Journal of King Saud University – Computer and Information Sciences 33 (2021) 364–373

Contents lists available at ScienceDirect

Journal of King Saud University – Computer and Information Sciences

journal homepage: www.sciencedirect.com

Dynamizing Dijkstra: A solution to dynamic shortest path problem through retroactive priority queue

Sunita [a,*], Deepak Garg [b]

[a] Computer Science and Engineering, Thapar University, Patiala 147004, India
[b] Computer Science Engineering Department, Bennett University, Greater Noida, Bennett University, Noida 201310, India

ARTICLE INFO

Article history:
Received 31 October 2017
Revised 2 March 2018
Accepted 3 March 2018
Available online 6 March 2018

Keywords:
Data structure
Algorithms
Dynamic shortest path
Retroactive data structure

ABSTRACT

Dynamic shortest path algorithms are the ones which are used to accommodate the online sequence of update operations to the underlying graph topology and also facilitate the subsequent query operations. Many solutions exist for the different versions of the problem, all of which identify the set of vertices whose shortest paths may be affected by the changes and then update their shortest paths according to the update sequence. In this paper, we are dynamizing the Dijkstra algorithm which helps to efficiently solve the dynamic single source shortest path problem. Dynamization is achieved by using the retroactive priority queue data structure. Retroactive data structure identify the set of affected vertices step by step and thus help to accommodate the changes in least number of computations. So, with a suitable dynamic graph representation and the use of retroactive priority queue, we have proposed algorithm to dynamize Dijkstra algorithm giving solution of dynamic single source shortest path problem with complexity O(nlg m) for the update time. We have performed experimental analysis by comparing the performance of the proposed algorithm with other algorithms. Our experimental results indicate that the proposed algorithm has better performance in terms of time and memory usage.

1. Introduction

General graph problems can be either static or dynamic. Dynamic graph problems are the problems in which the underlying graph has no restriction in terms of updation of its different parameters (like edges, vertices or weights). These problems are more challenging and time consuming as compared to the static ones. One of the most widely studied dynamic graph problems is the dynamic shortest path problem. The shortest path problem is a generic problem with applications in many different fields such as Operation Research, Management Systems, Computer Science and Artificial Intelligence. The main reason for its use in such diverse fields is that essentially any combinatorial optimization problem can be formulated as a shortest path problem (Rana and Garg, 2014; Sarnak and Tarjan, 1986; Yigit and Unsal, 2016).

Dynamic single source shortest path problem is a type of dynamic shortest path problem which gives shortest paths from a source vertex to all the other vertices of a graph in dynamic scenario. Almost all of the previous approaches for the solution of dynamic single source shortest path problems are based on identifying all the vertices which may be affected by the given changes in the graph and then updating the shortest paths accordingly. Another way to solve this problem is to make the Dijkstra algorithm dynamic. The Static Dijkstra algorithm is an iterative algorithm which is used to find the shortest path from a specific vertex of the graph called as source vertex to all the other vertices of the graph (Dijkstra, 1959). The working of the algorithm can be mathematically represented as:

$$\text{if } d[u] + w(u,v) < d[v]$$
$$\text{then } d[v] = d[u] + w(u,v)$$

where d[u] represents the estimated distance of vertex u from the source vertex and w(u,v) represents the weight of edge (u,v). The data structure used in the basic algorithm is the Priority Queue.

For dynamizing the algorithm, we are using the new data structuring paradigm, i.e. retroactive data structure introduced by

* Corresponding author.
E-mail addresses: sunita.tu@gmail.com (Sunita), deepakgarg108@gmail.com (D. Garg).

Peer review under responsibility of King Saud University.

Production and hosting by Elsevier

https://doi.org/10.1016/j.jksuci.2018.03.003
1319-1578/

Demaine et al. (2007) for this purpose. Kanat Tangwongsan (Acar et al., 2007) introduced a variant of these data structures called as non-oblivious retroactive (active) data structures. In the retroactive data structure, the changes arising in the data structure due to some modification in the past are automatically accommodated in the present but in the non-oblivious retroactivity the changes are propagated one by one i.e. the user is informed of the first operation that is effected due to the modification done in the past. We are basically using the active data structures (i. e. non-oblivious retroactivity) for the solution of our problem. Henceforth in the paper we will refer to non-oblivious retroactive data structure as retroactive data structure.

Primary work done in the paper is:

- We have proposed a novel approach for solving the dynamic shortest path problem by dynamizing the corresponding static algorithm of the problem by using retroactive data structure.
- For dynamizing the algorithm, we have given an implementation for the retroactive priority queue using red–black trees. We have validated the correctness of our approach with the help of an example graph.
- We have implemented proposed algorithm and also performed experimental analysis by comparing the performance of the proposed algorithm with other algorithms.

The paper is organized as: Section 2 reviews the basic terms and notations used in the paper. Section 3 introduces the proposed work and gives the proposed algorithm and theoretical validation with the help of an example graph. Section 4 describes the experimental analysis. Finally, Section 5 concludes the work.

2. Related work

Dynamic shortest path problems have been classified as: semi-dynamic and fully dynamic. Semi-dynamic can be either incremental or decremental. Incremental means only insertions/ edge weight decreases are allowed and decremental means only deletions/edge weight increases are allowed. Fully Dynamic problems can have both types of update operations applied to them. Some authors also considered the number of update operations as a classification criterion for the class of problems and proposed solutions to the problem which can handle a single update operation at a time (Buriol et al., 2003; Frigioni et al., 2000; Xiao et al., 2004). Two different approaches have been proposed in (Narviaez et al., 2001; Ramalingam and Reps, 1996) which allow update operations to be handled as a batch. Many approaches have also been proposed by the authors for the solution of the semi dynamic and fully dynamic variants of the problem like Gallo (Gallo, 1981) and Fujishige (Fujishige, 1981) gave solution for the semi-dynamic incremental problem and Even and Shiloach (Even and Shiloach, 1981) gave O(mn) update time algorithm for the decremental case.

Ramalingam and Reps (Ramalingam and Reps, 1996) gave the generalized algorithm for the incremental variant which was proved to be the best one in practice (Buriol et al., 2003; Demetrescu and Italiano, 2006) despite of its high worst case complexity. The idea behind the Ramalingam and Reps algorithm is to identify the set of affected nodes/vertices. This set of affected nodes is further divided into two different sets. One set represents those affected nodes whose shortest path could not be calculated using the current topology of shortest path tree. This set of nodes is stored in a queue (Q). Another set of affected nodes is the one whose updated shortest path could be calculated from the current topology of shortest path tree. This set is maintained in a heap (H) data structure. The shortest paths of nodes in queue (Q) are updated using the nodes in the heap (H). The algorithm is till date

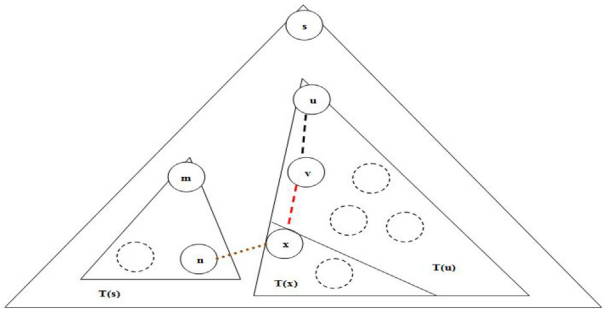

Fig. 1. Shortest path tree representation of Ramalingam and Reps approach.

the best one practically and has been used in the solution of many dynamic graph problems like betweeness centrality (Bergamini and Meyerhenke, 2015; Bergamini et al., 2015; Kas et al., 2013, 2014). Fig. 1 shows how the shortest path tree generated in the Ramalingam and Reps algorithm is used in the identification of affected vertices after an edge (u,v) weight is incremented.

The unaffected vertices in the shortest path tree of vertex u due to updation are represented as dotted. Dotted red edge between vertices v and x shows that it is no longer a shortest path (after updation of edge (u,v)). So, an alternative shortest path is identified (which is edge (n, x) and now the shortest path tree of edge is a part of the shortest path tree of vertex m instead of vertex u (as before updation).

Further, King and Thorup (King and Thorup, 2001) have given a specialization of the above algorithm. They have proposed to use a variation of the shortest path tree. In this shortest path tree each node along with maintaining the shortest paths, also maintain a special link pointing to the next shortest path. The advantage of storing this special tree is to be able to find an alternative path with no need to explore all outgoing arcs in the set.

Further, Demterscu et al. (Demetrescu and Italiano, 2003) proposed to use a simpler mechanism to identify the set of affected nodes belonging to queue (Q). Rest of the algorithm is same as that of the Ramalingam and Reps. So, the queue size in case of this algorithm may be larger than that of Ramalingam and Reps algorithm.

Furthermore, Thorup (Thorup, 2004) proposed solution for fully dynamic All Pairs Shortest Path problem with $O(n^2(\log n + \log^2((m + n)/n))$ amortized update time which has at least log factor improvement over the $\Omega(n^2)$ best-case time. Researchers attention moved towards approximation algorithms for these problems (Bergamini et al., 2015; Bernstein and Roditty, 2011; Bernstein, 2013; Buriol et al., 2003; Henzinger et al., 2015) by the seminal paper of Roditty and Zwick (Roditty and Zwick, 2012) proving that $O(mn)$ update time is optimal for exact distances and hence there is almost no scope in improving the already existing approaches. Mostafa Dahshan (Dahshan, 2013) have also given modified Dijkstra's algorithm and applied it to find the shortest path on the dual/network topology.

3. Proposed work: Dynamizing Dijkstra algorithm using retroactivity for solving shortest path problem

In this paper, we propose to dynamize the *Dijkstra algorithm* using retroactive data structures. These data structures help to maintain the historical sequence of events on a data structure. So, these can be effectively used for the dynamization of static algorithms. As priority queue is used in the static implementation of the algorithm, so using retroactive priority queue we can dynamize the algorithm. For solving dynamic shortest path problem using *Dijkstra* algorithm and the retroactive priority queue, we

need an implementation of the retroactive priority queue and a suitable dynamic graph representation.

First of all we give the basic notations used throughout the paper. Then we define how *Dijkstra* can be made dynamic by replacing the data structure i.e. priority queue by retroactive priority queue. After that the representation used for the underlying dynamic graph and implementation details for the retroactive priority queue using height balanced trees are given. In the next section, we specify the steps that need to be added to the static *Dijkstra* to convert it to its dynamic counterpart.

3.1. Basic notations

Let G = (V, E, w) be a simple directed graph, where V and E are the sets of vertices and edges, respectively, and w is a function from E to the set of non-negative real numbers i.e. w gives the weights of the corresponding edges. Let e = (u, v) ε E; then u is the head of e denoted as e_h, and v is the tail of e denoted as e_t. Each vertex v is represented by a key (like a, b etc.), and so is each edge e (*as* (*a,b*), (*a,c*) etc.). An edge e (u, v) in the graph is assigned with a weight w (u, v). Each vertex v contains the auxiliary information such as estimated distance which denotes the predicted shortest distance of that vertex from the source vertex and the predicted shortest path predecessor gives the immediate predecessor of that vertex on the predicted shortest path.

For the dynamic shortest path problem, we consider that the edge weight changes can be in any of the forms: edge weight increases only or edge weight decreases only or both edge weight increases and decreases. Although we are considering only edge weight changes, but edge insertion/deletion can also be handled by these as edge insertions can be considered as edge weight decreases from ∞ to the weight of the inserted edge. Likewise, edge deletions can be considered as edge weight increases by changing the edge weights of the deleted edge to ∞.

Now, we explain the priority queue, Let *Q* be a retroactive priority queue. An entry in *Q* is of the form (ver, ins_time, dist, pred, del_time) in which *ver* is a vertex, ins_time is the time at which that node has been inserted in the queue, *dist* contains the estimated shortest distance of the vertex from the source vertex, pred is the predecessor vertex for that vertex on the predicted shortest path and del_time is the time at which node is deleted from the queue. Nodes in the queue are ranked based on two parameters: first according to the distance value (dist) and then on time. If more than one entry has same dist, then the sequence among them is arbitrary. *Q* supports different operations, which are explained in Section 3.2.2 in detail.

As, we are dynamizing *Dijkstra* algorithm, so we need a graph representation that efficiently accommodates the dynamic updations of edge weights. We have chosen a simple but effective representation for our purpose. The graph is represented using adjacency matrix which stores graph as a matrix of neighbors (assuming no new vertices are added to the graph). No doubt the representation will be expensive in case the underlying graph is sparse, but in this representation any updation can be performed in constant time and thus changing the graph structure is very easy in this representation. Hence we have chosen the representation so that the time bounds are not affected due to the changes that need to be made on the graph.

3.2. Retroactive priority queue using balanced search trees

To dynamize Dijkstra algorithm, we wish to maintain a set of shortest paths for the graph whose edge weights change and hence the priority of edges to be selected in shortest paths changes over time. So, for this purpose we use the retroactive priority queue which allows to perform opeartions at any point of time. All the operations of a simple priority queue are done w.r.t time in retroactive priority queue, i.e. operations *Insert* and *Del_min* can be invoked as well as revoked at any time. So, the main operations, that are defined for the retroactive priority queue are as follows:

- ***Invoke (Insert (x, t))****:* performs an insertion of the item x at time t i.e. a new element x is inserted into the priority queue at time t according to its priority value.
- ***Invoke (Del_min (t)):*** performs the deletion of minimum value from the priority queue at time t.
- ***Revoke (Insert (t)):*** Undo the insert operation performed at time t.
- ***Revoke (Del_Min (t)):*** Undo the delete operation performed at time t. This is equivalent to inserting the item deleted at time t again in the priority queue.
- ***Find_min(t):*** Returns the minimum element existing in the priority queue at time t.

The detailed algorithms of these operations are given in next sub-section. Invoking an operation means performing the operation at any time present or past while revoking means undoing an operation performed at some time in the past. Ties in operation time are broken by the order in the sequence of operations. The time parameter t used in the operations, allows to maintain the historic sequence of operations.

In the remainder of this section, we shall review balanced binary search trees in brief and how retroactive priority queue data structure can be implemented using these.

3.2.1. Height-balanced search trees

A height-balanced binary search tree is a binary search tree that automatically keeps its height (i.e. balances its height) small in the presence of online insertions and deletions of items. These binary trees balance the height by performing transformations on the tree (such as tree rotations) at key times, in order to keep the height proportional to $log_2(n)$. No doubt an overhead is involved in performing these transformations, but it is overcome by fast execution of all other operations (Knuth, 1998). Various height balancing binary trees have been defined like: AVL trees (Adelson-Velskii and Landis, 1962), weight-balanced trees (Nievergelt and Reingold, 1973), red black trees (Guibas and Sedgewick, 1978) and treaps (Cecilia and Raimund, 1989) etc.

In all types of height–balanced search trees balance information is maintained in each node and rebalancing is done after each insertion or deletion by performing a series of transformations (rotations) along the access path (the path from the root to the inserted or deleted item). Transformations in the red–black trees are more efficient as compared to the transformations of other kinds of height-balanced trees (Sarnak and Tarjan, 1986). As we need to use these trees as a data structure for the implementation of priority queues incorporating dynamic changes, so the cost of transformations of the tree will affect the efficiency of our implementation and red black trees have the efficient transformations. So, as per our requirements, *red black tree* is the suitable representation.

3.2.2. Operation definitions

To implement the retroactive priority queue, we need to have a way to maintain the lifetimes of all the elements along with the elements themselves. So, to have such functionality, we maintain 2 red–black trees: T_{ins} and T_{d_m}, which maintain the sets *Insert* and *Del_Min* respectively. The set Insert maintains the key pair *(x, t)* where *x* denotes the item and *t* denotes the time of insertion of item *x*. Also set *Del_Min* maintains the times at which the del_min operations are performed. The tree T_{d_m} is ordered by time. The other tree T_{ins} is indexed by the item value and second indexing is according to the time, i.e. $(k_1, t_1) < (k_2, t_2)$ if and only if $(k_1 < k_2)$ or $((k_1 = k_2) \wedge (t_1 < t_2))$. The two trees have links between its

nodes, as each node of tree T_{d_m} is linked with a node of T_{ins} whose key is the return value for that del_min operation. On the similar lines each node of T_{ins} is linked with its corresponding del_min if that node is a result of Revoke (Del_Min(t)) operation. Whenever del_min operation is performed the node that is to be deleted from tree T_{ins} is not actually deleted, but is marked as invalid as we need to maintain the past information also. At a later time if del_min is revoked that node is again marked as the valid one. The retroactive priority queue also needs to identify the operations whose return values are affected, in addition to above operations. The definitions of all the above operations as pseudo code and in terms of trees T_{ins} and T_{d_m} have been represented with example as follows: Fig. 2 represents the configuration of both the trees before we start applying retroactive updations to these.

i) ***Invoke _Insert (x, t))*** performs an insertion of the key (x, t) into the tree T_{ins} at time t according to its priority value. Insertion in the tree is done in following steps:

(i) If t is the present time means no del_min has been performed after time t then search into the tree according to the key value and then, according to time to find the location of the key to be inserted. Where the search terminates, attach a new node containing the new key (x, t).

(ii) Else find the minimum value say k in tree T_{d_m} that has been deleted after time t and return this as the first inconsistent operation while inserting x at time t in the priority queue.

Fig. 3 shows the affect of applying operation Invoke_Insert(7, 11) to the trees of Fig. 2.

```
Invoke_Insert (x, t))
Insert (T_ins, x, t);
// Insert x into height balanced tree T_ins at time t.
P = Search_Min (T_d_m, t);
//Search in tree T_d_m for minimum key value deleted after
   time t.
if (P ! = NULL)
  return (P);
// Return the first inconsistent operation.
  else
  return (NULL);
```

ii) ***Invoke_Del_min (t))*** performs the deletion of minimum value from the priority queue at time t. To perform the deletion we have to check for the delete minimum operations performed before that time. Steps taken to perform del_min are as:

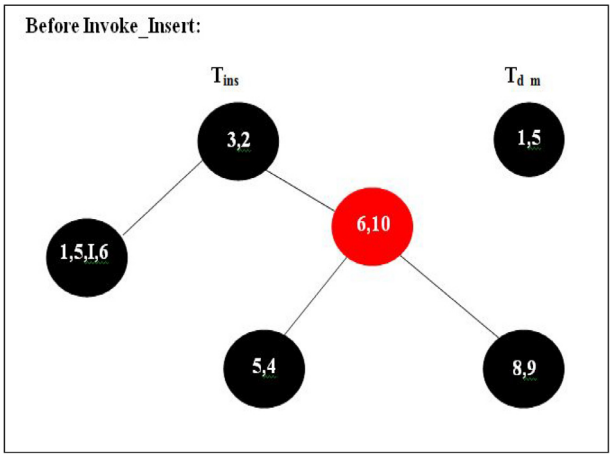

Fig. 2. Trees T_{ins} and T_{d_m}.

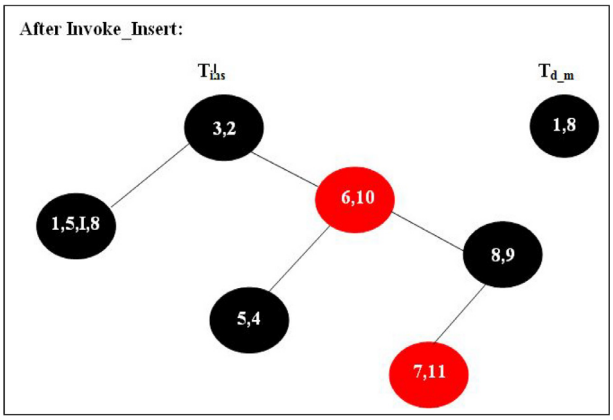

Fig. 3. Trees T_{ins} and T_{d_m} after Invoke_Insert(7,11) operation in priority queue.

i) If del_min is to be performed at the most recent time, then simply find the minimum key value from T_{ins} and delete it from T_{ins} and insert it into T_{d_m}.

ii) Else from the tree T_{d_m} find the key value k' that has been deleted at a time immediately before given time t. Now from tree T_{ins} find the minimum key k greater than key k'. k is the result of del_min operation.

Fig. 4 shows how the trees in Fig. 3 change when Invoke_Del_Min(8) operation is applied to them.

```
Invoke_Del_Min (t))
P = Search (T_d_m, t′);
// where t′ = max (t″ε T_d_m and t″ < t)
  if (P = Null)
// No del_min has been performed before time t
     N = Find_min(T_ins);
     Return (N);
  else
     K′ = P.Data;
// where k‘ is the maximum value deleted immediately before
   time t.
     Search (T_ins, k)
// where k = min (k″ ε T_ins and k″ > k′)
```

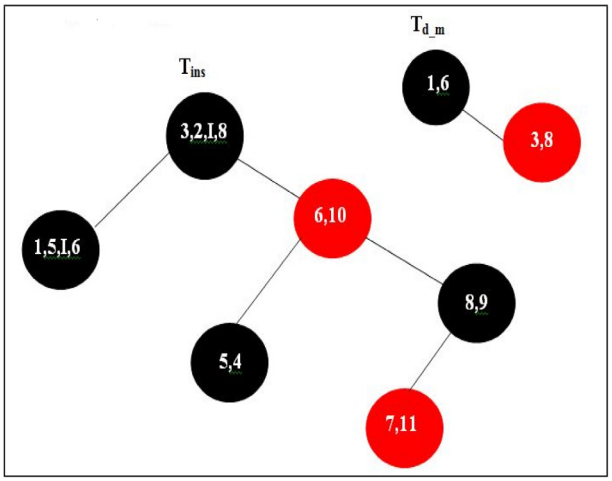

Fig. 4. Trees T_{ins} and T_{d_m} after Invoke_Del_Min(8) operation in priority queue.

iii) ***Revoke_Insert (t))*** undo the insert operation performed at time t. To do this, check if any del_min operation has been performed after this time t.
 i) If yes, return the first del_min performed after time t as the inconsistent operation.
 ii) Else, make the node inserted at time t as invalid and find the entry in tree T_{ins} corresponding to time t and mark this entry as invalid.

Fig. 5 shows how the trees in Fig. 4 change when Revoke_Insert (4) operation is applied to them.

```
Revoke (Insert (t))
P = Search_Min (T_d_m, t);
//Search in tree T_d_m for minimum key value deleted after
   time t.
     if (P ! = NULL)
     return (P); // Return the first inconsistent operation.
   else
     P = Search (T_ins, t);
     P.Valid = False;
```

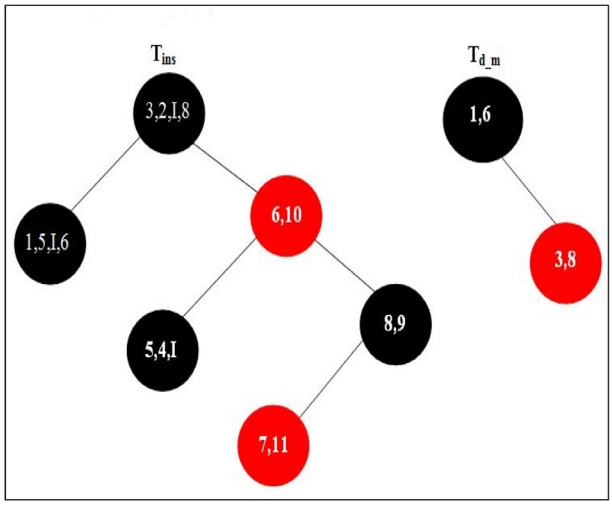

Fig. 5. Trees T_{ins} and T_{d_m} after Revoke_Insert(4) operation in priority queue.

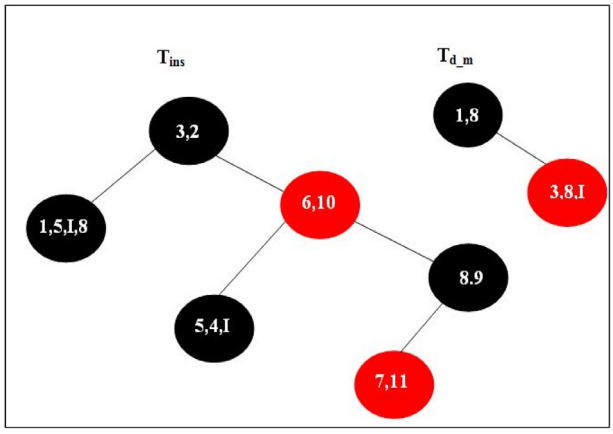

Fig. 6. Trees T_{ins} and T_{d_m} after Revoke_Del_Min(8) operation in priority queue.

iv) ***Revoke_Del_Min (t))*** Undo the delete operation performed at time t. This is equivalent to inserting the item deleted at time t again in the priority queue. Fig. 6 shows the affect of applying operation Revoke_Del_Min(8) on Fig. 5.

```
Revoke (Del_Min (t))
P = Search_Min (T_d_m, t);
  //Search in tree T_d_m for minimum key value deleted after
  time t.
    if (P ! = NULL)
      return (P);
  // Return the first inconsistent operation.
  else
    Insert (T_ins, x, t);
  // Insert x into height balanced tree T_ins at time t
```

v) ***Find_min(t):*** Returns the minimum element existing in the priority queue at time t. Simply perform the search operation in tree T_{ins} corresponding to time t if t is the recent time after which no del_min has been performed else find the maximum key value (say k‘ from the tree T_{d_m} that has been deleted from priority queue before given time t and then from tree T_{ins} find the minimum key that is greater than k‘ i.e. k‘‘= min{k ε T_{ins} and k is a valid entry and k‘‘ > k‘}. K‘‘ is the minimum value in the priority queue at time t.

Fig. 7 gives the final trees representing priority queue after applying operation Find_Min(13).

```
Find_min(t)
  P = Search_Min (T_d_m, t);
  //Search in tree T_dm for minimum key value deleted at time
  t.
    if (P = NULL)
      Q = Search_Min(T_ins, t);
  // Return the minimum key from T_ins at time t.
      return (Q.val);
    else
      return(P.val);
```

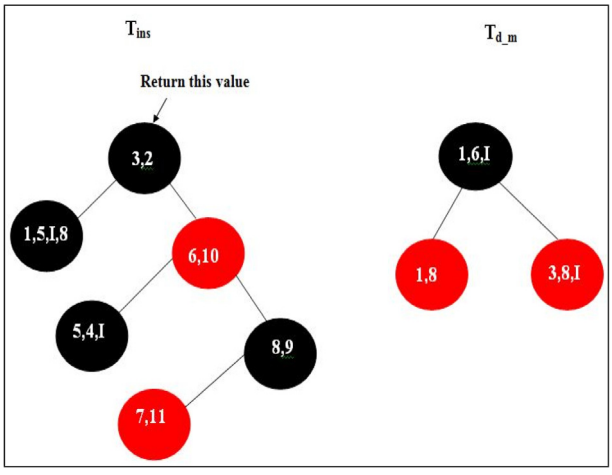

Fig. 7. Trees T_{ins} and T_{d_m} after Find_Min(13) operation in priority queue.

3.3. Proposed Algorithm: Dynamic Dijkstra using retroactive priority queue

The proposed algorithm is a modification of the standard *Dijkstra's* algorithm with the modification that the graph is dynamic one and any change in the edge weight of the graph is also input into the priority queue during the algorithm execution. Priority queue accordingly accommodates the change and returns the first operation in the operation sequence that has been affected due to the change in the edge weight.

3.3.1. Algorithm: Dynamic Dijkstra (D_Dij)

In the dynamic Dijkstra algorithm we are first checking whether the update operation is effecting the operations performed till now and if yes identify those operations and redo them to accommodate the change. The steps of the proposed algorithms are mentioned below:

step 1: Check whether the head vertex of updated edge is there in the retroactive priority queue (RPQ).
step 2: If it is not in RPQ (means no previously calculated path will be affected by this change) then go to Step 6.
step 3: Else If the entry corresponding to head vertex of updated edge is an active entry in the RPQ (the vertex is not used in any of the existing shortest paths), then go to Step 4, Else go to Step 5.
step 4: If update is negative (i.e. weight of edge has been decreased), then update the distance of tail vertex of updated edge and go to step 1, Else go to Step 6.
step 5: Else move in the priority queue immediately before the time that edge has been deleted and accommodate the change as required. Go to Step 1.
step 6: Exit

Further, these steps are described in detail through a flow chart Fig. 8) as well as pseudocode.

Dynamic Dijkstra (D_Dij)

// Weight of edge e has changed by say α(increased) or -α (decreased)
1: N = Search(T_{ins}, e_h);
// e_h denotes the head vertex of edge e
2: if (N == NULL) then
3: Exit
// No change in shortest paths, proceed normally
4: else
if N is active entry in the priority queue then
// N has not been deleted yet
5: if pred[e_h] == e_t then
6: Update estimated distance (d[e_t]) for the vertex e_t in priority queue.
7: else
Move in the priority queue before time when e_h was deleted say node m.
9: If (pred(e_h) == e_t) then
10: Update estimated distance (d[e_t]) for the vertex $\mathbf{e_t}$ in priority queue.
11: priority queue returns the del-min operation effected by this change, say node n.
/* Relax outgoing edges of the vertex of node n (n.v).*/
12: for each *e' ε n.v* do
13: if (pred [e'$_h$] == e_h) then
14: d[e'$_h$] = ∞
15: Relax edge e'
if relaxed insert(T_{ins},t.val)
16: insertion returns the next operation effected in the retroactive priority queue, say node m. Go back to step 11
17: end if
18: end for
19: end if
20: end if
21: end if

3.3.2. Example

Using the graph of Fig. 9, we are showing the actions performed by our algorithm when edge weights of the edges are allowed to change (i.e. increase as well as decrease). Each row below represents the state of the priority queue at time t, i.e. each entry in the priority queue is a valid entry in the queue at that specific time. Each entry in the queue is a triplet (v, val, pred) where v denotes the vertex, val denotes predicted distance of that vertex from source vertex and pred gives the predecessor of that vertex on the predicted shortest path.

Time				Priority_Queue
t = 0				O,0,Nil
t = 1	A,2,O		C,4,O	B,5,O
t = 2	C,4,O	B,4,A	D,9,A	F,14,A
t = 3	B,4,A	E,8,C	D,9,A	F,14,A
t = 4	E,7,B		D,8,B	F,14,A

At time t = 5 the weight of edge BD is increased to 5.

As Vertex D is still in the Priority queue that means edge BD has not been included in any shortest path till now. So, increment the estimated distance of vertex D by 1 and proceed normally.

t = 5	E,7,B		D,9,B	F,14,A

Or Suppose,

At time t = 5 the weight of edge BD is decreased to 1.

As Vertex D is still in the Priority queue that means edge BD has not been included in any shortest path till now. So, update the estimated distance of vertex D by 3 and proceed normally.

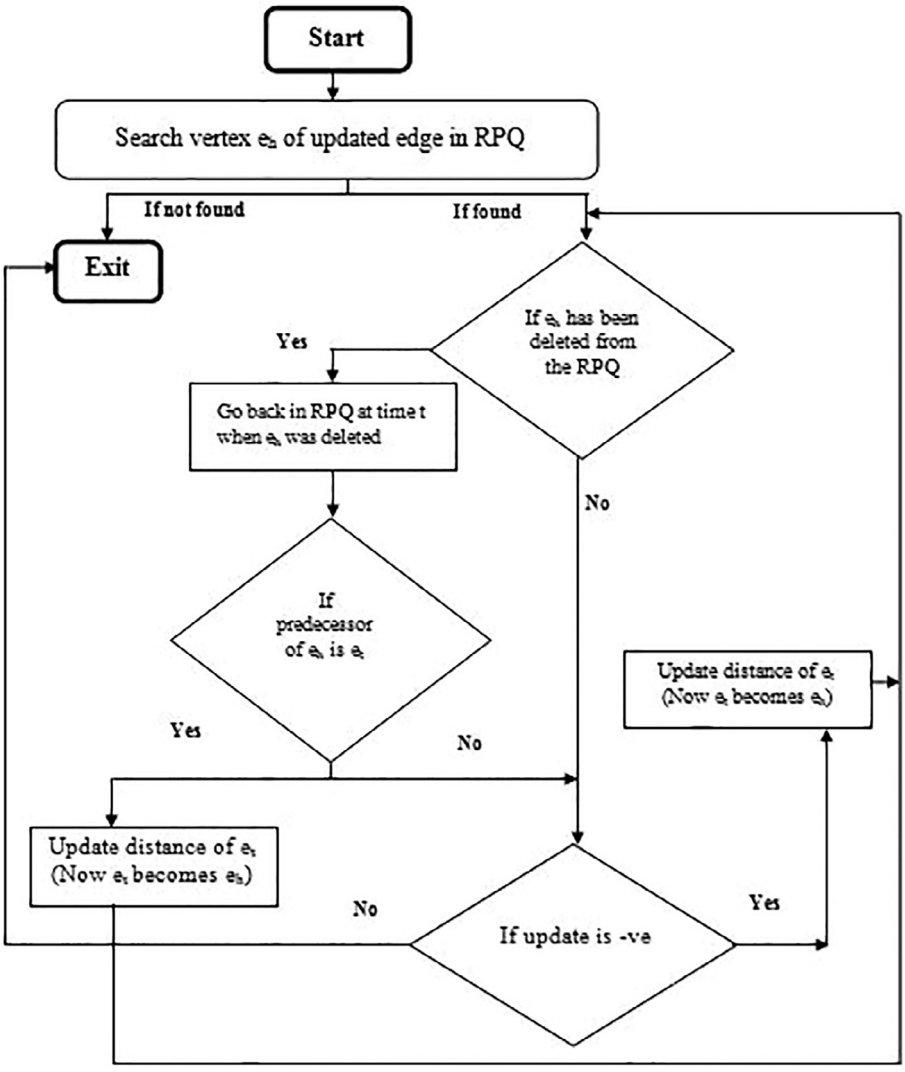

Fig. 8. Flow chart of Dynamic Dijkstra (D_Dij).

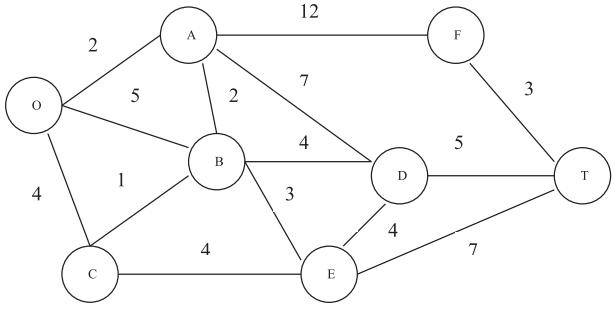

Fig. 9. Example Graph for dynamic Dijkstra algorithm.

t = 5	E,7,B	D,5,B	F,14,A
t = 6	D,9,B	F,14,A	T,14,E

At time t = 7 the weight of edge OB is increased to 7.

As vertex B has been previously in the priority queue, we need to follow the paths in which edge OB has been used, so that they can be updated accordingly. Move in the priority queue at time immediately before the deletion of vertex B i.e. at time 3. We need not change the value of shortest path of B as it is less than the new one including the changed weight for edge OB. So, there will be no change in shortest paths in this case also and we proceed normally.

t = 8	T,13,D	F,14,A

At time t = 9 the weight of edge AB is decreased to 1.

As vertex B has been previously in the priority queue, move in the priority queue at time immediately before the deletion of vertex B i.e. at time 3. Predecessor of B at that point is A, so its shortest path becomes 3. As value in the retroactive priority queue has

changed, it will automatically rearrange itself and the return the operation effected by this change.

At this point priority queue can be represented as:

t = 3	B,3,A	E,8,C	D,9,A	F,14,A

So, del_min for will be again performed with new value of distance for vertex B.

t = 10	E,6,B	D,7,B	T,13,D	F,14,A

Next operation effected is for vertex E. Now perform del_min for E.

t = 11	D,7,B	T,13,D	F,14,A

Next operation effected is for vertex D. Perform del_min for D.

t = 12	T,12,D	F,14,A

Now, we have reached to the point at which normal execution of Dijkstra should resume as all changes have been propagated completely.

In next section, the proposed algorithm is empirically analyzed and comparative results are presented.

4. Empirical analysis

An empirical analysis has been performed by comparing proposed alogorithm with two practically efficient algorithms. Here, the performance of our Dynamic Dijkstra (D_Dij) algorithm has been compared with the algorithms given by Ramalingam et al. (Ramalingam and Reps, 1996; Ramalingam and Reps, 1996) and Demetrescu et al. (Demetrescu and Italiano, 2003). The algorithm by Ramalingam et al. is known to be very fast in practice (Fortz and Thorup, 2000; Frigioni et al., 1998, 2000), it works by identifying the subset of vertices whose distance from source s is affected by the update. Distances are then updated by running a Dijkstra algorithm on those nodes. The algorithm by Demetrescu et al. uses the idea of uniform paths in Dijkstra-like algorithm so that the number of total edge scans can be reduced and performance bottleneck can be overcome. We have also compared our proposed algorithm with the heap reduction technique given by Buriol et al. (Buriol et al., 2003) applied to the algorithms of Ramalingam et al. (Ramalingam and Reps, 1996) and Demetrescu et al. (Demetrescu and Italiano, 2003).

The experiments were performed on a 2.30 GHz Intel Pentium III processorcomputer with 2 GB of RAM. The implementation or simulation has been performed using C Language (compiled with gcc-4.8.2 using the –O3 optimization option). CPU times were measured with the system function and memory profiling was done using tool Valgrind.

In our experiments, we have considered two types of graphs. One is random graph data sets as described in Table 1. Random graphs with n nodes, m edges and random integer edge weights are generated. Update sequence is also generated randomly by selecting any of the update operation on any random edge. Another graph data set has been taken from the experimental package (http://www.dis.uniroma1.it/~demetres/experim/dsp/ (Last accessed on 15 January, 2018) provided by Demetrescu et al. (Demetrescu and Italiano, 2006). The graph data used for experimentation is available in the package as XML files corresponds to the real world US road networks data (ftp://edcftp.cr.usgs.gov (Last accessed on 15 January, 2018). The table 2 describes the details of US Road Networks graphs from the package used for experimentation.

Table 1
Experimental data set: random graphs.

Graph	Number of Vertices |v|	Number of Edges |E|
R_G1	100	500
R_G2	250	1250
R_G3	325	1625
R_G4	515	2575
R_G5	625	3125
R_G6	735	3675
R_G7	800	4000
R_G8	885	4425
R_G9	950	4750

Table 2
Experimental data set: US road network.

Graph	XML File	Number of Vertices |v|	Number of Edges |E|
Graph-1	RI.xml	170	490
Graph-2	ID.xml	256	676
Graph-3	SD.xml	396	1132
Graph-4	MD.xml	490	1390
Graph-5	IN.xml	599	1890
Graph-6	MI.xml	695	2162
Graph-7	MO.xml	809	2510
Graph-8	CA.xml	945	2768
Graph-9	FL.xml	1368	3892

Further, the performance of the proposed Dynamic Dijkstra-(D-Dij) algorithm has been compared with the algorithms D_Rr (given by Ramalingam et al. (Ramalingam and Reps, 1996; Ramalingam and Reps, 1996), D_Up (given by Demetrescu et al. (Demetrescu and Italiano, 2003) and D_Rr_B & D_Up_B (based on heap reduction technique given by Buriol et al. (Buriol et al., 2003). The comparative results obtained through experimentations on these five three algorithms on two different types of graphs have been shown through Figs. 10–15.

Experiments with random graphs as well as US Road Network graphs point out that D_Dij has performance Figs. 10 and 14 comparable to that of D_Rr depending on the input graph topology, for the sparse graphs. Also memory usage Figs. 11 and 15for the proposed algorithm D_Dij is lesser as compared to all the other algorithms, because proposed algorithm does not use any extra information for performing the dynamic updations. But D_Rr maintain the extra information like the subset of effected vertices and D_Up also maintains the uniform paths for performing the dynamic updations. The memory usage decreases by using the heap reduction technique leading to lesser memory requiremnts of D_Rr_B and D_Up_B.

As is visible from the results Fig. 10 the performance of the proposed algorithm D_Dij is better than both other algorithms and also the time taken by the algorithms D_Rr and D_Up increases with increase in the underlying graph size (number of edges) due to increase in overhead in the D_Rr and D_Up algorithms. Moreover, Fig. 13 shows that memory usage in case of D_Rr is not much effected by the graph size but in case of D_Up memory usage increases significantly with increase in number of edges. For the proposed algorithm D_Dij, memory usage does not grow much with the increase in number of edges of the graph.

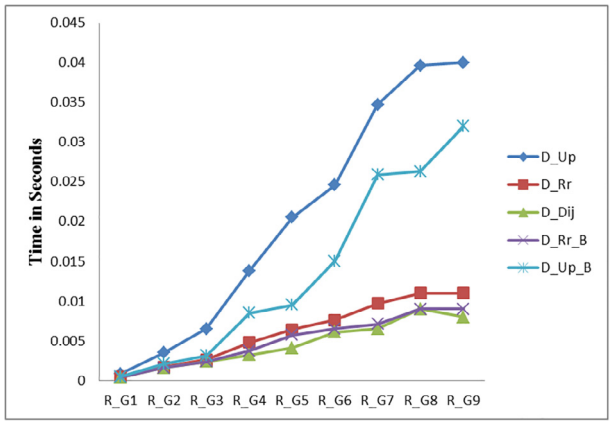

Fig. 10. Comparison time taken by D_Up, D_Rr, D_Up_B, D_Rr_B and D_Dij for Random Graphs.

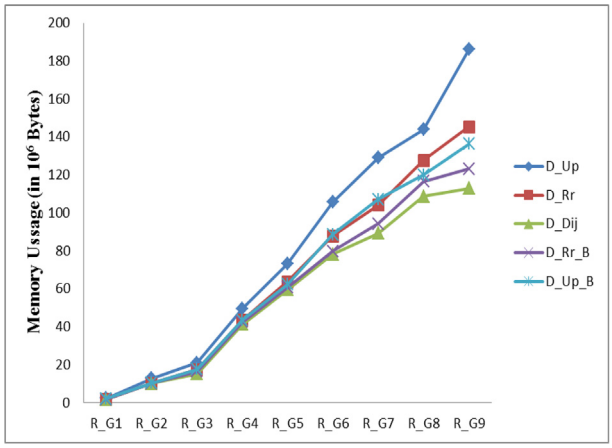

Fig. 11. Comparison Memory usage by D_Up, D_Rr, D_Up_B, D_Rr_B and D_Dij for Random Graphs.

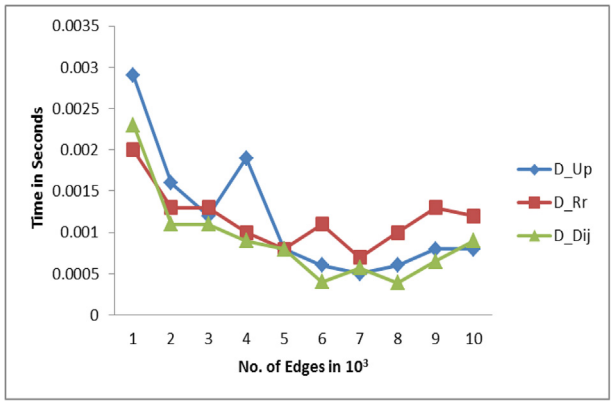

Fig. 12. Comparison time taken by D_Up, D_Rr and D_Dij for varying number of edges.

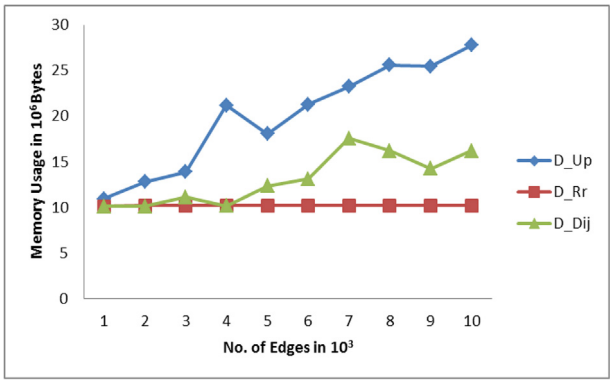

Fig. 13. Comparison memory usage by D_Up, D_Rr and D_Dij for varying number of edges.

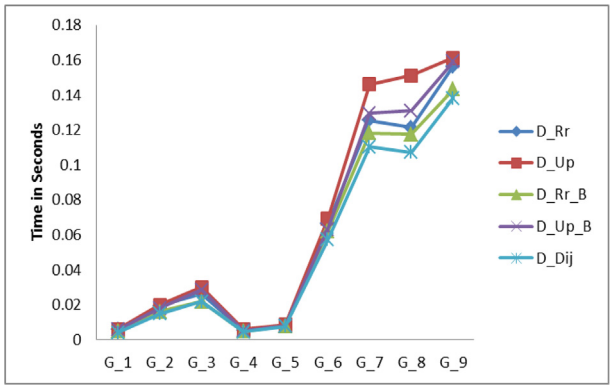

Fig. 14. Comparison time taken by D_Up, D_Rr, D_Up_B, D_Rr_B and D_Dij for US Road Netwok Data.

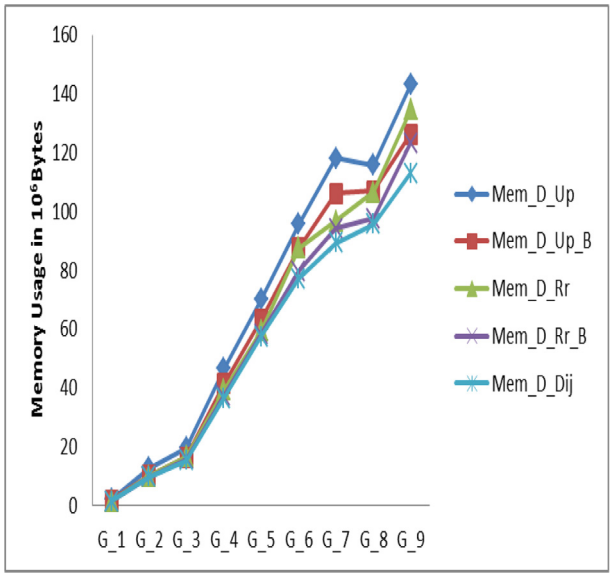

Fig. 15. Comparison Memory usage by D_Up, D_Rr, D_Up_B, D_Rr_B and D_Dij for US Road Netwok Data.

5. Conclusion

We have proposed to use the retroactive data structuring paradigm to represent the time-line required to be associated with each operation in the dynamic shortest path problem. Our method provides an efficient solution to the dynamic shortest path problem. As we are implementing the retroactive priority queue using the height balanced search trees, so the cost of all the operations

of priority queue is dependent on the height of the underlying tree data structure. Also, the No. of computations is less in case of dynamic changes in the graph as we are moving back in time at a point immediately before the time when the edge under updation is going to be used in any of the shortest paths. We are selectively choosing the point which has been actually affected by the change and applying the updation to that portion only.

The solution of shortest path problem using retroactive data structures can be easily adapted to relax the assumption we have made in our work. Transformations (Reweighing) can be applied to adapt our solution in case of general graphs (graphs with positive or negative edge weights). Also, our dynamic *Dijkstra* algorithm can be used to solve the dynamic all-pair shortest-path problem. The performance of the solution may be somewhat less than the best resource bounds known so far. Also, improving the efficiency of the underlying retroactive priority queue may further improve the performance of the approach. In future, we will plan to process a number of update operations as a single one, this may further reduce the time bounds for the problem.

References

Acar, U.A., Blelloch, G.E., Tangwongsan, K., 2007. Non-oblivious retroactive data structures. Technical Report CMU-CS-07-169.

Adelson-Velskii, G., Landis, E.M., 1962. An algorithm for the organization of information. Soviet Math. Doklady. 3, 1259–1263.

Bergamini, E., Meyerhenke, H., 2015. Fully-dynamic approximation of betweenness centrality. In the proceedings of European Symposium on Algorithms pp. 155–166.

Bergamini, E., Meyerhenke, H., Staudt, C., 2015. Approximating betweenness centrality in large evolving networks. In: Proceedings of the Meeting on Algorithm Engineering & Expermiments pp. 133–146.

Bernstein, A., 2013. Maintaining shortest paths under deletions in weighted directed graphs. STOC, 725–734.

Bernstein, A., Roditty, L., 2011. Improved dynamic algorithms for maintaining approximate shortest paths under deletions. In: proceedings of the ACM-SIAM Symposium on Discrete Algorithms pp. 1355-1365.

Buriol, L.S., Resende, M.G., Thorup M., 2003. Speeding up dynamic shortest path algorithms. AT&T Labs Research, TR TD-5RJ8B.

Cecilia, R.A., Raimund, G.S., 1989. Randomized search trees. In: Proceedings of the Foundations of Computer Science.

Dahshan, M., 2013. Shortest node-disjoint path using dual-network topology in optical switched networks. Int. Arab J. Inform. Technol. 10 (4), 365–372.

Demaine, E.D., Iacono, J., Langerman, S., 2007. Retroactive data structures. ACM Trans. Algorith. 3 (2), 13.

Demetrescu, C., Italiano, G. F., 2003. A new approach to dynamic all pairs shortest paths. In: Proceedings of the 35th Annual ACM Symposium on Theory of Computing (STOC'03), San Diego, CA, pp. 159–166.

Demetrescu, C., Italiano, G.F., 2006. Experimental analysis of dynamic all pairs shortest path algorithms. ACM Trans. Algorith. 2 (4), 578–601.

Dijkstra, E.W., 1959. A note on two problems in connexion with graphs. J. Numeris. Math. 1, 269–271.

Even, S., Shiloach, Y., 1981. An on-line edge-deletion problem. J. ACM 28 (1), 1–4.

Fortz, B., Thorup, M., 2000. Internet traffic engineering by optimizing OSPF weights. In: The proceedings of the IEEE Conference on Computer Communications pp. 519–528.

Frigioni, D., Ioffreda, M., Nanni, U., Pasqualone, G., 1998. Analysis of dynamic algorithms for the single source shortest path problem. ACM J. Exp. Algorithm 3.

Frigioni, D., Marchetti-Spaccamela, A., Nanni, U., 2000. Fully dynamic algorithms for maintaining shortest paths trees. J. Algorith. 34 (2), 251–281.

ftp://edcftp.cr.usgs.gov (accessed 15.01.2018).

Fujishige, S., 1981. A note on the problem of updating shortest paths. Networks 11, 317–319.

Gallo, G., 1981. Reoptimization procedures in shortest path problems. Rivista di Matematica per le Scienze Economiche e Sociali 3, 3–13.

Guibas, L.J., Sedgewick, R., 1978. A dichtomatic framework for balanced trees. Proc. IEEE Symp. Found. Comput. Sci., 8–21

Henzinger, M., Krinninger, S., Nanongkai, D., 2015. Dynamic approximate all-pairs shortest paths: breaking the O(mn) barrier and derandomization. Encyclop. Algorith., 1–3

http://www.dis.uniroma1.it/~demetres/experim/dsp/ (accessed 15.01.2018).

Kas, M., Carley, K.M., Carley, L.R., 2013. Incremental closeness centrality for dynamically changing social networks. Adv. Soc. Netw. Anal. Min., 1250–1258.

Kas, M., Carley, K.M., Carley, K.M., 2014. An incremental algorithm for updating betweenness centrality and k-betweenness centrality and its performance on realistic dynamic social network data. Soc. Netw. Anal. Min. 4 (1).

King, V., Thorup, M., 2001. A space saving trick for directed dynamic transitive closure and shortest path algorithms. In: Proceedings of the 7th Annual International Computing and Combinatorics Conference (COCOON), LNCS 2108, pp. 268–277.

Knuth, D., 1998. The art of Computer Programming. Sorting and Searching. Addison-Wesley, Massachusetts, pp. 458-481.

Narviaez, P., Siu, K., Tzeng, H., 2001. New dynamic SPT algorithm based on a ball-and-string Model. ACM Trans. Netw. 9 (6), 706–718.

Nievergelt, J., Reingold, E.M., 1973. Binary search trees of bounded balance. SIAM J. Comput. 2, 33–43.

Ramalingam, G., Reps, T.W., 1996. An incremental algorithm for a generalization of the shortest-path problem. J. Algorith. 21 (2), 267–305.

Ramalingam, G., Reps, T.W., 1996. On the computational complexity of dynamic graph problems. Theoret. Comput. Sci. 158, 233–277.

Rana, R., Garg, D., 2014. Anticipatory bound selection procedure (ABSP) for vertex k-center problem. Int. Arab J. Inform. Technol. 11 (5), 429–435.

Roditty, L., Zwick, U., 2012. Dynamic approximate all-pairs shortest paths in undirected graphs. SIAM J. Comput. 41 (3), 670–683.

Sarnak, N., Tarjan, R.E., 1986. Planar point location using persistent search trees. Communication ACM 29, 669–679.

Thorup, M., 2004. Fully-dynamic all-pairs shortest paths: Faster and allowing negative cycles. Algorithm Theory-SWAT pp. 384–396.

Xiao, B., Zhuge, Q., Sha, E.H.-M., 2004. Efficient algorithms for dynamic update of shortest path tree in networking. J. Comput. Appl. 11, 60–75.

Yigit, T., Unsal, O., 2016. Using the ant colony algorithm for real-time automatic route of school buses. Int. Arab J. Inform. Technol. 13 (5), 559–565.

Further reading

Ahuja, R.K., Orlin, J.B., Pallottino, S., Scutella, M.G., 2003. Dynamic shortest paths minimizing travel times and costs. Networks, 197–205.

Chan, T.M., 2001. Dynamic planar convex hull operations in near logarithmic amortized time. J. ACM 48, 1–12.

Sniedovich, M., 2006. Dijkstra's algorithm revisited: the dynamic programming connexion. J. Contr. Cybernet. 35, 599–620.

Tangwongsan, K., 2006. Active Data Structures and Applications to Dynamic and Kinetic Algorithms. Dissertation. Carnegie Mellon University.