Optimizing the Shortest Path Query on Large-Scale Dynamic Directed Graph

Phuong-Hanh DU Hai-Dang PHAM Ngoc-Hoa NGUYEN
Department of Information System
VNU University of Engineering and Technology, Hanoi, Vietnam
Email: {hanhdp, dangph, hoa.nguyen}@vnu.edu.vn

ABSTRACT

This paper presents our approach in order to optimize the shortest path query on a large-scale directed, dynamic graph such as a social network. For this kind of graph, edges can be dynamically added or removed while a user asks to determine the shortest path between two vertices. To solve this problem, we propose a strategic solution based on (i) an appropriate data structure, (ii) the optimized update actions (insertions and deletions) and (iii) by improving the performance of query processing by both reducing the searching space and computing in multithreaded parallel. Thus, graph is globally organized by the adjacent lists in order to improve the cache hit ratio and the update action performance. The reduction of searching space is based on the way of calculating the potential enqueued vertices. Cilkplus is chosen to parallelize the consecutive queries. Our strategy was validated by the datasets from SigMod Contest 2016 and SNAP DataSet Collection with the good experimental results.

Keywords

bi-directional search, large-scale directed and dynamic graph, multithreaded parallel computing

1. INTRODUCTION

Nowadays, we live on the networked society that networks are ubiquitous. These networks, such as social networks, biology networks, content distribution networks, transportation networks and semantic networks, have their data arising very quickly large-scale. To handle the big data challenge, the graph based approach is considered the most natural and suitable one [2]. By using the graph theory, vertices are commonly used to represent the entities, and edges are designed to denote the interactions between them. In the graph, the optimal (shortest) path problem is the problem of finding a path between two vertices such that the cost of its constituent edges is minimized. This is a fundamental and well-studied combinatorial optimization problem with many practical uses: from GPS navigation to routing schemes in computer networks; search engines apply solutions to this problem on website interconnectivity graphs and social networks apply them on graphs of peoples' relationships [3]. This problem is normally trivial, but in the context of having large-scale and quickly changing/elastic graph in reality, the task to answer optimal path queries is a big challenge [8].

In this paper, we present our solution in order to improve the performance of optimizing the query on large-scale directed, no-weighted and elastic graph. Our strategy is based on the ideas: (i) appropriate data structure, (ii) minimized searching space, and (iii) efficient implementation.

The rest of this paper is organized as follows. Section 2 presents preliminaries and related works. Section 3 details our strategic solution for improving the query processing performance. In Section 4, we summary our experiment to verify and benchmark our approach. Finally, the last section provides some conclusions and future works.

BDCAT'16, December 06-09, 2016, Shanghai, China

DOI: http://dx.doi.org/10.1145/3006299.3006321

2. PROBLEM FORMULATION AND RELATED WORKS

2.1 Data and Query Model

In this paper, we focus only on the directed and unweighted graphs. The data is represented as a graph $G(V, E)$ where V is the set of all vertices and $E = (v_i, v_j)|v_i, v_j \in V$ represents the set of all edges (v_i and v_j are connected with a single unweighted link). The total number of edges to (incoming) and from (outgoing) a vertex v_i is called the degree of v_i and is represented as $deg(v_i)$.

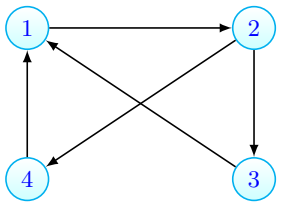

Figure 1: A directed unweighted graph

For the vertices, the most convenient way to represent them is to identify by a number. That leads to number the $|V|$ vertices from 0 to $|V|-1$. For the edges, there are three main ways to represent a relationship of graph: (i) edge lists, (ii) adjacency matrices and (iii) adjacency lists. In the large scale graph, the adjacency matrices representation

cannot be used because of the limit of main memory size. The edge list structure is simple, but the operations on the graph, such as insertion and deletion, are very difficult. The appropriate way to represent the large-scale edges of graph is the adjacency list structure.

For the directed, unweighted graph, there are two ways to organize its adjacency lists:

- **List of incoming edges from a given vertex:** Graph edges will be represented by a list of consecutive incoming vertices (*incoming_edges*) and an index list (*incoming_index*). The sub-list containing the incoming vertices of a vertex N will be addressed by the item at position N in the index list $incoming\_index[N]$. For managing the dynamic graph, we store also the number of incoming vertices. In order to increase the cache hit ratio, the locality of this number must be near the position of incoming edges. Thus, we use a pair (position,number) as indexing for each incoming vertices of a vertex. For example, the graph illustrated by the Fig.1 can be represented by the 2 following lists: $incoming\_edges = [3, 4, 1, 2, 2]$ and $incoming\_index = [(0, 0)(0, 2), (2, 1), (3, 1), (4, 1)]$
- **List of outgoing edges from a given vertex:** Similarly, the edges will be represented by a list of consecutive outgoing vertices (*outgoing_edges*) and an index list (*outgoing_index*). The sub-list containing the outgoing vertices of a vertex N will be addressed by the item at position N in the index list $outgoing\_index[N]$. For managing the dynamic graph, the number of outgoing vertices will be also stored. In order to increase the cache hit ratio, the locality of this number must be near the position of outgoing edges. Thus, we use a pair (position,number) as indexing for each outgoing vertices of a vertex. For example, the graph illustrated by the Fig.1 can be represented by the 2 following lists: $outgoing\_edges = [2, 3, 4, 1, 1]$ and $outgoing\_index = [(0, 0), (0, 1), (1, 2), (3, 1), (4, 1)]$

The basic operations, one could perform on the graph is read and write. Once writing on a graph, an edge is simply added or deleted, whereas a read on a graph is traversing his vertices. A graph traversal is also considered as a query on the graph. In a traversal, we are performing whether a Depth-first or a Breadth-first method[2].

From that, three operation types will be specified and described in more details as follows:

DEFINITION 1 ('Q'/QUERY: u v). *In this paper, we only focus on the operation which can be answered with the distance of the shortest path from the first node u to the second node v in the current graph. If there is no path between the nodes or if neither of them exists in the graph, the answer should be -1. The distance between any node and itself is always 0.*

DEFINITION 2 ('A'/ADD: u v). *This operation aims to modify the current graph by adding another edge from the first node u in the operation to the second node v. As was the case during the input of the original graph input, if the edge already exists, the graph remains unchanged. If one (or both) of the specified endpoints of the new edge does not exist in the graph, it should be added.*

DEFINITION 3 ('D'/DELETE: u v). *This operation is to modify the current graph by removing the edge from the first node u in the operation to the second node v. If the specified edge does not exist in the graph, the graph should remain unchanged.*

To validate this model, we use the testing protocol proposed by the ACM SigMod Programming Contest 2016. The testing workload is organized in batches. Each batch consists of a sequence of operations provided one per line. The last line must contain the single character F that signals the end of the batch. Each operation is represented by one character ('Q', 'A' or 'D') that defines the operation type, followed by a space and two positive integer numbers in decimal ASCII representation, also separated by a space. The two integer numbers represent node IDs. The query results in the output must reflect the order of the queries within the batch [3]. The following figure shows some example batches corresponding to the initial example graph above:

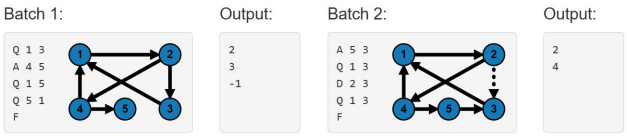

Figure 2: Testing workload in batches [3]

2.2 Related Works

To perform the actions on graph, there are a lot of tools and libraries that we can use to solve this problem. For example, NetworkX is a Python language software package for the creation, manipulation, and study of the structure, dynamics, and functions of complex networks and graph [16]. SNAP C++ library [6] is very popular for a general purpose, high performance system for analysis and manipulation of large graphs. These libraries use the bi-directional Breadth-First Search (BFS) algorithm to compute the shortest distance between two vertices. However, their implementation is not optimal due of general purpose requirements: the shortest distance is computed only in sequence (performed by only one CPU core) and the directional selection is based on number of enqueued vertices only. The later leads to the situation that there are a lot of enqueued vertices in the next traversal loop.

For the large-scale graphs, the researches of GraphLab in [1], PowerGraph in [14], GraphX in [15] are dedicated for processing the very big graph in both distributed and parallel computation. These systems are efficient for general purposes in case having a powerful computing platform like clusters and supercomputers [1]. However, like NetworkX and SNAP C++, they are not adequate for the shortest distance computation for the dynamic graph in the context of medium computing platforms.

For the dynamic graph, the efficient shortest path grabs the attention of researchers. [7], [8], [9] show the works on processing of shortest path traversal queries in online and dynamic transportation graphs. In order to harness the multicore architecture, [10], [11], [12], [13] present the work-efficient parallel breadth-first search algorithms on the large graph. Notwithstanding, the graph update actions were not got attentions in these works.

3. STRATEGIC SOLUTION

Our solution will be detailed in this section. The main actions in the directed, unweighted graph G are: (i) add an edge; (ii) delete an edge; and (iii) compute the shortest distance between two vertices. Because the workload was organized in batches, the following algorithm is proposed to execute it (we will detail the procedures *add_edge*, *del_edge*, *exec_queries* in the next sections):

Algorithm 1 Performing the testing workload

```
Require: QU,QV are the lists to store queries from testing workload
1: while having_batch do
2:     get all actions (a,u,v) of a batch into buffer B; //a: action; u,v: vertices
3:     query_num ← 0;
4:     for all (a,u,v) in B do
5:         if a =' A' then
6:             add_edge(u, v);
7:         else if a =' D' then
8:             del_edge(u, v);
9:         else if a =' Q' then
10:            store (u,v) into buffer QU,QV ;
11:            query_num+ = 1;
12:        end if
13:    end for
14:    if query_num > 0 then
15:        exec_queries();
16:    end if
17:    continue to next batch;
18: end while
```

3.1 Appropriate Data Structure of Graph

Normally, an instruction executed inside a CPU is very quick in comparison of data fetching time from the main memory to CPU (32 or 64 bits). The later forms a cache miss, meanwhile a cache hit has occurred when that data was already in the cache. CPU cache is normally organized by block of 64B. That means one cache block can contain 64/4= 8 vertices. Due to the CPU cache organization, when a process needs to handle a big data, the consecutive item list seems to be the best way to allow having the highest cache hit rate. Actually, the memory latency[1] can be specified in the table 1.

Table 1: Memory Latency 2016

	Time (ns)	Comment
L1 cache reference	0.5	> 2 ALU instruction latency
Branch mispredict	3	
L2 cache reference	4	8x L1 cache
Mutex lock/unlock	17	
Main memory reference	100	25x L2 cache, 200x L1 cache

All these lists are allocated consecutively in the main memory. That why each time we want to access a vertex, its adjacent vertices will be highly probability in CPU cache. Less main memory references offer the high performance of graph processing certainly.

From the above idea, the graph data is represented by the adjacency lists described as following:

- Each node is identified by a number
- All incoming/outgoing nodes of a node are stored in a adjacency trunk of array
- An index array is used to store the starting position of trunk in incoming/outgoing arrays

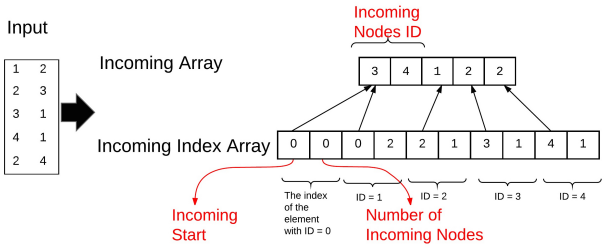

Figure 3: Data Structure of Graph

Thus, the graph G can now be represented by the following structure:

- *incoming_edges*, *incoming_index* are 4-bytes (integer) arrays to represent all edges by the incoming approach. Thus, each pair of two items of *incoming_index* are stored the position of incoming edges and number of incoming edges for a vertex.
- *outgoing_edges*, *outgoing_index* are 4-bytes (integer) arrays to represent all edges by the outgoing approach. Similarly, each pair of two items of *outgoing_index* are stored the position of outgoing edges and number of outgoing edges for a vertex.

By using both incoming and outgoing edge lists, we can quickly explore the graph by both directions. That allows to compute the shortest distance by the bi-directional BFS algorithm [10].

3.2 Optimizing the update actions

For the update actions in a directed graph, we can clearly form them by the insertions or deletions of edge.

- Adding a new edge:

There are two ideas for adding an edge based on our graph structure:

- *Preallocate a bucket*
  Each 32 nodes have a reserved buffer for adding new incoming/outgoing nodes. This buffer is called BUCKET. By default, BUCKET size is 8, that means a maximum of 8 nodes can be added in the interval of 32 vertices.
- *Move to the end of lists*
  Move all incoming/outgoing of two vertices (of the adding edge) to the end of incoming/outgoing edge lists. This idea can be illustrated as the figure 4.

Based on the deeply experiments, we found that the second idea offers more efficient than the first. The reason lies in their data structure: second idea allows initially consecutive lists meanwhile the first one has several slots (wholes) in the lists. Thus, the first one can probably implicate more cache miss rate than the second one.

And we decided to use the second idea for processing the adding action. To offer the multi-threaded parallel computation of shortest distance, before the real insertion, we will perform all the shortest distance queries in the queue. The

[1]Surf for more detail at http://people.eecs.berkeley.edu/~rcs/research/interactive_latency.html

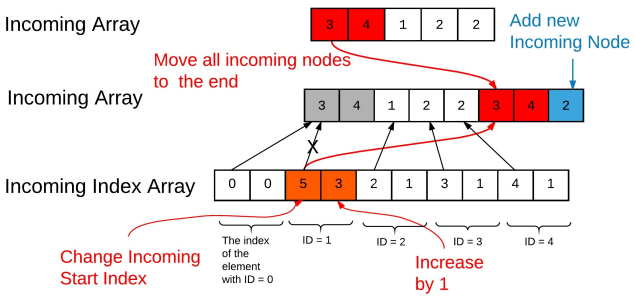

Figure 4: Performing an *Add an Egde* action

Algorithm 2 Add an edge (u,v) into the graph G

Require: G structured by 4-bytes arrays: *incoming_edges*, *incoming_index*, *outgoing_edges*, *outgoing_index*

1: **if** (u,v) exists in G **then**
2: return;
3: **end if**
4: *//Perform queries if having any before adding*
5: **if** *query_num* > 0 **then**
6: *exec_queries*();
7: **end if**
8: *//Update incoming info of the vertex v*
9: Copy all incoming list of v into end of *incoming_edges*;
10: Add new incoming node (u) into end of incoming list of v
11: Increase the number of incoming of v by 1 in *incoming_index*;
12: Update the index of incoming list of v to the new position in *incoming_index*;
13: *//Update outgoing info of the vertex u*
14: Copy all outgoing list of u into end of *outgoing_edges*;
15: Add new outgoing node (v) into end of incoming list of u
16: Increase the number of outgoing of u by 1 in *outgoing_index*;
17: Update the index of outgoing list of u to the new position in *outgoing_index*;

algorithm for adding an edge (u,v) into the graph G can be illustrated in the algorithm 2:

- Deleting an edge:

In our graph structure, the delete action is very simple. This action can be only done by updating the number of incoming/outgoing in the index arrays and removing the associated vertex in the incoming/outgoing edge lists. The operation of delete action can be shown by the figure 5:

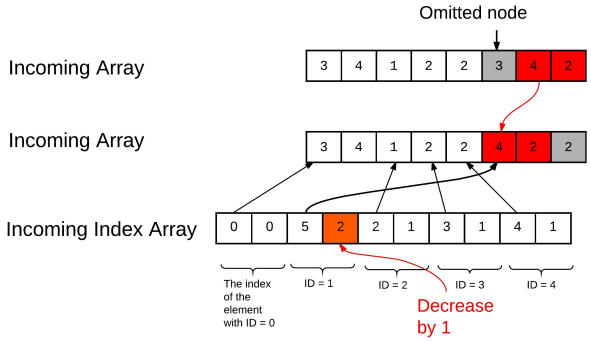

Figure 5: Performing a *Delete an Egde* action

To offer the multi-threaded parallel computation of shortest distance, before the real deletion, we will perform all the shortest distance queries in the queue. Thus, the operations for deleting an edge is shown as in the algorithm 3:

Algorithm 3 Delete an edge (u,v) of the graph G

Require: G structured by 4-bytes arrays: *incoming_edges*, *incoming_index*, *outgoing_edges*, *outgoing_index*

1: **if** (u,v) exists in G **then**
2: return;
3: **end if**
4: *//Perform queries if having any before deleting*
5: **if** *query_num* > 0 **then**
6: *exec_queries*();
7: **end if**
8: *//Update incoming info of the vertex v*
9: Remove the vertex u inside the incoming list of v *incoming_edges*;
10: Decrease the number of incoming list of v by 1 of *incoming_index*;
11: *//Update outgoing info of the vertex u*
12: Remove the vertex v inside the outgoing list of u *outgoing_edges* ;
13: Decrease the number of outgoing list of u by 1 in *outgoing_index*;

3.3 Optimizing the query processing

For the query processing, we interest firstly to the shortest distance computation between two vertices. Thus, Breadth First Search (BFS) algorithm is considered as the best way [10][12] for large-scale and dynamic directed/unweighted graph. Another point should be emphasized here: we have to exploit all the capability of modern multicore, multithread CPU. For that, our strategy has to offer in parallel processing for all consecutive queries. We describe in more detail our strategy of query processing:

i. Algorithm of shortest distance computation:

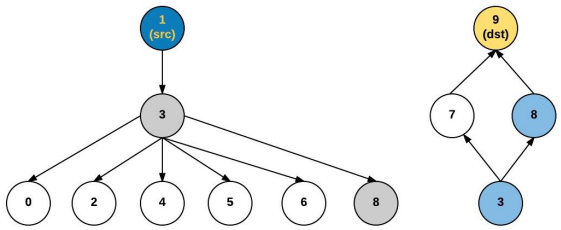

Figure 6: Bi-directional BFS shortest distance traverse

BFS is used in both directions by using both incoming arrays and outgoing arrays.

- 2 bitmap arrays are used for remarking traveled incoming/outgoing nodes.
- Strategies to reduce the search space: predict deeply on the graph. In the normal bi-direction BFS, we follow the branch that has the small number of traversing vertices in the incoming/outgoing queue cite SSSP-Parallel. However, this information is not enough. As

the figure 6 for the query of finding the shortest distance from 1(src) to 9(dst), by using the normal bi-directional BFS, we follow the outgoing queue of source vertex 1 (due to having only one child) and outgoing vertex 3 will be traversed at first. It means that (1+6+2 = 9) vertices must be enqueued.

Algorithm 4 Compute shortest distance (u,v) by the bi-directional BFS in the graph G

Require: G structured by 4-bytes arrays: $incoming\_edges, incoming\_index, outgoing\_edges, outgoing\_index$

```
if u == v then
    return 0;
end if
if (incoming_num[v] == 0)or(outgoing_num[u] == 0) then
    return -1; //No path between u and v
end if
Get incoming_map and outgoing_map from global arrays for this Cilk worker
in_queue ← v; //init in-queue
set_bit(v, in_map); //indicate v was visited
out_queue ← u; //init out-queue
set_bit(u, out_map); // indicate u was visited
in_cost = 1, out_cost = 1; //distance to/from u/v
count = 1; //vertices in queue to traverse
out_size = outgoing_num[u]; in_size = incoming_num[v];
while (count > 0) do
    if (out_size < in_size) then //following the out_queue
        out_size = 0; out_cost+ = 1; count = 0;
        for each vertex e in out_queue do
            for each vertex n in outgoing_edges[e] do
                if test_bit(n, out_map) then
                    if n == v then //reach to the destination vertex
                        Clear all bits of in/out maps;
                        Return out_cost;
                    end if
                    if test_bit(n, in_map) then
                        Clear all bits of in/out maps;
                        Return in_cost + out_cost;
                    end if
                    //ifnot, we push this vertex to out_queue
                    out_queue ← n; count+ = 1; set_bit(n, out_map);
                    out_size+ = outgoing_num[n];
                end if
            end for
        end for
        out_size+ = count;
    else//following the in_queue
        in_size = 0; in_cost+ = 1; count = 0;
        for each vertex e in in_queue do
            for each vertex n in incoming_edges[e] do
                if test_bit(n, in_map) then
                    if n == u then //reach from the source vertex
                        Clear all bits of in/out maps;
                        Return in_cost;
                    end if
                    if test_bit(n, out_map) then
                        Clear all bits of in/out maps;
                        Return in_cost + out_cost;
                    end if
                    //ifnot, we push this vertex to in_queue
                    in_queue ← n; count+ = 1; set_bit(n, in_map);
                    in_size+ = incoming_num[n];
                end if
            end for
        end for
        out_size+ = count;
    end if
end while
Clear all bits of in/out maps
Return -1;
```

To reduce the search space (reducing the traversal vertices), we modify the bi-directional BFS algorithm by the strategy: following only the direction that has the smaller sum of next level enqueued vertices and their children. For example, in the figure 6, at the beginning, we can realize that the sum of outgoing queue (only 1 vertex) and their children (6 vertices can be enqueued in the next level) is 7. Meanwhile, the sum of incoming queue (2 vertices) and their children (only 1 vertex can be enqueued in the next level) is 3. Thus, following the incoming direction can be more efficient than the outgoing direction.

Our strategy tuning the bi-directional BFS algorithm can be described in the algorithm 4.

ii. Query parallel processing:
Our solution for processing the consecutive queries is:

- Global incoming queue and outgoing queue are used for each call of searching the shortest length. Then, each searching thread will use only proper in/out queues determined by an interval of global in/out queues.

- Each searching thread will also own the proper in/out map slots, computed from the global in/out maps. After the searching is finished, these in/out slots will be cleared for the next search.

- Cilkplus is used for performing queries in parallel [4]. By conducting a lot of proper experiments, our solution is implemented with some multithreaded parallel computing methods like OpenMP[2], Pthread[3], we note that the Cilkplus method is the best one and offers the best performance for our solution.

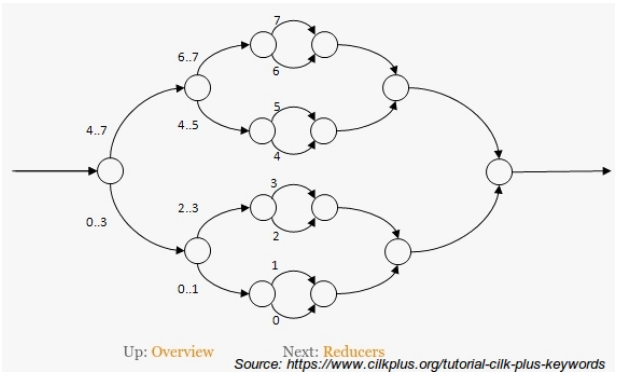

Figure 7: Paralleling the queries by Cilkplus method [4]

Thus, the parallelization of consecutive queries is illustrated as the following algorithm:

Algorithm 5 Execute all consecutive queries on the graph G

Require: G with all queries stores in the QU and QV lists

```
i = 0;
//Perform in parallel the queries by Cilkplus method
for i < query_num do
    distances[i] = shortest_distance(QU[i], QV[i]);
end for
```

[2]http://openmp.org/wp/
[3]https://computing.llnl.gov/tutorials/pthreads/

4. EXPERIMENT

To validate our strategic solution for optimizing the shortest path query on large-scale dynamic directed graph, we participated in the ACM SigMod Programming Contest 2016 and were selected as one of the five finalists. We use also other datasets from the Stanford Large Network Dataset Collection [5] to evaluate our solution.

4.1 SigMod Programming Contest 2016

In this contest, there are four initial big graphs used to issue the workloads consisting of a series of shortest path queries and graph update actions (insertions or deletions of edges). In answering each query, all graph updates preceding the query must be taken into account. Here are the statistics of these graphs:

Table 2: SigMod Contest Dataset Statistics

Parameter	Small	Medium	X-Large	XX-Large
Nodes	1574074	2000000	1971281	6009555
Edges	3232855	5000000	5533214	16518948
Max num of incoming nodes	114557	90412	12	779
Max num of outgoing nodes	114125	356364	12	770

Based on the proposed solution, we build and implement it by C language. The experiment was performed in the evaluation machine having 2 x Intel E5-2620v2 (15MB Cache, 6-cores/12threads), 20GB for the main memory, Ubuntu 14.04 Linux OS, gcc 5.2.1. Here is the final results we obtained from this contest.

Table 3: Final SigMod test results (in seconds)

Small	Medium	X-Large	XX-Large
0.118	0.494	1.284	2.878

4.2 Evaluation using SNAP datasets

In this experiment, two large graphs from SNAP dataset collection are selected to generate the workloads for testing: LiveJournal and Pokec. LiveJournal[4] is a free on-line community with almost 10 million members. For the Pokec dataset, Pokec[5] is the most popular on-line social network in Slovakia that connects more than 1.6 million people. More details of these datasets are shown in the table 4.2:

Table 4: LiveJournal&Pokec Dataset Statistics

Parameter	LiveJournal	Pokec
Nodes	4847571	1632803
Edges	68993773	30622564
Number of triangles	285730264	32557458
Diameter (longest shortest path)	16	11

Experiment Platform:

All test was conducted on our server that has an IntelÂő CoreâĎć i7-3720QM (6MB Cache, up to 3.60 GHz, 4 cores-8 threads), 8GB of main memory, CentOS 7 OS. Our implementation was compiled by GCC version 5.1.1 with optimization level O3 and SSE4 instructions.

Testing Workloads:

For the testing workloads, we built a tool in order to generate the workload by using the testing protocol of SigMod [3]. For each graph, we generated a workload including about 1,000,000 queries and update actions. The ratio of queries/insertions/deletions is 0.80/0.10/0.10 respectively (thus there are approximately 800,000 queries, 100,000 insertions, 100.000 deletions).

Evaluation Results:

To measure the time performance, we run our solution and other comparison tools 6 times for each dataset. The final results are illustrated as the table 5 with the values computed as the average values from those 6 time tests.

The first comparison tool is the original bi-directional BFS implemented by combining with the Cilkplus method for processing the queries in parallel. We also evaluated our solution in comparison with the simple reference solution proposed by SigMod Programming Contest 2016[6]. It is implemented in Python, using the *networkx* module. From this contest, we have also a dataset containing a graph of 1574074 vertices and 3232855 edges. This dataset is also used in our experiment.

Another tool from SNAP was also used for conducting the experiments. From the SNAP for C++[7], we implemented a tool respected our testing protocol.

The table 5 shows the evaluation results obtained from our experiments.

Table 5: Final test results for SigMod and SNAP datasets

DataSet	Our Solution	BFS Cilkplus	Ref. Solution	SNAP Solution
SigMod Test	**1.428s**	18.078s	3319.4s	5045s
LiveJournal	**32.045s**	55.630s	Can't execute	>24h
Pokec	**14.286s**	27.825s	>10h	>15h

The obtained results allow us to validate our strategic solution that has the outstanding performance in comparison with other approaches. By using the appropriate data structure, we can reduce amount of time while accessing the main memory for the graph data. This comes from the fact that the cache hit rate is increased during performing the workload. Another point that brings the strategic constraint we use in the bi-directional BFS algorithm for computing the shortest distance. By selecting the smaller queue for traversing, execution time could be significantly reduced. The last advantage of our solution lies in the Cilkplus multithreaded parallel computing method that allows to run the *shortest_distance* procedure in parallel. It offers a quick and easy way to harness the power of both multicore and vector processing.

All of codes and test results can be freely accessed from the GitHub link: https://github.com/nnhoa/shortest_path/.

[4]http://snap.stanford.edu/data/soc-LiveJournal1.html
[5]https://snap.stanford.edu/data/soc-pokec.html
[6]http://dsg.uwaterloo.ca/sigmod16contest/reference.tar.gz
[7]https://snap.stanford.edu/releases/Snap-2.4.zip

5. CONCLUSION AND FUTURE WORKS

In profit of multi-core/multi-chip and large main memory capacity of modern computer, we propose a method that allows to improve the performance of optimizing the query on large-scale directed, no-weighted and elastic graph. This method is based on the three main ideas: appropriate data structure for increasing the hit rate in the CPU cache, minimized searching space by smartly choosing the smallest queue for BFS algorithm, and efficient implementation by the Cilkplus-based parallel processing.

For future works, we aim to extend our method for the graph-based data model enabled Online Transaction Processing (concurrent query processing).

6. REFERENCES

[1] J. Wei, K. Chen, Y. Zhou, Q. Zhou and J. He: Benchmarking of Distributed Computing Engines Spark and GraphLab for Big Data Analytics, 2016 IEEE Second International Conference on Big Data Computing Service and Applications (BigDataService), pp.10-13 (2016).

[2] Jayanta Mondal and Amol Deshpande: Managing large dynamic graphs efficiently. In Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD '12), pp.145-156 (2012)

[3] The ACM SIGMOD Programming Contest 2016: http://dsg.uwaterloo.ca/sigmod16contest/

[4] Intel Cilkplus Documentation: https://www.cilkplus.org/cilk-documentation-full

[5] Stanford Large Network Dataset Collection: https://snap.stanford.edu/data/index.html

[6] D. Hallac, J. Leskovec, S. Boyd.: Network Lasso: Clustering and Optimization in Large Graphs. In ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD), 2015

[7] W. Nawaz, K. U. Khan and Y. K. Lee: CORE Analysis for Efficient Shortest Path Traversal Queries. In Social Graphs,Big Data and Cloud Computing (BdCloud), 2014 IEEE Fourth International Conference on, Sydney, NSW, 2014, pp.363-370.

[8] DL. H. U, H. J. Zhao, M. L. Yiu, Y. Li and Z. Gong: Towards Online Shortest Path Computation. In IEEE Transactions on Knowledge and Data Engineering, vol. 26, no. 4, pp. 1012-1025, April 2014.

[9] G. Scano, M. J. Huguet and S. U. Ngueveu: Adaptations of k-shortest path algorithms for transportation networks. Industrial Engineering and Systems Management (IESM), 2015 International Conference on, Seville, 2015, pp. 663-669.

[10] V. T. Chakaravarthy, F. Checconi, F. Petrini and Y. Sabharwal: Scalable Single Source Shortest Path Algorithms for Massively Parallel Systems. In: Parallel and Distributed Processing Symposium, 2014 IEEE 28th International, Phoenix, AZ, 2014, pp. 889-901.

[11] Q. Li and W. Wei: A parallel single-source shortest path algorithm based on bucket structure. In: 2013 25th Chinese Control and Decision Conference (CCDC), Guiyang, 2013, pp. 3445-3450.

[12] Charles E. Leiserson and Tao B. Schardl: A work-efficient parallel breadth-first search algorithm (or how to cope with the nondeterminism of reducers). In Proceedings of the twenty-second annual ACM symposium on Parallelism in algorithms and architectures (SPAA 2010)

[13] D. S. Banerjee, S. Sharma and K. Kothapalli: Work efficient parallel algorithms for large graph exploration. In: 20th Annual International Conference on High Performance Computing, Bangalore, 2013, pp. 433-442.

[14] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C.Guestrin: PowerGraph: Distributed Graph-Parallel Computation on Natural Graphs. OSDI, pages 17-30, 2012.

[15] J. E. Gonzalez, R. S. Xin, A. Dave, D. Crankshaw, M. J. Franklin, and I. Stoica: GraphX: Graph Processing in a Distributed Dataflow Framework. OSDI, Oct. 2014

[16] Aric A. Hagberg, Daniel A. Schult, Pieter J. Swar: Exploring network structure, dynamics, and function using NetworkX, In Proceedings of the 7th Python in Science Conference (SciPy), pp.11-15, 2008.