On *k*-Connectivity Problems in Distributed Systems

Vahid Khalilpour Akram Orhan Dagdeviren*
Ege University
International Computer Institute
Izmir, Turkey

*Corresponding Author Contact: orhandagdeviren@gmail.com

Abstract *k*-connectivity detection and restoration are important problems in graph theory and computer networks. A graph is *k*-connected if it remains connected after removing *k*-1 arbitrary nodes. The *k*-connectivity is an important property of a distributed system because a *k*-connected network can tolerate *k*-1 node failures without losing the network connectivity. To achieve the *k*-connectivity in a network, we need to determine the current connectivity value and try to increase connectivity if the current *k* is lower than the desired value. This chapter reviews the central and distributed algorithms for detecting and restoring the *k*-connectivity in graphs and distributed systems. The algorithms will be compared from complexity, accuracy and efficiency perspectives.

Keywords Graph Connectivity, Distributed Systems, *k*-Connectivity, Connectivity Detection, Connectivity Restoration, Distributed Algorithm, Central Algorithm, Fault Tolerance, Wireless Sensor Networks.

1 Introduction

The connectivity is one of the key properties in the graph theory and computer networks. Generally network topology is modeled as a graph and network algorithms are implemented over the graph data structure. A network is said to be connected if there is at least one path between every node in the network. Connectivity among the nodes is the most important necessity in all networks. Most of the networks such as LANs and WANs have communication infrastructures (e.*g*. cables, switches and routers) and provide higher reliability from the connectivity perspective. However it is not possible to prepare a communication infrastructure in all environments, whereas using ad hoc or mobile wireless sensor networks is inevitable or affordable in some applications. In this kind of networks there is no communication infrastructure and all nodes communicate with each other using some intermediate nodes.

In most ad hoc or wireless sensor networks, there is a special sink node which collects information from other nodes and transfers them to users. Typically sink is not reachable from the entire network because the areas that must be covered by the nodes is wide, harsh and include various obstacles. Hence, most of the nodes use other intermediate nodes to deliver their packets to sink. This means that, beside the main functionality, the nodes must work as routers to transfer the messages of other nodes among the network.

These kinds of networks provide lower connection reliability, because failure in some nodes can cut off the packet flow in the network. Even in the networks that have stable communication links, failure in a router may prevent the transmission of packets between some other hosts. Generally depending on the topology, it is quite possible that a failure in a node, divides the network to disconnected parts. In a network by losing such a node, the connection between other working nodes is lost and many active resources are wasted.

A network is called 1-connected if there is at least one path between every pair of nodes and there exists at least one node, which its removal, divides the network to separated segments. Such a node is called ***cut vertex***. Despite the application and network type, having a 1-connected topology puts the network in high division risk. Especially in the networks that are established in harsh environments recovering the cut vertices from crashes can be impossible or a very hard task. For this reason, there are many researches that have been made to find and resolve the cut vertices in a network (Atay & Bayazit, 2010)(Dagdeviren & Akram, 2014)(Tarjan, 1972). Generally 1-connected networks are not reliable and not fault tolerant from the connectivity perspective. In ad hoc and wireless sensor networks the nodes typically are battery powered and it is possible that some nodes die due to energy consumptions caused by packet transmissions. Hence a 1-connected ad hoc or wireless sensor network is not trustable and has high discontinuity potential. Having more independent paths between the nodes, increases the robustness of the connection of the network. If we have more than one path between each pair of nodes in the network, the failure in a node would not disconnect the communication path between other nodes because they can use alternative links.

A network is *k*-connected if we have at least *k* disjoint paths between each pair of nodes. A *k*-connected network can tolerate and remain connected if *k*-1 arbitrary nodes stop working. Higher values for *k* lead to stronger connectivity in the network. The *k*-connectivity not only increases the fault tolerance of the network, but also improves routing performance of the packets.

Figure 1 shows an example network where node 0 is sink. In this network, if nodes 4 and 6 stop working, more than half of the nodes lose their connection to the sink. In this case the sent packets from many active nodes are not delivered to the processing center. The connectivity value of this network is *k*=2.

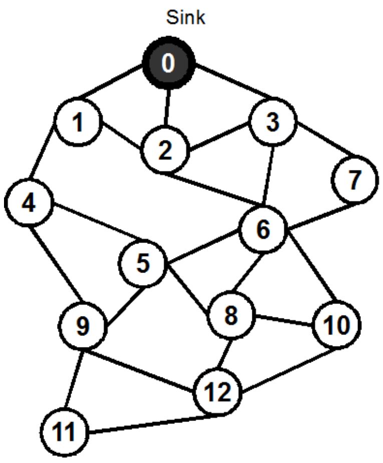


Figure 1: A sample network with *k*=2

The remaining parts of the chapter have been organized as follows; Section 2 defines some helpful terms and notations which are used in the explanation and evaluation of algorithms in next sections. Section 3 provides a formal definition of *k*-connectivity problem and also reviews the related topics and various aspects of this problem. Section 4 focuses on the algorithms that find the value of *k* in a given network. In this section the *k* detection algorithms have been classified in two central and distributed groups and a detailed explanation have been provided for the algorithms in each group. The *k*-connectivity restoration algorithms have been discussed in section 5. Just like section 4, the *k*-connectivity restoration approaches have been divided in two central and distributed algorithms which are discussed in sections 5.1 and 5.2 respectively. Finally a conclusion about the advantage of existence approaches and open issues on *k*-connectivity problem has been provided in section 6.

2 Definitions

The following symbols and terms have been used to explain and analysis the proposed algorithms in the next sections.

- The network is modeled as an undirected graph $G=(V,E)$ where V is the set of vertices that represents the nodes and E is the set of edges that indicates the links between nodes.
- d_v indicates the neighbor count of v or the degree of v in G.
- d is the minimum degree in G. In other words, all nodes in graph G have at least d neighbors.
- Δ is the maximum degree in graph G. So we have $\Delta = \max(d_v : v \in V)$.

3 *k*-connectivity problems

Various aspects of the *k*-connectivity problem have been investigated in several researches. The main outgrowth of *k*-connectivity is a fault tolerant network that can tolerate failure in many nodes without losing connection to active nodes. Generally we can divide the *k*-connectivity algorithms into two major parts:

- *k*-connected topology establishment.
- *k*-connectivity detection and restoration.

In the *k*-connected topology establishment category, some approaches use controlled node deployment and some other use radio power assignment methods to establish a *k*-connected network. Increasing the number of neighbors of each node in the network is a simplest solution to establish a *k*-connected network. Although there is no direct relation between *d* and *k* in a graph, the higher values for *d* increase the probability of *k*-connectivity.

In wireless networks increasing the radio range of nodes, increases the neighbors of node and leads to higher values for d. There are many researches about radio power assignment which try to achieve the *k*-connectivity by increasing the radio power of the nodes (Gupta & Gupta, 2013). (Nutov, 2008)(Younis, et. al., 2014). The main question in these researches is how to set the transmission ranges to establish a *k*-connected network. In the power assignment method, the radio range of each node is extended by increasing radio transceiver power, and more nodes connect to each other, which leads to achieve *k*-connectivity. The main constraint in this category is the limitation of radio power. Due to hardware limitations the radio transmission range of nodes cannot exceed a certain limit and it is not possible to assign the desired range for the nodes. The limit on the radio power is a significant constraint in this approach and it is impossible to have desired range for nodes all the times. Also increasing the radio power of nodes consumes more energy and discharges the battery of nodes in battery powered networks.

It is possible to create a *k*-connected network by new nodes deployment in the environment. The aim of some proposed approaches (Almasaeid, & Kamal, 2009)(Bai et. al., 2008) is to deploy new nodes in such a way that the resulting topology become *k*-connected. In the controlled deployment method, the nodes are deployed in certain positions such that the resulting topology becomes *k*-connected. The main disadvantage of this method is that it may not be possible to deploy nodes at desired positions all the times.

Another related topic is the *k*-connected topology maintenance which includes the researches that try to keep a topology in the *k*-connected form. These works suppose that the topology initially is *k*-connected and proposed algorithms try to hold the *k*-connectivity in case of failure in some nodes (Szczytowski, et. al., 2012) (Wang et. al., 2011). Finding the current connectivity value of the topology is not considered by these algorithms.

All of the mentioned topics are related to the *k*-connectivity problems but none of them find the value of *k* in a given network. Although the *k*-connectivity is an important property of the networks, finding the exact value of *k* in a topology is a complicated task and only a few algorithms have been presented for this problem.

In the next sections the proposed algorithms for *k*-connectivity detection and restoration will be discussed in more detail. The aim of *k*-connectivity detection is to find the current connectivity value of the network. The restoration algorithms increase the connectivity of the network by placing new nodes in certain positions or moving some nodes to new positions.

There are three major approaches for detecting the connectivity value of a graph:

- Approximating the connectivity value using probabilistic methods.
- Central algorithms.
- Distributed algorithms.

To restore the connectivity to a specific value of *k* there are two general approaches:

- Placing new nodes or activating reserved nodes.
- Moving mobile nodes to new locations.

Each of these approaches has some advantages and disadvantages that will be discussed in next sections.

4 The *k*-connectivity detection

In the *k*-connectivity detection problem, there is a network with arbitrary topology and we need to find the current connectivity value of this topology. The value of *k* in dense topologies is generally higher than the sparse topologies. Therefore it is reasonable that we define a probabilistic relation between density and connectivity of a topology. The most effective parameter on density is the number of nodes in the network and the area that the nodes placed in.

4.1 Probabilistic approaches for *k* detection

The probability of being a *k*-connected graph has been studied on many researches (Bettstetter, 2002) (Henzinger & Gabow, 2000) (Kallollu, et. al., 2014)(Ling & Tian, 2007) (Meghanathan & Gorla, 2010)(Penrose, 1999)(Reif & Spirakis, 1985) (Xing et. al., 2009) (Zhao, 2014)(Zhao, et. al., 2014) The aim of all of these approaches is to find a relation between the probability of having a *k*-connected graph and other parameters such as minimum or maximum node degrees, number of nodes and the area that the nodes are distributed. For example Penrose (Penrose, 1999) proved that the probability of having a *k*-connected graph is equal to the probability that the value of *d* in that graph is equal or greater than *k*.

$$P(G\ is\ k-connected) = P(d \geq k) \qquad (1)$$

Bettstetter (Bettstetter, 2002) and Ling (Ling & Tian, 2007) proved that the probability of having a wireless sensor network with minimum degree *d* can be calculated as relation (2).

$$P(d \geq n_0) = \left(1 - \sum_{j=0}^{n_0-1} \frac{(\rho\pi r_0^2)^j}{j!} e^{-\rho\pi r_0^2}\right)^n \qquad (2)$$
$$where\ \rho = \frac{n}{A}$$

In relation (2), n is the number of nodes, A is the area that the nodes are distributed and r is the range of each node. Using relations (1) and (2) it is possible to estimate a probabilistic value for the connectivity in an ad hoc or wireless sensor network. For example the probability of having a 5-connected network in a 100*100 area with *n*=50 and *r*=30 is:

$$P(G\ is\ 5-connected) = P(d \geq 5)$$

$$P(d \geq 5) = \left(1 - \sum_{j=0}^{5-1} \frac{(0.005 * 3.14 * 900)^j}{j!} e^{-0.005*3.14*900}\right)^{50} = 0.92$$

However these formulas only provide a probability of having a *k*-connected network according to the area, number of nodes and range of each node and are not useful to find the exact value of *k* in a network.

4.2 Central algorithms

Currently the only way to determine the exact value of k in a graph is to use a central algorithm. The simplest way to find k in a graph is to try all possible removal combinations of nodes and check the connectivity of the graph which leads to a brute force algorithm.

4.2.1 Brute Force Algorithm

In this algorithm we try to consider all possible removals of nodes and check whether the graph remains connected after removing any subset of nodes. The brute force algorithm generates the *power set* of nodes and removes each element in the power set from the graph and checks whether the remaining nodes in the graph are connected or not. The power set of any set S is the set of all possible subsets of S including S itself. For example the set of nodes in the graph of Figure 2 is $V = \{0,1,2,3,4\}$ and the power set of V is

$$PS(V) = \{\{0\},\{1\},\{2\},\{3\},\{4\},\{0,1\},\{0,2\},\{0,3\},\{0,4\},\{1,2\},\{1,3\},\{1,4\}, \\ \{2,3\},\{2,4\},\{3,4\},\{0,1,2\},\{0,2,3\},\dots,\{0,1,2,3,4\}\}$$

The algorithm sorts the power set according to the number of items in the subsets and starts to remove each element and then checks whether the graph remains connected after removing that item. If the graph remains connected the algorithm tries the next element in the power set. Otherwise the size of the removed set is reported as *k*.

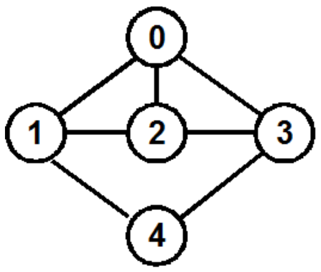


Figure 2: A graph with *k*=2

In the graph of Figure 2, removing the sets $\{0\}$, $\{1\}$, $\{2\}$, $\{3\}$, $\{4\}$, $\{0,1\}$, $\{0,2\}$, $\{0,3\}$, $\{0,4\}$, $\{1,2\}$, does not divide the graph into disconnected parts. But by removing the set $\{1,3\}$, node 4 disconnects from nodes 1 and 2. Because the size of removed set is 2 then the algorithm reports *k*=2 for this graph. Obviously to find the minimum subset that divides the graph, algorithm must remove the elements from power set in the ordered form according to the size of the subsets. For example all subsets with size 1 must be removed before the elements with size 2 and so on. The pseudo code of brute force algorithm is as follows:

```
Algorithm BruteForce (Graph G)
Begin
        ps=powerSet(G.V)       //generates the power set of  nodes of G
        sort(ps)               //sort ps according to the size of subsets
        foreach s in ps do
                T=remove(s,G)   //remove all nodes in s from G and store the resulting graph in T
                if  isConnected(T)=false then
                        return | s |
        return |G.V|
End
```

The *isConnected* function checks connectivity of the graph and returns true if all nodes in the graph is connected and false otherwise. The DFS[1] or BFS[2] algorithms can be used to implement the *isConnected* function. If we have a complete graph the algorithm returns the number of nodes in the

[1] Depth-First Search
[2] Breadth-First search

graph as the value of k, because removing the subsets in the power set does not divide the graph into disconnected partitions.

The brute force algorithm is a simple way to find the exact value of k in any graph. But obviously it is inefficient. The number of element in the power set of any set S is $2^{|S|}$. If we have a set of n items, to generate all subsets in the power set, any efficient power set generator algorithm will run at least with $O(2^n)$ time complexity, because the answer has 2^n items. Also the time complexity of BFS or DFS algorithms (to check the connectivity of the graph after each subset removal) is $O(n + |E|)$. Consequently in the worst case the time complexity of brute force algorithm is $O(2^n(n + |E|))$ and it is impossible to use this algorithm practically.

4.2.2 Network flow and connectivity testing

Network flow is one of the famous and widely used problems in graph theory. A flow network is a directed graph $G = (V, E)$ where each edge $e \in E$ has a non-negative capacity $c(e)$. The capacity *c(e)* is the upper bound for the amount of the flow that can pass from e. In a flow network the amount of incoming flow into a node $v \in V/\{source, sink\}$ equals to the amount of flow that goes out from *v*. The *source* and *sink* nodes are exceptions. The *source* vertex has only outgoing flow and the *sink* vertex has only incoming flow. In other words there is no incoming flow for the *source* and no outgoing flow from the *sink* in the network. A path with available capacity from *source* to *sink* is called an *augmenting path*.

In many applications it is required to find the maximum flow between two arbitrary pairs of *(s, t)* in the network. A simple greedy approach can be used to find the maximum flow between two nodes in a network. We can find a path with highest capacity from source to sink and use the maximum possible flow of that path and continue to find further paths until there is no capacity left between source and sink. But this approach may lead to non-optimal results. For example consider the graph in Figure 3. The goal is to send as much flow as possible from node *s* to node *t* with respect to the rules that total passed flow from each edge must be lower than or equal to the edge capacity, and for any node except *s* and *t*, the in and out going flows must be equal.

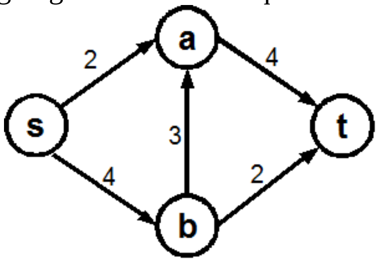


Figure 3: An example flow network

Applying the greedy idea on the flow network of Figure 3 has been illustrated in Figure 4. The first detected path can be $s \rightarrow b \rightarrow a \rightarrow t$ (Figure 4-a) with $flow = 3$ (the minimum capacity of edges in the path). In the second try we find $s \rightarrow a \rightarrow t$ with $flow = 1$ because 3 amount of $a \rightarrow t$ has been used in the previous path and only 1 amount is left for new path (Figure 4-b). In the third iteration we find the path $s \rightarrow b \rightarrow t$ with $flow = 1$ (Figure 4-c) and after this path there is no other path with positive flow between s and t. Therefore the maximum detected flow with greedy approach is 3+1+1=5, while the optimal value is 6. Selecting the paths $\rightarrow a \rightarrow t$, $s \rightarrow b \rightarrow t$ and $s \rightarrow b \rightarrow a \rightarrow t$ with $flow = 2$ for each path, provide optimal value 6 for the maximum flow in this network.

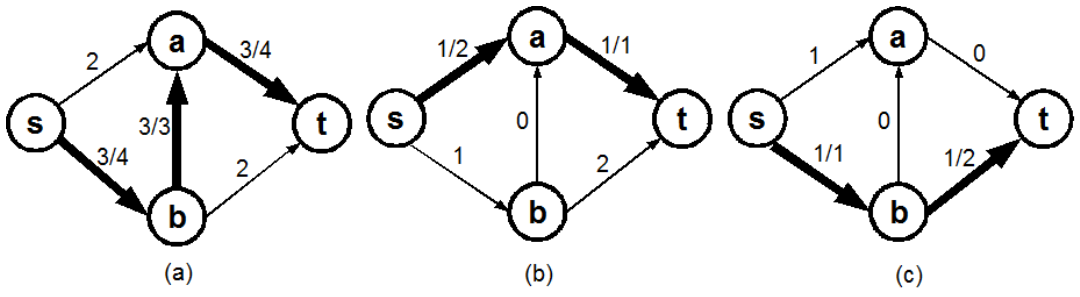


Figure 4: Steps in greedy algorithm to find the maximum flow

One of the first and efficient approaches that finds the maximum flow between two nodes has been provided by Ford–Fulkerson. The base idea of Ford–Fulkerson algorithm is similar to the greedy approaches: as long as there is a path from *s* to *t*, with unused capacity on all edges in the path, transfer maximum possible flow from that path and try to find another path. The key point of Ford-Fulkerson algorithm is *Residual Graph*.

Residual Graph of a flow network shows the additional possible augmenting paths in each step of Ford–Fulkerson algorithm. If there is an augmenting path from source to sink in residual graph, then it is possible to increase the current detected maximum flow. Every edge of a residual graph has a capacity value called residual capacity which initially is equal to original capacity of the edges in the flow network graph.

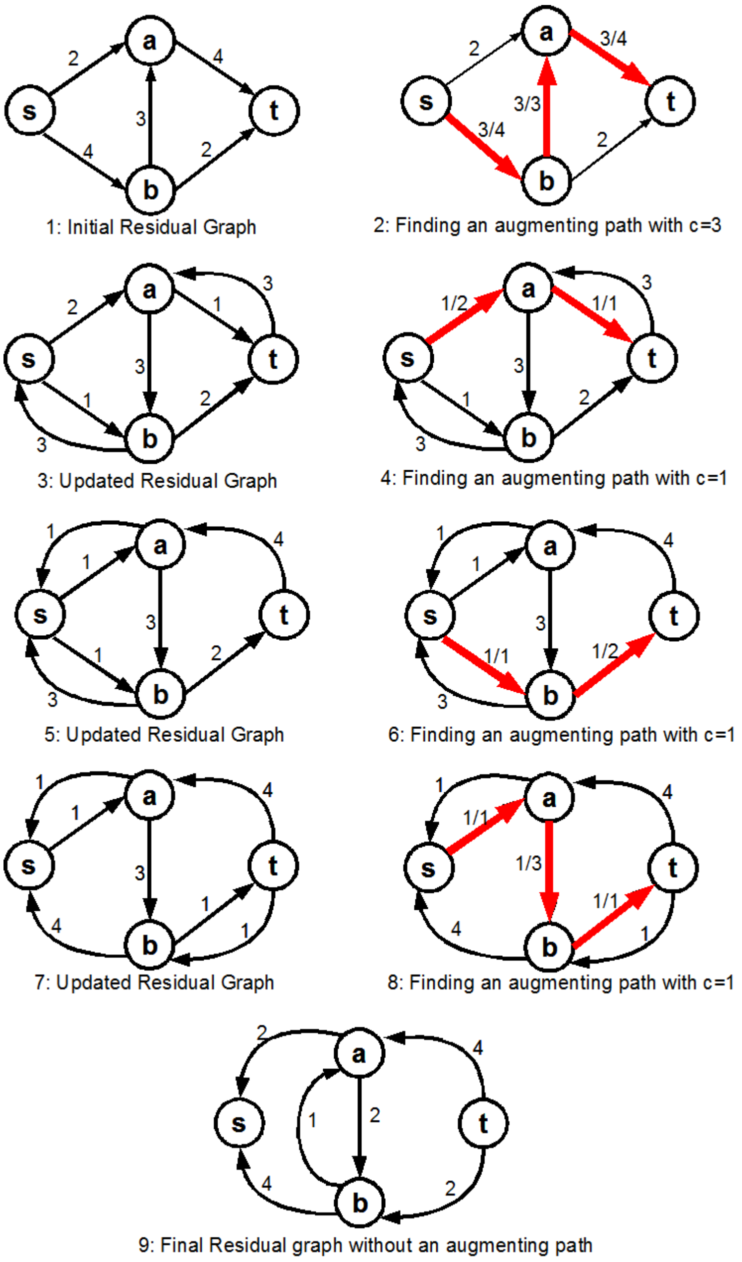

1: Initial Residual Graph
2: Finding an augmenting path with c=3
3: Updated Residual Graph
4: Finding an augmenting path with c=1
5: Updated Residual Graph
6: Finding an augmenting path with c=1
7: Updated Residual Graph
8: Finding an augmenting path with c=1
9: Final Residual graph without an augmenting path

Figure 5: Steps in Ford-Fulkerson algorithm to find the maximum flow

The Ford-Fulkerson algorithm continuously finds an augmenting path *p* between *s* and *t* in residual graph and updates it as follows:

- Find an augmenting path p between source and sink.
- For each $e \in p$ decrease the capacity of e by *c(p)* where *c(p)* is the amount of flow that can be sent using path p. Formally

$$\forall\, e \in p:\ \ c(e) = c(e) - c(p) \quad where \quad c(p) = \min(c(e))$$

- For each $e = (u, v) \in p$, add reverse edges $e' = (v, u)$ with the capacity *c(p)* to the residual graph.

Figure 5 shows the steps of Ford-Fulkerson algorithm on the graph of Figure 3. The residual graph initially equals to original graph. Suppose that the first detected augmenting path is $s \rightarrow b \rightarrow a \rightarrow t$ with $f = 3$. After sending the maximum possible flow from this path the residual graph is updated by adding reverse edges and decreasing the used flow from edges capacity as Step 3 in Figure 5. The next selected path is $s \rightarrow a \rightarrow t$ with *f*=1 which changes the residual graph as Step 5. The path $s \rightarrow b \rightarrow t$ is the next selected path to transfer 1 more flow from s to t. Selecting this path converts the residual graph to Step 7 in Figure 5. There is still another path $s \rightarrow a \rightarrow b \rightarrow t$ which increases the flow by 1. After using this path and updating the residual graph there is no more augmenting path left and the algorithm finds the maximum flow in this network as 3+1+1+1=6 which is optimum. The Ford-Fulkerson algorithm for maximum flow problem is as follows:

```
Algorithm Ford-Fulkerson (graph G, vertex s , vertex t )
Begin
        flow=0
          RG=G                    //create a copy of initial graph as residual graph
        while there is a path between s and t do
        begin
                  p=find_path(G,s,t)        //find a path from s to t in G
                  for each edge $e = (u, v) \in p$ do
                  begin
                            c(e)=c(e)- f(p)   //decrease the flow of path from capacity of edges on path
                            addEdge (G, v, u, f(p) )   //add a  reverse edge (v, u) with capacity f(p)
                            flow = flow+ f(p)
                  end
        end
        return flow
End
```

The original Ford–Fulkerson approach does not specify the way of finding augmenting paths in the residual graph (*find_paths* function in the above algorithm). Edmonds–Karp and Dinic completed Fold-Fulkerson approaches by using BFS algorithm to find new augmenting paths in each step. However to find the augmenting paths, it is possible to use either BFS or DFS algorithms.

The while loop in the above algorithm repeats until there is an augmenting path in the residual graph. By finding each augmenting path the *flow* increases at least by 1. In the worst case, the flow increases one unit in each iteration until it reaches to maximum possible flow. Therefore the while loop repeats at most max flow times. The number of repeats of for loop is equal to the number of edges in the detected path. Obviously the maximum possible edge count in the detected path is $|E|$. If we use the BFS or DFS algorithms to find next path in residual graph, the time complexity of *find_path* function will be $O(n + |E|)$. Consequently the time complexity of the Ford-Fulkerson algorithm is $O(maxFlow * (n + |E|)$. If we consider $O(|E|) = O(n^2)$ then the time complexity of this algorithm becomes $O(maxFlow * n^2)$.

It is possible to find the connectivity value of any graph using the maximum network flow algorithm (Even & Tarjan, 1975) (Galil, 1980). To do this we should create an initial residual graph G_R as follows:

- For any $v \in G$ add two vertices v_i and v_o to G_R
- Add a directed edge e from v_i to v_o with $c(e) = 1$.

- For any incoming edge $e = (u, v)$, add $e = (u, v_i)$ to G_R and set $c(e) = \infty$.
- For any outgoing edge $e = (v, u)$, add $e = (v_o, u)$ to G_R and set $c(e) = \infty$.

If the initial graph is undirected we can create two directed edge for each undirected edge between nodes. Figure 6 shows the Residual graph of the undirected graph of Figure 2. In Figure 6 the capacity of edges that have no label is infinite.

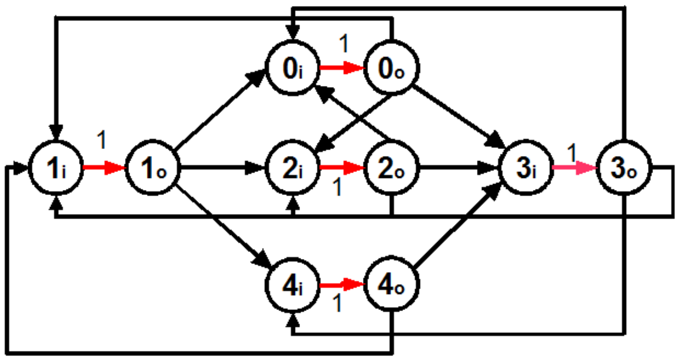


Figure 6: Residual graph of undirected graph of Figure 2

If we set the capacity of all inner edges (v_i, v_o) in the G_R to 1, at most one flow can passes over each v in G and the maximum flow between two arbitrary vertices in G_R indicates the number of disjoint paths between them in G because the flow capacity of each node in path is 1. In this case if we have *n* nodes in the graph, the maximum possible disjoint paths between two arbitrary nodes can be *n*-1 (in a complete graph each node has at most *n*-1 neighbors). Hence if the capacity of all nodes is 1 then the maximum possible flow between every pairs of nodes is *n*-1 and the time complexity of Ford-Fulkerson algorithm will be $O(n * n^2) = O(n^3)$. To find the value of *k* in a graph we should find the disjoint path count between all pairs of nodes and select the smallest value as *k*. In a graph with *n* nodes there is $\frac{n(n-1)}{2}$ possible pairing among the nodes which leads to $O(n^2)$ time complexity. Consequently the complexity of finding the value of *k* using network flow algorithm is $O(n^2 * n^3) = O(n^5)$. The algorithm is as follows:

```
Algorithm findConnectivityValue (Graph G )   //G=(V,E)
Begin
        flow=0
        k=|V|
        GR=createResidualGraph(G) //create initial residual graph
        for each vertex v∈V do                  //generate all possible pairings among node
              for each vertex u∈V do
                      if  v ≠ u  then
                      begin
                              flow=Ford-Folkerson(GR, v, u)   //  find max flow between u and v
                              if  flow < k  then     //select minimum detected flow between nodes
                                      k = flow
                      end
        return  k
End
```

The above algorithm is base of several other connectivity detection algorithms that try to find the value of *k* faster. Various improvements have been made on the above algorithm which results significant reduction in time complexity. Even proposed an improved version of the above algorithm which finds the connectivity value of *G* with $O(|V|^{\frac{1}{2}}|E|^2)$ time complexity (Even & Tarjan, 1975). Gomory and Hu proved that it is possible to solve the multi terminal maximum network flow problem by considering only n-1 pairs of nodes instead of $\frac{n(n-1)}{2}$ possible pairs. Therefore the time complexity of above algorithm can be reduced to $O(n^4)$. Kleitman has improved the above algorithm and provided a method which can run in $O(k^2n^3)$ time complexity. Obviously we always have $k \leq n$ and Kleitman 's algorithm runs faster than Gomory's algorithm if $k \leq \sqrt{n}$.

Also Even provided another algorithm that accepts a graph G and a connectivity value k and checks whether G is at least k-connected or not (Even, 1975). This algorithm selects a random subset $S \subseteq V$ where $|S|=k$. Then the algorithm runs Ford-Fulkerson algorithm to check whether there are at least k disjoint paths between all pairs $(u, v) \in S$ or not. If there is a pair $(x, y) \in S$ which the number of disjoint paths between them is smaller than k then the algorithm finishes immediately and returns false. The algorithm starts second phase if all nodes in S have at least k disjoint paths to each other. In the second phase a graph G' is created from G by adding a single vertex a which is connected to all nodes $u \in S$. Obviously each $u \in S$ has k disjoint paths to a because a connected to all nodes in S and every node in S has k disjoint paths to other nodes. The most important property of node a is that if any node $u \in V - S$ has k disjoint paths to a then it has k disjoint paths to each node $v \in S$ because a connected to other nodes only with k vertices in S and every node that has k disjoint paths to a must use one of the nodes in S. We can conclude that if each node $u \in V - S$ has k disjoint paths to a then it has a disjoint path to each node in $v \in S$ and because S is k-connected then each node $u \in V$ has k disjoint paths to all other nodes and G is k-connected. Figure 7 shows the main steps of Even algorithm for testing k=3.

In Figure 7-a we see the initial graph. Suppose that the algorithm is called with k=3 to check whether the graph is at least 3-connected or not. Hence a set with 3 vertices is selected. Suppose the selected vertices are the nodes 3,5 and 7 (Figure 7-b). In the next step the algorithm checks that the selected vertices have at least k disjoint paths to each other (Figure 7-c). If all nodes in S have k disjoint paths to each other, the algorithm starts second phase and adds a dummy vertex a to graph G and connects it to all nodes in S. After that if each vertices in V-S has k disjoint paths to a the algorithms returns true otherwise it return false.

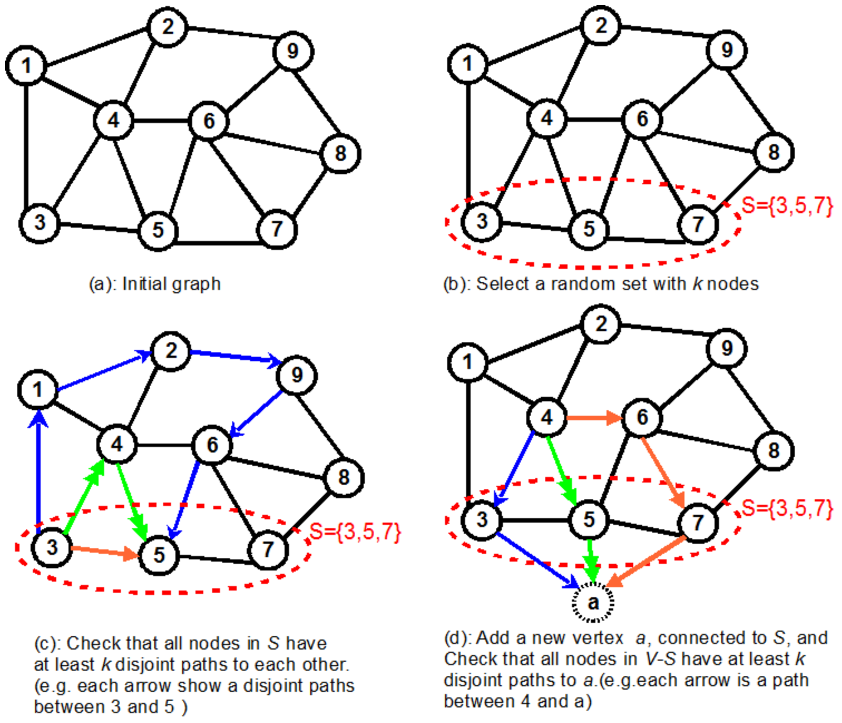

(a): Initial graph
(b): Select a random set with k nodes
S={3,5,7}
(c): Check that all nodes in S have at least k disjoint paths to each other. (e.g. each arrow show a disjoint paths between 3 and 5)
(d): Add a new vertex a, connected to S, and Check that all nodes in V-S have at least k disjoint paths to a.(e.g.each arrow is a path between 4 and a)
1
2
3
4
5
6
7
8
9
a

Figure 7: Main steps in Even Algorithm for checking the graph connectivity

To pseudo code of this algorithm is as follows:

```
Algorithm checkConnectivity (Graph G, k )                //G=(V,E)
Begin
        //step 1: create and test S
        GR=createResidualGraph(G)  //create initial residual graph
        for i=1 to k  do                    //select first k node from V as S
                for j=i+1 to k  do
                begin
                        //find the disjoint  path count between V[i] and V[j]
                        c= Ford-Fulkerson( GR, G.V[i],G.V[j] )
                        if c < k then    // finish if there is a pair with lower disjoint paths than k
                                return false
                end

        //step 2 add dummy node a to G and check the other nodes
        $G' = ( G.V \cup a, G.E)$
        for i=1 to k  do
                connect ($G'.V[i]$ , a)
        GR=createResidualGraph($G'$)//create residual graph for $G'$

        for i=k+1 to |G.V|  do
                begin
                        //find disjoint path count between V[i] and a
                        c= Ford-Fulkerson(GR, $G'.V[i]$, a )
                        if c < k then    // finish if there is a pair with lower disjoint paths than k
                                return false
                end
        return  true
End
```

It is mentioned before that the Ford-Fulkerson algorithm finds the disjoint path count between two vertices with $O(maxFlow * n^2)$ time complexity. Here *maxFlow* is at most *k* and we have two nested for loops which call Ford-Fulkerson algorithm. Hence the time complexity of step one is $(k^2 * k * n^2) = O(k^3n^2)$. Also the for loop in step 2 repeats at most n times and each time calls Ford-Fulkerson algorithm with $O(kn^2)$ complexity. Hence the time complexity of step 2 is $O(kn^3)$ and the total time complexity of algorithm is $O(kn^3 + k^3n^2)$. If we suppose $k < \sqrt{n}$ then we have:

$$k < \sqrt{n} \rightarrow \quad k^2 < n \quad \rightarrow \quad k^3 < kn \quad \rightarrow \quad n^2k^3 < kn^3$$

Therefore for $k \leq \sqrt{n}$ the time complexity of algorithm is $O(kn^3)$. However this algorithm needs an initial value for *k* and it cannot find the connectivity value of the graph. Central algorithms cannot be directly implemented in a distributed system. To apply a central *k*-connectivity detection algorithm on a distributed system, we must collect all information about the topology of the network in a single node which needs huge amount of data transmission in the network. In the other hand, finding the exact value of *k* using local information in a distributed system is a hard task.

4.3 Distributed algorithms

In a distributed system the nodes in the network not only may have no information about the general topology but also they may run asynchronously without any clock synchronization. These factors (lack of global clock and global knowledge of the network topology) make the design and analysis of distributed algorithms harder than the central algorithms. Meanwhile the *k*-connectivity is one of the complicated problems in graph theory and become more complicated in distributed environment. To solve this problem in a distributed system, all nodes should be able to start a distributed algorithm and find the value of *k* using local information or some message exchanges with other nodes. Currently a distributed algorithm that can find the exact value of k in a network has not provided yet and in all proposed algorithms the nodes try to estimate k according to local information.

4.3.1 Three localized distributed algorithms

Jorgic presented three localized distributed algorithms to estimate the k-connectivity in a given network (Jorgic, et. al., 2007). In these algorithms the value of k is estimated using local neighborhood information. In the first algorithm named LND[3], each node finds the number of its neighbors and broadcasts it to p-hop away where p can be 1, 2 or 3. Therefore, each node learns the degree of its p-hop neighbors and sets the k to the minimum visited degree. In the start of algorithm each node broadcast a *Hello* message to find its degree. After finding local d each node broadcasts the d for p hop away and updates k each time it receives lower value for d from other nodes. The LND algorithm is as follows:

```
Algorithm LND
    Broadcast a Hello message
    Find the local degree d by counting the number of received Hello messages.
    Broadcast d to3-hop away by setting the message ttl to 3
    Set k to the minimum of local d and received degrees from other nodes,
End.
```

In the LND algorithm each node broadcasts exactly one *Hello* message and at most *p-hop* degree messages. Hence the upper bound of the total broadcasted messages in the entire network is *n*(1+p-hop)*. The value of *p-hop* determines the level of locality in the algorithm. Higher values for *p-hop* provide to increase the accuracy of estimation but causes to transmit more messages in the network. If we consider *p-hop*=3, the number of exchanged messages in the network will become *4n* messages. Each message carries a single number indicating the degree of a node. The maximum possible value for degree in a graph is Δ so the size of each message is $log_{2}(\Delta)$ bits. Therefore the total bits of exchanged messages in LND are $O(nlog_{2}(\Delta))$ which make LND an applicable approach from performance perspective. But in term of correctness, LND is not a favorable approach. The value of k in any graph is at most equal to the minimum degree of the graph. Hence even if the minimum degree of the graph reaches to all nodes after *p-hop* broadcast, the probability of correct estimation is very low. Generally most of the times k is smaller than d and the detected value by LND is generally higher than the real k. In Figure 1, for example, the minimum degree of the graph is 3 while k is 2.

The second proposed algorithm by Jorgic is Local Subgraph Connectivity Detection (LSCD). In this algorithm each node finds its one hop neighbor list, by broadcasting a *Hello* message. In the second phase, each node broadcasts its one hop neighbor list to *p-hop* away. In this way each node learns its *p-hop* local sub-graph. Finally the nodes find the connectivity value of this sub-graph using a central algorithm and accept the result as an estimation of global k. The pseudo code of LSCD algorithm is as follows:

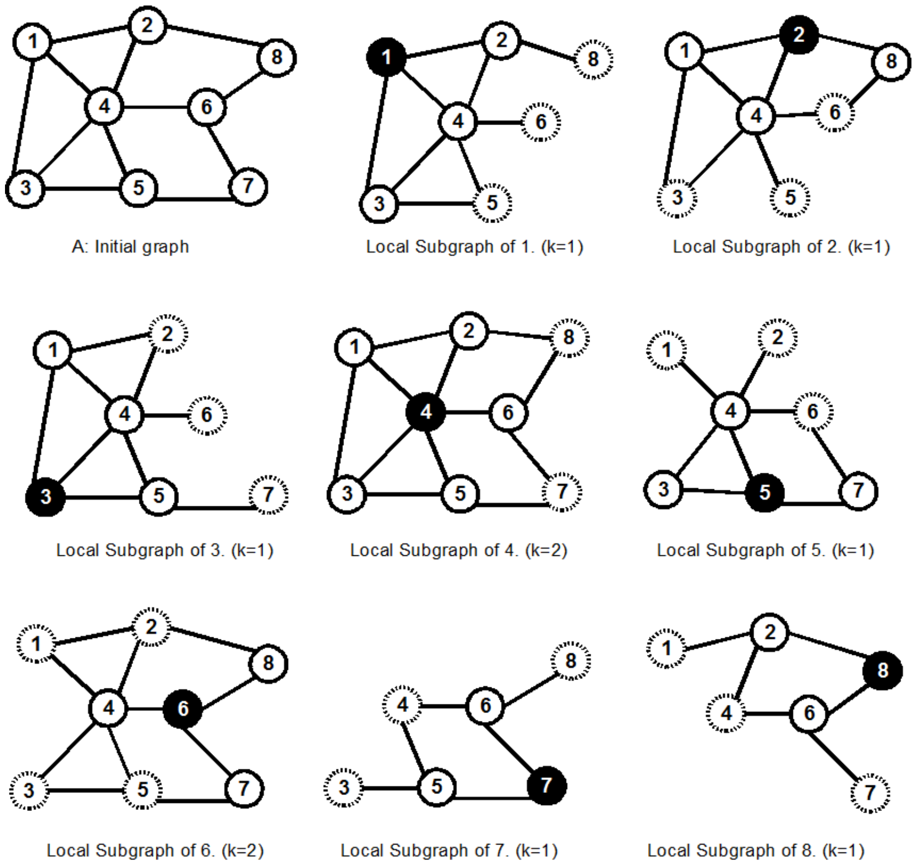
```
Algorithm LSCD
    Find one hop neighbor list by broadcasting Hello messages.
    Construct p-hop local subgraph $G^p$ by exchanging one hope neighbor list.
    Run a central algorithm to find the connectivity value of $G^p$.
    Set k to the connectivity value of $G^p$.
End
```

In LSCD algorithms all nodes need to exchange one-hop neighbor lists. Each item in the neighbor list is identifier of a node and consumes $log_{2}(n)$ bits in the message. The maximum possible neighbor count for each node is Δ. Hence the maximum size of each message is $\Delta log_2(n)$. Considering n nodes in the network which send one hop lists for the first time, the total exchanged bits in the network is $n\Delta log_{2}(n)$ bits. Each message is rebroadcasted by the neighbors of the sender,

[3] Local Neighborhood Detection

p-hop times. If we let *p-hop*=3, the total size of exchanged bits in the network will be $O(n\Delta^3 log_2(n))$.

Figure 8 shows an example graph and the *2-hop* local subgraphs of each node in LSCD algorithm. In this figure the black nodes are root nodes that run the algorithm. The one hop neighbors of root nodes are shown with solid circles and two hop neighbors are distinguished by dashed circles. The correct *k* value in the initial graph is 2, but only two nodes find the correct value of *k*.


A: Initial graph
Local Subgraph of 1. (k=1)
Local Subgraph of 2. (k=1)
Local Subgraph of 3. (k=1)
Local Subgraph of 4. (k=2)
Local Subgraph of 5. (k=1)
Local Subgraph of 6. (k=2)
Local Subgraph of 7. (k=1)
Local Subgraph of 8. (k=1)
Figure 8: An example graph and its 2-hop local subgraphs

Except the local subgraphs of nodes 4 and 6, all other subgraphs include cut edges and central algorithm running on them find *k*=1. Nodes 4 and 6 find *k*=2 which is correct. Generally the probability of having a cut edge in local subgraph of each nodes is very high and hence most of the nodes find *k*=1 in their local subgraphs.

The third algorithm is Local Critical Node Detection (LCND) which is very similar to LSCD. This algorithm exchange the neighbor list information just like LSCD and create local subgraphs but try to finds cut vertices in the subgraph by removing the nodes one by one. The connectivity value of a graph that has cut vertex is 1 and there is no need to call a central *k* detection algorithm in such a graph. If there is no cut vertex in subgraph then the local subgraph is 2 connected and node remove itself from graph and recall the cut vertex detection algorithm. If the resulting graph has no cut vertex then *k* is 3. This process continues until finding a cut vertex in the local subgraph. After removing each node from local subgraph the nodes increase *k* by one. Although this approach may provide a different detection method for *k*, but it still has same problems as LSCD and exchange large amount of messages in the network without providing a correct estimation for *k*.

4.3.2 Safe Algorithms

Cornejo and Lynch provided two distributed algorithms for *k*-connectivity on unit disk graphs (Cornejo & Lynch, 2010) Both algorithms are safe and need an initial value for *k* to test whether the topology of the network is at least *k*-connected or not. An algorithm is safe which never estimate higher values for *k* than the real connectivity value. They proposed a Natural Local Test (NLT) for *k* connectivity as first algorithm which is very similar to LSCD algorithm. In NLT algorithm each node finds the local subgraph of the network and launches a central algorithm to accept or reject the provided k for the algorithm. If all nodes accept the given value, it is accepted as general k. The pseudo code of NLT algorithm is as follows:

```
Algorithm Natural Local Test (k)
Begin
    Construct t hop local subgraph $G^t$ by exchanging one hop neighbor list.
    Run a central algorithm to find $k(G^t)$, the connectivity value of $G^t$.
    if $k(G^t) < k$ then
        accept
    else
        reject
End
```

In the above algorithm the network is considered as k-connected if all nodes accept the given value for k. The local subgraph construction can be done in the same way as LSCD algorithm and the bit complexity for both algorithms are same. The NLT algorithm needs a consensus mechanism among the nodes. All nodes must learn about the other nodes decisions to know whether the current k is accepted by all nodes or not. This implies a communication overhead and increases the bit complexity of algorithm. This algorithm needs an initial value for k to be checked on the graph. If we try to find an estimated value for k using NLT we need to run algorithm with incremental inputs until the algorithm rejects a value for k which may transmit huge amount of data in the network. However NLT is a safe algorithm. It never estimates higher value for k than the real connectivity value. In other words if all nodes accept the input value x, then the real k in the network is equal or higher than x.

The second proposed approach by Cornejo is k-connectivity with small edges (CWSE) algorithm. In this algorithm each node finds its local subgraph and eliminates the edges that their distance is longer than $\frac{1}{k}$ where k is the input value for the algorithm. If the graph remains connected after elimination and has at least k+1 nodes, the node accepts, otherwise rejects the provided value for k. The idea behind this algorithm is that in a dense network the probability of having a k-connected topology is higher than a sparse network. In a dense network the distance between nodes may be very smaller than sparse network. Hence in a k-connected dense topology with high probability each node has a connected edge with length smaller than $\frac{1}{k}$. The pseudo code of CWSE algorithm is as follows:

```
Algorithm CWSE ( k )
Begin
    Construct t hop local subgraph $G^t$ by exchanging one hope neighbor list.
    Remove all edges with length longer $\frac{1}{k}$ from $G^t$.
    if $G^t$ is connected and $N(G^t) > k+1$ then
        accept
    else
        reject
End
```

Figure 9 shows an example network and the execution steps of CWSE algorithm in node 7. The 2-hop local subgraph of node 7 has been presented in Figure 9-b. After creating local subgraph all edges that are longer than $\frac{1}{k}$ must be removed from this graph. The resulting graph after removing these edges is Figure 9-c and the node 7 incorrectly rejects k=3. However for k=2 all nodes accept the given value because every node has at least one connected edge which its length is smaller than 0.5.

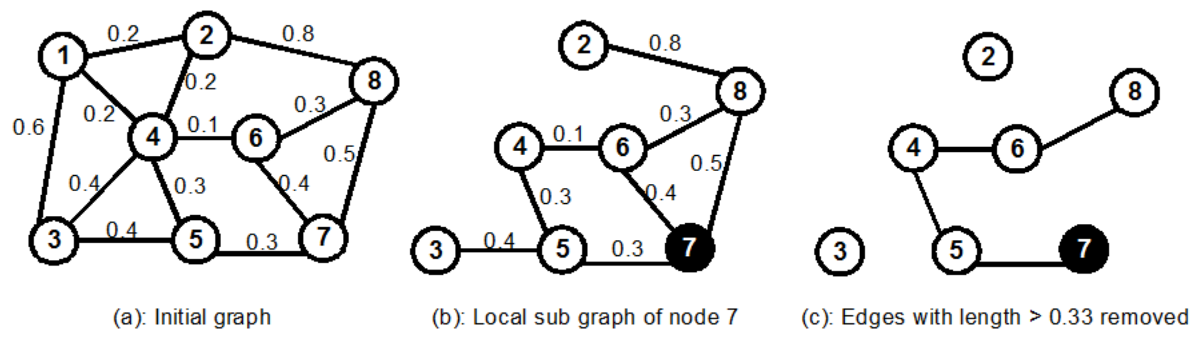


Figure 9: The execution steps of CWSE in node 7 for k=3

In the above algorithm $N(G^t)$ is the number of nodes in G^t. The bit complexity of this algorithm is exactly same as LSCD. Also just like the NLT, this algorithm needs consensus between nodes to flood all accept or reject decisions in the network. The algorithm is safe and never accepts a value higher than the real k. It cannot find the connectivity value and needs an initial guess about the k to check the graph for that connectivity. Unlike the NLT or LSCD algorithms, CWSE uses two heuristics with lower time complexity instead of calling a central connectivity algorithm to decide about the given k, and this is the most important aspect which differs CWSE from previous algorithms.

4.3.3 Distributed k-Connectivity Maintenance

Distributed k-Connectivity Maintenance (DKM) is another distributed algorithm which has two phases; detection and restoration (Szczytowski, et. al., 2012). In this algorithm each node estimates its disjoint path count to sink. This algorithm assumes that sink is a special reliable node which never stops working. If each node has k disjoint paths to sink then all nodes have k disjoint paths to each other and the network is k-connected. The hop count between sink and each other node is used to estimate the number of disjoint paths between sink and that node. To estimate the number of disjoint paths to sink all nodes find their Support Node Set (SNS). The support node set of node v is the set of one and two hop neighbors of node v which are closer to sink. For example in Figure 10-a if we suppose the sink is node 0 then the SNS(3)= {2,1}, SNS(4)=SNS(6)={3} and SNS(5)={4,6}. The support node set also includes the nodes that are in two hop neighbor list and connected by a node which have same hop number from the sink. For example, in Figure 10-b, SNS(5)={4,6,3} because node 6 has smaller hop value than node 5 and node 4 connected to node 3 with smaller hop value than node 5.

The size of $SNS(v)$ is considered as local k estimation in node v. For example in Figure 10-a nodes 4 and 6 find k=1 and node 5 finds k=2. In Figure 10-b nodes 4 and 5 find incorrect k=3. The pseudo code of DKM algorithm is as follows:

```
Algorithm DKMestimateLocalConnectivity (curNode )
Begin
        SNS = Ø;
        for all u in oneHopNeighbors do
                if u.hop < curNode.hop then
                        SNS ← SNS ∪ u
                else if u.hop = curNode.hop then
                        for all v  in twoHopNeighbors do
                                if v.hop < curNode.hop and v ∉ oneHopNeighbors then
                                        SNS ← SNS ∪ v
                        closerNeigh ← closerNeigh ∪ u
       return |SNS|
End
```

In the DKM algorithm each node must find and broadcast its one hop neighbor list. The one hop neighbor lists are rebroadcasted with other nodes one more time. Therefore we have $O(n\Delta^2 log_2(n))$

bit complexity. DKM provides an approximate value for the number of disjoint paths between each node and sink. It is quite possible that the algorithm overestimates or underestimates the disjoint path count.

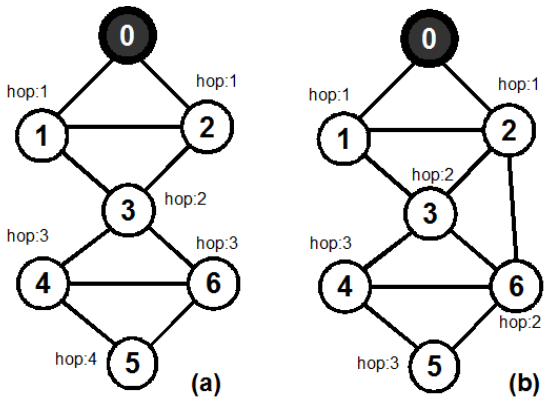


Figure 10: Two sample topologies

5 The k-connectivity restoration

Developing a strategy for connectivity restoration after or before a failure is one of the essential and unavoidable task in most of the networks. The connectivity restoration in a distributed system is the process of automatically detecting and resolving the connection lost between the nodes in the network. A k-connectivity restoration is the process of increasing or restoring the links in the network such that the resulting topology becomes k-connected. There are many algorithms that restore 1-connectivity between active nodes in the network. But only a few approaches have been proposed for k-connectivity restoration.

A k-connectivity restoration algorithm should be able to increase the connectivity of a network to at least k. Placing or activating new nodes and moving mobile nodes to new locations are two common way to restore the connectivity of a network. Both methods have some advantages and disadvantages. Placing or activating new nodes, need to have redundant nodes in the network. In the other hand, moving a node to a new position is an energy consuming and complicated task which can terminate some existence links.

5.1 Central Algorithm

Node failure in a k-connected network can pull down the k-connectivity and reduce the value of k. In a k-connected network, if all nodes work in their predefined positions the network remains k-connected. From another view if we only consider the location of nodes in a k-connected network, as long as we have working nodes in that locations the network is k-connected. Failure in some locations may not affect the connectivity value of the network but it is possible that a failure in a node which converts one of these locations to an empty slot reduce the value of k. So after each failure we have one of the following situations:

1. Working nodes located on remaining locations form a k-connected network.
2. Working nodes located on remaining locations do not form a k-connected network.

In case 1 we are lucky and the stopped node has no effect on connectivity value. In this case there is no need to do any restoration activity. In case 2 we should select and move a working node from its original location to the failed node location. Obviously we must select a node that its vacation does not affect the k-connectivity. We call the location of this node as safe location. Also to achieve the optimality we must select a node from a safe location which its moving consume lowest energy in the entire network. Wang (Wang et. al., 2011) proposed to use the maximum weighted matching algorithm on a bi-partite graph of active nodes and default locations of nodes to achieve an optimum solution. Let P be the set of default positions of all nodes, V be the set of all nodes, v_f be the failed node and p_f be the position of failed node. The Wang's algorithm first checks whether the resulting graphs $H = P/p_f$ is k-connected or not. If H is k-connected then the algorithm finishes immediately. Otherwise the algorithm removes one of the positions $p_i \in P$ from P and checks whether the resulting graph of P/p_i is k-connected and then runs a maximum weighted matching on a bi-partite graph of V/v_f and P/p_i. This process repeats for every $p_i \in P$ and finally the maximum weighted matching which produces a k-connected graph is selected as result. Obviously in selected matching the failed node position p_f has been matched with another position p_j which means that the node in p_j must moves to new location p_f. The weight of edges of bi-partite graph must have reverse relation with the cost of moving between locations.

This central algorithm provides an optimum solution for the k-restoration problem. The time complexity of best central algorithm that can check the connectivity of a network with predefined candidate for k is $O(kn^3)$. Also the maximum weighted matching algorithm runs on $O((2n)^3)$ time complexity (we have $2n$ vertices in bi-partite graph). Hence each iteration of testing and matching has $O((2n)^3 + kn^3) = O(kn^3)$ time complexity. Repeating this computation for all locations in the graph leads to $O(kn^4)$ time complexity for Wang's algorithm.

5.2 Distributed Algorithm

The only proposed distributed algorithm for k-connectivity restoration, that supports both detection and restoration phases is DKM. In the restoration phase, DKM algorithm increases the number of active resources in the nodes that estimate lower value for k than the desired connectivity. DKM assumes that there is unlimited number of resources (nodes) in the location of each node and every node can increase active resources to desired values to achieve the k-connectivity. In the restoration phase, the algorithm supposes that each node knows the desired value for *k*. Having this value, every node computes a *vote* value as follows:

$$vote = k_{desired} - k_{estimated}$$

Then each node v that has a positive $vote$ sends this value to the nodes which are in $SNS(v)$. Each node in the network that receives at least one *vote* from their neighbors, calculates a response to the votes as follows:

$$responce.value = \sum_{i \in V} vote_i$$

$$responce.minVote = \min(vote_i)$$

$vote_i$ is the $i'th$ received vote by the node. In this way every node that receives some votes, collects them and broadcasts a response that includes the sum up of all votes and also the minimum received votes.

After receiving response messages, node v selects node u which is the sender of biggest $responce.value$ and sends a *select* message to u. In this way, node u learns that it must activate at least $responce_u.minVote$ resources to increase the connectivity of at least one neighbor to *k*. The nodes that receive more than one *select* message for a *response*, just take into account one of them and ignore the others. In other words, if a node receives more than one *select* for a single *response* it just actives $minVote$ number of its resources and waits for next *vote-response-select* cycle. The algorithm finishes if all nodes have at least *k* nodes in their support node set and do node broadcast any *vote*.

```
Algorithm DKMresolveNode (curNode, $k_{desired}$)
k=estimateLocalConnectivity(curNode)
while k < $k_{desired}$ do
begin
    vote = $k_{desired}$ –k
    for all v in SNS do
        sendVote(v, vote);
    responses= collectResponses( )
    maxResponse=0
    for each response r in  responses do
        if r.sender.hop < curNode.hop and response.value > maxResponse then
            maxResponse = response;
    sendSelection ( maxResponse.sender )
    k = estimateLocalConnectivity ( curNode )
end
```

In the above algorithm each node estimates a local value for *k* and repeats a while loop until the estimated *k* equals to desired *k* value. In the while loop a *vote* is calculated and sent to all nodes in SNS and *responses* are collected from them. Then the sender of highest *response* is selected and a *select* message is sent to that node. Finally the estimation process is called again to find the new connectivity value. To send appropriate response messages, each node runs the following algorithm.

```
Algorithm  voterResponce ( curNode , k )
begin
    votes = collectVotes( )
     response.value=0
    response.minVote = max
    for all vote in votes do
         response.value += vote;
        response.minVote = min(response.minVote, vote);
    response.sender = curNode;
    broadCastResponse(vote.sender, response)
    s=collectSelections( maxDelay )
    if |s|>0 then
        active more  response.minVote  resources
end
```

In the above algorithm each node collects votes and calculates the total and minimum received votes. Then a *response* message which includes the total and minimum of votes is broadcasted. After that the node waits for *maxDelay* time to receive select messages. If a *select* message arrives, the node activates more *minVote* resources. Activating these resources increases the number of connectivity of another node to *k*. This process continues until all nodes reaches to *k*-connectivity. The main drawback of restoration phase in DKM algorithm is the assumption about the unlimited number of resources in each node.

In another algorithm (Atay & Bayazit, 2010) introduced a new metric, named *k-redundancy*, which provides a criterion to identify critical parts of a network. The *k-redundancy* of node *v* is the minimum number of nodes that their removal disconnects two neighbors of *v*. This provides a measure to represent the importance of a node in keeping the connectivity of graph. With this definition the cut vertices in the topology are 0-redundant. It means that there are no supporting nodes for cut vertices in the network and if we lose a cut vertex the graph is divided to disconnected partitions. If we lose a 1-redundnunt node there is still one node that keeps the connectivity of all nodes in the network. In this way a 2-redundunt node has two supporting nodes and there is a reverse relation between redundancy value and the importance of node. With increasing the redundancy value of each node its importance from connectivity perspective decreases because the failure of that node can be easily tolerated by redundant nodes. Figure 11 shows a sample graph and the *k*-redundancy value of each node.

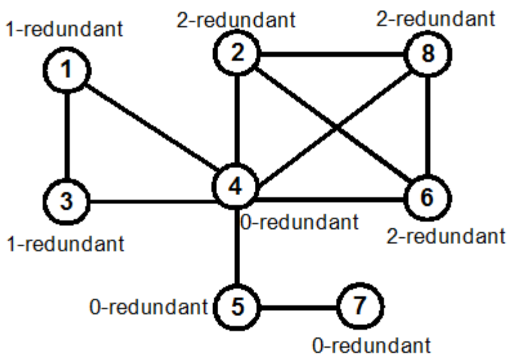


Figure 11: *k*-redundancy of nodes in a sample graph

The authors provide two approaches, to maintain and repair the connectivity using *k*-redundancy information. The proposed methods do not support *k*-connectivity detection and restoration and only restore a disconnected network to 1-connected form. However using *k*-redundancy as a metric of importance of a node in connectivity is another application of *k*-connectivity which shows the importance of this issue in the networks.

5 Conclusion

The connectivity robustness is one of the most essential properties in every network. The *k*-connectivity is an appropriate property to measure the connectivity robustness of any network. Especially in wireless sensor and ad hoc networks that intermediate nodes is used to deliver the packets, 1-connectivity put the network in high disconnection risk. For this reason, finding the current connectivity value of the network and increasing it to a desired value is one of the fundamental tasks which need efficient algorithms to reduce the communication and power consumption overhead. In this chapter the proposed central and distributed algorithms for both *k*-connectivity detection and *k*-connectivity restoration were discussed. Generally the central algorithms use network flow algorithm to find the connectivity value. Currently the best central algorithm can find the value of k with $O(n^4)$ time complexity. Distributed algorithms try to estimate the value of k using some local information such as nodes degree or one hop and two hop neighbor lists. Generally these estimations are not equal to the real k value.

To restore the connectivity of a network to a desired level of k, there are two major approaches: moving other active nodes and placing new nodes in the network. Adding new nodes is not possible in all environments. In the other hand moving other active nodes is a very complicated task which can produce new link failure in the graph. There is a central algorithm which use maximum matching algorithm to determine the best moving for optimal result. The time complexity of this algorithm is $O(kn^4)$. A distributed algorithm named DKM estimates the current value of k in the network and increases it to desired value by activating more resources in each node.

The main drawback of central algorithms is the need of gathering the entire graph topology in a single node which needs huge amount of data exchange in large networks. In the other hand distributed algorithms can only estimate the connectivity value of the network using local information. This estimation may provide wrong results that are far away from the correct answer. These deficiencies make the *k*-connectivity detection and restoration topic as an attractive open problem which are considered by many researchers.

References

Almasaeid, H. M., & Kamal, A. E. (2009, June). On the minimum *k*-connectivity repair in wireless sensor networks. In IEEE International Conference on Communications, 2009. ICC'09. (pp. 1-5). IEEE.

Atay, *N*., & Bayazit, B. (2010). Mobile wireless sensor network connectivity repair with *k*-redundancy. In Algorithmic Foundation of Robotics VIII (pp. 35-49). Springer Berlin Heidelberg.

Bai, X., Xuan, D., Yun, Z., Lai, T. H., & Jia, W. (2008, May). Complete optimal deployment patterns for full-coverage and *k*-connectivity ($k \leq 6$) wireless sensor networks. In Proceedings of the 9th ACM international symposium on Mobile ad hoc networking and computing (pp. 401-410). ACM.

Bettstetter, C. (2002, June). On the minimum node degree and connectivity of a wireless multihop network. In Proceedings of the 3rd ACM international symposium on Mobile ad hoc networking & computing (pp. 80-91). ACM.

Cornejo, A., & Lynch, *N*. (2010). Fault-tolerance through *k*-connectivity. In Workshop on Network Science and Systems Issues in Multi-Robot Autonomy: ICRA (Vol. 2, *p*. 2010).

Dagdeviren, O., & Akram, V. *K*. (2014). An Energy-Efficient Distributed Cut Vertex Detection Algorithm for Wireless Sensor Networks. The Computer Journal, 57(12), 1852-1869.

Even, S. (1975). An algorithm for determining whether the connectivity of a graph is at least *k*. SIAM Journal on Computing, 4(3), 393-396.

Even, S., & Tarjan, R. E. (1975). Network flow and testing graph connectivity. SIAM journal on computing, 4(4), 507-518.

Galil, Z. (1980). Finding the vertex connectivity of graphs. SIAM Journal on Computing, 9(1), 197-199.

Gupta, B., & Gupta, A. (2013, March). On the *k*-Connectivity of Ad-Hoc Wireless Networks. In IEEE 7th International Symposium on Service Oriented System Engineering (SOSE), 2013 (pp. 546-550).

Henzinger, M. R., Rao, S., & Gabow, H. *N*. (2000). Computing vertex connectivity: New bounds from old techniques. Journal of Algorithms, 34(2), 222-250.

Jorgic, M., Goel, *N*., Kalaichevan, *K*. A. L. A. I., Nayak, A., & Stojmenovic, I. (2007, August). Localized detection of *k*-connectivity in wireless ad hoc, actuator and sensor networks. In Proceedings of 16th International Conference on Computer Communications and Networks, 2007. ICCCN 2007. (pp. 33-38). IEEE.

Kallollu Narayanaswamy, *N*., Satyanarayana, D., & Prasad, M. *G*. (2014). An Analytical Expression for *k*-connectivity of Wireless Ad Hoc Networks. TELKOMNIKA (Telecommunication Computing Electronics and Control), 12(1), 179-188.

Ling, Q., & Tian, Z. (2007, November). Minimum node degree and *k*-connectivity of a wireless multihop network in bounded area. In Global Telecommunications Conference, 2007. GLOBECOM'07. IEEE (pp. 1296-1301). IEEE.

Meghanathan, *N*., & Gorla, S. (2010). On the Probability of *K*-Connectivity in wireless Ad Hoc networks under Different mobility models. International Journal on Applications of Graph Theory in Wireless Ad Hoc Networks and Sensor Networks (GRAPH-HOC), 2(3).

Nutov, Z. (2008). Approximating minimum-power *k*-connectivity. In Ad-hoc, Mobile and Wireless Networks (pp. 86-93). Springer Berlin Heidelberg.

Penrose, M. D. (1999). On *k*-connectivity for a geometric random graph. Random Structures & Algorithms, 15(2), 145-164.

Reif, J. H., & Spirakis, *P*. *G*. (1985). *k*-connectivity in random undirected graphs. Discrete mathematics, 54(2), 181-191.

Szczytowski, *P*., Khelil, A., & Suri, *N*. (2012, January). DKM: Distributed *k*-connectivity maintenance in 9th Annual Conference on Wireless Sensor Networks. In Wireless On-demand Network Systems and Services (WONS), 2012 (pp. 83-90). IEEE.

Tarjan, R. Depth-first search and linear graph algorithms. SIAM Journal on Computing, 1, 146–160, 1972.

Wang, S., Mao, X., Tang, S. J., Li, M., Zhao, J., & Dai, *G*. (2011). On “Movement-Assisted Connectivity Restoration in Wireless Sensor and Actor Networks”. Parallel and Distributed Systems, IEEE Transactions on, 22(4), 687-694.

Xing, X., Wang, *G*., Wu, J., & Li, J. (2009, November). Square region-based coverage and connectivity probability model in wireless sensor networks. In Collaborative Computing: Networking, Applications and Worksharing, 2009. CollaborateCom 2009. 5th International Conference on (pp. 1-8). IEEE.

Younis, M., Senturk, I. F., Akkaya, *K*., Lee, S., & Senel, F. (2014). Topology management techniques for tolerating node failures in wireless sensor networks: A survey. Computer Networks, 58, 254-283.

Zhao, J. (2014, June). Minimum node degree and *k*-connectivity in wireless networks with unreliable links. In 2014 IEEE International Symposium on Information Theory (ISIT), (pp. 246-250). IEEE.

Zhao, J., Yagan, O., & Gligor, V. (2014). Results on vertex degree and *k*-connectivity in uniform s-intersection graphs. Technical Report CMU-CyLab-14-004, CyLab, Carnegie Mellon University.