

TRS: A New Structure for Shortest Path Query

Qi Wang^(✉), Junting Jin, and Hong Chen

Renmin University of China, Beijing, China
vicky0105.wq@gmail.com, jjt0901@126.com, chong@ruc.edu.cn

Abstract. Shortest Path Query is being applied to more and more professional scopes, such as social networks and bioinformatics. But the exponential growth of data makes it much more challenging since traditional BFS-based algorithms are hard to scale due to the requirement of huge memory.

Different from the traditional algorithm such as Dijkstra algorithm, our method is based on Depth-First-Search, which first constructs the DFS tree with interval-based encoding, and then isolates non-tree edges to generate the TRS structure for the graph. Shortest path queries between arbitrary nodes are performed upon this structure. The final result could be a detail path with exact path cost. This algorithm is quite easy to scale to large graphs, since the TRS algorithm automatically divide the graph into a set of connected components, each of which has a single TRS structure. Our experiments has proved that the algorithm fits large sparse graph quite well in real world.

Keywords: Information network · Large graph · TRS · Shortest path

1 Introduction

Traditional algorithms for SPQ are mainly based on Breadth-First-Search of the graph. The classical Dijkstra’s algorithm [8] with $O(n^3)$ time cost and $O(n^3)$ space cost is one of them. And these methods are all memory based, and show great limitations in scalability with the fast growth of data, especially the web data. However, now even a small community network may have thousands of nodes, and storing all of the shortest path trees for this large graph is infeasible at all.

In general, graphs for real large networks are mostly sparse. That is, the average degree of each node is close to 1 or even lower. Accordingly, the storage structure also changes to storing edges instead of storing adjacency matrix. Respect to this property, we proposed a new approach to find the shortest path in sparse graph, and the time cost only associated with the number of edges, while a little more space is needed.

This work is supported by a grant from “Special Research on Key Technology of Domestic Database with High Performance and High Security” for National Core-High-Base Major Project of China (No. 2010ZX01042-001-002-002).

Our algorithm first call Depth-First-Search (DFS) algorithm on a directed acyclic graph(DAG), and converts the graph into our new structure TRS(s), in which each node has its interval-based three tuple code. TRS divides all edges into three sets: DFS spanning tree edges (TE), separation edges (SE), and remaining edges (RE). For each TRS, if it only contains DFS spanning tree edges, answering any shortest path query upon it only takes $O(n)$ time if the code is beforehand sorted; If either separation edges or remaining edges are not null, the time cost only depends on the number of non-tree edges. And this is why our algorithm is quite efficient on sparse graphs.

There are three main contributions of our work in total. The first one is we introduce Interval Based Encoding into SPQ to prune some impossible directions; the second one is we proposed a new structure TRS to represent a graph, and TRS makes it possible to answer any SPQ with space cost of $O(n+m)$, where n is number of nodes and m is number of edges; and the last one is our algorithm scales well by using divide-and-conquer strategy. This method is suitable for parallel processing system for big data like hadoop.

The rest of this paper is organized as follows: Sect. 2 depicts related work, then Sect. 3 introduces our TRS framework. In Sect. 4 we give out the full view of TRS algorithm for answering shortest path queries. Section 5 provides the complexity in both time and space for our algorithm. Section 6 shows all the experiments. Section 7 is conclusions and future work.

2 Related Work

The very famous algorithm for shortest path query is Dijkstra [8], and some improvement could be made by using heap data structure for priority queues [9]. Although this algorithm has been extended to external memory [4], it cannot handle well with respect to response time. In order to deal with SPQ, Agrawal and Jagadish [10] introduced the idea of graph partitioning. Later, [7] gave a more efficient way by materializing some local shortest path which could be held in memory. R. Gutman [5] used the reach value of each node to find the shortest path for road networks. Depending on the basic idea of reach value, AV. Goldberg, H. Kaplan and R. Werneck [3] introduced several variants of the reach algorithm, and some do not need explicit lower bounds. Since in the search process, some unrelated branches may be taken into account and it may cause a lot of time cost. Subsequently, methods using geometric attributes of a graph are proposed, and a lot of work has been done. Reference [6] used a geometric way to prune some unrelated branches in order to narrow the search. And [14] divides the graph into several regions, and put an edge into the priority queue in Dijkstra's algorithm when a path from the source region to the destination region passes this edge. The most recent research using geometric attribute is [2], which exploits symmetry of the graphs to compress BFS-trees. Experiments shows that the space cost is reduced in comparison with the un-compressed BFS-trees, but still, space cost is too high when the graph is large enough. Another new research

done by F. Wei [1] proposed TEDI as an indexing and query processing scheme for SPQ. Time cost for index construction is $O(n^2)$, while query time depends on the argument of the decomposition.

3 TRS Framework

3.1 Background Knowledge

In this paper, our research object is Directed-Acyclic-Graph with non-negative weight. Let n be the number of nodes, and m be the number of edges. For a complete graph, there's an arc between them for any two nodes, thus number of edges is $M(n) = n * (n - 1)$. But in most graphs, number of edges is much less than $M(n)$, so we define the ratio of m to $M(n)$ as the saturation of G , that is, $saturation(G) = m/n * (n - 1)$. Suppose the saturation of a graph is lower than a given low ratio, e.g. 0.5 %, then we call this graph a sparse graph.

Enlightened by the sparse feature and pruning idea, we propose the TRS structure. In this structure, we classify edges into three sets, TE, SE, and RE. Edges in TE form the spanning tree(or forest) of the original graph. When we only consider the TE edges, if node u could reach node v , the only path is already the shortest path. But at most time, SE and RE must be considered. If the graph is sparse, the number of edges is less than $0.005n^2$. Thus the number of edges in SE and RE is also small, so that the additional time for searching in SE and RE will not be too much. In our experiments, this has been well proved.

3.2 TRS Structure

In this section, we will first introduce Interval-Based Encoding.

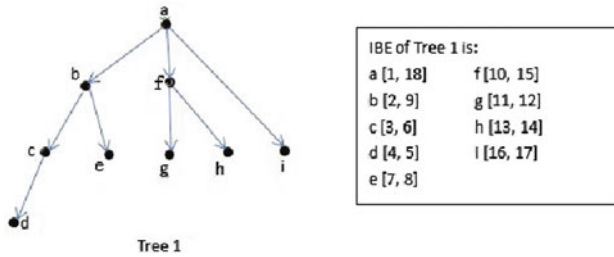


Fig. 1. Example of interval based encoding

Interval-Based Encoding. Interval-Based Encoding was first put forward to be applied to XML files [11], then it is used to solve reachability problems [12, 13] well. Interval-Based Encoding utilizes Depth-First-Search to assign both the pre-order code and post-order code for every node of a tree. When a node is visited at the first time it is assigned the pre-order code, and it gets post-order code after all its offsprings are visited. Figure 1 shows an example of Interval-Based Encoding of a spanning tree. Relationship between any two nodes could be represented through their intervals, that is, root a has the most wide interval, which contains all the intervals of a 's offsprings, and intervals of b and f has no intersections because they belong to different branches.

In Fig. 1, if we want to find the shortest path from node a to i , we can use the IBE of Tree 1 to directly prune the b -branch and f -branch. Because neither of their intervals satisfy the condition containing i 's interval. Considering its $O(1)$ time cost, we think of making it useful in SPQ to avoid a BFS which only depends on the weight of an edge. But as the example indicates that the encoding only fits for tree structure, and if we want to take advantage of it to answer SPQ in an arbitrary graph, we must modify the encoding.

TRS for Graph. For an arbitrary directed graph, it has more edges than its corresponding spanning tree or forest which could be interpreted by the Interval-Based Encoding, thus, in order to answer the shortest path query, these non-tree edges must also be recorded to maintain the complete path information of the original graph.

In our encoding process, we first select a root node and then perform a DFS on the graph from this node to get its spanning tree. And then select next root node as such, as a result, spanning forest comes into being. During this process, the interval for each node is also assigned. If the arc head of the current scanning edge directs a closed node, this edge is a non-tree edge.

Theorem 1. *For a non-tree edge, the start node is ancestor or sibling (we call two nodes siblings if they belong to two different subtrees, without regard to the level of subtrees) of the end node.*

Proof. Nodes in the DFS trees have three relationship, which are ancestor, offspring, and sibling. Assume a non-tree edge $e = \langle u, v \rangle$, v is a closed node in the DFS trees. When the current scanning edge is e , u 's post-order is bigger than v 's post-order, and thus u cannot be offspring of v , but only can be ancestor or sibling.

Theorem 2. *For a non-tree edge, if the start node is a sibling of the end node, its pre-order is bigger than the post-order of the end node.*

Proof. Edge $e = \langle u, v \rangle$ is a non-tree edge, and u is a sibling of v , and their intervals are respectively $[u_a, u_b]$ and $[v_a, v_b]$. We already know from Theorem 1's proof that $u_b > v_b$. Because different nodes' intervals can't overlap, there may be $u_a < v_a$ or $u_a > v_b$. However, if $u_a < v_a$, then $[v_a, v_b] \subset [u_a, u_b]$, and according to Interval-Based Encoding, u should be v 's ancestor, which conflicts with their sibling relationship. Thus the relationship between u_a and v_b can only be $u_a > v_b$.



Fig. 2. Example of TRS

According to Theorem 1 and 2, we classify all non-tree edges into two classes: SE(Separation Edges) and RE(Remaining Edges). SE are those edges whose two endpoints are siblings in the DFS tree, and connect two different branches. RE are those edges whose arc tail is ancestor of head in the DFS tree. We store the Interval-Based Encoding instead of the whole DFS tree because the code actually presents the topology of the tree, and the location of each node can be deduced from the codes. So, as to a spanning tree with tree edges and nodes' codes, as well as SE and RE, we call it a TRS structure, reconstructing the graph and keeping all topology information except the weights. Figure 2 is an example.

Weight information is also easy to maintain by adding another item in the Interval-Based Encoding. For each node, we assign it a weight as the sum of weights along the path on the DFS tree from the root node. E.g. in Fig. 2, $w(b) = w(ab)$, in which $w(b)$ represents the weight of node b, and $w(ab)$ represents the weight of edge $\langle a, b \rangle$. If one node are reachable from the other node in a tree, and if we want to get the path weight between the two nodes in the DFS tree,

Algorithm 1 TRS Construction

```

INPUT: DAG G
OUTPUT: TRS(s) for G
ALGORITHM:
01:  Find root nodes set R of G
02:  For 1..n
03:    visited[n]=false
04:  End
05:  For each root  $\in R$  Do
06:    start DFS from root
07:    If v is not visited
08:      visited[v]=true
09:    Else v is boundary node, back to v's parent
10:  End
11:  Assign pre-order if not assigned
12:  Assign post-order if all its offspring is visited
13:  End
    
```

Fig. 3. Algorithm of TRS construction

we can directly use the weight of end node subtracting the weight of start node. Because of this feature, we record the path weight from root to the current node as the node weight. And for non-tree edges, we need to record the edge weight. By adding one more item in the code in this way, the weight information of the graph is also maintained.

Sometimes, the graph is not well connected in one spanning tree. For example, if a graph has two connected components, one contains edges $\langle a, b \rangle$ and $\langle a, d \rangle$, and the other contains edge $\langle c, d \rangle$, the DFS scanning would generate a spanning forest rooted at a and c for this graph, and node d is shared by both of them. We call nodes (shared by more than one TRS) boundary nodes like d . In Fig. 3, Algorithm 1 describes how to generate TRS structure for an arbitrary DAG.

4 Answering Shortest Path Query

4.1 The Framework

In general, the DFS process is performed upon a DAG G at first, then it returns the TRS structure of G . If there're more than one spanning tree of G , $\text{TRS}(G)$ returns a group of TRS and identified by their root nodes. In each TRS, the nodes as well as their codes, the responding SE and RE are recorded. For each TRS of G , we call AnswerSPQ algorithm to deal with and return the shortest path between the query nodes. When there are more than one TRS structure in a connected DAG, there must be some boundary nodes as intermediate nodes to connect adjoining TRS. For example, If node u is the current encoding node, and it has already been visited and belongs to another TRS, we mark u a boundary node and set it to be closed, so that its offspring (if exist) will not appear in the current TRS, and this avoids mark one node repeatedly.

4.2 Answer SPQ On TRS

For the given start node s and end node t , if there is no non-tree edges and t 's interval is within s 's interval, we search in the encoded nodes from s to t , and the shortest path is a sequence which are ordered by their pre-orders of these nodes, with weight of $w(t)-w(s)$. But mostly, neither SE nor RE is null, and shortest paths take the same chance to pass both of them. Since it's easy to find a path on the DFS tree, the main problem focuses on paths passing these two sets. When two query nodes are reachable, and no SE edges exist in their branch, we only need to replace some tree paths with RE edges if they have smaller weight. When there are SE edges between the branches which contain the two nodes respectively, we need to find the shortest path between the query nodes and end points of SE edges and connect these parts together to get the global shortest path. The algorithm is depicted in Fig. 4.

The first action in Algorithm 2 is to sort edges in SE. According to Theorem 2, we know that edges starts from a laterly-visited node to a formerly-visited node,

Algorithm 2 AnswerSPQ

```

INPUT: TRS of G, query nodes s, t
OUTPUT: Shortest path from s to t if exists
ALGORITHM:
01:  qsort(SE) by pre-order of start node in descending order
02:  If Reachable(s, t) Do
03:    current = RE-SP(s, t)
04:  Else Do
05:    current.weight = INFINITY
06:  End if
07:  For  $e \in SE$  Do
08:    If e's start is reachable from s Do
09:      tmp = SE-SP(s, t, e)
10:      If  $tmp.weight < current.weight$  Do
11:        current = tmp
12:      End If
13:    End If
14:  End For
15:  return current
    
```

Fig. 4. Algorithm of AnswerSPQ

and this sorting makes the search starts from the right-most SE edge in a DFS tree (if the tree spreads from left to right) to guarantee no route is missed. If the two nodes are reachable in the DFS tree, we find out a temporary shortest path by taking RE into account, or set them unreachable if not reachable. Then SE is considered. The searching starts from each SE edge whose start node is reachable from s, and when the process gets to the end node of this edge, we treat the end node as a new source node to run SE-SP, until it reaches t or has searched all edges in SE but not reaching t. Algorithm 3 and 4 illustrate the process of RE-SP and SE-SP.

Algorithm 3 only consider RE edges while processing SPQ, because whether or not the shortest path passes SE edges, it needs to consider both DFS tree and RE. Here we offer a schematic figure to explain the relationship of RE edges that appear on the same branch of DFS tree as Fig. 6.

The bold line represents a trunk of DFS, and the arc lines represent RE edges on this branch. Here we must make sure that each RE edge is valuable, that is, for $(a \rightarrow b) \in RE$, $w(a \rightarrow b) < w(b) - w(a)$. Because only when it is valuable, shortest path may pass it, if not, the shortest path chooses the tree path instead of this RE edge. The schema indicates that there are two kinds of relationship between two RE edges: serial and overlying. In Fig. 6, edge $(a \rightarrow b)$ and edge $(e \rightarrow f)$ are serial; edge $(a \rightarrow b)$ and edge $(c \rightarrow d)$, as well as edge $(e \rightarrow f)$ and edge $(g \rightarrow h)$ are overlying. Algorithm 3 describes how to answer shortest path query under the two situations.

Algorithm 3 Shortest Path only Considering RE (RE-SP)

INPUT: query nodes s, t
 OUTPUT: Shortest path from s to t
 ALGORITHM:
 01: Find RE Candidate whose end points are on the path from s to t
 on DFS-Tree
 02: qsort(RE Candidate) by pre-order of end nodes in ascending order
 03: Initial path queue Q , $Q.push(s, s)$
 04: For each RE $edge(rs, re) \in RECandidates$ Do
 05: Current = $path(s, rs) + edge(rs, re)$
 06: While ! $Q.empty$ Do
 07: $Sp(s, t') = Q.pop()$
 //Pop Q 's first element as the shortest path from s to t in DFS tree.
 08: If (rs, re) overlap $sp(s, t')$ Do
 09: $tmp = sp(s, t') + path(t', re)$
 10: Else if (rs, re) is serial to $sp(s, t')$ Do
 11: $tmp = sp(s, t') + path(t', rs) + path(rs, re)$
 12: $brk = true$
 13: End If
 14: If $tmp.weight < current.weight$ Do
 15: $current = tmp$
 16: End If
 17: $Q.push(current)$
 18: If $brk == true$ break
 19: End While
 20: End For
 21 $sp(s, t') = Q.pop()$
 22: return $sp(s, t') + path(t', t)$

Fig. 5. Algorithm of RE-SP

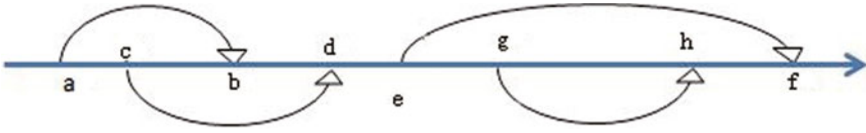


Fig. 6. Schema of RE-SP

Then let's consider the SE edges. From Algorithm 2 we can get a very naive thought, that is, starting from the right most SE edge, and then treating the end of this SE edge as a new query source node, repeating this procedure until we find out the path. But this will lead to traversing all SE edges, so we do a one-hop ahead computation, that is, connecting SE edges who are reachable.

Definition 1. For any two SE edges, se_1 and se_2 , if end of se_1 can reach start node of se_2 , we can define se_2 is a SE-child of se_1 .

Algorithm 4 Shortest Path Considering SE (SE-SP)

```

INPUT: query nodes s, t, SE edge se
OUTPUT: Shortest path from s to t
ALGORITHM:
01: result.weight = INFINITY
02: part1 = RE-SP(s, se.start)
03: part1=part1 + se
04: For p ∈ SE-children Path Array Do
05:     If Reachable(se.end, p.start) Do
06:         pse = first se edge of p
07:         part2 = SE-SP(se.end, t, pse)
08:     End If
09:     If part2.weight < result.weight Do
10:         If Reachable(se.end, t) Do
11:             part3 = RE-SP(part2.last, t)
            //part2.last stands for the end point of the last se edge of p
12:         End If
13:         If part1.weight + part2.weight + part3.weight < result.weight Do
14:             result = part1 + part2 + part3
15:         End If
16:     End If
17: End For
18: Return result
    
```

Fig. 7. Algorithm of SE-SP

With this definition, for an SE edge, we first find out all its SE-children. Then compute the shortest path from its end node to the start nodes of its SE-children by calling algorithm RE-SP. Next, add its own weight to get the shortest path from its start node to the start nodes of its SE-children, and store them in a path array. After all these shortest paths are attained for each SE edge, we order them by the pre-order of start nodes in descending order. When a query comes, we only need to search in this array, until we find out a path that could reach t . Algorithm 4 shows the detail of using this two-hop shortest paths.

If there are more than one TRS structures in a connected DAG, there must be some boundary nodes connecting them. These boundary nodes help to find out the shortest path. If two nodes can be connected through some boundary nodes, we get these paths and select the shortest one as the final result. Some paths may have only one boundary node lies on, while some may have more than one, in this case, shortest path between boundary nodes can be pre-computed.

5 Algorithm Analysis

In this section, we will give the mathematic analysis on the complexity of our algorithm using TRS structures. According to Algorithm 2, time cost for computing shortest path on TRS mainly focuses on SE-SP, let's name it $T(\text{SE-SP})$,

and SE-SP calls itself recursively. But we cannot use this recursive algorithm to derive time complexity because for each SE edge, the number of SE edges which it could reach (we call these SE edges the child edge) is unclear, and actually, the use of SE is to enumerate all possible paths. So we utilize the average child SE edges to get the result. Let the average number of child edge a SE edge has be a , and all possible paths to reach the destination node forms an a -tree, and the number of leaf node is the number of searching paths. If the a -tree has k levels, then we get the following equation:

$$1 + a + a^2 + \dots + a^{k-1} = |SE| \quad (1)$$

From Eq. (1), we can know that

$$a^{k-1} = (|SE| * (a - 1) + 1) / a \quad (2)$$

and this is equal to the number of leaf node in this tree. So the search upon SE edges is totally conducted $(|SE| * (a - 1) + 1)$ times, and each time k SE edges are considered. Also from Algorithm SE-SP, each time a SE edge is counted in, RE-SP is called, so theoretically, for each such path, RE-SP is called $(k-1)$ times, but many of them are called more than once, and the actual number is

$$1 + a + a^2 + a^3 + \dots + a^{k-1} \quad (3)$$

which is $|SE| + 1$.

Here, we still use the average assumption that on each branch that is separated by SE edges, the number of RE edges are almost the same, so that in each branch, the number of RE edge is $|RE|/|SE|$. And time cost for RE-SP is $|RE'|^2/|RE'|$ is the number of RE edges on the current single branch), so that the total time cost for one a -tree path is $(k - 1) * (|RE|/|SE|)^2$. And then we get that

$$T(SE-SP) = (|SE|(a - 1) + 1) * (k - 1) * |RE|^2/|SE|^2$$

And time complexity is $O(|RE|^2/|SE|)$. Now, let's turn to Algorithm 2, we can induce that time cost for one TRS is $O(|RE|^2)$. And worst condition for fulfilling a path is $|SE| * n$. Therefore, when a graph only has one TRS, the worst time cost for answering shortest path is $O(|RE|^2 * |SE| * n)$, while the best is $O(|RE|^2 * n)$, and for sparse graph, $|RE|$ and $|SE|$ are both much smaller than n . Our experiment also shows that time cost for sparse graph speeds up.

Space cost for TRS is also quite small. $O(n)$ for node encoding, $O(|SE|)$ for SE edges, and $O(|RE|)$ for RE edges. And clearly, the total space cost does not exceed $O(n+m)$. And this significant space saving will show great advantage in large graphs.

6 Experiments

All experiments in this paper were run on a Intel Core 2 2.66GHz PC with 4G memory. Programming language is C/C++.

Table 1. TRS structure on artificial graph

Node no.	Edge no.	Saturation	Time (ms)	Space (KB)	RE no.	SE no.
100	129	0.013	0	2.05	15	15
200	237	0.006	0	4.29	7	31
300	348	0.004	0	6.91	15	34
400	463	0.003	0	8.99	26	38
500	577	0.002	0	11.0	21	57
600	671	0.002	0	13.5	29	43
700	807	0.002	0	15.7	54	54
800	907	0.001	0	17.9	37	71
900	994	0.001	0	19.8	35	60
1000	1152	0.001	0	23.1	78	75

The experiments first record the time and space cost for constructing TRS for graphs. We generated an artificial graph, whose number of nodes ranging from 100 to 1000, and the saturations are mostly lower than 1%. In the first experiment, the graph contains one TRS. The first three columns of Table 1 are the number of nodes, number of edges and saturation. And the next two columns list the time cost and space cost for TRS construction.

From this table, we can see the time cost and space cost for construction is quite small for graphs with such scale. And other features such as RE number and SE number are also shown in the table.

To test the correctness and efficiency of our TRS structure and algorithm, we designed a group of experiments. To test the correctness, we run AnswerSPQ on TRS and Dijkstra algorithm using a group of identical queries, and then compare the results. Table 2 indicates that our algorithm can find out the correct shortest path between any two pair of nodes if the path exists.

Table 2. Correctness verification of TRS

Query Points(s, t)	Result using Dijkstra	Result using TRS
(221, 88)	<221, 64, 166, 217, 1, 49, 181, 93, 59, 184, 302>	<221, 64, 166, 217, 1, 49, 181, 93, 59, 184, 302>
(118, 82)	<118, 82>	<118, 82>
(37, 359)	<37, 35, 2, 41, 117, 65, 262, 220, 237, 264, 123, 225, 125, 134, 251, 73, 212, 209, 72, 14, 414, 164, 359>	<37, 35, 2, 41, 117, 65, 262, 220, 237, 264, 123, 225, 125, 134, 251, 73, 212, 209, 72, 14, 414, 164, 359>
(48, 483)	<48, 51, 57, 248, 350, 287, 224, 429, 193, 203, 328, 288, 425, 148, 483>	<48, 51, 57, 248, 350, 287, 224, 429, 193, 203, 328, 288, 425, 148, 483>
(444, 439)	<444, 439>	<444, 439>

Table 3. Result on graph

Node no.	Edge no.	Dijkstra (ms)	TRS (ms)
100	129	0.016	0.109
200	237	0.094	0.047
300	348	0.219	0.093
400	463	0.438	0.110
500	577	0.718	0.172
600	671	0.922	0.110
700	807	1.391	0.157
800	907	1.734	0.250
900	994	2.390	0.141
1000	1152	2.860	0.281

To test the efficiency of TRS, 1000 queries using AnswerSPQ on TRS and Dijkstra algorithm respectively are carried on the graph, and the average time cost is recorded as follows in Table 3.

In order to clearly see our improvement, we give out Fig. 8. The blue line is the cost for Dijkstra algorithm and the red dotted line is the cost for TRS. In the figure, we can see when the graph is small, Dijkstra performs better than TRS. But as the size of graph grows, time cost of Dijkstra increases sharply, while our TRS shows a slow rate of rise, and lead to a much better performance than Dijkstra when graph size is 1000.

In addition, we collected the wikipedia data for our real graph experiments. In this data set, node represents the web page and edge represents the link from one page to another. Total number of nodes for this graph is 1100000, total number of edges is 1133321, and the saturation is 0.0000113 %. Construction of this graph totally outputs 37048 TRSs, and 151639 boundary nodes. The biggest TRS contains 5667 nodes and 6547 edges, and 1934 boundary nodes. Table 4 gives the result performing AnswerSPQ on this real graph.

Because the real graph is so large, that it's necessary to make preparation for computing. The pre-computation phase can be divided to three parts. The first is to parse the graph to forests, the second is to generate TRS for each tree in the forest, and the third is to compute local shortest paths between boundary nodes. Since we do not directly store all these information in memory, all of the three parts include disk I/O time cost. The query time 17.5 milliseconds is also the average query time for 1000 different queries, and for such a big graph, the response time is acceptable while the BFS method cost 32.47 milliseconds in a graph contains 592,983 nodes.

Table 4. Result on Wikipedia graph

Node No.	Edge No.	Space (MB)	Pre-computation time (min)	Query time (ms)
1,100,000	1,133,321	36.853	107	17.5

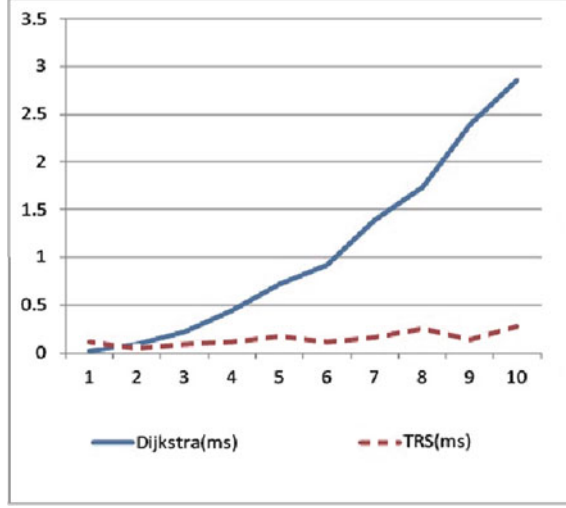


Fig. 8. Result on graph

7 Conclusion and Future Work

Answering shortest path query on a graph which does not fit in memory is now a very important problem with the development of networks. In this paper, we first introduce Interval-Based Encoding into the graph re-construction stage and give out a new structure TRS of the graph. For most graphs, the construction time is short. Proved by experiments, TRS structure performs quite well, and time cost increase almost linear respect to the graph scale. When AnswerSPQ on TRS algorithm computes shortest paths in graph with more than one TRS, this method does not perform quite well when graph is small, but our experiment results shows that, it performs well when graph is large.

In our experiments, we notice that the pre-computation time cost is high, and it mainly consumes on disk I/O and computing local shortest path. For disk I/O, our next attempt is to maintain graph in memory when the scale is not large. And if a TRS contains too many boundary nodes, the cost is high. E.g., in the TRS we mentioned above which contains 1,935 boundary nodes, total 3,744,225 queries need to be answered. So we will focus on how to improve efficiency on graph with a lot of boundary nodes. Besides, the graph tends to be dynamic in real world, how to maintain the query efficiency using TRS structure when either some nodes or edges change remains to be studied further.

References

1. Wei, F.: TEDI: Efficient shortest path query answering on graphs. In: SIGMOD'10 (2010)
2. Xiao, Y., Wu, W., Pei, J., Wang, W., He, Z.: Efficiently indexing shortest paths by exploiting symmetry in graphs. In: EDBT'09 (2009)
3. Goldberg, A.V., Kaplan, H., Werneck, R.: Reach for A*: efficient point-to-point shortest path algorithms. In: Workshop on Algorithm Engineering and Experiments, pp. 129–143 (2006)
4. Arge, L., Meyer, U., Toma, L.: External memory algorithms for diameter and all-pairs shortest-paths on sparse graphs. In: Díaz, J., Karhumäki, J., Lepistö, A., Sannella, D. (eds.) ICALP 2004. LNCS, vol. 3142, pp. 146–157. Springer, Heidelberg (2004)
5. Gutman, R.: Reach-based routing: a new approach to shortest path algorithms optimized for road networks. In: Proceedings of the 6th International Workshop on Algorithm Engineering and Experiments, pp. 100–111 (2004)
6. Wagner, D., Willhalm, T.: Geometric speed-up techniques for finding shortest paths in large sparse graphs. *Konstanzer Schriften in Mathematik und Informatik*, pp. 1430–3558 (2003)
7. Chan, E.P.F., Zhang, N.: Finding shortest paths in large network systems. Department of Computer Science, University of Waterloo (2001)
8. Dijkstra, E.W.: A note on two problems in connexion with graphs. *Numerische Mathematik* (1959)
9. Cormen, T.H., Leiserson, C., Rivest, R., Stein, C.: *Introduction to Algorithms*. MIT Press, Cambridge (2001)
10. Agrawal, R., Jagadish, H.V.: Algorithms for searching massive graphs. *IEEE Trans. Knowl. Data Eng.* **6**, 225–238 (1994)
11. Eietz, P., Sleator, D.: Two algorithms for maintaining order in a list. In: Proceeding of the 19th Annual ACM Symposium on Theory of Computing (STOC), pp. 365–372 (1987)
12. TRßl, S., Leser, U.: Fast and practical indexing and querying of very large graphs. In: SIGMOD'07 (2007)
13. Wang, H., He, H., Yang, J., Yu, P.S., Yu, J. X.: Dual labeling: answering graph reachability queries in constant time. In: Proceedings of the 22nd International Conference on Data Engineering (ICDE), p. 75 (2006)
14. Lauther, U.: An extremely fast: exact algorithm for finding shortest paths in static networks with geographical background. *IfGIprints 22*, Institut fuer Geoinformatik, Universitaet Muenster (ISBN 3-936616-22-1), pp. 219–230 (2004)

Social Media Retrieval and Mining

ADMA 2012 Workshops, SNAM 2012 and SMR 2012,

Nanjing, China, December 15-18, 2012. Revised

Selected Papers

Zhou, S.; Wu, Z. (Eds.)

2013, X, 167 p. 49 illus., Softcover

ISBN: 978-3-642-41628-6