Parallel and Distributed Graph Algorithms and their Applications

Aditi Majumder
Department of Computer Science
University of North Carolina at Chapel Hill

Abstract

This paper surveys the advent of graph algorithms in the field of parallel and distributed computing, both because of the need for fast graph algorithms and also because of the applicability of graph algorithms in the distributed systems arena. A few parallel implementations of commonly used graph algorithms like minimum spanning tree, all pairs shortest path, and connected components, are discussed. A few applications in which these graph algorithms are critical, are also discussed. In the field of distributed systems, a few graph algorithms are used for effective communication in a wide/local area network. This paper also touches upon a few graph algorithms like connected components, and bi-connected components and elaborates on the use of distributed shortest path algorithm for routing techniques in ARPANET.

Keywords: Distributed Graph Algorithms, Parallel Graph Algorithms, ARPANET algorithms.

1 Introduction

The last four decades have witnessed an upsurge of interest and activity in graph theory, particularly among applied mathematicians and engineers. There are a large number of applications of graph theory to some areas in physics, chemistry, communication science, computer technology, electrical and civil engineering, architecture, operational research, genetics, psychology, sociology, economics, anthropology, and linguistics. The theory is also intimately related to many branches of mathematics, including group theory, matrix theory, numerical analysis, probability, topology and combinatorics. The fact is that graph theory serves as a mathematical model for any system involving a binary relation. Partly because of their diagrammatic representation, graphs have an intuitive and aesthetic appeal.

The first two or three decades saw an extensive research in computational and algorithmic aspects of graphs. But with the advent of the notion of parallel and distributed systems, there has been a swing in the research on graph algorithms from sequential domain to the parallel and distributed domain. It is important to mention here, that there is a clear-cut distinction between parallel graph algorithms and distributed graph algorithms, and it is absolutely necessary to understand this. In parallel algorithms the computation is distributed among systems while in the distributed algorithms computation is done working with geographically distributed systems. So while parallel graph algorithms are designed to increase efficiency of computation, distributes graph algorithms are a necessity, specially in networking. Any practical problem, be it in electrical network analysis, circuit layout in VLSI design, operational research or social sciences, almost always leads to very large graphs. In such cases, parallel algorithms are designed to make computations faster. So the application of such algorithms is more or less the same as that of the sequential algorithms. But when the nodes of a graph are geographically distributed, it becomes impossible to compute structures like MST, Shortest Paths, with the conventional algorithms since no node has the global topology, and the topology is ever changing with the presence of node and link failures. In such cases a completely different approach is required to solve problems, and that is essentially the domain of distributed graph algorithms [1] [2].

The rest of the paper is organized as follows. Section 2 presents the parallel graph algorithms and their applications. Section 3 presents some frequently used distributed graph algorithms and their applications. Section 4 deals in details with the application of one such distributed algorithm, the shortest path algorithm, in routing techniques for networks. Section 5 presents the conclusions and the scope of future work in this area.

2 Parallel Graph Algorithms

A graph $G = (V, E)$ can be of two types based on the number of edges it has. G is *sparse* if $|E|$ is much smaller than $O(|V|^2)$; otherwise it is *dense*. The adjacency matrix representation is used for dense graphs while that of adjacency list is used for sparse graphs.

It is easy to predict that parallel graph algorithms will require some kind of *data partitioning*, and equal partitioning of the adjacency matrix is much easier than that of adjacency list. In case of dense graphs, partitioning of the adjacency matrix is done in such a way that each processor does an equal amount of work and communication is localized. This lead to proper load balancing and less communication overhead. However for sparse graphs the adjacency list representation is essential to reduce the computational overhead taking the advantage of its sparseness. But then it is very difficult to achieve an even work distribution. So parallelism cannot achieve much enhanced efficiency for sparse graphs. However, for some particular kind of application specific sparse graphs, there are some efficient parallel algorithms.[1] Though, here our main focus will be on the algorithms for dense graphs.

2.1 Connected Components

The *connected components* of an undirected graph $G = (V, E)$ are the maximal disjoint sets $C_1, C_2, \ldots, C_k$ such that $V = C_1 \cup C_2 \cup \ldots \cup C_k$, and $u, v \in C_i$ iff u is reachable from v and v is reachable from u. The connected components of an undirected graph are the equivalence classes of vertices under the ‘ is reachable from’ relation.

The algorithms uses the concept of finding the connected components of a graph by using depth first search. The outcome of this depth first traversal is a forest of depth first trees. Each tree in the forest contains vertices that belong to a different connected component. This takes $O(|E|)$ time.

2.1.1 Parallel Formulation

The adjacency matrix of the graph G is partitioned in p parts, where p is the number of processors. Each part is assigned to one of the p processors. Each processor P_i has a subgraph G_i of G, where

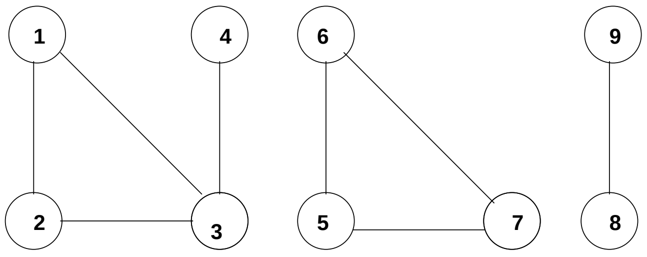


Fig 2.1 A graph with three connected components: { 1,2,3,4} , {5,6,7} , and {8,9}

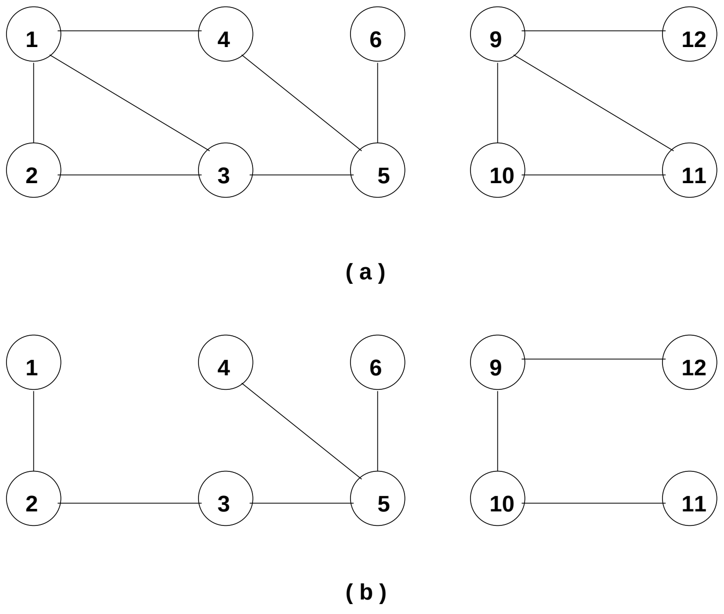


Fig 2.2 : Part (b) is a depth first forest obtained from the depth first traversal of the graph in part (a) . Each of these trees is a connected component of the graph in part (b).

$G_i = (V, E_i)$ and E_i are the edges that correspond to the portion of the adjacency matrix assigned to this processor. In the first step, each processor P_i computes the depth first spanning forest of the graph G_i. At the end of this step, p spanning forests has been constructed. During the second step, the spanning forests are merged pairwise until only one spanning forest remains. This spanning forest has the property that two vertices are in the same connected component of G if they are in the same tree.

Merging is done using disjoint sets of edges. Each spanning tree of a subgraph of G is represented by a set. The sets for the different trees are pairwise disjoint. Suppose we want to merge forest A into forest B. For each edge (u, v) of A, a find operation is performed for each vertex to determine if the two vertices are already in the same tree of B. If not, then the two trees of B containing u and v are united by union operation. Otherwise no union operation is necessary. Merging of A and B requires at most $2(n-1)$ find operations and $(n-1)$ union operations. Implementing this disjoint-set data structure with ranking and path compression takes $O(n)$ time. Fig 2.3 shows how the algorithm works for 2 processors.

2.1.2 Complexity Analysis on a Hypercube

For *data distribution* the adjacency matrix is partitioned into p stripes, each stripe comprising of n/p consecutive rows. This is called *block striped partitioning.* For computing the individual spanning forests, each processor takes $\Theta(n^2/p)$ time. The pairwise merging of the spanning forests are done by embedding a virtual tree in the hypercube. There are $\log p$ merging steps each taking $O(n)$ time. Thus the cost due to merging is $\Theta(n \log p)$. During each merging stage, spanning forests are sent between nearest neighbours. Now, $\Theta(n)$ edges of the spanning forest is transmitted. Thus communication cost is $\Theta(n \log p)$. Thus the parallel run time is

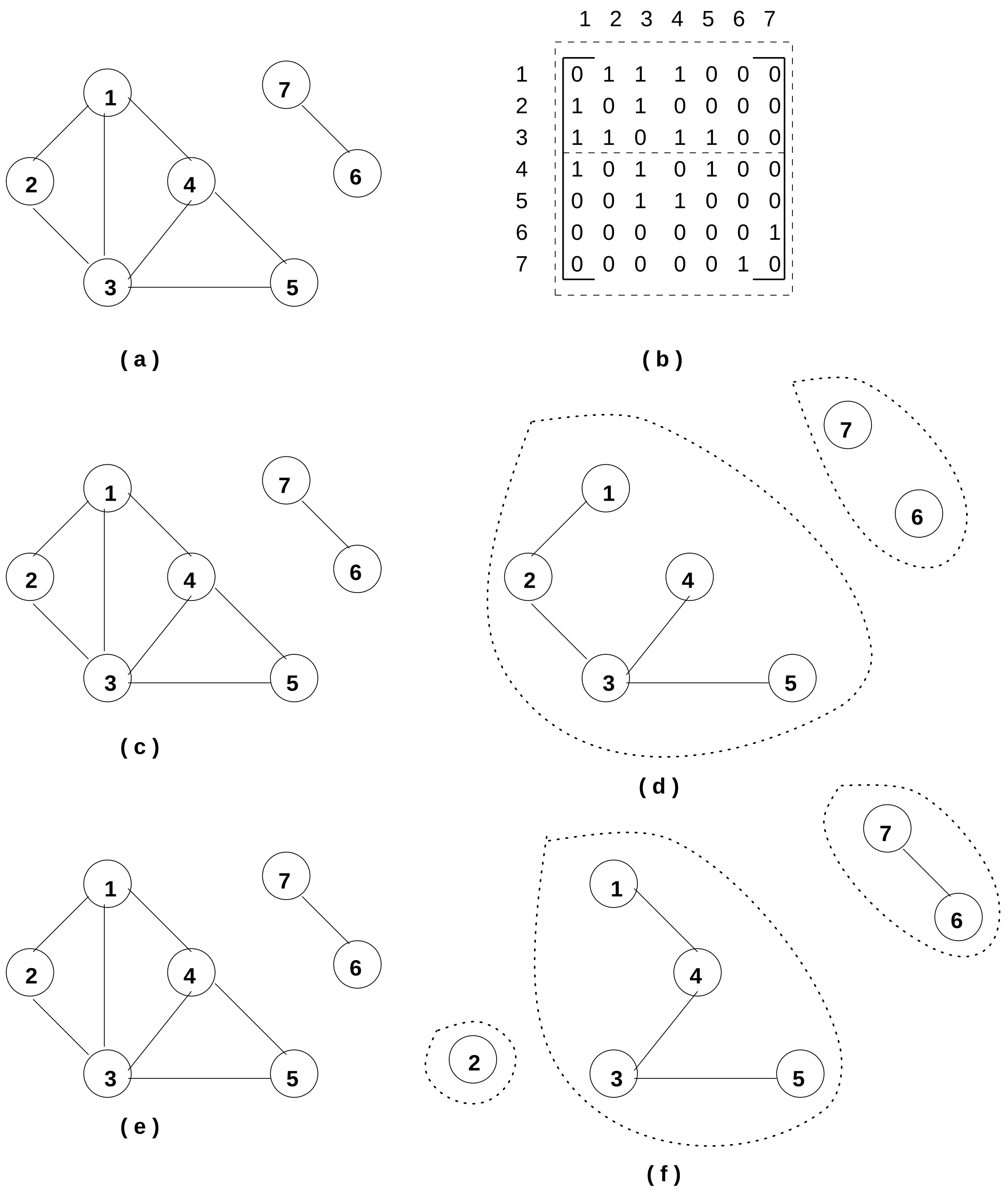
$$T_P = \Theta(n^2/p) + \Theta(n \log p)$$

.

2.2 Minimum Spanning Tree

A *spanning tree* of an undirected graph G is the sum of the weights of the edges in the subgraph. A *minimum spanning tree* (MST) for a weighted undirected graph is a spanning tree with the minimum weight. We use for this Prim's greedy method of finding an MST.

```
**procedure** PRIM-MST(V,E,w,r)
**begin**
  $V_T := [r]$;
  $d[r] := 0$;
    **for** all $v \in (V - V_T)$ **do**
      **if** edge $(r, v)$ exists set $d[v] := w(r, v)$;
      **else** set $d[v] := \infty$;
    **while** $V_T \neq V$ **do**
```



$$\begin{array}{c|ccccccc}
 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\
\hline
1 & 0 & 1 & 1 & 1 & 0 & 0 & 0 \\
2 & 1 & 0 & 1 & 0 & 0 & 0 & 0 \\
3 & 1 & 1 & 0 & 1 & 1 & 0 & 0 \\
\hline
4 & 1 & 0 & 1 & 0 & 1 & 0 & 0 \\
5 & 0 & 0 & 1 & 1 & 0 & 0 & 0 \\
6 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\
7 & 0 & 0 & 0 & 0 & 0 & 1 & 0
\end{array}$$

Fig 2.3 : Computing connected components in parallel. The adjacency matrix of the graph G in (a) is partitioned into two parts as shown in (b). Next each process gets a subgraph of G as shown in (c) and (e) . Each processor then computes the spanning forest of the subgraph, as shown in (d) and (f). Finally the two spanning trees are merged to form the solution.

```
    begin
      find a vertex $u$ such that $d[u] = min[d[v] | v \in (V - V_T)]$;
      $V_T := V_T \cup u$;
      for all $v \in (V - V_T)$ do
        $d[v] = min[d[v], w(u, v)]$;
    endwhile
end PRIM-MST
```

The working of this algorithm is shown in Figure 2.4. The overall complexity of Prim's algorithm is $\Theta(n^2)$.

2.2.1 Parallel Formulation

Prim's algorithm is iterative. Each iteration adds a new vertex to the minimum spanning tree. Since the value of $d[v]$ for a vertex v may change every time a new vertex u is added in V_T, it is impossible to select more than one vertex to include in the minimum spanning tree. Now, different iterations of the while loop cannot be performed in parallel. However each iteration can be performed in parallel as follows.

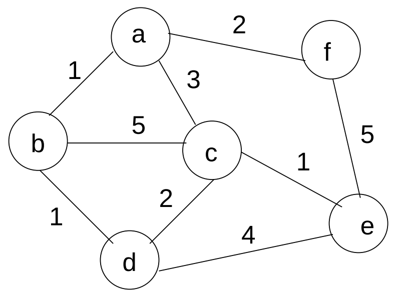
Let p be the number of processors, and let n be the number of vertices in the graph. The set V is partitioned into p subsets using the block striped partitioning. Each subset has n/p consecutive vertices, and the work associated with each subset is assigned to different processors. Let V_i be the subset of vertices assigned to the processor P_i for $i = 0, 1, \ldots, p-1$. Each processor P_i stores a part of the array d that corresponds to V_i . Fig 2.5 illustrates the partitioning. Each processor P_i computes $d_i[u] = min d_i[v] | v \in (V - V_T) \cap V_i$ during each iteration of the while loop. The global minimum is then obtained over all $d_i[u]$ by using a single node accumulation operation and is stored in processor P_0. Processor P_0 now holds the new vertex u that will be inserted in V_T. Processor P_0 broadcasts u to all processors by using one to all broadcast. The processor P_i responsible for vertex u marks u as belonging to set V_T. Finally each processor updates the values of $d[v]$ for its local vertices.

When a new vertex u is inserted into V_T, the values of $d[v]$ for $v \in (V - V_T)$ must be updated. The processor responsible for v must know the weight of the edge (u, v). Hence, each processor P_i needs to store the columns of the weighted adjacency matrix corresponding to the set V_i of vertices assigned to it.

2.2.2 Space and Time Complexity

The space required to store the required part of the adjacency matrix at each processor is $\Theta(n^2/p)$. Fig 2.5 illustrates the partitioning of the weighted adjacency matrix. The computation performed at each iteration to minimize and update the values of $d[v]$ is $\Theta(n/p)$. And since there is n iterations, hence the total order is $\Theta(n^2/p)$. The communication performed in each iteration is due to the single-node-accumulation and one-to-all broadcast.

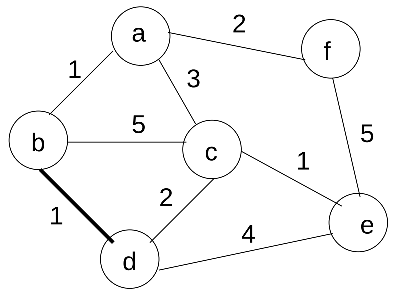
(a) Original graph



	a	b	c	d	e	f
d[ ]	1	0	5	1	-	-

$$\begin{array}{c|cccccc} a & 0 & 1 & 3 & - & - & 2 \\ b & 1 & 0 & 5 & 1 & - & - \\ c & 3 & 5 & 0 & 2 & 1 & - \\ d & - & 1 & 2 & 0 & 4 & - \\ e & - & - & 1 & 4 & 0 & 5 \\ f & 2 & - & - & - & 5 & 0 \end{array}$$

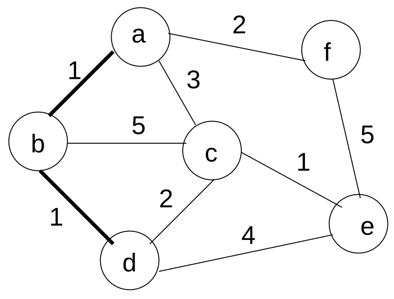
(b) After selection of 1st edge



	a	b	c	d	e	f
d[ ]	1	0	2	1	4	-

$$\begin{array}{c|cccccc} a & 0 & 1 & 3 & - & - & 2 \\ b & 1 & 0 & 5 & 1 & - & - \\ c & 3 & 5 & 0 & 2 & 1 & - \\ d & - & 1 & 2 & 0 & 4 & - \\ e & - & - & 1 & 4 & 0 & 5 \\ f & 2 & - & - & - & 5 & 0 \end{array}$$

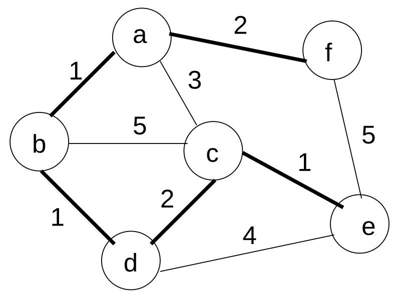
(c) After selection of 2nd edge



	a	b	c	d	e	f
d[ ]	1	0	2	1	4	2

$$\begin{array}{c|cccccc} a & 0 & 1 & 3 & - & - & 2 \\ b & 1 & 0 & 5 & 1 & - & - \\ c & 3 & 5 & 0 & 2 & 1 & - \\ d & - & 1 & 2 & 0 & 4 & - \\ e & - & - & 1 & 4 & 0 & 5 \\ f & 2 & - & - & - & 5 & 0 \end{array}$$

(d) Final minimum spanning tree



	a	b	c	d	e	f
d[ ]	1	0	2	1	1	2

$$\begin{array}{c|cccccc} a & 0 & 1 & 3 & - & - & 2 \\ b & 1 & 0 & 5 & 1 & - & - \\ c & 3 & 5 & 0 & 2 & 1 & - \\ d & - & 1 & 2 & 0 & 4 & - \\ e & - & - & 1 & 4 & 0 & 5 \\ f & 2 & - & - & - & 5 & 0 \end{array}$$

Fig 2.4 : Prim's minimum spanning tree algorithm .

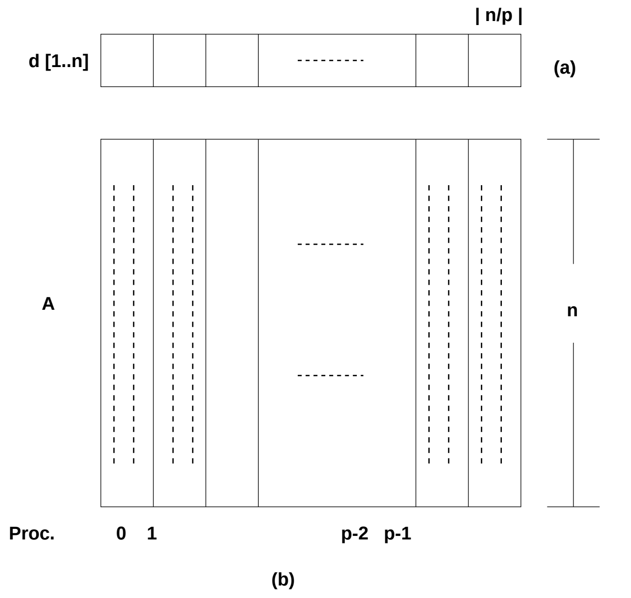


Fig 2.5 : The partitioning of the distance array d and the adjacency matrix A among p processors

2.3 All Pairs of Shortest Paths

The following section presents two algorithms to solve the all pairs shortest path (APSP) problem. The first uses Dijkstra's single source shortest path algorithm and the second uses Floyd's Algorithm. Dijkstra's algorithm requires non-negative edge weights while the other work with negative edge weights provided there is no negative cycle.

2.3.1 Dijkstra's Single Source Shortest Path Algorithm

For a weighted graph $G = (V, E)$, the *single source shortest paths* problem is to find the shortest paths from a vertex $v \in V$ to all other vertices in V. The algorithm is presented here. The runtime of this algorithm is $O(n^2)$.

```
procedure DSSSP(V,E,w,s)
begin
    $V_T := [s]$;
    for all $v \in (V - V_T)$ do
        if $(s, v)$ exists set $l[v] := w(s, v)$;
        else set $l[v] := \infty$;
    while $V_T \neq V$ do
    begin
        find a vertex $u$ such that $l[u] := min[l[v] | v \in (V - V_T)]$;
        $V_T := V_T \cup [u]$;
        for all $v \in (V - V_T)$ do
        $l[v] := min[l[v], l[u] + w(u, v)];$
    **endwhile**
**end** DSSSP
```

2.3.2 Parallel Formulation of DSSSP

The parallel formulation of this is very similar to that of Prim's minimum spanning tree. The weighted adjacency matrix is partitioned using *block striped mapping*. Each of the p processors is assigned n/p consecutive columns of the weighted adjacency matrix and computed n/p values of the array l. During each iteration, all processors perform computation and communication similar to that performed by the parallel formulation of Prim's algorithm.

2.3.3 Parallel Formulation of Dijkstra's APSP

DSSSP can be used to solve APSP by executing the single-source algorithm on each vertex v. Since the complexity of DSSSP is $\Theta(n^2)$, the complexity of this algorithm is $\Theta(n^3)$.

Dijkstra's all pairs of shortest paths problem can be parallelized in two distinct ways. One approach partitions the vertices among different processors and has each processor compute the DSSSP for all the vertices assigned to it. This is called the *source partitioned formulation*. In the other approach, computation of DSSSP for each vertex is assigned to a set of processors and use the parallel formulation of the DSSSP to solve the problem on each set of processors. This is called the *source parallel formulation.*

Source Partitioned Formulation The source partitioned formulation of Dijkstra's algorithm uses n processors. Each processor P_i finds the shortest path from vertex v_i to all other vertices by executing sequential DSSSP. It requires no inter-process communications. Thus the parallel run time of this is given by

$$T_P = \Theta(n^2)$$

. It may seem that, due to absence of inter-process communication this is an excellent parallel formulation. However, if the number of processors is greater than n, other algorithms will outperform this because of poor scalability.

Source Parallel Formulation The source parallel formulation is similar to the source partitioned one, except that the single source algorithm runs on disjoint subset of processors. Specifically p processors are divided into n partitions, each with p/n processors. Each of the n DSSSP is solved by one of the n partitions. Thus the total number of processors that can be used efficiently here is $O(n^2)$. Thus this exploits more parallelism than the previous one.

2.3.4 Floyd's Algorithm

Floyd's Algorithm for solving APSP is based on the following observation. Let $G = (V, E, w)$ be the weighted graph, and let $V = v_1, v_2, \ldots, v_n$ be the vertices of G. Consider a subset $S = v_1, v_2, \ldots, v_k$ of vertices for some k where $k \leq n$. For any pair of vertices $v_i, v_j \in V$, consider all paths from v_i

to v_j whose intermediate vertices belong to S. Let $p_{i,j}^{(k)}$ be the minimum weight path among them, and let $d_{i,j}^{(k)}$ be the weight of $p_{i,j}^{(k)}$. If the vertex is not on the shortest path from v_i to v_j, then $p_{i,j}^{(k)}$ is the same as $p_{i,j}^{(k-1)}$. However, if v_k is in $p_{i,j}^{(k)}$, then we can break $p_{i,j}^{(k)}$ into two paths - one from v_i to v_k and the other from v_k to v_j. Each of these paths uses vertices from $v_1, v_2, \ldots, v_{k-1}$. Thus, $d_{i,j}^{(k)} = d_{i,k}^{(k-1)} + d_{k,j}^{(k-1)}$. These observations lead to the following recurrence relation.

$$
\begin{aligned}
d_{i,j}^{(k)} &= w(v_i, v_j) && \text{if } k = 0 \\
&= min[d_{i,j}^{(k)}, d_{i,k}^{(k-1)} + d_{k,j}^{(k-1)}] && \text{if } k \geq 1
\end{aligned}
$$

The length of the shortest path from v_i to v_j is given by $d_{i,j}^{(n)}$. In general the solution is a matrix $D^{(n)} = d_{i,j}^{(n)}$. Floyd's algorithm solves the above recurence in a bottom up order of increasing values of k. The algorithm is presented below.

```
1.  procedure FLOYD
2.  begin
3.      $D^{(0)} = A$;
4.      for $k = 1$ to $n$ do
5.          for $i = 1$ to $n$ do
6.              for $j = 1$ to $n$ do
7.                  $d_{i,j}^{(k)} = min[d_{i,j}^{(k)}, d_{i,k}^{(k-1)} + d_{k,j}^{(k-1)}]$;
8.  end FLOYD.
```

As it pretty evident from the algorithm, the runtime is $\Theta(n^3)$.

2.3.5 Parallel Formulation of Folyd's Algorithm of APSP

A generic parallel formulation of of Floyd's Algorithm assigns the task of computing matrix $D^{(k)}$ for each value of k to a set of processors. Let p be the number of processors available. Matrix $D^{(k)}$ is partitioned into p parts, and each part is assigned to a processor. Each processor computes the $D^{(k)}$ values of its partition. To accomplish this, a processor must access the corresponding segments of k^{th} row and column of matrix $D^{(k-1)}$. The following describes one technique of partitioning matrix $D^{(k)}$

Block-Checkerboard Mapping One way to partition matrix $D^{(k)}$ is to divide matrix $D^{(k)}$ into p squares of size $(n/\sqrt{p})x(n/\sqrt{p})$, and assign each square to one of the p processors. It is helpful to think of the p processors as arranged in a logical grid of size $\sqrt{p}x\sqrt{p}$. Note that this is only a conceptual layout and does not actually reflect the actual processor interconnection network. The processor in the i^{th} row and j^{th} column is referred to as $P^{i,j}$. Processor $P^{i,j}$ is assigned a square subblock of $D^{(k)}$ whose upper left corner is $((i-1)n/\sqrt{p}+1, (j-1)n/\sqrt{p}+1)$ and whose lower right corner is $(in/\sqrt{p}, jn/\sqrt{p})$. Each processor updates its part of the matrix during each iteration. During the k^{th} iteration of the algorithm, each processor $P_{i,j}$ needs certain segments of the k^{th} row and the k^{th} column of the $D^{(k-1)}$ matrix. For example, to compute $d_{l,r}^{(k)}$ it must get $d_{l,k}^{(k-1)}$ and $d_{k,r}^{(k-1)}$. Now $d_{l,k}^{(k-1)}$ resides along the same row and $d_{k,r}^{(k-1)}$ resides along the same column as $P_{i,j}$. So during the k^{th} iteration $\sqrt{p}$ processors containing part of the k^{th} row send it to $\sqrt{p}-1$ processors in the same column. Same thing happens for rows. The parallel version of Floyd's algorithm using this partitioning is presented here.

```
1.  procedure FLOYD-PAR
2.  begin
3.     for $k = 1$ to $n$ do
4.     begin
5.        each processor $P_{i,j}$ that has a segment of the $k^{th}$ row of $D^{(k-1)}$;
             broadcasts it to $P_{*,j}$ processors;
6.        each processor $P_{i,j}$ that has a segment of the $k^{th}$ column of $D^{(k-1)}$;
             broadcasts it to $P_{i,*}$ processors;
7.        each processor wait to receive the needed segments;
8.        each processor $P_{i,j}$ computes its part of the $D^{(k)}$ matrix;
9.     end
10. end FLOYD-PAR
```

The running time of this algorithm is $\Theta(n^3/p)$.

3 Distributed Graph Algorithms

These kind of algorithms are finding more and more applications in information technology, networking and communications. A few of the important distributed graph algorithm along with their analysis will be presented here.[2]

3.1 Information propagation

The problem of propagating information through the nodes of G is the problem of broadcasting throughout G information originally held by only a subset of G. In this section we consider two variations of the problem known as the *Propagation of Information Problem or the PI Problem* and the *Propagation of Information with Feedback Problem or the PIF problem.* In the PI problem all that is asked is that all nodes in G receive the information while in the PIF problem the requirement is that not anly should all the nodes receive information but the originating node should also be informed that all the nodes has got that information.

There may be a very elegant solution to this problem. Find a spanning tree T on G. Broadcast the message along this spanning tree. But there are essentially three reasons why this approach cannot be adopted. Firstly, the spanning tree may not be available initially and has to be computed at the expense of additional message and time complexity. Secondly in case of failures new trees has to be calculated. Thirdly by repeatedly using the same lines to distribute messages we are not using the resources properly.

3.1.1 PI Algorithm

The algorithm we consider here does the broadcast by 'flooding the network', and for this reason has a higher message complexity than the one based on the spanning tree. The very simple idea behind the approach is that all nodes possessing *inf* initially send it all of their neighbours at the beginning (they all start concurrently). Every other node, upon receiving *inf* for the first time, sends it on to all of its neighbours, including the one from which it was received. As a result a node receives *inf* from all its neighbours. So, in this strategy, information is propagated like waves from the nodes that possess it, and reaches all nodes as fast as possible, regardless of all such edge failures that do not disconnect the graph. The algorithm to be followed by each node is presented here.

```
**Algorithm** PI
    **Variables**
        $reached_i$ = **false**
    **Input:**
        $msg_i$ = **nil**
    **Actions if** $n_i \in N_0$
        $reached_i$ = **true**
        Send $inf$ to all $n_j \in Neigh_i$
    **Input:**
        $msg_i = inf$
    **Action:**
        **if not** $reached_i$ **then**
            **begin**
                $reached_i$ := **true**;
                Send $inf$ to all $n_j \in Neigh_i$
            **end**
```

Here N_0 is the set of nodes who have the inf initially. One point is important to mention here is that, it is impossible for a node n_i for which $reached_i$ = **true** to tell whether a copy of inf it receives is a copy of a message that was already in transit, or the response to the message he sent out.

3.1.2 Complexity Analysis

Let $G = (V, E)$ where $|V| = n$ and $|E| = m$. In G there exists at least one path between every nose and all the nodes in N_0. So it is trivial matter to see that inf does indeed get broadcasted to all other nodes in the same component as N_0. Secondly, exactly one message traverses each edge in each direction, totalling $2m$ messages. So the msg complexity of this algorithm is $O(m)$ and the time complexity is $O(n)$.

3.1.3 Algorithm PIF

The solution by flooding to PIF is an extension of the flooding solution we gave for PI. A variable $parent_i$ is employed at each node n_i to indicate one of the neighbours of n_i. When n_i receives inf for the first time, $parent_i$ is set to point to the neighbour n_j from whom n_i received it, and forwards the inf to all its neighbours except $parent_i$.Upon receiving a copy of inf from each of its neighbours, n_i may then send inf to $parent_i$ as well. Node n_1 possesses the information that all the nodes have received inf upon receiving inf from all its neighbours.

```
Algorithm PIF
    Variables
        $reached_i$ = false
        $count_i = 0$;
        $parent_i$ = nil
    Input:
        $msg_i$ = nil
    Actions if $n_i \in N_0$
        $reached_i$ = true
        Send $inf$ to all $n_j \in Neigh_i$
    Input:
        $msg_i = inf$ such that $origin_i(msg_i) = (n_i, n_j)$
    Action:
        $count_i = count_i + 1$;
        if not $reached_i$ then
            begin
                $reached_i$ = true;
                $parent_i = n_j$;
                Send $inf$ to all $n_j \in Neigh_i, n_k \neq parent_i$
            end
        if $count_i = |Neigh_i|$ then
            if $parent_i \neq$ nil then
                Send $inf$ to $parent_i$;
```

3.1.4 Complexity Analysis

It follows easily that the collection of variables $parent_i$ for all $n_i \in N$ establishes on G a spanning tree rooted at n_1. The leaves in this tree are nodes from which no other node receives inf for the first time. The construction of this tree can be viewed as a wave of information that propagates outwards from n_1 and farther reaches G. Clearly this construction involves

$$|Neigh_i| + \sum_{n_i \in N - N_0} (|Neigh_i| - 1) = 2m - n + 1$$

messages and $O(n)$ time.

There are two important theorem that has to be stated from which the correctness of this algorithm is obtained very simply.

Theorem 3.1 *In Algorithm PIF, node $n_i \neq n_1$ sends inf to $parent_i$ within at most $2d$ time of having received inf for the first time, where d is the number of edges on the longest tree path between n_i and a leaf in T_i, where T_i is a sunset of N containing the nodes in the subtree rooted at node n_i. In addition. at the time this message is sent , every node in T_i has received inf.*

Theorem 3.2 *In algorithm PIF , node n_1 receives inf from all of its neighbours within time $O(n)$ of having executed and at the time the last inf is received every node in N has received inf.*

From these two the correctness of the algorithm follows trivially.

3.2 Connected Components

The problem that we treat in this section is the problem of discovery, by each node N, of identifications of all the other nodes to which it is connected by a path in G. The relevance of this problem becomes apparent when we consider the myriad of practical situations in which portions of G may fail, possibly disconnecting the graph and thereby making thereby unreachable from each other a pair of nodes that could previously communicate over a path of finite number of edges. The ability to discover the identification of the nodes that still share a connected component of the system in an environment that is prone to such changes may be crucial in many systems.

3.2.1 Algorithm

The algorithm is called Test-Connectivity and its essence is the following. It can be started by any of the nodes in N, either spontaneously or on receipt of the first message. In either case, what a node n_i does to initiate its participation in the algorithm is to broadcast its identification, id_i. This is coupled with the assumption that the links are FIFO. At any node n_i there are four array variables. They will be denoted by $parent_i, count_i, reached_i$. Other than this every node has a var called *initiated*. For any node n_i, $parent_i^k$ indicates the node in $Neigh_i$ from which the first id_k was received, $count_i^k$ indicates the number of times id_k has been received, $reached_i^k$ indicates whether id_k has been received atleast once or not. The *initiated* implies whether $n_i \in N_0$ or not. The algorithm is presented here.

```
Algorithm Test-Connectivity
  Variables
    $parent_i^k = \textbf{nil}$ for all $n_k \in N$;
    $count_i^k = 0$ for all $n_k \in N$;
    $reached_i^k = \textbf{false}$ for all $n_k \in N$;
    $initiated_i = \textbf{false}$;
  Input
```

```
    $msg_i$ = nil
  Action if $n_i \in N_0$
    $initiated_i$ = true
    $reached_i^i$ = true
    Send $id_i$ to all $n_j \in Neigh_i$
  Input
    $msg_i = id_k$ such that $origin_i(msg_i) = (n_i, n_j)$ for some $n_k \in N$
  Action
    if not $initiated_i$ then
      begin
        $initiated_i$ = true
        $reached_i^i$ = true
        Send $id_k$ to all $n_l \in Neigh_i$
      end
    $count_i^k = count_i^k + 1$;
  if not $reached_i^k$ then
      begin
        $reached_i^k$ = true
        $parent_i^k = n_j$;
        Send $id_k$ to every $n_l \in Neigh_i$ such that $n_l \neq parent_i^k$;
      end;
    if $count_i^k = |Neigh_i|$ then
        if $parent_i^k \neq$ nil then
          Send $id_k$ to $parent_i^k$
```

3.2.2 Complexity Analysis

Let $G = (V, E)$ be the graph such that $|V| = n$ and $|E| = m$. If the algorithm is studied a little carefully it will be found that its message complexity is n times than that of PIF algorithm. So the message complexity of this algorithm is $O(mn)$. Since the lengths of messages depend upon n, it is appropriate in this case to compute the algorithms bit complexity as well. Since each nodes identity can be expressed in $\log p$ bits, then the bit complexity is $O(mn \log p)$. The time complexity of the algorithm is essentially that of PIF, plus the time for a node in N_0 to trigger the initiation of another node as far from it as $n - 1$ edges, in summary, $O(n)$ as well.

3.3 Biconnected Components

A distributed algorithm to compute incrementally the biconnected components in a dynamically changing graph is presented here. A failure free distributed system is the model where each node can send msgs to any other node, messages between each pair of nodes is delivered in the order sent and eventual delivery is guaranteed. The algorithm finds the biconnected components of a dynamically changing *logical configuration graph* that defines the communication pattern of the application. Initially the logical configuration graph has no edges. The environment may cause changes to the graph by requesting the insertion or deletion of an edge at the node corresponding to either endpoint of the edge. The node at which the request is made is called the *requesting node*, and the node at the other endpoint is called the *peer*.

3.3.1 Algorithm

Here a very brief overview of the algorithm will be presented. The actual implementation of this is very complex and an interested reader is referred to [3].

In the serial algorithm, assumption is made that the environment makes the requests one at a time and the processing of one request is complete before the next request is made. The nodes of the environment are numbered. In any biconnected component (bcc) the node with the minimum number is designated as the coordinator. He maintains datastructures that stores

- The topology of that bcc.
- The articulations points of that bcc.
- The way it is connected to other bcc's through these articulation points.

The other nodes only maintain information about the members present in its own bcc but not the topology. Nodes participate in the algorithm by message passing.

There can be two kinds of request. One for *edge insertion* and another is for *edge deletion* that can change the topology. Each request is handled in three stages of message passing.

- coordinator notification
- classification
- Update

Edge Insert

Coordinator Notification Stage The requesting node sends a *Req-Update* message to its coordinator that instructs it to process the update.

Classification Stage In this stage, the graph is searched for the peer node in a broadcast over a tree of coordinators rooted at the coordinator of the requesting node. Relevant topology and coordinator mapping information are collected in a convergecast.

The coordinator first checks if both the requesting node r and the peer node p lies in its own bcc. Then it goes to the update stage for a *very easy link*. Otherwise it checks if it is a *easy link* for any of its neighbouring bcc's if r is an articulation point, through some message passing with p.

If it is not an easy link case it initiates a *Find-Peer* broadcast over the tree of coordinators in its own connected component. This message is broadcasted in a probe-echo fashion and ultimately the initiating coordinator receives a *Return-Peer* message if p is in the same connected component, otherwise it gets a *No-Peer* message. In the former we have a case of *path condensation* while in latter we have the case of *Component Link*, and the algorithm proceeds to its update stage.

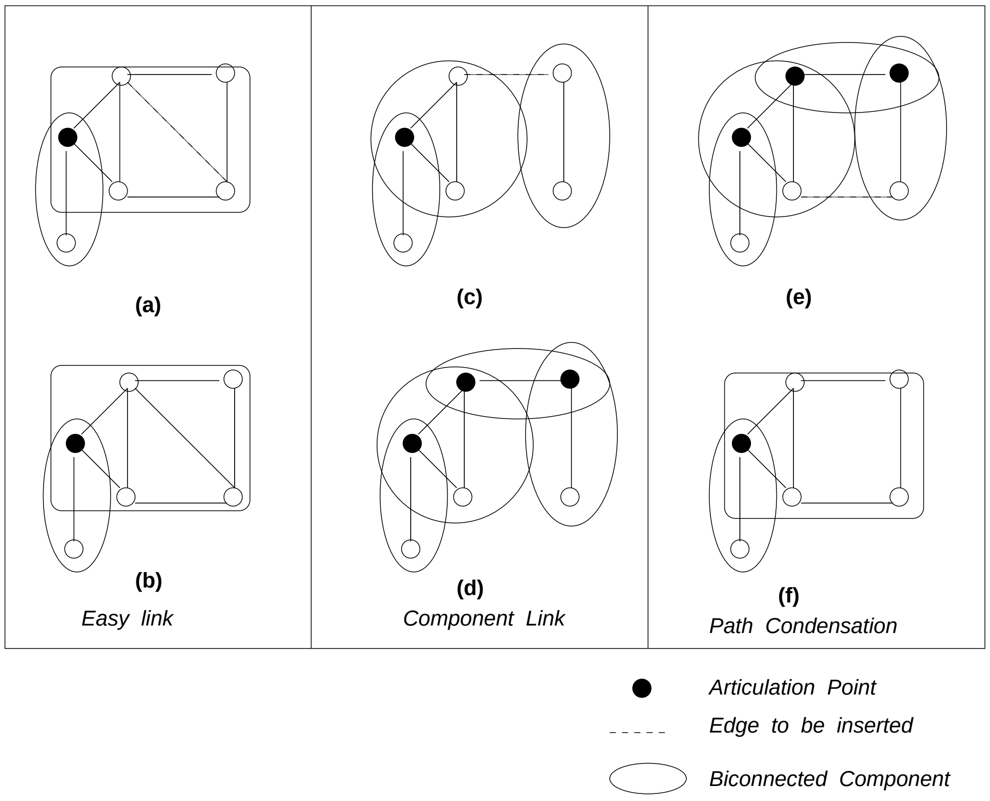


Fig 3.1 : Edge Insert

Update Stage In this stage the local bcc information is updated at every affected node, according to the information collected in the classification stage. There are four cases.

- In the very easy link case the coordinator updates its local bcc data structures.
- In the easy link case, a message is passed through the peer asking the neighbouring coordinator to add the required edge and update its local data structures. After this is done by the neighbouring coordinator it sends an acknowledgement.
- In case of the component link, the coordinator sends a *New-Bcc* message to both r and p, where the new bcc consists of only r and p. r and p communicate amongst themselves to decide who will be the coordinator. They exchange information about each others neighbouring coordinator topology. After that they send a *Note-Coord* message to all the neighbouring coordinators so that they get informed about the new bcc and update their own data structures accordingly. After this they send their ack to the initiating coordinator.
- In case of path condensation, the coordinator first updates its own local bcc. Then it sends an *Adjust-Mapping* message to all the coordinators that are being merged. All these communicate amongst themselves to decide upon the new coordinator and all the other coordinators which are going to be eliminated send their topology to the new coordinator through a *Coord-Update* message. The new coordinator updates its data structures, sends *Bcc-Update* message to all its members so that they can update the newly added members in the bcc, and then it sends a *Note-Coord* message to its neighbouring coordinator, informing them that it has become the coordinator of the new bcc.

After completion of the whole process, the new or the old coordinator sends a *done* message to the requesting node indicating end of processing for that request.

Edge Delete

The coordinator notification stage is exactly the same.

Classification Stage If r and p are in some larger bcc of the coordinator, then (r, p) is classified as an *internal edge*. If the set (r, p) is a bcc in the coordinator's local bcc topology or (after message passing with the peer) it is found that (r, p) is a bcc in the local topology of the peer, then we classify (r, p) as a *bridge edge*. Otherwise it is an *external edge*

Update Stage If the edge to be deleted is a *bridge edge*, then the coordinator sends a *Delete-Bcc* message to both r and p. Then these two nodes will send messages to their neighbouring coordinators so that they adjust their datastructures. After that they cease to be a bcc notifying this to the initiator coordinator.

If the edge is not a *bridge edge* the coordinator sends a *Delete-Edge* message to itself in case of an *internal edge*, and forwards through the peer node to the coordinator of the peer node in case of an *external edge*. The coordinator removes the edge, changes its local topology, sends *Coord-Update* message to each of the new coordinators that are formed.

In both cases, the coordinator sends a *done* message to the requesting node after the update is complete.

Concurrent Algorithm The concurrent algorithm can be obtained from this with a very slight modification. There the environment is allowed to issue multiple requests simultaneously. The coordinators use a timestamping technique to serialize these out, within each connected component. As a result each coordinator maintains a queue of request, and this is made consistent by communication over all the coordinators in the same connected component, so that they are all working on the same request at a time. So the only modification is after the coordinator notification stage there is timestamp collection stage after which it is decided on which request to process. And after the update stage there is a queue management stage which updates the queue of requests.

3.3.2 Complexity Analysis

Messages Note that the broadcast or a convergecast over the connected components takes $O(c)$ messages, where c is the number of bcc's in the connected component , and the coordinator notification takes $O(1)$ messages. The serialization of requests requires broadcast and convergecast and hence take $O(c)$ messages.

While inserting an edge, classification stage takes $O(c)$ messages since it involves the search for the peer. In the update stage, each node in the new bcc is sent a *Bcc-Update* message and neighbouring coordinators are sent a *Note-Coord* message. Since number of neighbours are bounded by $O(c)$, an update takes $O(b + c)$ messages, where b is the number of nodes in the resulting bcc. Similarly in deleting an edge also it takes $O(b + c)$ messages.

Total Message Length The message length is dominated by the update stage of the insert edge operation. Since each node in the resulting bcc could, in the worst case, be sent a new topology of the bcc formed, the total message length is $O(b(b + e))$, where b and e are the number of nodes and edges in the resulting bcc respectively. Such a case can arise when each node in the new bcc is a coordinator for some other bcc.

Time The time is measured in terms of units of message transmission time, where each message takes one unit of time. The serialization part of the algorithm takes $O(c)$ time, where c is the number of biconnected components in the connected component. Hence it takes $O(c)$ time for both insert and delete. Note that the update stage takes concurrent time since the messages can be processed concurrently.

4 Routing : An Application of Distributed Shortest Path Algorithm

4.1 The Sequential Bellman-Ford Algorithm

The Bellman-Ford Algorithm solves the single-source shortest paths problem in the more general case in which edge weights can be negative. Given a weighted directed graph $G = (V, E)$ with source s and weight function $w : E \rightarrow \mathbf{R}$, the Bellman-Ford algorithm returns a boolean value indicating whether or not there is a negative-weight cycle that is reachable from the source. If there is such a cycle, the algorithm indicates that no solution exists. If there is no such cycle, the algorithm produces the shortest path and their weights. Bellman-Ford uses a technique of relaxation, progressively decreasing an estimate $d[v]$ on the weight of a shortest path from the source s to each of the vertices $v \in V$. until it achieves the actual shortest path.[4]

The trick in the Bellman-Ford algorithm is to first find the shortest path weights subject to the constraint that the path contains at most one arc, then to find the shortest path weights subject to the constraint that the path contains at most two arcs, and so forth. The solution is given when we find the shortest path lengths subject to the constraint that the path contains at most $|V| - 1$ arcs.

Let $D_i^{(h)}$ be the shortest path weight from node 1 (source node) to node i subject to the constraint that the path contains at most h arcs. By convention we take $D_1^{(h)} = 0$ for all h. The Bellman-Ford algorithm is then as follows:

Initially,

$D_i^{(0)} = \infty$ for all $i \neq 1$

For each successive $h \geq 0$,

$D_i^{(h+1)} = min_j[D_j^{(h)} + d_{ji}]$ for all $i \neq 1$

In the worst case the algorithm must be iterated $N - 1$ times, each iteration must be done for $N - 1$ nodes. and for each node, the minimization must be taken over no more then $N - 1$ alternatives. So the runtime complexity of this algorithm is $O(N^3)$.

4.2 Requirements of a Good Routing Algorithm

A good routing algorithm should have the following properties

- **Correct** This means that message aimed at destination i should eventually reach i.
- **Simple** The algorithm should be simple and elegant to implement.
- **Robust** The algorithm should be able to withstand system wide failures of any kind for years. The routing algorithm should be able to cope up with the changes in the topology and traffic without requiring all jobs in all hosts to be aborted and the network to be rebooted whenever any node crashes.
- **Stable** The system to converge to an equilibrium. A very important implication of this property is that messages should not loop round and round. As we will see in the subsequent section that this is very difficult to achieve.
- **Fair** All the nodes in the network should get a fair share of the network resources.
- **Optimal** The network should have the resources allocated optimally. Fairness and Optimality often put contradicting demands on the routing algorithm. So a trade-off must be found between the two.

4.3 Distributed Asynchronous Bellman-Ford Algorithm

In this subsection we consider an algorithm similar to the one originally implemented in the ARPANET in 1969. The idea is to compute the shortest distances from every node to every other node by means

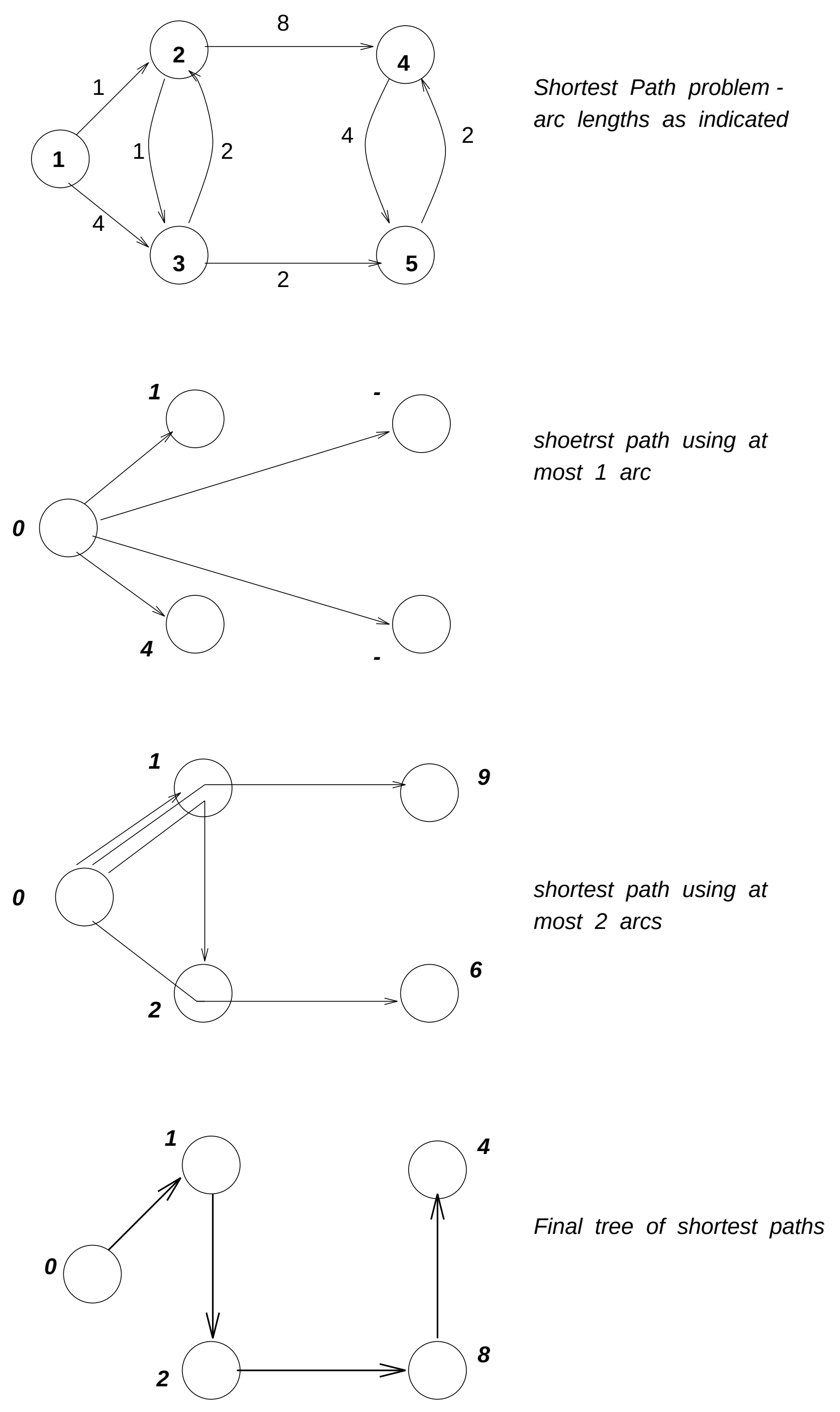

8
2
4
1
1
2
4
2
1
4
3
2
5
Shortest Path problem - arc lengths as indicated
1
-
shoetrst path using at most 1 arc
0
4
-
1
9
0
shortest path using at most 2 arcs
6
2
1
4
Final tree of shortest paths
0
2
8

Fig 4.1 : The working of Bellman-Ford Algorithm

of a distributed version of the Bellman-Ford Algorithm. An interesting aspect of the algorithm is that it requires very little information to be stored at the network nodes. Indeed, a node need not know the detailed network topology. It suffices for a node to know the weight of its outgoing links and the identity of every node in the network.

4.3.1 The Algorithm

It is assumed that each link (i, j) has a positive weight d_{ij}. It is also assumed that the network stays strongly connected, and if (i, j) is a link then (j, i) is also a link. In practical situation, weights d_{ij} can change with time. In the analysis, however, it is assumed that the lengths d_{ij} are fixed while the initial conditions can be arbitrary. These assumptions provide an adequate model for a situation where the link weight stay fixed for some time t_0 following a number of changes that occurred before t_0.

The focus is on the shortest distance D_i from each node i to a generic destination, taken for the sake of concreteness to be node 1. In reality, a separate version of the algorithm must be executed for each destination node. From Bellman's equation

$$\begin{aligned} D_i &= min_{j \in N(i)}[d_{ij} + D_j] \qquad & i \neq 1 \\ D_i &= 0 & \end{aligned}$$

where $N(i)$ denotes the set of current neighbours of node i, i.e. the nodes connected to i via an uplink. These equations are equivalent but slightly different from the ones given in the previous section. There the problem was to find the shortest distances *from* a given node 1 to all other nodes, while here the problem is to find the shortest distance *to* a given node 1 from all other nodes. One version of the problem can be obtained from the other by simply reversing the directions of each link while keeping the weight unchanged.

The algorithm given is well suited for distributed computation since the iteration for each h can be executed at each node i in parallel with every other node. One possibility is for all the i nodes to execute one iteration simultaneously, exchange information of their computation with their neighbours, and execute again with the index h incremented by 1. In that case the algorithm will terminate in at most $N - 1$ iterations (N is the number of nodes) with each node i knowing both the shortest distance D_i and the outgoing link on the shortest path to the node 1.

Unfortunately, implementing this algorithm in such a synchronous fashion is not as easy as it appears. Firstly, a mechanism is needed for all nodes to agree to start the algorithm. Secondly, a mechanism is needed to abort the algorithm and start a new version if a link status or weight changes when the algorithm is running.

A simpler alternative is used that does not insist on maintaining synchronization between nodes. This eliminates the need for either an algorithm restart or algorithm initiation protocol. The algorithm simply operates indefinitely by executing from time to time at each node $i \neq 1$ the iteration

$$D_i = min_{j \in N(i)}[d_{ij} + D_j]$$

using the latest estimates D_j received from the neighbours $j \in N(i)$, and the latest status or weight of the outgoing links from node i. The algorithm also requires that each node i, from time to time, transmit its latest estimate D_i to all its neighbours. Furthermore, no assumptions are made on the initial values $D_j, j \in N(i)$ available at each node i. The only requirement is node i will eventually execute the iteration if this changes D_i, and will eventually transmit the results to the neighbours. Thus a totally asynchronous mode of operation is envisioned.

It has been proved that the algorithm is still valid when executed asynchronously as described above. It can also be proved that if a number of link weight changes occur upto some time t_0, and no other changes occur subsequently, then within a finite time from t_0 the asynchronous algorithm finds the correct shortest distance of every node. The shortest distance estimates available at time t_0 can be arbitrary non-negative numbers, so it is not necessary to reinitialize the algorithm after each link status or link weight change.

4.3.2 Implementation

The original 1969 ARPANET algorithm was based on the asynchronous scheme described above. The actual implementation of the scheme will be described here briefly.

- Attempt to route packets along the path of least delay.
- The total path is not determined in advance, rather, each node decides which line to use in forwarding the packet to the next node.
- Each node maintain a table of estimated delay to each other node, and sent its table to all its adjacent nodes every 128secs.
- When node i receives the table from the adjacent node j, it first measures the delay from itself to j.
- Then it computes the delay via j to all other nodes by adding to each entry in j's table its own delay to j.
- Once node i has received the table from all its adjacent nodes it can easily determine which adjacent node will result in the shortest delay, and accordingly updates its routing table.

4.3.3 Technical Problems

- Since each node forwards the estimate of all the nodes in the network, the update message is very long and uses a lot of link and node bandwidth.
- Since the updates are transmitted every 128ms the network does not get enough time to reach equilibrium. Morover, each node does some computation and then hands over the result to its neighbour, so changes take time to propagate. As a result of which before a change has propagated properly, another one comes in, and the network never stabilizes.
- Delay is measured to be proportional to the queue length at any node. Since here packet size is variable, a shorter queue length may not necessarily mean shorter traffic. As a result of

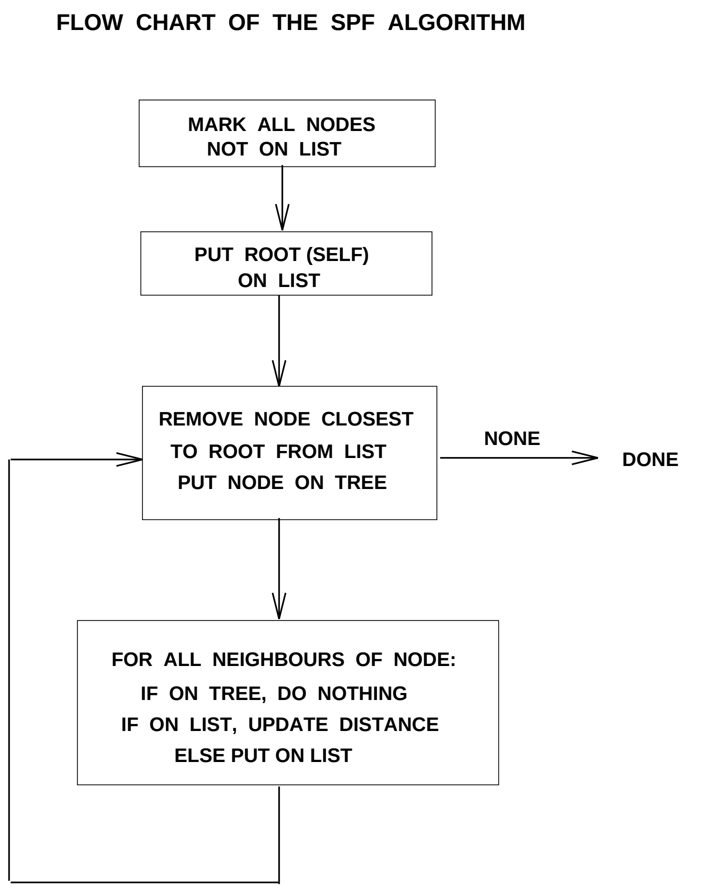

FLOW CHART OF THE SPF ALGORITHM
MARK ALL NODES NOT ON LIST
PUT ROOT (SELF) ON LIST
REMOVE NODE CLOSEST TO ROOT FROM LIST PUT NODE ON TREE
NONE
DONE
FOR ALL NEIGHBOURS OF NODE:
IF ON TREE, DO NOTHING
IF ON LIST, UPDATE DISTANCE
ELSE PUT ON LIST

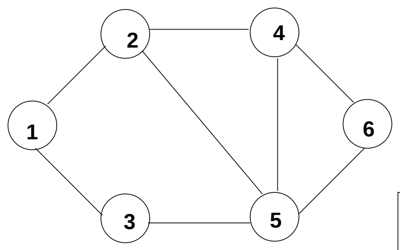
Fig 4.2 : The SPF Algorithm

these, the network responds very quickly to minor changes but takes lot of time to adapt to major failures.

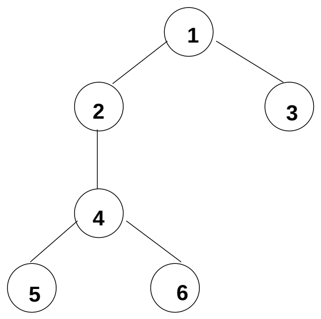
4.3.4 Modification

To do away with all these problems, a slightly modified protocol was designed which is presented here.[6] The algorithm for this protocol was called the shortest path first(SPF) and is described in Fig 4.2. An example using SPF is explained in Fig 4.3

- Work done by each node
  - Maintains a database containing the network topology and line delays.
  - Calculates the best paths to all other nodes independently using the database, routing outgoing packets accordingly.
  - Measures the delays along its outgoing links periodically and forwards this 'routing update' to all other nodes.



(a) NETWORK



(b) SHORTEST PATH TREE

destination node	route traffic via node
2	2
3	3
4	2
5	2
6	2

(c) ROUTING DIRECTORY

Fig 4.3 : An example of the SPF Algorithm

  - Forwards any 'routing update' information before processing it and then updates the database according to the 'routing updates received.

- Distribution of 'routing updates'
  - The size of the update packet is pretty small(176 bits on an average) and contains information independent of all other nodes. This uses little line or node bandwidth.
  - The update generated by a node travels unchanged to all the other nodes in the network.
  - The updates are handled with highest priority, therefore they propagate very fast in the network.
  - Flooding without allowing duplicates is used for this purpose.
- Measurement of line delays
  - The actual delay of each packet flowing over each of the outgoing links from a node is measured, and an average is taken over them every 10secs.
  - The delay is reported to all other nodes only if it is significantly different from the previous delay, or in other words, it is greater than a particular threshold value. This threshold value is a decreasing function of time. This is done to assure that minor changes do not affect the network frequently and at the same time minor changes do not accumulate and lead to major changes. This threshold function is designed in such a manner that each node is bound to transmit a update message once in 1 minute.

The algorithm is also modified a little to take care of failures.
Response of node I to change in line delays

- Delay of line AB from node A to B increases by X
  - Identify the nodes in B's subtree and increase their delays from I by X.
  - For each node S in B's subtree, examine S's neighbours which are not in the subtree to see if there is a shorter path from I to S via those neighbours. If such a path is found, put node S on LIST.
  - Invoke a slightly modified version of SPF on the nodes present in the LIST to find the best paths.(if it is non-empty)
- Delay of line AB from node A to B decreases by X
  - Identify the nodes in the subtree and decrease their distances from I by X.
  - Try to find a shorter distance for each node K that is not in the subtree but us adjacent to a subtree node by identifying a path to K via an adjacent node which is in the subtree. If such a node is found put K on the LIST.
  - Invoke a slightly modified version of SPF on the nodes in LIST to restructure the tree.(if it is non-empty)

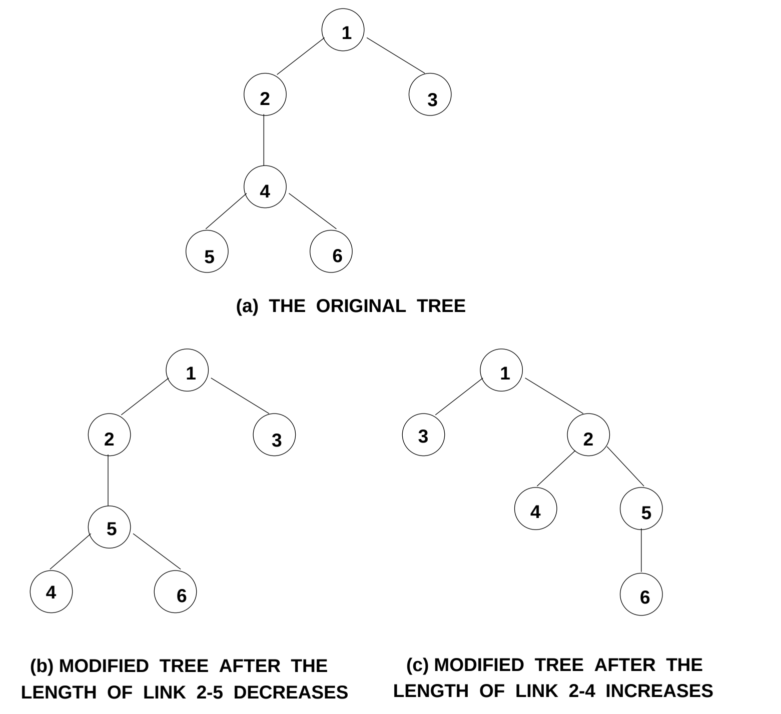

(a) THE ORIGINAL TREE
(b) MODIFIED TREE AFTER THE LENGTH OF LINK 2-5 DECREASES
(c) MODIFIED TREE AFTER THE LENGTH OF LINK 2-4 INCREASES

Fig 4.4 : SPF modifies to link and node failures

4.3.5 Complexity of SPF

Since the average subtree size provides a measure of the SPF performance, it is useful to understand how this quantity varies with the size of the network. Let N denote the number of network nodes, and let h_i represent the number of hops on the path from the source node, $i = 1$, to node i, in other words, if the weight of each line is 1 then h_i is the weight of the path to node i. Clearly, node i appears in i's subtree and in the subtrees of all nodes along the path to i. Thus h_i is equal to the number of subtrees in which node i is present, so that total number of all subtree nodes

$$= \sum_{i=2}^{N} h_i$$

and since there are $N - 1$ subtrees, the average subtree size is given by

$$= (1/N - 1) \sum_{i=2}^{N} h_i$$

But this expression is identical to the average hop length of all paths, and thus the average subtree size is equal to the average hop length from the root to all the nodes. This result is significant since the average hop length increases quite slowly as the number of nodes increases. For a network of uniform connectivity $c > 2$, the average hop length increases roughly as $\log N / \log(c - 1)$.

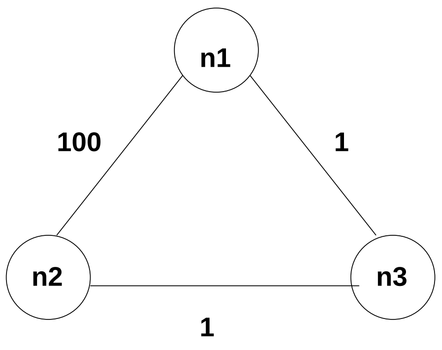


Fig 4.4 : Network topology showing Bouncing Effect

4.3.6 Analysis

Both asynchronous Bellman-Ford and the SPF algorithm does not take care of loop free transmission. In fact in both these algorithms, there are three major shortcomings. They are

- Bouncing Effect
- Continuing-to-Infinity
- Looping

Bouncing Effect: Consider the destination node n_1. Assume that, each node obtains the correct distance and routing tables by running any of the above algorithms. It is easy to see that the node n_2 and n_3 will choose nodes n_1 and n_2 as their next neighbours respectively. Now if link (n_1, n_2) fails, node n_2 will choose node n_3 as the next node. Thus a routing table loop occurs between n_2 and n_3. Furthermore nodes n_2 and n_3 have distances 3 ans 2 respectively, which are much less than 100 to the destination n_1. Now node n_3 will update its distance to 4, which again will cause node n_2 to increase its distance to be 5. Clearly, nodes n_2 and n_3 will keep on increasing their distances until n_3 reaches a distance of 102 to n_1, and chooses n_1 as its next neighbour. This scenario is called the bouncing effect.

Continuing-to-Infinity: This problem becomes much worse if now link (n_1, n_3) also fails, which is equivalent to link (n_1, n_3) having an infinite weight. Apparently, node n_2 and n_3 will continue to increasing their distance to n_1 without bounds. This is called the continuing-to-infinity problem.

Looping: Not only in the case of failures, but at any moment, when the updates are being done, the paths implied by the routing table may have loops. If such a loop persists for a long time then looping of data may occur resulting in considerable overhead.

4.4 Loop Free Bellman-Ford Algorithm

In this section a refined shortest path algorithm is presented that avoids the bouncing effect and continuing to infinity.[7] As it is evident that the bouncing effect and the continuing-to-infinity behaviour arise due to the fact that a node may offer its neighbour a distance corresponding to a path that has the neighbour as an internal node of the path. Thus once the distance is updated it may correspond to a non-simple path (path with cycles). The routing table contains for each destination node the distance and the next preferred node on the path to the destination. Now at any moment, any finite distance maintained in the routing table of a node i for a destination j is generated over the network by sequentially accumulating a set of link weights, which either existed at some previous moment or currently exist in the network. Also, this set of links form a path from i to j. For example, a distance entry D in the routing table of node i maintained at time t, can be expressed as

$$D = \sum_{m=1}^{r} d_{n_m n_{m+1}}(tm)$$

where $n_1 = i$, $n_{r+1} = j$ and $d_{n_m n_{m+1}}(t_m)$ is the weight of the link (n_m, n_{m+1}) existing in the network at some time $t_m \leq t$. Indeed node i gets informed about D via nodes $j, n_r, \ldots, n_2, i$. We shall refer to this path as *path implicit.* The bouncing effect and continuing-to-infinity is due to the fact that path implicit in a distance entry in a nodal routing table may be non-simple. So these effects can be avoided if the protocol searches for the shortest paths among only simple paths. This can be attained in a very straight forward fashion so that at each node in addition to storing distances, also store the implicit paths for these distances in the routing table. However the overhead in this approach is large, because an entire path corresponding to each distance entry is recorded. consequently the message size and the local storage for each node becomes $O(n)$ where n is the number of nodes in the network.

A main feature of the protocol presented here is the notion of the *head of path*, which permits each node to infer the implicit path in the distance entry without adding excessive overhead or update message or local storage. Each entry in the routing table is associated with a head of the path (last node just before the destination) information with each entry.

4.4.1 The Protocol

The routing table at node i is a vector with each entry a quadruple specifying the destination j, the next node P_{ij} for reaching j, the current shortest distance $d_i(j)$ and the head $h + i(j)$ of the path corresponding to $d_i(j)$. The distance table D at node i is a matrix with an entry being a pair (D^k_{ij}, h^k_{ij}) where the former represents the distance from i to j via k, and the latter represents the head of the node in the corresponding path. In fact, the row of the distance table correspond to destinations and columns to neighbours.

$DEST\#$	d	P	h
1	0	*	*
2	3	6	5
3	4	6	2
4	6	6	7
5	2	6	6
6	1	6	1
7	5	6	3

This table illustrates how a node determines if its neighbour is in the path from it to the destination. Say 1 wants to determine if 7 is in its shortest path destination to 2. 1 starts search from destination 2, finds 5 as its head, goes to 5, finds 6 as its head. This continues until he traces the whole path as $2 - 5 - 6 - 1$ and determines that 7 is not on the implicit shortest path to 2. If 7 was on the path, then at some point of tracing 7 would have been encountered and the trace could have stopped there. This path derived from the information in the routing table is called the *extracted path*. It is to be noted here that checking if the neighbour is in the path extracted from a shortest path distance by the sender is done by assuming that such a path extracted from the routing table using the head information is the path implicit in this shortest path distance. However it can be proved that this is not always true. To conquer this, the rule to generate the routing table is modified such that, the distance in the routing table to go to any node j is determined by choosing from column k iff D^k_{ij} is the minimum among row j of distance table and each node v in the path extracted from D^k_{ij}, which is assumed to be on the same path as the path implicit in D^k_{ij}, must be that D^k_{iv} is also minimum among row v and is chosen to be put in the routing vector.

For convenience, weight change on any link (i, j) is treated as if (i, j) fails and comes up immediately with the new weight. In addition, failure and recovery of a node is treated as if all the link adjacent to that node fail or recover. Therefore, the events that the protocol can encounter are link failures and recoveries.

4.4.2 Properties

The properties of this algorithm are presented here very briefly. An interested reader is referred to [7].

- At any moment in computation, the path extracted from any distance maintained at each node, say i, to any destination, say j, is a simple path and is equivalent to the path implicit in such distance.
- This algorithm is without bouncing effect.
- This does not have continuing-to-infinity behaviour.
- This algorithm terminates in a finite time after the last topological change.

- When the algorithm terminates, any link weight maintained in the distance table must be in the final graph.
- When the algorithm terminates, the distance for any node i to amy other reachable node j maintained in the routing table of i is the shortest distance between i and j in the final graph and the next neighbour on the path to the destination is maintained correctly; also the distance from i to any unreachable node is marked as undetermined.
- This is not loop-free, however, only multiple failures can cause loops.
- The time complexity for this algorithm is $O(h)$ for a single link failure/recovery, where h is the maximum height experienced during computation ; and $O(N)$ for multiple failures, where N is the total number of nodes in the network.

From all this properties the correctness of this protocol follows trivially. However it must be mentioned here that this protocol can be slightly modified to form what is called a *enhanced protocol* whose performance is bounded by $O(h)$.

5 Conclusion

In this paper a small subset of the actually existing algorithms has been presented. A parallel algorithm for finding the maximum flow in a weighted graph, has been proposed by Shiloach and Vishkin [8]. Detailed discussion on parallel graph algorithms can be found in [1]. This literature also discusses about the sparse graph algorithms which have orthogonal issues to be considered, when compared with dense graph algorithms in the parallel domain. Readers can find distributed algorithms for minimum spanning tree and maximum flow graph algorithms in [2]. More about the correctness proofs of the most recent loop-free routing algorithms used in ARPANET can be found in [7].

The direction of development in this field points us towards the need for different algorithms for the same graph problems based on different application domains in the field of distributed computing, and various implementation domains in the field of parallel computing. The issues related to their performance are also specific to the implementation and application. Therefore, comparison of these algorithms need to have a common platform. Thus, the research might go in the direction of finding a parameter to compare these algorithms and also towards finding new algorithms for various upcoming applications and architectures.

References

[1] Vipin Kumar, Ananth Grama, Anshul Gupta and George Karypis
Introduction to Parallel Computing, The Benjamin/Cummings Publishing Company, Inc. 1994.

[2] Valmir C. Barbosa
An Introduction to Distributed Algorithms, The MIT Press, Cambridge, Massachusetts, London, England.

[3] Bala Swaminathan and Kenneth J. Goldman
An Incremental distributed Algorithm for Computing Biconnected Components, Eighth International Workshop on Distributed Algorithms, Terschelling, Netherlands, September 1994.

[4] Mischa Schwartz and Thomas E. Stern
Routing Techniques used in Computer Communication Networks, IEEE Transactions on Communications, Vol. Com-28, No. 4, April 1980.

[5] Jeffrey M. Jaffe and Franklin H. Moss
An Responsive Distributed Routing Algorithm for Computer Networks, IEEE Transactions on Communications, Vol Com-30, No. 7, July 1982.

[6] John M. McQuillan and Ira Richer
The New Routing Algorithm for ARPANET, IEEE Transactions on Communications, Vol Com-28, No. 5,May 1980.

[7] Chunhsiang Cheng, Ralph Riley and Srikanta P.R. Kumar
A Loop Free Extended Bellman-Ford Routing Protocol without Bouncing Effect, 1989 ACM.

[8] Y.Shiloach and U.Vishkin.
An $O(n^2 \log n)$ Parallel Max-flow algorithm, Journal of Algorithms, 3:128-146, 1982.