IJCSN International Journal of Computer Science and Network, Volume 2, Issue 6, December 2013
ISSN (Online): 2277-5420 www.IJCSN.org

A Survey on Algorithms for Optimal Path Search in Dynamic Networks

[1] **Neeraj Navlakha,** [2] **Manisha J. Nene**

[1] **Department of Applied Mathematics, Defence Institute of Advanced Technology**
Pune, Maharashtra, 411025, India

[2] **Department of Computer Engineering, Defence Institute of Advanced Technology**
Pune, Maharashtra, 411025, India

Abstract - Dynamic networks are prevalent everywhere right from Internet, Local Area Networks, Mobile Ad hoc networks to Wireless Sensor Networks. Communication networks, social networks, Web, transportation networks infrastructures represent dynamic networks due to the dynamics that influence its topologies, connectivity, reliability and fault-tolerance. The network dynamics is rigorously influenced by the input parameters like data, communication, computational overloads, data traffic, data congestion, inclusion and exclusion of data resources over time. Hence, to identify optimal paths in such dynamic networks where communication nodes come and go, network communication edges may crash and recover, is a non-trivial problem. In this paper we present a rigorous study on various shortest path finding algorithms in dynamic networks. Although the research on this problem spans over more than thirty years, in the last couple of years many novel algorithmic techniques have been proposed. An arduous effort is made to abstract some combinatorial and algebraic properties. Also, common data-structural tools that are at the base of those techniques are described and compared.

Keywords- *Dynamic networks, dynamic graph problems, dynamic shortest paths, complexity*

1. Introduction

A fundamental problem in communication networks is finding optimal routes for transmitting data packets. To minimize communication time and cost, many routing schemes deliver packets along shortest routes. To speed up the task of finding optimal paths, routers use pre-computed lookup tables. In many scenarios, however, the costs associated with network links and the structure of the network itself may change dynamically over time, forcing continuous recalculations of the routing tables. A typical example is routing in ad hoc wireless networks, where collections of wireless mobile hosts form a temporary network without the aid of any established infrastructure or centralized administration. In such an environment, it may be imperative for one mobile host to find intermediate hosts in forwarding a data packet to its destination, due to the restricted range of each mobile hosts wireless transmissions. Transmission protocols need to adapt fast to routing changes when host movement is persistent, while requiring little or no overhead during periods in which hosts move less frequently. [4]

In this paper a survey is made on a number of recent algorithmic techniques for maintaining shortest routes in dynamic graphs. For the sake of generality, focus is done on the all-pairs version of the problem (APSP), where one is interested in maintaining information about shortest paths between each pair of nodes. Many of the techniques described in this paper can be specialized for the case where only shortest paths between a subset of nodes in the network have to be maintained.

A dynamic graph algorithm maintains a given property P on a graph subject to dynamic implementations, such as edge insertions, edge deletions and edge weight updates and should process queries on property P quickly, and perform update operations faster than re-computing from scratch, as done by the fastest static algorithm. If an algorithm is fully dynamic it can handle both edge insertions and edge deletions efficiently. A partially dynamic algorithm can handle either edge insertions or edge deletions. We say that it is incremental if it supports insertions only, and if it supports deletions only it is decremental. In this paper, survey is done for fully dynamic algorithms for maintaining information about shortest path [1].

1.1 Literature Survey

The first papers on dynamic shortest paths date back to 1967 [5,6,7]. In 1985, Even, Gazit and Rohnert [8,9] presented algorithms for maintaining shortest paths on directed graphs with arbitrary real weights. Their algorithms required $O(n^2)$ per edge insertion and also the worst-case bounds for edge deletions were comparable to re-computing APSP from scratch.

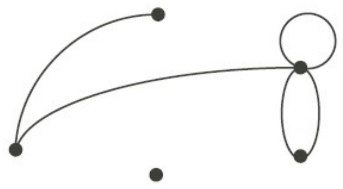
Two classic algorithms for the single-source shortest-path (SSSP) problem are the Bellman Ford algorithm [11] and the Dijkstra's algorithm [10]. They have $O(nm)$ and $O(n^2)$ time complexities, respectively, where n is the number of vertices and m is the number of edges in a graph. The Dijkstra's algorithm is used in a network where each edge has a positive length (or weight) value. The Bellman-Ford algorithm can be applied in the situation when edge lengths are negative. For the APSP problem, the Floyd-Warshall algorithm is a classic algorithm which has the time complexity of $O(n^3)$.

Many algorithms have been proposed by improving or combining the above classical algorithms. For example, based on the Dijkstra's algorithm, Orlin et al. [12] proposed a faster algorithm for the SSSP problem in networks with few distinct positive lengths.

Goldberg et al. [13] proposed an efficient shortest path algorithm which combines with A^* search. Roditty and Zwick [14] studied the dynamic shortest path problems and proposed a randomized fully dynamic algorithm for the APSP problem in directed unweighted graphs. For directed graph with real edge weights, Pettie's [15] algorithm solves the APSP problem in $O(nm + n^2 \log\log n)$ time. Chan [16] firstly obtained an algorithm with time complexity $O(n^3/\log n)$ in 2005, which is the best-known result for the APSP problem in a directed graph.

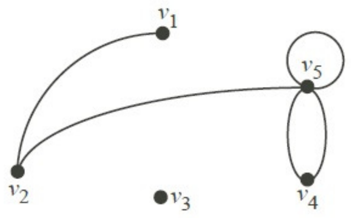
1.2 Some Definitions and Terminology

Conceptually, a graph is formed by vertices and edges connecting the vertices.



Formally, a graph is a pair of sets (V, E), where V is the set of vertices and E is the set of edges, and formed by pairs of vertices. E has elements which can occur more than once so that every element has a multiplicity. Usually, we label the vertices with letters (for example: v_1, v_2, . . . or a, b, c, . . . or) or numbers 1, 2, . . . Throughout this paper, we will label the elements of V in this way.

We label the vertices as follows:



We have $V = \{v_1, . . . , v_5\}$ for the vertices and $E = \{(v_1, v_2), (v_2, v_5), (v_5, v_5), (v_5, v_4), (v_5, v_4)\}$ for the edges. Similarly, we often label the edges with letters (for example: a, b, c, . . . or e_1, e_2, . . .) or numbers 1, 2, . . . for simplicity.

A path is a graph P = (V, E) of the form $V = \{v_1, v_2, ..., v_n\}$ $E = \{v_1\text{-}v_2, v_2\text{-}v_3, ...v_{n-1}\text{-}v_n\}$.
where, $n \geq 1$ and vertices v_1, . . . , v_n are all distinct. Vertices v_1 and v_n are the endpoints of the path. Note that a path may consist of a single vertex, in which case both endpoints are the same.

The problem of finding a path between two vertices (or nodes) in a graph such that the sum of the weights of its constituent edges is minimized is shortest path problem.

1.3 Organisation of the paper

The remainder of this paper is organized as follows. In Section 2 we describe the newest dynamic shortest path algorithms. In Section 3 we made comparisons among the experimental results of all the algorithms discussed. In Section 4 we list some concluding remarks and open problems.

2. Algorithmic Techniques

2.1 Algorithmic Techniques for Maintaining Shortest Routes in Dynamic Networks

The Problem:

Let G = (V, E) be a weighted directed graph. The authors [1] consider the problem of maintaining a data structure for G under an intermixed sequence of update and query operations of the following kinds:
• Decrease (v, w): decrease the weight of edges incident to v in G as specified by a new weight function w. They call this kind of update a v-Centered decrease in G.
• Increase (w): increase the weight of edges in G as specified by a new weight function w.
• Query(x, y): return the distance between x and y in G.

They considered generalized update operations where they can modify a whole set of edges, rather than a single edge. Also, they do not address the issue of returning actual paths between vertices, and they just consider the problem of answering distance queries.

In this section the authors [1] describe how to maintain all pairs shortest paths in a directed graph with non-negative integer edge weights less than C in $O\left(n^{2.5}\sqrt{C \log n}\right)$ amortized time per update operation.
Data Structure:
Given a weighted directed graph G, they maintain for each vertex v:

- A shortest paths tree Out_v of G of depth $d \leq \sqrt{nC \log n}$ rooted at v;

- A shortest paths tree $\mathrm{In_v}$ of $\widehat{\mathrm{G}}$ depth $\mathrm{d} \leq \sqrt{\mathrm{nC}\log\mathrm{n}}$ rooted at v, where $\widehat{\mathrm{G}}$ is equal to G, except for the direction of edges, which is reversed;
- A set S containing $\sqrt{\mathrm{nC}\log\mathrm{n}}$ vertices of G chosen uniformly at random, referred to as "blockers";
- A complete weighted directed graph $\mathrm{G_s}$ with vertex set S such that, with very high probability, the weight of (x, y) in $\mathrm{G_s}$ is equal to the distance between x and y in G.
- An integer distance matrix dist.

They [1] maintained the trees with instances of the data structure adapted to deal with weighted directed graphs and to include only short paths, i.e., vertices of distance up to d from the root. Information about longer paths will be obtained by stitching together these short paths.

Implementation of Operations:

The main idea of the algorithm [1] is to exploit the long paths property, maintaining dynamically only shortest paths of (weighted) length up to $\sqrt{\mathrm{C}\log\mathrm{n}}$ with the data structure, and stitching together these paths to update longer paths using any static $\mathrm{O}(\mathrm{n}^3)$ all pairs shortest paths algorithm on a contraction with $\mathrm{O}(\sqrt{\mathrm{nC}\log\mathrm{n}})$ vertices of the original graph. Following are the operations:

- Decrease (v, w): rebuild $\mathrm{In_v}$ and $\mathrm{Out_v}$ to update $\mathrm{G_s}$, i.e., paths of length up to d. Apply the algorithm Stitch below to update longer paths.
- Increase (w): update edges with increased weight in any $\mathrm{In_v}$ and $\mathrm{Out_v}$ which contains them, and then update $\mathrm{G_s}$, i.e., paths of length up to d. Apply the algorithm Stitch below to update longer paths.
- Query(x, y): return dist(x, y).

Features of the stitching algorithm are given below:

```
Stitch()
  (1) Let $dist^{(d)}(x,y)$ be the distance from x to y of length up to
      d, obtained from all the trees $In_v$ and $Out_v$.
  (2) Compute the distances $dist()$ between all vertices in S
      using any static $O(n^3)$ APSP algorithm on $G_s$.
  (3) Compute the distances from vertices in V to vertices in S.
      This can be done for a pair
      $x \in V$ and $s \in S$ by computing:

      $dist(x, y) \rightarrow min\{dist^{(d)}(x, y), \min_{s \in S}\{dist(x, s)+dist^{(d)}(s, y)\}.$
```

Analysis:
The stitching algorithm is dominated by the last step, which takes time $\mathrm{O}(\mathrm{n}^2|\mathrm{S}|) = \mathrm{O}(\mathrm{n}^2(\mathrm{n}\log\mathrm{n}/\mathrm{d}))$. Shortest path trees of length up to d can be maintained in $\mathrm{O}(\mathrm{n}^2\mathrm{d})$ amortized time. Choosing $\mathrm{d} = \Theta(\sqrt{\mathrm{nC}\log\mathrm{n}})$ yields an amortized update bound of $\mathrm{O}(\mathrm{n}^{2.5}\sqrt{\mathrm{C}\ \log\mathrm{n}})$.

2.2 Algorithm for Time Dependent Shortest Safe Path on Transportation Networks

Proposed Algorithm:

The authors [2] initially outline the proposed algorithm, named $\mathrm{T_{DSSP}}$. Then they explain a sub-algorithm named Arrival-Time, which is to calculate the earliest arrival-time functions for $\mathrm{T_{DSSP}}$.

Let R be the set of the arrived edges with danger factor larger than the given upper bound Ω. The algorithm $\mathrm{T_{DSSP}}$ searches for the shortest safe path P* from $\mathrm{v_s}$ to $\mathrm{v_e}$ on the graph $\mathrm{G_T}$ utilizing the set R and the earliest arrival-time function $\mathrm{g_i(t)}$ for all is, that are generated in the first step by the sub-algorithm Arrival-Time (to be described later). Then, it determines the predecessor of a node on P* utilizing the backward manner from $\mathrm{v_e}$ to based $\mathrm{v_s}$ on $\mathrm{g_i(t)}$ and the best starting t*. $\mathrm{T_{DSSP}}$ works on the following three cases.

- Case 1: both safe path with relatively long travel-time and unsafe path with relatively short travel-time exist concurrently. In this case it returns the safe path.
- Case 2: all $\mathrm{v_s} - \mathrm{v_e}$ paths are safe. In this case it returns the shortest one.
- Case 3: no safe path exists for the given Ω. In this case it returns an unsafe but shortest path.

```
Input: a time-dependent graph $G_T$, source $v_s$, destination $v_e$, and
starting-time interval $T = [t_s, t_e]$;
Output: a shortest safe path P* from $v_s$ to $v_e$ for starting time t*;
Algorithm $T_{DSSP}(G_T, v_s, v_e, T, \Omega)$;
/* return(x) means outputting x and terminating the algorithm. */
Begin

1.   call Arrival-Time $(G_T, v_s, v_e, T, \Omega)$ to generate $g_i(t)s$ and R;
2.   if R is not empty then
     2.1.  Let $\Gamma = \{t_1, t_2, \cdots t_r\}$, that is the set of the inflection points of
           $g_e(t)$;
     2.2.  Sort Γ in ascending order according to $g_e(t_i) - t_i$, where
           $1 \leq i \leq r$;
     2.3.  for each $t_i \in T$ do
               Generate a $v_s - v_e$ path $P_i$ with starting time $t_i$;
               if all edges in $P_i$ are safe then return $P_i$;
               end of if;
3.   $t^* := argmin_{t \in T}\{g_e(t) - t\}$; /* select a shortest path */
4.   Generate a $v_s - v_e$ path P* with starting time t*;

Return P*;

End
```

In the first case, let Γ be the set of the inflection points of $\mathrm{g_e(t)}$ as described in step 2.1. They sort Γ in ascending order according to the travel time calculated by $\mathrm{g_e(t_i)} - \mathrm{t_i}$ in step 2.2. Then they select $\mathrm{t_i}$ from Γ iteratively and calculate the path $\mathrm{P_i}$. If the path is safe, algorithm terminates in step 2.3.
In the second case, R is empty. The best starting time t* with the minimal $\mathrm{v_s} - \mathrm{v_e}$ travel time satisfies that $\mathrm{g_e(t^*)} - \mathrm{t^*} = \min_t\{\mathrm{g_e(t)} - \mathrm{t}\}$. It can be identified

according to the earliest arrival-time functions calculated by algorithm T_{DSSP} in step 3. They initialize v_j to destination v_i, and the optimal path P* is set to empty in step 4. Assume that there is no waiting time at each edge and the predecessor of v_j is v_i. If there exists an edge (v_i, v_j) in E, such that $g_j(t^*) = g_i(t^*) + w_{i,j}(g_i(t^*))$ i.e., the arrival time at $v_j, g_j(t^*)$, is the arrival time at $v_i, g_i(t^*)$ plus the edge delay from v_i to v_j, then they iteratively find a predecessor v_i of v_j and add (v_i, v_j) into the path P* till the path reaches the source node v_s.

The third case corresponds to that, R is not empty and no safe path is returned in the step 2.3. In this case, similar calculations are performed as in the second case. But, the algorithm returns an unsafe but shortest path calculated by steps 3 and 4, as no safe path exists in this case.

Now they [3] introduce how to calculate the earliest arrival-time function $g_i(t)$ for the node v_i. They propose the algorithm Arrival-Time for time refinement on FIFO graphs. It consists of two sections: refinement for starting-time interval and refinement for arrival-time function. The former refines the starting-time subinterval $I_i = [t_s, \tau_i]$ for each node $v_i \in G_T$ and the latter refines $g_i(t)$ i.e., $g_i(t)$ specifies the earliest $v_s - v_i$ arrival time for any starting time t in subinterval $[t_s, \tau_i]$. It is highlighted that the algorithm Arrival-Time can handle the time-dependent edge-delay graph by refining the arrival time for each node with the increase of the time interval.

From the formal description, it takes five parameters as the input: time-dependent graph G_T with danger factors, source node v_s, destination node v_e, starting time interval $T = [t_s, t_e]$, and the given upper bound Ω of danger factors for the shortest safe path.

Input: *a time-dependent graph G_T, a query $SSP(v_s, v_e, T)$ with source v_s and destination v_e, and starting-time interval $T = [t_s, t_e]$; danger factor function set $\{d_{i,j}(t)\}$ on edge (v_i, v_j), and a given upper bound Ω of the danger factor for the shortest path.*
Output: *$\{g_i(t) | v_i \in V\}$ – all earliest arrival-time functions.*
Algorithm Arrival-Time$(G_T, v_s, v_e, T, \Omega)$

```
/* $D = \{d_{i,j}(t)\}$ */
begin
1.  $g_s(t) := t$; $\tau_s := t_s$; /*for $t \in T$*/
2.  for each $v_i \neq v_s$ do $g_i(t) := \infty$ for $t \in T$ , and $\tau_i = t_s$;
3.  Let Q be a priority queue initially containing pairs $(\tau_i, g_i(t))$ for
    all nodes vi $\epsilon$ V, ordered by $g_i(\tau_i)$ in ascending orders;
4.  while |Q |≥ 2 do
    4.1.  $(\tau_i, g_i(t)) :=$ dequeue(Q)and $(\tau_k, g_k(t)) :=$ head(Q);
    4.2.  $\Delta := min\{w_{f,i}(g_k(\tau_k)) | (v_f, v_i) \in E\}$;
    4.3.  $\tau'_i := max\{t | g_i(t) \leq g_k(\tau_k) + \Delta\}$;
    4.4.  for each $(v_i, v_j) \in E$ do
                  if $d_{i,j}(\tau_i) \leq \Omega$ then begin
                  $g'_j(t) := g_i(t) + w_{i,j}(g_i(t))$ for $t \in [\tau_i, \tau'_i]$;
                  $g_j(t) := min\{g_i(t), g'_i(t)\}$ for $t \in [\tau_i, \tau'_i]$
                  update$(Q, (\tau_j, g_j(t)))$
          end of if and for;
    4.5. if $d_{i,j}(\tau_i) > \Omega$ for all $(v_i, v_j)$ then /*i is fixed now*/
                  begin
                  Let $d_{i,j^*}$ be $min_j\{d_{i,j}(g_i(\tau_i))\}$;
                  $g'_{j^*}(t) := g_i(t) + w_{i,j^*}(g_i(t))$ for $t \in [\tau_i, \tau'_i]$;
                  $g_{j^*}(t) := min\{g_{j^*}(t), g'_{j^*}(t)\}$ for $t \in [\tau_i, \tau'_i]$;
                  update$(Q, (\tau_{j^*}, g_{j^*}(t)))$;
                  add $\langle (v_i, v_j), g_i(\tau_i) \rangle$ into R;
                  end of if;
    4.6. $\tau_i := \tau'_i$;
    4.7. if $\tau_i \geq t_e$ then
                  if $v_i = v_e$ then return $\{g_i(t) | v_i \in V\}$;
       else enqueue$(Q, (\tau_i(g_i(t))))$;
    end of while;
5.  return$\{g_i(t) | v_i \in V\}$;
end.
```

Firstly, $g_s(t)$ and τ_s are initialized for source node v_s: $g_s(t) := t$ and $\tau_s := t_s$ (see step 1). This is corresponding to a trivial case: departing from the source node v_s to itself at any timet_0, resulting in the travel time of value $g_s(t_0) - t_0$, i.e., 0. For all other nodes, the earliest arrival-time function $g_i(t)$ is initialized as $g_i(t) := \infty$ which implies that these nodes are undetermined yet, and τ_i is initialized as $\tau_i = t_s$ for each i (see step 2).

In step 3, Q is a priority queue containing pairs $(\tau_i, g_i(t))$ for each node $v_i \in G_T$, and these pairs are in the ascending order according to$g_i(\tau_i)$, initially the top pair in Q is $(\tau_s, g_s(t))$ (see step 3). The while loop in step 4 conducts time-refinement consisting of the refinement of starting-time interval (steps 4.1-4.3 and step 4.6) and the refinement of arrival-time function (steps 4.4 and 4.5).In each iteration, the earliest arrival-time function $g_i(t)$ is ensured to be well-refined in the starting-time subinterval $I_i = [t_s, \tau_i]$ for the node v_i. The algorithm terminates in two cases. The first one is that Q contains no more than one pair (see step 5), and the other is that the arrival-time function $g_e(t)$ of the destination node is well-refined in the entire interval T (see step 4.7).

On the refinement of starting-time interval, the algorithm dequeues the top pair, denoted as $(\tau_i, g_i(t))$ in Q in each iteration (see step 4.1). Then the current top pair namely $(\tau_k, g_k(t))$ is regarded as the basis of the refinement of starting time interval. Note that the operation head (Q) retrieves the current top pair but does not dequeues it from Q. Due to the order of pairs in Q, they obtain that $(g_i(\tau_i))$ is the earliest arrival time and $(g_k(\tau_k))$ is the second earliest arrival time from source node v_s.

According to the property of FIFO graph G_T, it is impossible to arrive at the node v_f(except v_i) before $g_k(\tau_k)$ if they start from $v_s$at time taken in$[\tau_f, t_e]$. Now, they fix the node v_i and consider an edge $(v_f, v_i) \in E$ at time $g_k(\tau_k)$. Assume that the arrival time to the node v_f is $g_k(\tau_k)$. Thus, the minimum travel time from v_f to v_i is Δ , that is calculated by $min\{w_{f,i}(g_k(\tau_k)) | (v_f, v_i) \in E\}$ (see step 4.2). Therefore, the next earliest arrival time from v_s to v_i via any edge (v_f, v_i) is $g_k(\tau_k) + \Delta$, if starting time t is not less than τ_f. Suppose t is the latest starting time which satisfies $g_i(t) \le g_k(\tau_k) + \Delta$, then they set it to τ'_i (see step 4.3). From $g_i(t) \le g_k(\tau_k) + \Delta$ for $t \in [\tau_i, \tau'_i]$; they conclude that the function $g_i(t)$ is well-refined in(t_s, τ'_i). $g_i(t)$ is the earliest arrival time from v_s to v_i for starting time $t \in [t_s, \tau'_i]$. Therefore, $g_i(t)$ is well-refined in $I'_i = [t_s, \tau'_i]$ where $I_i \subset I'_i$. In the end, they set τ_i and I_i to τ'_i and I'_i(see step 4.6), respectively, to refine the starting time interval.

Now they describe the refinement of the arrival-time function. Noting that $g_i(t)$ is well-refined in the enlarged subinterval, they can refine $g_j(t)$ for all v_i, where $(v_i, v_j) \in E$, in starting-time subinterval $[\tau_i, \tau'_i]$ (see step 4.4). It is noteworthy that they only need to refine $g_j(t)$ on $[\tau_i, \tau'_i]$, (because $g_j(t)$has been already refined on $[t_s, \tau_i]$ and then they [3] update Q with the latest refined $g_j(t)$.

If the danger factor of edge (v_i, v_j) at time τ_i, $d_{i,j}(\tau_i)$ is larger than the given danger boundary Ω for all $(v_i, v_j) \in E$, there algorithm then selects an edge (v_i, v_{j^*}) with the minimal danger factor at time $g_i(\tau_i)$ to calculate the function $g_{j^*}(t)$. Firstly, it computes the arrival time $g'_{j^*}(t)$ at v_{j^*} via edge (v_i, v_{j^*}) for starting time $t \in [\tau_i, \tau'_i]$. Then, it updates $g_{j^*}(t)$ to $\min\{g_{j^*}(t), g'_{j^*}(t)\}$ in the interval $[\tau_i, \tau'_i]$ (see step 4.5). After that, it updates the priority queue Q, and records the time $g_i(\tau_i)$and the selected edge (v_i, v_{j^*}) by inserting them into R.

2.3 New Algorithms for All-Pairs Shortest-Paths Problem

Basic Algorithm:

The well-known Dijkstra's algorithm uses a simple breadth-first search approach to find all shortest paths from a single source to all other vertices in a given graph. It has the advantages of efficiency and simplicity. To find all-pairs shortest-paths, the authors [3] apply the Dijkstra's algorithm on each vertex iteratively. Such an intuitive approach is simple but not very efficient.

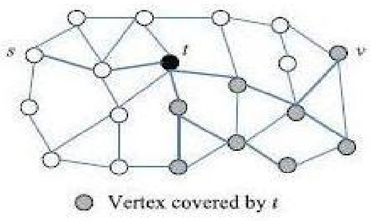


Figure 1: Example of covered nodes

To improve the Dijkstra's algorithm on the APSP problem, the authors proposed a new algorithm that utilizes information obtained in previous steps to accelerate the latter process. Let G = (V, E) denotes a weighted directed graph, where V and E are the set of vertices and edges respectively. The edge from the vertex u and v is denoted as (u, v) and its weight is w(u, v). They give the following definition to illustrate the main idea of the algorithm.

Definition: If the vertex t is an intermediate vertex on the shortest path from the vertex u to v, then they say that v is covered by t.

The Dijkstra's algorithm uses a breadth-first search (BFS) method to get shortest paths starting from a single source. When the vertex t is visited in the search process, if the shortest paths from t have been obtained in previous steps, then they can immediately obtain the shortest paths from sources to all other vertices using t as an intermediate vertex. In other words, given a source s, they need not visit other vertices which are covered by t. Based on this idea; they proposed the following algorithm which improves the Dijkstra's algorithm on solving the APSP problem. Many vertices may be covered by an intermediate vertex t (see Figure 1 for an example), so the total time used may be reduced dramatically.

The following data structures are used in the proposed algorithm:

- L: the matrix containing the edge weights, where $L[u, v]$ the weight of edge $[u, v]$. If the edge $[u, v]$ does not exist, then $L[u, v] = \infty$;
- D: the distance matrix, where $D[u, v]$ is the distance from the vertex u to v. Initially, $D[u, v] = \infty$; for all vertex pairs;
- flag: the vector to indicate whether the shortest paths from a vertex to other vertices have been calculated. All elements of the vector are set to zero initially. After the shortest paths for vertex u are calculated, $\text{flag}[u]$ is set to 1.
- Q: the min-priority queue containing the vertices to be visited. It is the same queue as that used in the classic Dijkstra's algorithm.

Procedure 1: *Modified Dijkstra's Procedure*
Input: *Graph G = (V, E), source s, weight matrix L, distance matrix D, vector flag*
Output: *updated distance matrix D, updated vector flag*

```
1.  D[s, s] = 0
2.  Q = {s}
3.  while Q is not empty do
    3.1.  t=DeQueue(Q)
    3.2.  if flag[t] = 1, then
                  for each vertex v ϵ V do
                  if D[s, t] + D[t, v] < D[s, v] then
                          D[s, v] =D[s, t] + D[t, v]
                          end of if
                  end for
    3.3. else
                  for each edge (t, v) outgoing from t do
                          If D[s, t] + L[t, v] < D[s,v] then
                          D[s, v] =D[s, t] + L[t, v]
```

```
                    Enqueue (Q, v)
                end if
            end for
        end if
end while
4.  flag[s] =1
```

```
Algorithm 1: Basic Algorithm for the APSP Problem
Input: Graph G= (V, E), weight matrix L
Output: distance matrix D
1.  for each vertex v ϵ V do
        f lag[v] = 0
2.  for each vertex pair (u, v) do
        D[u, v] = ∞
3.  for each vertex v ϵ V do
4.  call the modified Dijkstra's procedure to get shortest paths
    starting from v.
```

To find all the shortest paths in a graph G, the algorithm [3] firstly initializes the flag vector by setting all elements as zero (step 1-2 in Algorithm 1) and initializes all elements of the distance matrix to be infinity (step 3-4 in Algorithm 1). Then the modified Dijkstra's procedure (Procedure 1) is called iteratively to find shortest paths starting from every vertex. Compared to the classic Dijkstra's algorithm, the step 3.2 in procedure 1 [3] are main stuff newly added. When a vertex t is visited, if the shortest paths starting from it have been obtained (flag[t] is set to 1) (step 3.2 in Procedure 1), then the shortest paths from the source s to other vertices are updated by using t as an intermediate vertex (step 3.2). After all shortest paths starting from s have been obtained, they set flag[s] to 1 to indicate it. Other steps are the same as those in the classic Dijkstra's algorithm. The procedure Enqueue(Q, v) adds a vertex v in the min-priority queue Q. The procedure DeQueue(Q) gets a vertex from the queue Q which has the smallest shortest-path starting from s.

The newly-added steps do not change the upper bound on the time complexity of the algorithm. The proposed algorithm has the same time complexity than the Dijkstra's algorithm. Assume the time complexity of operations on the min-priority queue is δ, then the time complexity of proposed algorithm is $O(n(\delta n + m))$. For directed graph with real edge weights, if the queue is implemented simply as an array, then $\delta = O(n)$and the time complexity of the algorithm is $O(n^3 + nm) = O(n^3)$. If the queue is implemented with a Fibonacci heap, then $\delta = O(\log n)$ and the time complexity of the algorithm is $O(n^2 \log n + nm)$ .For an unweighted undirected graph, $\delta = O(1)$ and the time complexity of the algorithm is $O(n^2 + nm) = O(nm)$

Optimization for Complex Networks:
Since few nodes in a complex network have large number of neighbors(neighboring nodes), it is very possible for these nodes to be in the middle of shortest paths of other nodes. If the shortest paths from these high-degree nodes are obtained in advance, then the visiting of other nodes covered by them can be saved in the modified Dijkstra's procedure. Therefore, the order of vertices to be selected as sources is important for the algorithm performance in the context of complex networks.

To determine the order of vertices as sources, one can simply sort the vertices by their degrees. For example, one can sort vertices in descending order and select vertices with high degrees as sources before vertices with low degrees.

Because a large number of vertices in a complex network have low degrees, one need not determine the order for all vertices so that the time complexity of the selection process can be reduced. For example, one can use a ratio parameter r $(0 < r < 1)$ to control the selection process. If nr nodes have been selected, then the selection process can be stopped and the vertices which have not been selected will be used as sources in a random order.

Based on the above ideas, the authors [3] proposed an optimized algorithm for the APSP problem. The following data structures are added:

- deg: the vector containing the degrees of vertices. deg[i] is the degree of the i-th vertex;
- order: the vector containing the indices of vertices to be used as sources. order[i] is the index of i-th source vertex.

The optimized algorithm is illustrated by Algorithm 2

```
Algorithm 2: Optimized Algorithm for the APSP Problem
Input: Graph G = (V, E), weight matrix L, ratio r
Output: distance matrix D
1.  for each vertex v ϵ V do
        Flag[v]=0
2.  for each vertex pair (u, v) do
        D[u, v]= ∞
3.  for i = 1 to n do
        Calculate the degree of the i-th vertex
4.  for i = 1 to n do
        order[i] = i
5.  for i = 1 to rn do
        5.1  for j = i + 1 to n do
                 If deg[order[ j]] > deg[order[i]] then
                     swap(order[ j], order[i])
6.  for i = 1 to n do
        call the modified Dijkstra's procedure using the source of
index order[i]
```

Step 3 is used to calculate degrees of all vertices and vertices are selected according to their degrees in steps 4-5. The swap(a, b) operation swaps the values of two variables a and b.

Calculating degrees of all vertices requires $O(m)$ time. The selection procedure has the time complexity of

$O(rn^2)$. Thus, the newly-added steps have the time complexity of $O(m + rn^2)$.[3]

Adaptive Optimization Algorithm:

An interesting fact is that there is heuristics information in the modified Dijkstra's procedure and that information can be utilized to help selecting sources. In step 3.3 of the modified Dijkstra's procedure, if the condition is true, then the shortest path from source s to a vertex v may traverse the edge (t, v) temporarily. It hints that the vertex t is a possible intermediate vertex on some shortest paths. If the vertex t is used as predecessor more frequently, it will be possible for it to cover more vertices. Thus, it should be used as source in advance.
Based on the observations, the authors [3] proposed an improved algorithm which can select sources adaptively. For a vertex t, they use the variable deg[t] to store its priority to be selected as the source. The variable deg[t] is initially set to the degree of t, and later updated in the modified Dijkstra's procedure. If the condition in step 3.3 of the modified Dijkstra's procedure is true then

$$\mathrm{deg}[t] = \mathrm{deg}[t] + c \qquad (1)$$

where c is a predefined constant, e.g., c = 1. Other steps in the modified Dijkstra's procedure are not changed.
The main procedure is stated in Algorithm 3. Since the vector deg changes in the modified Dijkstra's procedure, the algorithm will select vertices as sources adaptively.

```
Algorithm 3: Adaptive Algorithm for the APSP Problem
Input: Graph G = (V, E), weight matrix L
Output: distance matrix D
1.  for each vertex v ϵ V do
        Flag[v]=0
2.  for each vertex pair (u, v) do
        D[u, v]= ∞
3.  for i = 1 to n do
        Calculate the degree of the i-th vertex
4.  for i = 1 to n do
        order[i] = i
5.  for i = 1 to n do
        5.1  for j = i + 1 to n do
                If deg[order[ j]] > deg[order[i]] then
                        swap(order[ j], order[i])
                end if
        end for
6.  call the modified Dijkstra's procedure using the source of index order[i]
```

Calculating degrees of all vertices requires $O(m)$ time. The selection procedure in steps 5 has the time complexity of $O(n)$Thus; the algorithm 3 has the same time complexity than the algorithm 1.

3. Experimental Results and Comparison

3.1 For algorithm technique in section 2.1

The main idea of the algorithm [1] is to exploit the long paths property, maintaining dynamically only shortest paths of (weighted) length up to $\sqrt{C\log n}$ with the data structure, and stitching together these paths to update longer paths using any static $O(n^3)$ all pairs shortest paths algorithm on a contraction with $O(\sqrt{nC\log n})$ vertices of the original graph.

The stitching algorithm is dominated by the last step, which takes time$O(n^2|S|) = O\big(n^2(n\log n/d)\big)$. Shortest path trees of length up to d can be maintained in $O(n^2d)$ amortized time. Choosing $d = \Theta\big(\sqrt{nC\log n}\big)$ yields an amortized update bound of $O\big(n^{2.5}\sqrt{C\ \log n}\big)$.

3.2 For algorithm technique in section 2.2

In this algorithm [2] the authors conducted extensive experimental studies to compare algorithm T_{DSSP} with $T_{DSP}08$, the most efficient discrete-time algorithm[17]. $T_{DSP}08$ focuses on finding optimal answers for the TDSP problem using a continuous-time approach.

distance	$\Omega = 0.2$	$\Omega = 0.5$	$\Omega = 0.8$	$\Omega = 1.0$
4	1.9	8.8	41.9	79.7
8	2.0	13.9	60.8	109.8
12	2.3	19.6	89.6	176.8
16	6.8	27.8	146.9	218.9
20	10.2	37.9	188.9	274.6
24	14.8	42.8	246.1	300.9

Table 1: T_{DSSP} runtime (s) for instances with different upper bounds of danger factor and different distance between source node and destination node, on a graph with 33 nodes and 100 edges.

Initially, it computes the earliest arrival-time function $g_i(t)$ for each node v_i. Then it checks whether there exists a $v_s - v_e$ path by calculating the best starting time t^*. After that, it generates a path P* corresponding to the arrival time $g_e(t^*)$ for the best starting time t^*. Finally, it returns the path P* together with the best starting time. $T_{DSP}08$ is able to deal with FIFO time-dependent graphs, as well as the general time-dependent graphs, with arbitrary edge-delay functions.

The two algorithm are implemented using C++, on a 3.0GHz CPU/1G memory PC running XP. They test the two algorithms on a real data set with 16326 nodes and 26528 edges, representing the road-map in the Maryland State in US. The nodes represent the starts, ends, and intersections of roads, while the edges represent road segments.

The continuous piecewise-linear danger-factor functions $\{d_{i,j}(t)\}$ are generated randomly in the following manner. They set four parameters: the average value of the danger factors (denoted as $\overline{d}$), the range of the danger

factor (denoted as $\hat{d}$), the length of the domain (denoted as L_T), and the number of the segments N_T. In the domain T = [0, L_T], T is randomly divided into N_T sub-intervals. In each sub-interval, $d_{i,j}(t)$ is set to a linear function. The value of $d_{i,j}(t)$ at the start/end of each subinterval is randomly generated as a number in [$\bar{d}$−$\hat{d}$, $\bar{d}$+$\hat{d}$] uniformly, where $\bar{d}$ is the average value of danger factor on the edge. The continuous piecewise-linear edge-delay functions $\{w_{ij}(t)\}$ are generated randomly.
Note that, T_{DSSP} generates the same solution as T_{DSP}08 when Ω = 1.0.

They collect data on a sub-graph of the road-map. The running time and memory dissipation of the algorithms are shown in figures 2 and 3 [2], respectively; for different upper bounds of danger factor that vary from 0.15 to 1.00.

In figure 2 [2], the running time of T_{DSP}08 is 2.06 seconds, as T_{DSP}08 is independent of the danger factor. The algorithm T_{DSSP}, however, is not. T_{DSSP} is superior for the case of the danger factor less than 0.5.

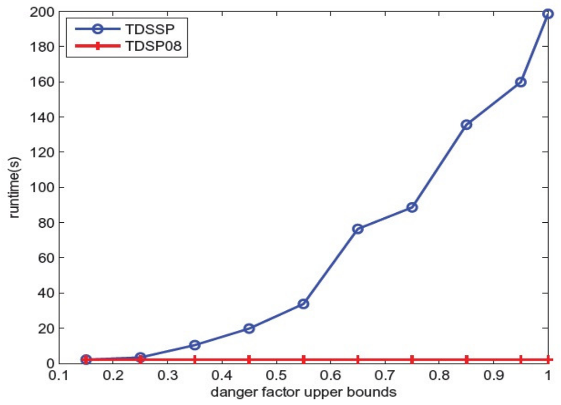


Figure 2: Running time of algorithms T_{DSSP} and T_{DSP}08 for different danger factors, on a graph with 33 nodes and 100 edges.

For example, the running time of T_{DSSP} is about 3 seconds for Ω = 0.25, that is very close to the running time of T_{DSP}08. The running time gradually increases to 20 seconds when Ω increases to 0.5. From that, T_{DSSP} becomes relatively slow with the increasing boundary of danger factor. This is because, there are more edges becoming safe in the graph, with the increasing of the boundary of danger factor. In this case, the algorithm T_{DSSP} has to take more processing time to calculate the earliest arrival time functions for all nodes to select the shortest safe path.

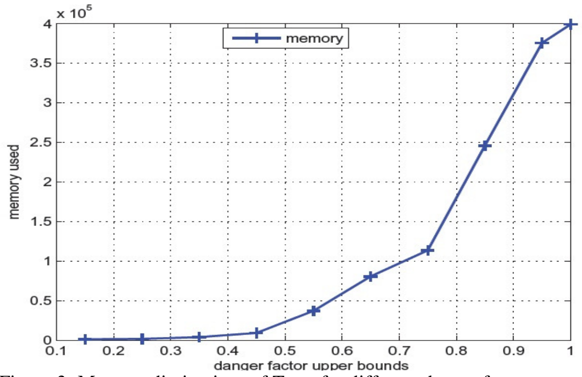


Figure 3: Memory dissipations of T_{DSSP} for different danger factors, on a sub-graph with 205 nodes and 1000 edges.

On the other hand, the running time of T_{DSSP} increases with the increasing distance between the source node and the destination node. It is not difficult to understand that, the far between the source node and the destination node, the more calculations are required. But this increase in running time is valuable as they can find a path that is both shortest and safe. In addition, these increases in running time is acceptable for relatively smaller distance values and smaller upper bounds of danger factor, in comparison to the running time of T_{DSP}08 shown in figure 2 [2], that is about 2 seconds.
They also collect the memory dissipations for both algorithms on the graph with different sizes, for different upper bounds of the danger factor. Figure 3 [2] show the simulation results on a relatively larger sub-graph that consists of 205 nodes and 1000 edges. From figure 3, they concluded that the corresponding memory dissipation has the similar changes as the running time. They also confirm that similar results are also true for other sub-graphs with different sizes, according to our extra experimental results. [2]

3.3 For algorithm technique in section 2.3

The newly-added steps do not change the upper bound on the time complexity of the algorithm. The proposed algorithm [3] has the same time complexity than the Dijkstra's algorithm. Assume the time complexity of operations on the min-priority queue is δ, then the time complexity of proposed algorithm is $O(n(\delta n + m))$. For directed graph with real edge weights, if the queue is implemented simply as an array, then $\delta = O(n)$ and the time complexity of the algorithm is $O(n^3 + nm) = O(n^3)$. If the queue is implemented with a Fibonacci heap, then $\delta = O(\log n)$ and the time complexity of the algorithm is $O(n^2 \log n + nm)$. For an unweighted and undirected graph $\delta = O(1)$ and the time complexity of the algorithm is $O(n^2 + nm) = O(nm)$.

Introduction of Experiments:

The authors [3] evaluate the performance of the proposed algorithms on random networks. They compare their performance in complex networks with the Erd̈os and Rényi (ER) [18] random graph model and the scale-free Albert-Barab á si (AB) [19] network model.

In the ER model, each vertex pair is uniformly connected with a probability p in a network of n nodes.

There are about $pn(n-1)/2$ edges in an ER graph. The average degree of vertices is $\langle k \rangle = p(n-1) \approx pn$. The degrees of vertices can be represented by the Poission distribution.

The AB model is an extension of the original Barabási - Albert (BA) model. In the model, a network initially contains m_0 nodes. Then the network grows using the following operations[19]:

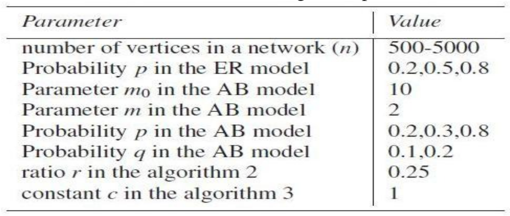
1. With probability p, add m new edges. For each edge, one of its end-points is selected randomly from the existing nodes. Another end-point is selected with probability

   $$\pi(k_i) = \frac{k_i+1}{\sum_j k_j+1} \qquad (2)$$

   where k_i is the degree of the i-th node;

2. With probability q, rewire m edges selected randomly in the network. For each edge, one of its end-points is changed to another node which is selected with a probability given by the Equation (2);
3. With probability $1-p-q$, add a new node and m new edges connecting the new node to other existing nodes. Another end-point of each new edge is also selected with a probability given by the Equation (2).

Table 2: Parameter Settings in Experiments



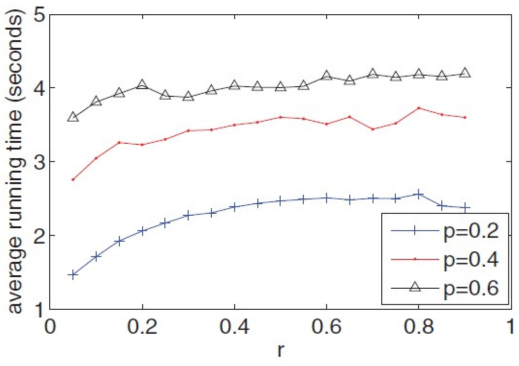
Parameter	*Value*
number of vertices in a network (n)	500-5000
Probability p in the ER model	0.2,0.5,0.8
Parameter m_0 in the AB model	10
Parameter m in the AB model	2
Probability p in the AB model	0.2,0.3,0.8
Probability q in the AB model	0.1,0.2
ratio r in the algorithm 2	0.25
constant c in the algorithm 3	1

Depending on the model parameter values, the AB model can not only generate networks with power-law degree distributions, but also networks with exponential degree distributions. When q < 0.5 then networks with power-law degree distributions will be generated. The model parameter values used in their experiments are given in Table 2 [3].

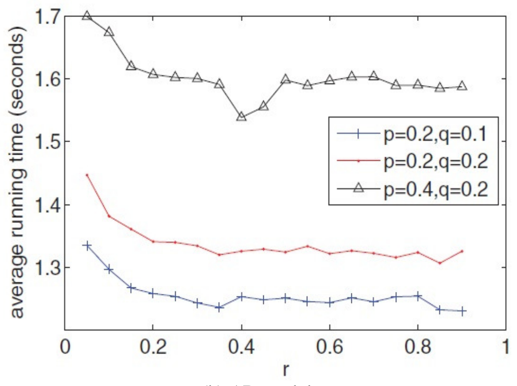
The algorithms to be compared include the classic Dijkstra's algorithm which is called iteratively using every vertex as source, algorithm 1, algorithm 2 and algorithm 3 [3].

For each parameter setting, they [3] randomly generate 50 network instances and run the algorithms on them. Then, the running time is averaged on all instances.
The test platform is a laptop with Intel Core i7-2640M CPU and 4GB memory, running Fedora 16 (Linux core 3.1.0-7) operating system.

Effects of Algorithm Parameters:



(a) ER model

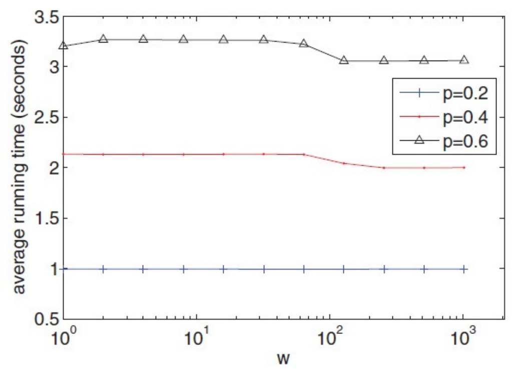


(b) AB model
Figure 4: Effects of parameter r

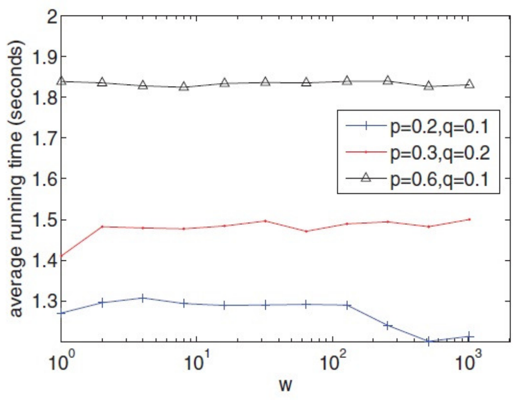
The Authors [3] test the performance of algorithms on different algorithm parameter values. The node number is set to 1000 for the ER model and 3000 for the AB model.

With different values of parameter r, the performance of algorithm 2 is demonstrated in Figure 4 [3]. It is shown that the average running time slightly increases with the increasing of r in cases of ER model. It seems that the optimization strategy is not useful in random networks of the ER model. In scale-free networks of the AB model, however, the average running time decreases with increasing of r. The average running time does not change significantly when r > 0.3. Thus, they can use a small value of r in algorithm 2 to improve the algorithm performance in scale-free networks.

The performance of algorithm 3 with different values of parameter c is demonstrated in Figure 5. The average running time does not change much in most cases.
Therefore, they can assume that the value of c has little impact on the performance of the algorithm 3. [3]



(a) ER model



(b) AB model
Figure 5: Effects of parameter c

Comparison of Algorithm Performance:

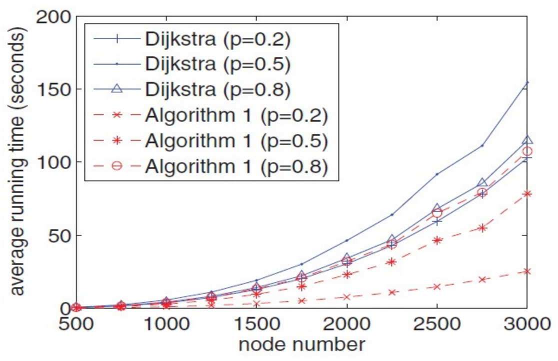
The Authors [3] compare the performance of the classic Dijkstra's algorithm and the algorithm 1 on random networks of ER model and AB model.

Case	a_0	a_1	a_1/a_0
$p = 0.2$	3.94856×10^{-9}	0.88043×10^{-9}	0.2230
$p = 0.5$	5.82049×10^{-9}	3.28129×10^{-9}	0.5637
$p = 0.8$	4.36906×10^{-9}	3.73352×10^{-9}	0.8545

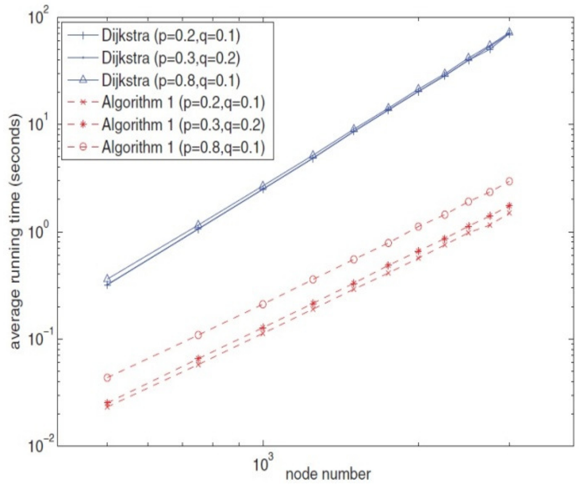
Table 3: Coefficients on Cases of ER Model

From Figure 6a, [3] they show that the average running time increases with the increasing of node number. The algorithm 1 outperforms than the Dijkstra's algorithm in random networks of ER model with different link probability. Both algorithms have the same level of time complexity which is $O(n^3)$, but with different factors.
Let the time complexity of Dijkstra's algorithm be $O(a_0 n^3)$ and the time complexity of algorithm 1 be $O(a_1 n^3)$.

Using polynomial regression method, they obtain the values of a_0 and a_1 in each case. The results are shown in Table 3. From the table, one can see that the coefficient ratio (a_1/a_0) increases with the increasing of probability p.



(a) ER model



(b) AB model
Figure 6: Comparison with Dijkstra's Algorithm

When p = 0.8, the average running time of the algorithm 1 is about 85% of the time of the Dijkstra's algorithm. When p drops down to 0.2, the algorithm 1 takes only about 22% running time of the Dijkstra's algorithm. Thereby, our algorithm is very efficient in cases of sparse random networks.

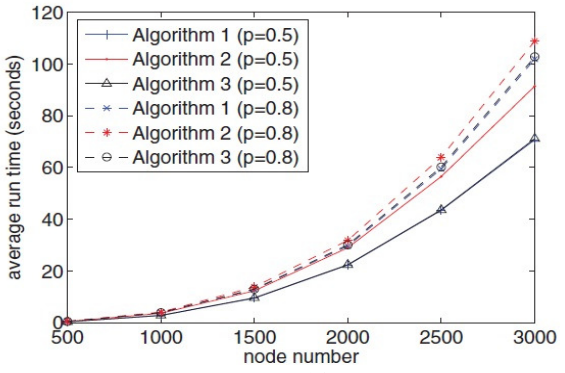
Figure 6b [3], shows the log-log curve of the average running time in scale-free networks of AB model. Interestingly, they find that the performance of algorithm 1 is significantly better than the performance of the Dijkstra's algorithm. The time complexity of algorithm 1 is even reduced in scale-free networks. From figure 6b, they show that the average running time follows the paw-law distribution for both algorithms. Denote the average running time by y and the node number by n. The relationship between them can be expressed by:

$$\log(y) = b_0 + b_1 \log(n) \qquad (3)$$

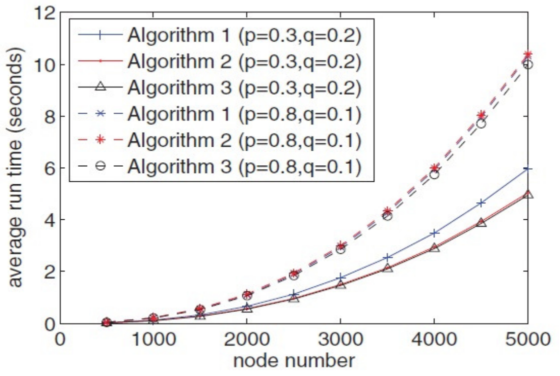
where b_0 and b_1 are coefficients.
The values of b_0 and b_1 are calculated using the linear regression method and the results are shown in Table 4. It is shown that the time complexity of Dijkstra's

algorithm is still $O(n^3)$, but the time complexity of Algorithm 1 is reduced to about $O(n^{2.4})$.

The performance of three proposed algorithms [3] is compared at last. The results are shown in Figure 7 [3]. It can be observed that the average running time of algorithm 2 [3] is slightly higher than the other two algorithms in cases of ER model, while the performance of algorithm 3 [3] is comparable to that of algorithm 1 [3].



(a) ER model



(b) AB model

Figure 7: Comparison of Proposed Algorithms

However, in cases of AB model, the average running time of algorithm 2 and the time of algorithm 3 are lower than the time of algorithm 1 when p is low.

Table 4: Coefficients on Cases of AB Model

	Dijkstra's Algorithm		Algorithm 1	
Case	b_0	b_1	b_0	b_1
$p = 0.2, q = 0.1$	-19.80237	3.00059	-18.22028	2.32293
$p = 0.3, q = 0.2$	-19.84295	3.00704	-18.37842	2.36342
$p = 0.8, q = 0.1$	-19.45874	2.96218	-17.83301	2.35931

When p is high, the performance of algorithm 2 is comparable to the performance of algorithm 1 while the performance of algorithm 3 is slightly better than the other two algorithms. [3]

4. Conclusions and Open Problems

Identifying optimal paths in dynamic networks is a non-trivial problem. The applications, computations, processes, data etc. migrate from blade servers to handheld computational devices and vice-versa which belongs to a dynamic network demand for runtime optimal path detections.

Throughout the paper we attempted to present all the algorithmic techniques within a unified framework by abstracting the algebraic and combinatorial properties and the data-structural tools that lie at their foundations.

Simulation results on real-world graph of Maryland State in US show that the running time and memory dissipation are acceptable on relatively small network, and also on enquiry of safe path between a pair of nodes with relatively short distance. The time complexity is only about $O(n^{2.4})$ when algorithm 3 is applied in scale-free networks generated by the AB model. The algorithm performance is slightly improved with the optimization strategies in scale-free networks.

Recent work has raised some new and perhaps intriguing questions. First, can we reduce the space usage for dynamic shortest paths to $O(n^2)$? Second, and perhaps more important, can we solve efficiently fully dynamic single-source reachability and shortest paths on general graphs? Lastly, are there any general techniques for making increase-only algorithms fully dynamic, remains unanswered.

Potential Application Areas:

Identifying Optimal Path is a broadly useful problem-solving model for:

- Maps
- Robot navigation.
- Texture mapping.
- Typesetting in TeX.
- Urban traffic planning.
- Optimal pipelining of VLSI chip.
- Subroutine in advanced algorithms.
- Telemarketer operator scheduling.
- Routing of telecommunications messages.
- Approximating piecewise linear functions.
- Network routing protocols (OSPF, BGP, and RIP).
- Exploiting arbitrage opportunities in currency exchange.
- Optimal truck routing through given traffic congestion pattern.

References

[1] Camil Demetrescu and Giuseppe F. Italiano. Algorithmic Techniques for Maintaining Shortest Routes in Dynamic Networks. Electronic Notes in Theoretical Computer Science 171 (2007) 3–15.

[2] Wu Jigang, Song Jin, Haikun Ji, Thambipillai Srikanthan. Algorithm for Time-dependent Shortest Safe Path on Transportation Networks. International Conference on Computational Science, ICCS 2011, Procedia Computer Science 4 (2011) 958–966.

[3] Wei Peng, Xiaofeng Hu, Feng Zhao, Jinshu Su. A Fast Algorithm to Find All-Pairs Shortest Paths in Complex Networks. International Conference on Computational Science, ICCS 2012, Procedia Computer Science 9 (2012) 557 – 566.

[4] D. B. Johnson and D. A. Maltz. Dynamic source routing in ad hoc wireless networks. In Imielinski and Korth, editors, Mobile Computing, volume 353. Kluwer Academic Publishers, 1996.

[5] P. Loubal. A network evaluation procedure. Highway Research Record 205, pages 96–109, 1967.

[6] J. Murchland. The effect of increasing or decreasing the length of a single arc on all shortest distances in a graph. Technical report, LBS-TNT-26, London Business School, Transport Network Theory Unit, London, UK, 1967.

[7] V. Rodionov. The parametric problem of shortest distances. U.S.S.R. Computational Math. and Math. Phys., 8(5):336–343, 1968.

[8] S. Even and H. Gazit. Updating distances in dynamic graphs. Methods of Operations Research, 49:371–387, 1985.

[9] H. Rohnert. A dynamization of the all-pairs least cost problem. In Proc. 2nd Annual Symposium on Theoretical Aspects of Computer Science, (STACS'85), LNCS 182, pages 279–286, 1985.

[10] E. W. Dijkstra. A Note on Two Problems in Connexion with Graphs. Numerische Mathematik, vol.1, pp.269-271, 1959.

[11] R. Bellman. On a Routing Problem. Quarterly of Applied Mathematics, vol.16, no.1, pp.87-90, 1958.

[12] J. B. Orlin, K. Madduri, K. Subramani, and M. Williamson. A Faster Algorithm for the Single Source Shortest Path Problem with Few Distinct Positive Lengths. Journal of Discrete Algorithms, vol.8, no.2, pp.189-198, June 2010.

[13] A. V. Goldberg, H. Kaplan, and R. F. Werneck. Reach for A*: Efficient Point-to-Point Shortest Path Algorithms. In Proc. of the Eighth Workshop on Algorithm Engineering and Experiments (ALENEX'06), Miami, Florida, USA, pp.129-143, Jan. 2006.

[14] L. Roditty and U. Zwick. On Dynamic Shortest Paths Problems. Algorithmica, March 2010 (published online).

[15] S. Pettie. A new approach to all-pairs shortest paths on real-weighted graphs. Theoretical Computer Science, vol.312, no.1, pp.47-4, Jan. 2004.

[16] T. M. Chan. All-pairs shortest paths with real weights in $O(n^3/\log n)$ time. in: Proc. 9th Workshop Algorithms Data Structures, in: Lecture Notes in Computer Science, vol.3608, Springer, Berlin, 2005, pp.318-324.

[17] D. Bolin and Y. J. Xu and Q. Lu, Finding time-dependent shortest paths over large graphs, in Proceedings of the 11th international conference on Extending database technology, France, March 2008, pp. 205-216.

[18] P. Erd¨os and A. R´enyi. On the Evolution of Random Graphs. Publications of the Mathematical Institute of the Hungarian Academy of Sciences, vol.5, pp.17-61, 1960.

[19] R. Albert and A.-L. Barab´asi. Topology of Evolving Networks: Local Events and Universality. Physical Review Letters, vol.85, no.24, Dec. 2000.