduce the cost.

6.2 Future work

Although the paper has shown that is faster than the naive method, we feel that there are some limitations in work. There is a large scope for the future plan which is separated in several directions.

1. In the experimental work, we have tested the efficiency and accuracy of the ADS, but we have not mentioned the memory requirements of these algorithms. In the future work, whether these sketches are saving space can be considered.
2. Two statistics have been used as the application to show the performance of ADS (distance distribution and closeness centrality), more statistics could be applied in the sketches. For example, timed influence [7], distance-based and the degree statistics [8].
3. A larger dataset could be used as an input graph. In the experimental, we use pol blogs as dataset which contains 1200 nodes and 14000 edges; we can use a larger graph do the test and observe the relative error and the running time.
4. Edith Cohen proposed Historic Inverse Probability (HIP) estimators. The HIP estimator uses for estimate large class statistics with ADS, and this estimator is simple and flexible. It can be a part of the future plan.

It is clear that there are several tasks left for the future work. Therefore, the tasks are studied one by one.

REFERENCES

- [1] E. Cohen. MinHash Sketches: A Brief Survey 2016. full version: http://www.cohenwang.com/edith/Surveys/minhash.pdf
- [2] P. Crescenzi, R. Grossi, L. Lanzi, and A. Marino. A comparison of three algorithms for approximating the distance distribution in real-world graphs. In TAPAS, 2011.
- [3] Statista. Number of monthly active Facebook users worldwide as of 2nd quarter 2017 (in millions) https://www.statista.com/statistics/264810/number-of-monthly-active-facebook-users-worldwide/.
- [4] E. Cohen, D. Delling, T. Pajor, and R. F. Werneck. Computing classic closeness centrality, at scale. In COSN. ACM, 2014
- [5] A. Bavelas. Communication patterns in task oriented groups. Journal of the Acoustical Society of America, 22:271–282, 1950.
- [6] E. Cohen All-Distances Sketches, Revisited: HIP Estimators for Massive Graphs Analysis. In COSN. ACM, 2015
- [7] E. Cohen, D. Delling, T. Pajor, and R. F. Werneck. Timed influence: Computation and maximization. Technical Report cs.SI/1410.6976, arXiv, 2014.
- [8] M. Richardson and P. Domingos. Mining knowledge-sharing sites for viral marketing. In KDD. ACM, 2002.
- [9] P. Flajolet and G. N. Martin. Probabilistic counting algorithms for data base applications. J. Comput. System Sci., 31:182–209, 1985.
- [10] Cherkassky, Boris V.; Goldberg, Andrew V.; Radzik, Tomasz Shortest Paths Algorithms Theory and Experimental Evaluation Mathematical Programming. Ser. A. 73 (2): 129–174
- [11] T H Cormen C E Leiserson and R L Rivest Introduction to Algorithms MIT Press Cambridge MA 1990

- [12] Mehlhorn, Kurt; Sanders, Peter (2008). "Chapter 10. Shortest Paths" Algorithms and Data Structures: The Basic Toolbox. Springer. ISBN 978-3-540-77977-3.
- [13] E. Cohen. All-distances sketches, revisited: Scalable estimation of the distance distribution and centralities in massive graphs. Technical Report cs.DS/1306.3284, arXiv, 2013.
- [14] Border, Andrei Z. "On the resemblance and containment of documents", Compression and Complexity of Sequences: Proceedings, Positano, Amalfitan Coast, Salerno, Italy, June 11-13, 1997 (PDF), IEEE, pp.21-29
- [15] E. Cohen. Size-estimation framework with applications to transitive closure and reachability. J. Comput. System Sci., 55:441–453, 1997.
- [16] P. Flajolet, E. Fusy, O. Gandouet, and F. Meunier. Hyperloglog: The analysis of a nearoptimal cardinality estimation algorithm. In Analysis of Algorithms (AofA). DMTCS, 2007.
- [17] Gábor Csárdi, Tamás Nepusz, igraph Reference Manual http://igraph.org/c/doc/igraph-docs.pdf
- [18] E. Cohen and H. Kaplan. Summarizing data using bottom-k sketches. In ACM PODC, 2007.
- [19] P. Li, A. B. Owen, and C-H Zhang. One permutation hashing. In NIPS, 2012.
- [20] M. Rosenblatt. Remarks on some nonparametric estimates of a density function. The Annals of Mathematical Statistics, 27(3):832, 1956.
- [21] E. Cohen and H. Kaplan. Spatially-decaying aggregation over a network: Model and algorithms. J. Comput. System Sci., 73:265–288, 2007. Full version of a SIGMOD 2004 paper.
- [22] Mehlhorn, Kurt; Sanders, Peter (2008). "Chapter 10. Shortest Paths" Algorithms and Design: The Basic Toolbox. Springer. ISBN 978-3-540-77977-3.
- [23] Tauhid R. Zaman. Information Extraction with Network Centralities: Finding Rumor Sources, Measuring Influence, and Learning Community Structure Department of Electrical Engineering and Computer Science on August 17, 2011

- [24] S. VENU MADHAVA SARMA Assistant Professor of Mathematics K. L. UNIVERSITY Vaddeswaram. APPLICATIONS OF GRAPH THEORY IN HUMAN LIFE page 26

- [25] Keith Willman. Algorithms in C Robert Sedgewick Addison-Wesley publishing company ISBN 0-201-51425-7 chapter 29 chapter 32

- [26] gladius -topcoder member Introduction to Graphs and Their Data Structures: Section 1 Topcoder

- [27] Lawrence L. Larmore DFS and BFS Algorithms using Stacks and Queue UNLV http://www.egr.unlv.edu/~larmore/Courses/CSC477/bfsDfs.pdf

- [28] Yefim Dinitz. Rotem Itzhak. Hybrid Bellman–Ford–Dijkstra algorithm Ben-Gurion University of the Negev, POB 653, Beer-Sheva, ScienceDirect.com, 2017

- [29] Michael Brinkmeier and Thomas Schank Network Statistics Springer-Verlag Berlin Heidelberg 2005

APPENDICES

The naive method

```
1.  #include <igraph.h>
2.  #include <iostream>
3.  #include <stdlib.h>     /* srand, rand */
4.  #include <time.h>       /* time */
5.
6.  using namespace std;
7.
8.  //print the result
9.  void print_vector(igraph_vector_t *v, FILE *f) {
10.     long int i;
11.     for (i=0; i<igraph_vector_size(v); i++) {
12.         fprintf(f, " %li", (long int) VECTOR(*v)[i]);
13.     }
14.     fprintf(f, "\n");
15. }
16.
17. int main() {
18.
19.     //CAALCULATE TIME
20.     clock_t start,finish;
21.     double totaltime;
22.     start=clock();
23.
24.     igraph_t g;
25.     FILE *ifile;
```

```
26.
27.     igraph_vector_t weight;
28.
29.     //OPEN FILE
30.     ifile=fopen("/users/chenhuiping/Documents/ADSexample/ADSexamp
    le/conpolblogs.ncol", "r");
31.
32.     if (ifile==0) {
33.         printf("File not found\n");
34.         return 10;
35.     }
36.     igraph_i_set_attribute_table(&igraph_cattribute_table);
37.
38.     //READ A GRAPH FROM FILE
39.     igraph_read_graph_ncol(&g, ifile, NULL, 0, IGRAPH_ADD_WEIGHTS
    _YES, IGRAPH_DIRECTED);
40.
41.     //INIT
42.       igraph_vector_init(&weight, igraph_ecount(&g));
43.
44.     // SET WEIGHT
45.     EANV(&g,"weight",&weight);
46.
47.     //NOW I will  assign random weights to each edges
48.     for (int i=0; i<igraph_ecount(&g); i++)
49.     {
50.         int m=(rand()%100);
51.         VECTOR(weight)[i] = m+1;
52.     }
53.
54.
55.     // Let's do shortest path with the  weights
56.     // I will do a shortest path - here I prepare the arguments
```

```
57.     igraph_vector_t vertices_of_path;
58.     igraph_vector_init(&vertices_of_path, igraph_vcount(&g));
59.     igraph_vector_t edges_of_path;
60.     igraph_vector_init(&edges_of_path, igraph_ecount(&g));
61.
62.
63.     //THE number of nodes
64.     long int nodes=igraph_vector_size(&vertices_of_path);
65.
66.     //the value of shortest path for all vector
67.     int sum[nodes];
68.     for(int j=0;j<=nodes-1;j++)
69.     {
70.         sum[j]=0;
71.     }
72.         /*++++++++++++++++++++NAIVE METHOD++++++++++++++++++++++
    +++++*/
73.
74.
75.     //SET A COUNTER
76.     int counter=1;
77.     int dis;
78.
79. //      shortest path from node 0 to all nodes
80.
81.         cout<<"Distanced distribution:"<<endl;
82.
83.         for(int j=1;j<=nodes-1;j++){
84.             dis = 0;
85.
86.             //DIJKSTRA'S ALGORITHM
87.             igraph_get_shortest_path_dijkstra(&g, &vertices_of_pa
    th, &edges_of_path, 0, j, &weight, IGRAPH_OUT);
```

```
88.
89.
90.             for (int i=0; i<igraph_vector_size(&edges_of_path); i
    ++){
91.
92.                 int edges=VECTOR(edges_of_path)[i];
93.                 dis=dis+VECTOR(weight)[edges];
94.             }
95.             if(dis!=0){
96.                 sum[counter]=dis;
97.                 counter++;
98.             }
99.
100.          }
101.
102.     //sort by shortest path
103.
104.     for(int i=0;i<=counter-2;i++){
105.
106.         for(int j=counter-1; j>i; j--){
107.
108.             if(sum[i]>sum[j]){
109.                swap(sum[i],sum[j]);
110.             }
111.         }
112.     }
113.
114.     int d;
115.     cout<<"entry a distance d:";
116.     cin>>d;
117.     //distanced distribution
118.     int Ndv = 1;
119.     int k=1;
```

```
120.      int Nd[sum[counter-1]];
121.      //print the result
122.      for(int i=0;i<=d;i++){
123.
124.          for(int j=k;j<=counter-1;j++){
125.              if (i>=sum[j]){
126.                  Ndv=Ndv+1;
127.                  k=j+1;
128.              }
129.              else break;
130.          }
131.
132.          Nd[i]=Ndv;
133.          cout<<"When distance d="<<i<<", |Nd(v)| = "<<Ndv<<";"<<
    endl;
134.      }
135.
136.  /***************** Closeness centrality ********************/
137.
138.      cout<<endl<<"The Closeness centrality of graph G:"<<endl;
139.      int close;  //the sum of N_d(0)
140.      cout<<sum[counter-1]<<endl;
141.      for(int i=0;i<=sum[counter-1];i++){
142.
143.          close += Nd[i];
144.      }
145.
146.      double n;   // the number of node
147.      n = igraph_vcount(&g)*1.0-1;
148.      cout<<n/close<<endl;
149.      finish=clock();
150.      totaltime=(double)(finish-start)/CLOCKS_PER_SEC;
```

```
151.      cout<<endl<<"THE TIME OF RUNNING THIS CODE::"<<totaltime<<"
    SECONDS！"<<endl<<endl;
152.      fclose(ifile);
153.      return 0;
}
```

Bottom-k

```
1.  #include <igraph.h>
2.  #include <iostream>
3.  #include <stdlib.h>    /* srand, rand */
4.  #include <time.h>      /* time */
5.  using namespace std;
6.  //COUNTER FOR NODES AT DISTANCE D
7.  int h=1;
8.  //GET SHORTEST PATH
9.  int get_shortest_path(igraph_t *g, igraph_integer_t
    node,igraph_vector_t *weight,int d)
10. {
11. igraph_vector_t vertices,edges;
12. igraph_vector_init(&vertices, 0);
13. igraph_vector_init(&edges, 0);
14. int distance = 0;
15. //GET THE SHORTEST PATH EDGE ID
16. igraph_get_shortest_path(g, &vertices, &edges, 0,
    node,IGRAPH_OUT);
17. //CALCULATE THE WEIGHT OF SHORTEST PATH
18. for(int j=0;j<igraph_vector_size(&edges);j++){
19. int a =VECTOR(edges)[j];
20. distance=distance+VECTOR(*weight)[a];
21. }
22. if(distance<=d){
```

```
23. return distance;
24. }
25. else
26. return -1;
27. }
28. //GET NODE AT MOST AT DISTANCE D
29. void get_node_at_distance_d(int distance[][2],igraph_integer_t
    node,igraph_t *g,igraph_vector_t &father,igraph_vector_t
    *weight,int d)
30. {
31. int dis;

32. for(int i=0;i<igraph_vector_size(&father);i++){

33. if(VECTOR(father)[i]==node){
34. dis =  get_shortest_path(g, i, weight, d);

35. if(dis!=-1){
36. distance[h][0]=i;
37. distance[h][1]=dis;
38. h=h+1;
39. //CALL SELF
40. get_node_at_distance_d(distance, i, g, father,  weight, d);
41. }
42. }
43. }
44. }
45. //PRINT VECTOR
46. void print_vector(igraph_vector_t *v, FILE *f) {
47. long int i;
48. for (i=0; i<igraph_vector_size(v); i++) {
49. fprintf(f, " %li", (long int) VECTOR(*v)[i]);
50. }
```

```
51. fprintf(f, "\n");
52. }

53. //FIND Kth SMALLEST
54. float find_kSmall(int MinHash[][2], int k, igraph_vector_t
    *vrank,int h){
55. if(k>h+1){
56. return -1;

57. }
58. else{
59. for(int i=0;i<h;i++){
60. for(int j=0;j<h-i;j++){

61. if (VECTOR(*vrank)[MinHash[j][0]] >=
    VECTOR(*vrank)[MinHash[j+1][0]]){
62. swap(MinHash[j][0],MinHash[j+1][0]);
63. swap(MinHash[j][1],MinHash[j+1][1]);
64. }
65. }
66. }
67. return MinHash[k-1][0];
68. }
69. }
70. int main() {
71. //CAALCULATE TIME
72. clock_t start,finish;
73. double totaltime;
74. start=clock();

75. igraph_t g;
76. FILE *ifile;
```

```
77. igraph_vector_t weight,vrank;

78. // THE FILE
79. ifile=fopen("/users/chenhuiping/Documents/ADSexample/ADSexample/c
    onpolblogs.ncol", "r");
80. if (ifile==0) {
81. printf("File not found\n");
82. return 10;
83. }
84. igraph_i_set_attribute_table(&igraph_cattribute_table);

85. // READ GRAPH FROM FILE
86. igraph_read_graph_ncol(&g, ifile, NULL, 0,
    IGRAPH_ADD_WEIGHTS_YES, IGRAPH_DIRECTED);
87. //INIT
88. igraph_vector_init(&vrank, igraph_ecount(&g));
89. igraph_vector_init(&weight, igraph_ecount(&g));

90. //SEED OF RAND()
91. srand (time(NULL));
92. //NOW I will  assign random weights to each vector
93. SETEANV(&g, "vrank", &vrank);
94. for (int i=0; i<igraph_vcount(&g); i++){
95. VECTOR(vrank)[i] =  (double) rand() / RAND_MAX;
96. }
97. //ASSIGN WEIGHT
98. EANV(&g,"weight",&weight);
99. // Let's do shortest path with the  weights
100.   // I will do a shortest path - here I prepare the arguments
101.   igraph_vector_t vertices_of_path;
102.   igraph_vector_init(&vertices_of_path, igraph_vcount(&g));
103.   igraph_vector_t edges_of_path;
104.   igraph_vector_init(&edges_of_path, igraph_ecount(&g));
```

```
105.   // NOW i do a shortest path from vertex 0 to vertex 1
106.   // THE result vertices are in vertices_of_path
107.   // THE result edges are in edges_of_path
108.   //THE number of nodes
109.   long int nodes=igraph_vector_size(&vertices_of_path);
110.   //FIND NODES NEIGHBORS AND GET THE SHORTEST PATHE FROM SOURCE
    NODE
111.   igraph_vector_t v;
112.   igraph_vector_init(&v, 0);
113.   igraph_neighbors(&g, &v, 1, IGRAPH_ALL);
114.   //GET THE LENGTH OF TWO NODES
115.   igraph_vector_t vertices,edges;
116.   igraph_vector_init(&vertices, 0);
117.   igraph_vector_init(&edges, 0);
118.   //DISANTANCE
119.   int d;
120.   int k;
121.   cout<<"enter an integer distance d:";
122.   cin>>d;
123.   cout<<"enter an integer k:";
124.   cin>>k;
125.   //COUNTER FOR MINHASH
126.   int z=0;
127.   //***************bfs***************
128.   igraph_vector_t order, rank, father, pred, succ, dist;
129.   igraph_vector_init(&order, 0);
130.   igraph_vector_init(&rank, 0);
131.   igraph_vector_init(&father, 0);
132.   igraph_vector_init(&pred, 0);
133.   igraph_vector_init(&succ, 0);
134.   igraph_vector_init(&dist, 0);
```

```
135.   igraph_bfs(&g, /*root=*/0, /*roots=*/ 0, /*neimode=*/
    IGRAPH_OUT,
136.   /*unreachable=*/ 0, /*restricted=*/ 0,
137.   &order, &rank, &father, &pred, &succ, &dist,
138.   /*callback=*/ 0, /*extra=*/ 0);
139.   //CEATE AN ARRAY FOR SKETCHES
140.   int MH[igraph_vcount(&g)][2];
141.   MH[0][0]=0;
142.   MH[0][1]=0;
143.   for(int i=1;i<igraph_vcount(&g);i++)
144.   {
145.   MH[i][0]=-1;
146.   MH[i][1]=0;
147.   }
148.   //CREATE AN ARRAY SOTRE NODES AT MOST AT DISTTANCE D
149.   int distance[igraph_vcount(&g)][2];
150.   distance[0][0]=0;
151.   distance[0][1]=0;
152.   for(int i=1;i<igraph_vcount(&g);i++)
153.   {
154.   distance[i][0]=-1;
155.   distance[i][1]=0;
156.   }
157.   //CALCULATE NODES AT MOST AT DISTTANCE D
158.   get_node_at_distance_d(distance, 0, &g, father, &weight, d);
159.   int dists[h][2];
160.   // PUT THE RESULT IN A  SMALLER ARRAY
161.   for(int i=0;i<h;i++)
162.   {
163.   dists[i][0]=distance[i][0];
164.   dists[i][1]=distance[i][1];
165.   }
```

```
166.   //SORT ARRAY WHICH STORE THE NODES AT DISTANCE D IN A
    ASCENDING ORDER
167.   for(int i=0;i<h;i++)
168.   {
169.   for(int j=h-1;j>i;j--)
170.   {
171.   if (dists[i][1] >dists[j][1]){
172.   swap(dists[i][0],dists[j][0]);
173.   swap(dists[i][1],dists[j][1]);
174.   }
175.   }
176.   }
177.   //BOTTOM-K
178.   for(int i=1;i<h;i++)
179.   {
180.   //FINDE THE K-TH SMALLEST
181.   int node = dists[i][0];
182.   int n = find_kSmall(MH, k, &vrank, z);
183.   if(n==-1){
184.   MH[z+1][0] = node;
185.   MH[z+1][1] = dists[i][1];
186.   z++;
187.   }
188.   else{
189.   if(z+1<=k){
190.   MH[z+1][0] = node;
191.   MH[z+1][1] = dists[i][1];
192.   z++;
193.   }
194.   else{
195.   if(VECTOR(vrank)[node]<VECTOR(vrank)[n]){
196.   MH[z+1][0] = node;
197.   MH[z+1][1] = dists[i][1];
```

```
198.    z++;
199.    }
200.    }
201.    }

202.    }
203.    cout<<"Distanced distribution:"<<endl;
204.    for(int i=0;i<z;i++){
205.    int tmp;
206.    tmp = MH[i][1];

207.    for(int j=z; j>i; j--){
208.    if(tmp > MH[j][1]){
209.    MH[i][1] = MH[j][1];
210.    MH[j][1] = tmp;
211.    tmp = MH[i][1];
212.    swap(MH[i][0],MH[j][0]);
213.    }
214.    }
215.    }

216.    //distanced distribution
217.    int Ndv = 1;
218.    int y=1;
219.    int Nd[MH[z][1]];

220.    //print the result
221.    for(int i=0;i<=MH[z][1];i++){
222.    for(int j=y;j<=z;j++){
223.    if (i>=MH[j][1]){
224.    Ndv=Ndv+1;
225.    y=j+1;
226.    }
```

```
227.   else break;
228.   }
229.   Nd[i]=Ndv;
230.   cout<<"When distance d="<<i<<", |Nd(v)| = "<<Ndv<<";"<<endl;
231.   }

232.   /****************** Closeness centrality ****************/
233.   cout<<endl;
234.   cout<<"The Closeness centrality of graph G:"<<endl;
235.   int total=0;
236.   for(int i=0;i<=MH[z][1];i++){
237.   total+=Nd[i];
238.   }
239.   cout<<"For node 0"<<", Closeness ="<<((z)*1.0-1)/total<<endl;
240.   finish=clock();
241.   totaltime=(double)(finish-start)/CLOCKS_PER_SEC;
242.   cout<<"\n THE TIME OF RUNNING THIS
    CODE::"<<totaltime<<endl<<"SECONDS! "<<endl;
243.   fclose(ifile);
244.   return 0;
245.   }
```

K-mins

```
1.  #include <igraph.h>
2.  #include <iostream>
3.  #include <stdlib.h>    /* srand, rand */
4.  #include <time.h>      /* time */

5.  using namespace std;
```

```
6.  //COUNTER FOR NODES AT DISTANCE D
7.  int h=1;

8.  //GET SHORTEST PATH
9.  int get_shortest_path(igraph_t *g, igraph_integer_t
    node,igraph_vector_t *weight,int d)
10. {
11. igraph_vector_t vertices,edges;
12. igraph_vector_init(&vertices, 0);
13. igraph_vector_init(&edges, 0);

14. int distance = 0;

15. //GET THE SHORTEST PATH EDGE ID
16. igraph_get_shortest_path(g, &vertices, &edges, 0,
    node,IGRAPH_OUT);
17. //CALCULATE THE WEIGHT OF SHORTEST PATH
18. for(int j=0;j<igraph_vector_size(&edges);j++){
19. int a =VECTOR(edges)[j];
20. distance=distance+VECTOR(*weight)[a];
21. }
22. if(distance<=d){
23. return distance;
24. }
25. else
26. return -1;
27. }

28. //GET NODE AT MOST AT DISTANCE D
29. void get_node_at_distance_d(int distance[][2],igraph_integer_t
    node,igraph_t *g,igraph_vector_t &father,igraph_vector_t
    *weight,int d){
30. int dis;
```

```
31. for(int i=0;i<igraph_vector_size(&father);i++){
32. if(VECTOR(father)[i]==node){
33. dis =  get_shortest_path(g, i, weight, d);
34. if(dis!=-1){
35. distance[h][0]=i;
36. distance[h][1]=dis;
37. h=h+1;
38. get_node_at_distance_d(distance, i, g, father,  weight, d);
39. }
40. }
41. }

42. }

43. void print_vector(igraph_vector_t *v, FILE *f) {
44. long int i;
45. for (i=0; i<igraph_vector_size(v); i++) {
46. fprintf(f, " %li", (long int) VECTOR(*v)[i]);
47. }
48. fprintf(f, "\n");
49. }

50. //FIND THE SMALLEST RANK
51. float find_Smallest(int MinHash[][2],igraph_vector_t *vrank,int
    h){
52. for(int i=0;i<h;i++){
53. for(int j=h;j>i;j--){
54. if
    (VECTOR(*vrank)[MinHash[i][0]] >VECTOR(*vrank)[MinHash[j][0]]){
55. swap(MinHash[i][0],MinHash[j][0]);
56. swap(MinHash[i][1],MinHash[j][1]);
57. }
58. }
```

```
59. }
60. return MinHash[0][0];
61. }

62. int main() {

63. //CAALCULATE TIME
64. clock_t start,finish;
65. double totaltime;
66. start = clock();

67. igraph_t g;
68. FILE *ifile;

69. igraph_vector_t weight,vrank;

70. ifile=fopen("/users/chenhuiping/Documents/ADS_k_mins/ADS_k_mins/c
    onpolblogs.ncol", "r");

71. if (ifile==0) {
72. printf("File not found\n");
73. return 10;
74. }
75. igraph_i_set_attribute_table(&igraph_cattribute_table);

76. igraph_read_graph_ncol(&g, ifile, NULL, 0,
    IGRAPH_ADD_WEIGHTS_YES, IGRAPH_DIRECTED);

77. //INIT
78. igraph_vector_init(&vrank, igraph_ecount(&g));
79. igraph_vector_init(&weight, igraph_ecount(&g));

80. SETEANV(&g, "vrank", &vrank);
```

```
81. EANV(&g,"weight",&weight);

82. // Let's do shortest path with the  weights
83. // I will do a shortest path - here I prepare the arguments
84. igraph_vector_t vertices_of_path;
85. igraph_vector_init(&vertices_of_path, igraph_vcount(&g));
86. igraph_vector_t edges_of_path;
87. igraph_vector_init(&edges_of_path, igraph_ecount(&g));

88. //THE number of nodes
89. long int nodes=igraph_vector_size(&vertices_of_path);

90. //FIND NODES NEIGHBORS AND GET THE SHORTEST PATHE FROM SOURCE
    NODE

91. igraph_vector_t v;
92. igraph_vector_init(&v, 0);

93. igraph_neighbors(&g, &v, 1, IGRAPH_ALL);

94. //GET THE LENGTH OF TWO NODES

95. igraph_vector_t vertices,edges;
96. igraph_vector_init(&vertices, 0);
97. igraph_vector_init(&edges, 0);
98. //DISANTANCE
99. int d;
100.   int k;

101.   cout<<"enter an integer distance d:";
102.   cin>>d;
103.   cout<<"enter an integer k:";
104.   cin>>k;
```

```
105.   //COUNTER FOR MINHASH
106.   int z=0;
107.   int p;

108.   //***************bfs****************

109.   igraph_vector_t order, rank, father, pred, succ, dist;

110.   igraph_vector_init(&order, 0);
111.   igraph_vector_init(&rank, 0);
112.   igraph_vector_init(&father, 0);
113.   igraph_vector_init(&pred, 0);
114.   igraph_vector_init(&succ, 0);
115.   igraph_vector_init(&dist, 0);

116.   igraph_bfs(&g, /*root=*/0, /*roots=*/ 0, /*neimode=*/
    IGRAPH_OUT,
117.   /*unreachable=*/ 0, /*restricted=*/ 0,
118.   &order, &rank, &father, &pred, &succ, &dist,
119.   /*callback=*/ 0, /*extra=*/ 0);

120.   //    cout<<"**************** "<<igraph_vcount(&g)<<"
    ********"<<endl;


121.   //CREATE AN ARRAY SOTRE NODES AT MOST AT DISTTANCE D
122.   int distance[igraph_vcount(&g)][2];

123.   distance[0][0]=0;
124.   distance[0][1]=0;

125.   for(int i=1;i<igraph_vcount(&g);i++)
```

```
126.   {
127.   distance[i][0]=-1;
128.   distance[i][1]=0;
129.   }

130.   //CALCULATE NODES AT MOST AT DISTTANCE D
131.   get_node_at_distance_d(distance, 0, &g, father, &weight, d);

132.   //CEATE AN ARRAY FOR MINHASH
133.   int MH[igraph_vcount(&g)][2];
134.   MH[0][0]=0;
135.   MH[0][1]=0;
136.   for(int i=0;i<igraph_vcount(&g);i++){
137.   MH[i][0]=-1;
138.   MH[i][1]=0;
139.   }

140.   //K-mins
141.   srand( (unsigned)time(NULL));

142.   for(int i=0;i<k;i++)
143.   {
144.   p=0;
145.   //NOW I will  assign random weights to each vector
146.   for (int j=0; j<igraph_vcount(&g); j++)
147.   {
148.   VECTOR(vrank)[j] =  (long double) rand() / RAND_MAX;
149.   }

150.   //TEMPORARY ARRAY
151.   int TMH[igraph_vcount(&g)][2];
152.   TMH[0][0]=0;
153.   TMH[0][1]=0;
```

```
154.   for(int i=1;i<igraph_vcount(&g);i++){
155.   TMH[i][0]=0;
156.   TMH[i][1]=0;
157.   }

158.   for(int j=0;j<igraph_vcount(&g); j++){
159.   if(distance[j][0]==-1){
160.   break;
161.   }
162.   else{
163.   //FINDE THE K-TH SMALLEST
164.   int node = distance[j][0];
165.   int n = find_Smallest(TMH, &vrank, p);
166.   if(TMH[n][1]<=distance[j][1] &&
    VECTOR(vrank)[node]<=VECTOR(vrank)[n]){

167.   TMH[p+1][0] = node;
168.   TMH[p+1][1] = distance[j][1];
169.   if(MH[node][0]==-1){
170.   MH[node][0]=node;
171.   MH[node][1]=distance[j][1];
172.   z++;
173.   }
174.   p++;
175.   }
176.   }
177.   }

178.   }
179.   int MHF[z][2];
180.   int u=0;
181.   for(int i=0;i<igraph_vcount(&g);i++){
```

```
182.   if(MH[i][0]!=-1){
183.   MHF[u][0]=MH[i][0];
184.   MHF[u][1]=MH[i][1];
185.   u++;
186.   }
187.   }

188.   /*++++++++++++++DISTANCE DISTRIBUTION++++++++++++*/

189.   cout<<"Distanced distribution:"<<endl;
190.   for(int i=0;i<z;i++){
191.   int tmp;
192.   tmp = MHF[i][1];

193.   for(int j=z; j>i; j--){
194.   if(tmp > MHF[j][1]){
195.   MHF[i][1] = MHF[j][1];
196.   MHF[j][1] = tmp;
197.   tmp = MHF[i][1];
198.   swap(MHF[i][0],MHF[j][0]);
199.   }
200.   }
201.   }


202.   //distanced distribution

203.   int Ndv = 0;
204.   int y=1;
205.   int Nd[MHF[z][1]];
206.   //print the result
207.   for(int i=0;i<=MHF[z][1];i++){
208.   for(int j=y;j<=z;j++){
```

```
209.   if (i>=MHF[j][1]){
210.   Ndv=Ndv+1;
211.   y=j+1;
212.   }
213.   else break;
214.   }
215.   Nd[i]=Ndv;
216.   cout<<"When distance d="<<i<<", |Nd(v)| = "<<Ndv<<";"<<endl;
217.   }

218.   /****************** Closeness centrality **************/


219.   cout<<endl<<"The Closeness centrality of graph G:"<<endl;
220.   int close;  //the sum of N_d(0)

221.   for(int i=0;i<=MHF[z][1];i++){
222.   close += Nd[i];
223.   }

224.   double n;   // the number of node
225.   n = Nd[MHF[z][1]]*1.0-1;
226.   cout<<n<<","<<close<<endl;
227.   cout<<n/close<<endl;

228.   finish=clock();
229.   totaltime=(double)(finish-start)/CLOCKS_PER_SEC;
230.   cout<<"\nTHE TIME OF RUNNING THIS
    CODE::"<<totaltime<<endl<<"SECONDS! \n"<<endl;
231.   fclose(ifile);
232.   return 0;
233.   }
```

K-partition

```
1.  #include <igraph.h>
2.  #include <iostream>
3.  #include <stdlib.h>    /* srand, rand */
4.  #include <time.h>      /* time */

5.  using namespace std;

6.  //COUNTER FOR NODES AT DISTANCE D
7.  int h=1;

8.  //GET SHORTEST PATH
9.  int get_shortest_path(igraph_t *g, igraph_integer_t
    node,igraph_vector_t *weight,int d){
10. igraph_vector_t vertices,edges;
11. igraph_vector_init(&vertices, 0);
12. igraph_vector_init(&edges, 0);

13. int distance = 0;
14. //GET THE SHORTEST PATH EDGE ID
15. igraph_get_shortest_path(g, &vertices, &edges, 0,
    node,IGRAPH_OUT);

16. //CALCULATE THE WEIGHT OF SHORTEST PATH
17. for(int j=0;j<igraph_vector_size(&edges);j++){
18. int a =VECTOR(edges)[j];
19. distance=distance+VECTOR(*weight)[a];
20. }
```

```
21. if(distance<=d){
22. return distance;
23. }
24. else
25. return -1;
26. }

27. //GET NODE AT MOST AT DISTANCE D
28. void get_node_at_distance_d(int distance[][2],igraph_integer_t
    node,igraph_t *g,igraph_vector_t &father,igraph_vector_t
    *weight,int d)
29. {
30. int dis;
31. for(int i=0;i<igraph_vector_size(&father);i++){
32. if(VECTOR(father)[i]==node){
33. dis =  get_shortest_path(g, i, weight, d);
34. if(dis!=-1){
35. distance[h][0]=i;
36. distance[h][1]=dis;
37. h=h+1;
38. get_node_at_distance_d(distance, i, g, father,  weight, d);
39. }
40. }
41. }

42. }

43. void print_vector(igraph_vector_t *v, FILE *f) {
44. long int i;
45. for (i=0; i<igraph_vector_size(v); i++) {
46. fprintf(f, " %li", (long int) VECTOR(*v)[i]);
47. }
48. fprintf(f, "\n");
```

```
49. }

50. //FIND THE SMALLEST RANK
51. float find_Smallest(int bucket[][3],igraph_vector_t *vrank,int
    p,int o){
52. for(int i=p;i<=o;i++){
53. for(int j=o;j>i;j--){
54. if (VECTOR(*vrank)[bucket[i][1]] >VECTOR(*vrank)[bucket[j][1]]){
55. swap(bucket[i][2],bucket[j][2]);
56. swap(bucket[i][1],bucket[j][1]);
57. }
58. }
59. }
60. return bucket[p][1];
61. }

62. int main() {

63. //CAALCULATE TIME
64. clock_t start,finish;
65. double totaltime;
66. start = clock();

67. igraph_t g;
68. FILE *ifile;

69. igraph_vector_t weight,vrank;

70. ifile=fopen("/users/chenhuiping/Documents/ADS_k_mins/ADS_k_mins/c
    onpolblogs.ncol", "r");

71. if (ifile==0) {
72. printf("File not found\n");
```

```
73. return 10;
74. }
75. igraph_i_set_attribute_table(&igraph_cattribute_table);

76. igraph_read_graph_ncol(&g, ifile, NULL, 0,
    IGRAPH_ADD_WEIGHTS_YES, IGRAPH_DIRECTED);

77. //INIT
78. igraph_vector_init(&vrank, igraph_ecount(&g));
79. igraph_vector_init(&weight, igraph_ecount(&g));
80. srand (time(NULL));

81. SETEANV(&g, "vrank", &vrank);
82. EANV(&g,"weight",&weight);

83. for (int j=0; j<igraph_vcount(&g); j++){
84. VECTOR(vrank)[j] =  (long double) rand() / RAND_MAX;
85. }
86. // Let's do shortest path with the  weights
87. // I will do a shortest path - here I prepare the arguments
88. igraph_vector_t vertices_of_path;
89. igraph_vector_init(&vertices_of_path, igraph_vcount(&g));
90. igraph_vector_t edges_of_path;
91. igraph_vector_init(&edges_of_path, igraph_ecount(&g));

92. //GET THE LENGTH OF TWO NODES
93. igraph_vector_t vertices,edges;
94. igraph_vector_init(&vertices, 0);
95. igraph_vector_init(&edges, 0);

96. //DISANTANCE
97. int d;
98. int k;
```

```
99. cout<<"enter an integer distance d:";
100.   cin>>d;
101.   cout<<"enter an integer k:";
102.   cin>>k;

103.   //***************bfs***************
104.   igraph_vector_t order, rank, father, pred, succ, dist;

105.   igraph_vector_init(&order, 0);
106.   igraph_vector_init(&rank, 0);
107.   igraph_vector_init(&father, 0);
108.   igraph_vector_init(&pred, 0);
109.   igraph_vector_init(&succ, 0);
110.   igraph_vector_init(&dist, 0);

111.   igraph_bfs(&g, /*root=*/0, /*roots=*/ 0, /*neimode=*/
    IGRAPH_OUT,
112.   /*unreachable=*/ 0, /*restricted=*/ 0,
113.   &order, &rank, &father, &pred, &succ, &dist,
114.   /*callback=*/ 0, /*extra=*/ 0);

115.   //CREATE AN ARRAY SOTRE NODES AT MOST AT DISTTANCE D
116.   int distance[igraph_vcount(&g)][2];
117.   distance[0][0]=0;
118.   distance[0][1]=0;

119.   for(int i=1;i<igraph_vcount(&g);i++){
120.   distance[i][0]=-1;
121.   distance[i][1]=0;
122.   }

123.   //CALCULATE NODES AT MOST AT DISTTANCE D
```

```
124.   get_node_at_distance_d(distance, 0, &g, father, &weight, d);
125.   int counter=0;
126.   //CEATE AN ARRAY THE NODES AT DISTANCE D
127.   int MHF[igraph_vcount(&g)][2];
128.   MHF[0][0]=0;
129.   MHF[0][1]=0;
130.   for(int i=0;i<igraph_vcount(&g);i++){
131.   if(distance[i][0]!=-1){
132.   MHF[counter][0]=distance[i][0];
133.   MHF[counter][1]=distance[i][1];
134.   counter++;
135.   }
136.   }
137.   //USEFOR K-PARTITION
138.   int MH[igraph_vcount(&g)][2];
139.   MH[0][0]=0;
140.   MH[0][1]=0;
141.   for(int i=1;i<igraph_vcount(&g);i++){
142.   MH[i][0]=-1;
143.   MH[i][1]=0;
144.   }
145.   //K-PARTITION
146.   int c_MH=0;
147.   for(int j=0;j<h;j++){
148.   if(k>c_MH){
149.   MH[MHF[j][0]][0]=MHF[j][0];
150.   MH[MHF[j][0]][1]=MHF[j][1];
151.   c_MH++;
152.   }
153.   else{
154.   //USE TO STORE THE MINIMUM NODE IN EACH BUCKET
155.   float min_bucket[k][3];
156.   for(int i=0;i<k;i++){
```

```
157.   min_bucket[i][0]=1.0;
158.   min_bucket[i][1]=0;
159.   min_bucket[i][2]=0;
160.   }
161.   float flag= 1.0/k;
162.   // ASSIGN BUCKET RANDOMLY
163.   srand( (unsigned)time( NULL ) );
164.   for(int i=0;i<=j;i++){
165.   float b=  (float) rand() / RAND_MAX;
166.   int bucket_number=int(b/flag);
167.   if(VECTOR(vrank)[MHF[i][0]]<min_bucket[bucket_number][0]){
168.   min_bucket[bucket_number][0]=VECTOR(vrank)[MHF[i][0]];
169.   min_bucket[bucket_number][1]=MHF[i][0];
170.   min_bucket[bucket_number][2]=MHF[i][1];
171.   }
172.   }
173.   for(int t=0;t<k;t++){
174.   int node_min=min_bucket[t][0];
175.   int node=min_bucket[t][1];
176.   if(node_min==1.0){
177.   break;
178.   }
179.   else{
180.   if(MH[node][0]==-1){
181.   MH[node][0]=node;
182.   MH[node][1]=min_bucket[t][2];
183.   c_MH++;
184.   }
185.   }
186.   }
187.   }
188.   }
189.   //CEATE AN ARRAY THE NODES AT DISTANCE D
```

```
190.   int MHB[c_MH][2];
191.   MHB[0][0]=0;
192.   MHB[0][1]=0;
193.   int c=0;
194.   for(int i=0;i<igraph_vcount(&g);i++){
195.   if(MH[i][0]!=-1){
196.   MHB[c][0]=MH[i][0];
197.   MHB[c][1]=MH[i][1];
198.   c++;
199.   }
200.   }
201.   /*++++++++++++++++++++DISTANCE
    DISTRIBUTION++++++++++++++++++++++++++++*/
202.   cout<<"Distanced distribution:"<<endl;
203.   for(int i=0;i<c_MH;i++){
204.   int tmp;
205.   tmp = MHB[i][1];
206.   for(int j=c_MH-1; j>i; j--){
207.   if(tmp > MHB[j][1]){
208.   MHB[i][1] = MHB[j][1];
209.   MHB[j][1] = tmp;
210.   tmp = MHB[i][1];
211.   swap(MHB[i][1],MHB[j][1]);
212.   swap(MHB[i][0],MHB[j][0]);
213.   }
214.   }
215.   }
216.   //distanced distribution
217.   int Ndv = 1;
218.   int y=1;
219.   int Nd[MHB[c_MH-1][1]];
220.   //print the result
221.   for(int i=0;i<=MHB[c_MH-1][1];i++){
```

```
222.   for(int j=y;j<c_MH;j++){
223.   if (i>=MHB[j][1]){
224.   Ndv=Ndv+1;
225.   y=j+1;
226.   }
227.   else break;
228.   }
229.   Nd[i]=Ndv;
230.   cout<<"When distance d="<<i<<", |Nd(v)| = "<<Ndv<<";"<<endl;
231.   }

232.   //   /******************* Closeness centrality ************/
233.   cout<<endl<<"The Closeness centrality of graph G:"<<endl;
234.   int close;  //the sum of N_d(0)
235.   for(int i=0;i<=MHB[c_MH-1][1];i++){
236.   close += Nd[i];
237.   }
238.   double number;   // the number of node500
239.   number = Nd[MHB[c_MH-1][1]]*1.0-1;
240.   cout<<number/close<<endl;

241.   finish=clock();
242.   totaltime=(double)(finish-start)/CLOCKS_PER_SEC;
243.   cout<<"\nTHE TIME OF RUNNING THIS
    CODE::"<<totaltime<<endl<<"SECONDS! \n"<<endl;
244.   fclose(ifile);
245.   return 0;
246.   }
```