Analysis of Computer Network Routing Algorithms

1st Junbin Yang
Honors Discrete Mathematics
jyang853@gatech.edu

2nd Yunjie Zhang
Honors Discrete Mathematics
yzhang3942@gatech.edu

3rd William Hudson
Honors Discrete Mathematics
whudson40@gatech.edu

***Abstract*—This paper explores the different types of computer routing algorithms, analyzes their implementation and time complexity, and considers their drawbacks. We will also explore different types of shortest path algorithms and relate them back to routing.**

***Index Terms*—routing, algorithms, networks, packets**

I. Introduction

Routing is an essential and fundamental concept in computer networks. It is a necessary component that determines how data packets are moved between computers and machines. There are many different routing implementations, each with its predetermined rules to select the best path to its destination. The design of network routing algorithms involves optimizations and graph theory to achieve quick and efficient routing times.

II. Types of routing algorithms

Computer network routing algorithms can fall under three categories: Adaptive, Non-Adaptive, and Hyrbid.

A. Adaptive Algorithm

Adaptive algorithms decide routing paths based upon the network structure (network topology). The algorithm utilizes dynamic information such as current load, delay, and network topology when determining routes.

B. Non-Adaptive Algorithm

Non-Adaptive Algorithms decide routing information and rules during the network boot-time. Hence, non-adaptive algorithms are not affected or influenced by network topology or other dynamic information.

C. Hybrid Algorithms

Hybrid algorithms incorporate aspects of both Adaptive and Non-Adaptive Algorithms.

III. Elementary Graph Theory

To understand some of these algorithms and their implementations, some basic understanding of graphs is required. Let us define some relevant key concepts.

A. Graph

A graph is made up of vertices and edges. The graph's node are represented by vertices, while the connection between two nodes are represented by edges. The term graphs and networks are often interchangeable.

The degree of a vertex, denoted $\deg(v)$, refers to the number of edges incident of the vertex v.

The number of vertices can be denoted as $|V|$.

The number of edges can be denoted as $|E|$.

B. Cycles

A cycle in a graph indicates that the path starts and ends at the same vertex.

C. Trees

A tree is simply a connected graph with 0 cycles. A tree will also have $|E| = |V| - 1$

D. Minimum Spanning Tree (MST)

A spanning tree is defined as a subset of a graph, where all vertices are connected using the minimum amount of edges. A minimum spanning tree is the spanning tree with the minimum total edge weight.

IV. Adaptive Algorithms

Adaptive algorithms intend to have a deliberate approach to change their behavior based off of traffic and topology of the network. These types of algorithms can be divided into two overarching categories based on the level of communication between routers, these being isolated and centralized routing. Isolated routing algorithms do not check the status of connections between routers using only locally known information, while centralized algorithms have this information stored in one location. Since there are two subcategories of isolated routing, these will be divided separately into 'Hot Potato' and 'Backwards Learning' to reflect their difference.

A. Isolated Routing

1) Hot Potato Routing: Hot Potato Routing is an algorithm that intends to reduce the stress on individual elements (whether on the network or router level) by pushing them to out of the system as quickly as possible. The nature of these algorithms means that packets may be sent in a not preferred or optimal direction, called deflection. These algorithms themselves have many different implementations some taking more information and others not in their decision alongside the behavior of the algorithm when there is no traffic, being possible to define its behavior according to predefined protocols or dynamically if they use a shortest path algorithm, where they can be classified further based on how they deal with multiple packets meeting inside of the same router, with the following types:

1) Minimum Advance: One packet is sent to its preferred destination
2) Weakly Stable: If the packet is deflected, then there was no free path for it to reach its destination
3) Stable: One cannot change the edges assigned to ensure that packets get closer to their destination
4) Maximum Advance: The maximum number of packets advance

In addition, to solve conflicts, different systems may be used, primarily using priority for each packet, where either priority is randomly assigned when the packet is created or decremented as it approaches its destination. Due to the differing nature of each of these algorithms and the fact that they are very traffic dependent with their own properties, limitations, and worst, average, and best case efficiencies based on how they treat the incoming packages general analysis of these algorithms is impossible without delving into each possible subcategory. It is important to note that with some of these algorithms there exists a possibility for a condition called 'livelock', where the packets endlessly cycle being unable to reach their destination, with certain algorithms being able to perform very well in certain types of networks and go to livelock on others, this property being dependent both on traffic and topology of the network in addition to algorithm type.

2) Backwards Learning Routing: Backwards Learning Routing is an isolated routing algorithm which requires more metadata than the previous information. For this, the packets are required to have three elements: their end destination, their source destination, and a 'hop' counter. This hop counter increments every time the packet is sent through a router. The hop counter of an incoming node is compared to the hop counter inside of the node's routing table, this table updating if the hop count is less than the one current stored, thus it is learning 'backwards' information about the nodes above. This allows for efficient learning of the network focusing on having packets travel to as few nodes as possible while not having complex connections. This however does not take into consideration network traffic or congestion when determining the path for any packet, thus it may be sent into a congested area of the network. A potential implementation of a backwards learning can be seen below using a hashmap implementation of a routing table:

```java
// @author William Hudson

import java.util.HashMap;
import java.util.LinkedList;
import java.util.List;

public class TableEntry {
   private int hopCount;
   private Router nextNode;
   public Router gethopCount(){return
       nextNode;}
   public void updateEntry(Router
       newNextNode, int newHopCount){
        nextNode=newNextNode;
        hop=newHopCount;
   }
   /**
   Other methods omitted
    */

}

public class Router {
   private boolean messageRead;
   private Packet packet;
   private List<Router> neighbors;
   private String name;
   private HashMap<Router, TableEntry>
       routingTable;

   /**
   Constructors ommitted
    */

   public void send(Packet packet, Queue
       queue, int hop) {
      TableValue tv =
          routingTable.get(packet.getSource());
      if(tv==null){
         routingTable.put(packet.getSource(),
             new
             TableEntry(packet.getLastNode(),
             packet.getHopCount());
      } else
          if(tv.getHopCount()>packet.getHopCount()){
         routingTable.put(packet.getSource(),
             new
             TableEntry(packet.getLastNode(),
             packet.getHopCount());
      }
      TableValue destinationToGo =
          routingTable.get(packet.getDestination());
      if(destinationToGo==null){
         \\implementation omitted, it uses a
             flooding algorithm which is a
             different type of algorithm
      } else{
         queue.add(destinationToGo.getNextNode().add(pac
      }
   }
   /**
   Other methods ommitted
    */
}
```

In this version of backwards learning, there exists a Hashmap that serves as the routing table for that specific router. It first when it takes in a packet seeks to update its own table, adding it to the table if it is unknown, then when it seeks to send out the table, it uses the table if its destination is stored, otherwise uses a different algorithm to send the packet to its destination.

B. Centralized Routing

Centralized Routing Algorithms are those whose storage and control is done by a centralized router which contains the routing table for the entire network. This allows for easy manipulation and the centralization of data that would be impossible in a more isolated system, however means that an outage of the central router will cripple the entire network. This category is extremely diverse, as unlike in the isolated networks, the central routing server has full information over the entire system. Unfortunately due to this great flexibility, analysis of centralized routing systems in terms of efficiency are nonexistent, however most centralized algorithms take inspiration from the hybrid algorithms such as distance vector and link state in order to create the routing tables for each router in the system.

V. Non-Adaptive Algorithms

In computer networking, there are two Non-Adaptive routing algorithms: Flooding and Random Walk.

A. Flooding Algorithm

The flooding algorithm is a quite simple, but rather an expensive algorithm. Starting from the source router, the router sends incoming packets to every neighboring routers except the source router itself. Every router will become a sender and a receiver of the packet, eventually leading to the packet reaching all routers in the network. This can lead to uncontrolled flooding, where some routers may keep distributing the same packet to already visited routers. To optimize this, control flooding using some some of restraint can be implemented to contain this flooding. Let us analyze the pseudocode of a controlled flooding algorithm.

Algorithm 1 Controlled Flooding with Bit State

Source Router sends packet to neighbors
Source Router seen_message bit $\leftarrow 1$
Sent_Message $\leftarrow 1$
if Sent_Message == 1 **then**
  for every router receiving packet **do**
    Sent_Message $\leftarrow 0$
    if Router.seen_message == 0 **then**
      Router.seen_message $\leftarrow 1$
      Router sends packets to neighbors simultaneously
    end if
  end for
end if

In this version of control flooding, a bit is used to indicate whether a router has seen the message or not. If a router indicates it has already seen the packet, that respective router stops and does not send the packet to its neighbors. If the router has not seen the packet, it continues to flood and send its neighbors the packet. This process is repeated until there are no available routers left.

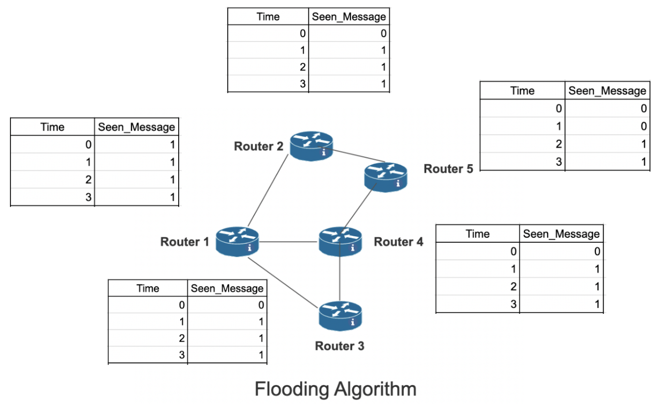

Fig. 1. Example of flooding algorithm

In the figure above, we are assuming Router 1 wants to send a packet to Router 5. At time $t = 0$, Router 1 marks its seen_message bit and sends the packet to all neighbors: Router 2, 3, 4 simultaneously. Now at time $t = 1$, since all the routers receiving a packet has their seen_message bit not set, they mark their seen_message bit and proceed to send the packet to their neighbors. At $t = 2$, router 5 is the only router that has their seen_message bit not set, so it receives the packet and sends to its neighbors. Since all routers in this network has seen the packet, the algorithm terminates.

Another implementation of control flooding is the use of hop counts. Essentially, the algorithm keeps track of how many times the packet has been sent, and terminates once it has reach a hop limit set by the algorithm. Unlike the control flooding algorithm with the seen_message bit, using hop counts allows packets to be resent to routers that have already seen the packet.

```
// @author Junbin Yang

import java.util.Queue;
import java.util.LinkedList;
import java.util.List;

public class Flooding {
   public static final int NUM_NETWORKS = 50;
   public static final int DATA_TO_SEND = 3;
   public static final int HOP_COUNT = 8; //
       prevents flooding indefinitely
   /**
   Other methods omitted
    */

   public static void flood(NetworkGraph
       graph, Router start, Packet packet,
       Queue<Router> networkQueue) {
      start.send(packet, networkQueue,
          packet.getHopCount());
      while(!networkQueue.isEmpty()) {
         Router next = networkQueue.remove();
         next.send(next.getPacket(),
             networkQueue,
             next.getPacket().getHopCount());
      }
```

```java
    }
}

public class Router {
    private boolean messageRead;
    private Packet packet;
    private List<Router> neighbors;
    private String name;

    /**
    Constructors ommitted
     */

    public void send(Packet packet, Queue
        queue, int hop) {
        if (hop < Flooding.HOP_COUNT) {
            for (Router n : neighbors) {
                if (!n.messageRead) {
                    n.messageRead = true;
                    n.packet = new Packet(this,
                        this.packet.getTargetRouter(),
                        this.packet.getData(),
                        this.packet.getHopCount()
                        + 1);
                    queue.add(n);
                }
            }
        }
    }
    /**
    Other methods ommitted
     */
}
```

The Java implementation above behaves like the Breadth First Search (BFS). The BFS algorithm does a graph traversal, first exploring all vertices at a current depth before traversing the next depth. Because graphs can contain cycles, BFS needs a way to differentiate visited and not visited vertices. This can be achieved by having a list of all the visited vertices. Consider the pseudocode below.

Algorithm 2 Breadth First Search

```
Create queue
Create list of visited nodes
Mark root node as visited
Enqueue root node
while Queue not empty do
    x = Dequeue
    for all immediate neighbors of x do
        if not visited then
            Enqueue
            Mark as visited
        end if
    end for
end while
```

Let us prove that BFS is $O(|V|+|E|)$.

Proof: We proceed directly. We first recognize that initializing data structures (i.e. queues/lists) take $O(1)$ time. In the worst case, each vertex is only visited once. Once added to the visited set, the vertex will not be visited again. This gives us $O(|V|)$. We also recognize that each edge can only be visited once when the loop visits its neighbors. In the worst case, every edge is visited once, giving $O(|E|)$.
Combining these visits, we get $O(|E|+|V|)$. It is important to note that BFS is considered to be linear time. ■

Thus, we can conclude that the flooding algorithm is also $O(|E|+|V|)$. Although this gives us linear time, it is important to consider the drawbacks of flooding. Sending the packets to all neighbors throughout the network can hog a lot of bandwidth in the network. Suppose a packet with one destination router in a network with 1000 routers. The flooding algorithm wastes a tremendous amount of traffic and energy.

B. *Random Walk Algorithm*

Type here

VI. Hybrid Algorithms

A. *Link State*

Link state algorithms is a type of hybrid algorithm that helps to determine the best path for data packets to go from the source to its destination. The following is a description of link state and its associated steps:

Network Topology Discovery:
Within link-state routing, each router in the network collects information about its directly connected neighbors and the state of the links connecting them. This information includes the cost of each link. For example, such costs may represent factors such as bandwidth/delay. Routers will use protocols that utilize link-state routing such as OSPF(Open Shortest Path First) or IS-IS(Intermediate System to Intermediate System).

Link-State Advertisement(LSA):
This occurs after the router has gathered information about its network neighbors and the costs of the links. The router then constructs a packet dubbed as the Link-State Advertisement/-Packet (LSA/LSP). This packet contains information about the router itself and its neighboring routers. The router proceeds to flood the network with these packets to ensure that every router within a network has a consistent outline of the network topology.

Building the Link-State Database:
When the routers receive LSA/LSPs from each other, they will utilize the information from these packets to update a database that contains information reflecting the state of the network. This database is called the link-state database and contains a representation of the entire network. This information proves incredibly valuable when calculating the shortest path to destinations within a network.// **Routing Table Generation**
This is completed after the shortest path calculation returns a result for each router specifically. The router will then use its specific result to construct a routing table that is specific to itself. This routing table contains information such as the next-hop router and path taken, along with that path's costs. This information proves useful when forwarding packets to their destinations.

Update Propagation:
An important aspect of link state routing is its ability to constantly monitor the network for any change in topology or link state. When there is a change, such as a link failure or a new link becoming available, the routers will recreated LSAs and flood the network. The routers then proceed to update their link-state databases, recalculate their shortest paths, and recreate their routing tables.
Algorithms Applied:
When routers are calculating a shortest path under a link-state routing protocol, they utilize a greedy algorithm called Dijkstra's Algorithm.

Algorithm 3 Dijkstra's Algorithm

procedure dijkstra(G,l,s)
Input: Graph $G = (V, E)$, directed or undirected; positive edge lengths $\{l_e, e \in E\}$; vertext $s \in V$
Output: For all vertices u reachable from s, dist(u) is set to the distance from s to u.
for all $u \in V$ **do**
  $dist$(u) $= \infty$
  $prev$(v) $= nil$
end for
dist(s) = 0
$H =$ makequeue(V), using dist-values as keys
while H is not empty **do**
  u = deletemin(H)
  for all edges $(u, v)E$ **do**
    if $dist(v) > dist(u) + l(u, v)$ **then**
      $dist(v) = dist(u) + l(u, v)$
      $prev(u) = u$
      decreasekey(H, v)
    end if
  end for
end while

The time complexity of Dijkstra's Algorithm is $O(E + V \log(V))$ where E is the number of edges and V is the number of vertices in a graph.
Let us proceed by induction to prove that Dijkstra's Algorithm is correct.
Base case (H = 1): Since H only grows

B. Distance Vector Algorithm

The Distance Vector Algorithm is also referred to as the Bellman-Ford Algorithm. This algorithm's hybrid nature allows for changes in topology or other dynamic information to adjust the algorithm. Each router has its own Distance Vector Table, which contains the distances between the respective router to all destinations in the network. Each router shares its Distance Vector Table (DVT) with its neighbors to update distances. To further clarify, consider the Bellman-Ford Equation.
Let $D_x(y)$ be the cost of least-cost path from x to y. Then,

$$D_x(y) = min_v\{c_{x,v} + D_v(y)\}$$

Where:
min_v: min taken over all neighbors v of x
$c_{x,v}$: is the direct cost from x to v
$D_v(y)$: v's estimated least-cost-path cost to y

The Bellman-Ford equation is used to relax edges. Edge relaxation is the process of repeatedly estimating the shortest path between the source router and all possible destination routers. But how does the routers get an updated shortest path? For every time t, every router will send its Distance Vector Table to its neighbors. Then, routers use this information to calculate the shortest-path using the bellman-ford equation (edge relaxation). This process is repeated until convergence. Consider the pseudocode below for the Bellman-Ford algorithm.

Algorithm 4 Bellman Ford Algorithm

$\forall v \in V, d[v] \leftarrow \infty$ // All distances in DVT are infinity
$d[s] \leftarrow 0$ // Distance from router to itself is 0
for i from $1 \rightarrow n - 1$ **do**
  for $(u, v) \in E$ **do**
    $d[v] \leftarrow \min\{d[v], d[u] + w(u, v)\}$ // Relaxing edge
  end for
end for
for $(u, v) \in E$ **do**
  if $d[v] > d[u] + w(u, v)$ **then**
    return null // Negative cycle
  end if
end for

Let us prove that the Bellman-Ford Algorithm correctly computes distances. Some notation clarification:

- $\delta(s, v)$: denote shortest path from s to v
- $w(s, v)$: denote the edge weight of the path from s to v

Proof: We need to show that if a graph has no negative cycles, then $d_{n-1}[v] = \delta(s, v) \quad \forall v \in V$
By using induction, let us prove $d_k[v]$ is the minimum weight path from s to v using $\leq k$ edges.

- Let 's' denote source node
- Let 'v' denote destination node

Base Case: If $k = 0$, then $d_k(v) = 0$ when $v = s$. For other destinations v, where $v \neq s$, $d_k(v)$ is ∞.

Inductive Hypothesis: Assume $d_k[s]$ is the minimum weight path for some $k \in \mathbb{N}$.

Inductive Step: For all vertices $\ell, d_{k-1}[\ell]$ is the minimum from S to ℓ that uses $\leq k - 1$ edges.
When $v \neq s$, let α be the shortest path from s to v with $\leq k$ edges. Let β be the node right before $v \in \alpha$. Let Q be the path from S to β. Then, Q has $\leq k - 1$ nodes in the path and thus, must be the shortest path from S to β due to our definition of ℓ. So by inductive hypothesis, we know that $w(Q) = d_{k-1}[\beta]$. During iteration $k, d_k[v]$ undergoes edge relaxation. Thus, $d_k[v]$ = min$(d_{k-1}[v], d_{k-1}[\beta] + w(\beta, v))$. So, we know that $d_{k-1}[\beta] + w(\beta, v)$ is equal to $w(Q) + W(\beta, v) = w(\alpha)$. This shows that $d_k[v] \leq w(\alpha)$. We know that $d_{k-1}[v]$

is the shortest path from s to v with $\leq k-1$ edges, so it must be as large as $w(\alpha)$ because α has more edges.

Hence. we can conclude that $d_k[v] = w(\alpha)$, so $d_k[v]$ is the minimum weight path from s to v with $\leq k$ edges. ■

Now, let us visually see how what is contained in the Distance Vector Tables.

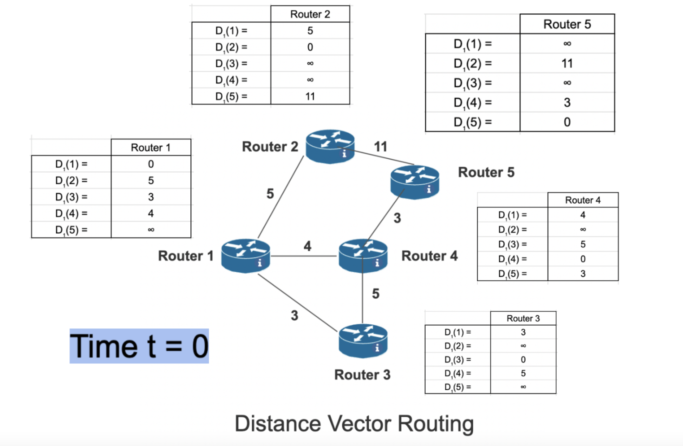

Fig. 2. Distance Vector Routing at $t = 0$

At boot time of the network $(t = 0)$, the distance vector tables are initialized and sets every entry to ∞. Then, the distance vector for each router to itself is set to 0. One aspect different to the Bellman-Ford Algorithm, is that the direct neighbors of each router have their distances estimated at $t = 0$. Before edge relaxation is performed, all routers send to every neighbor their distance vector table.

At time $t = 1$, the Bellman-Ford algorithm is used to update the estimated costs to each router. Once this update is implemented to every router's Distance Vector Table, the process repeats again and sends a distance vector table to every neighbor. This process continues until convergence, or an agreement between routers, is achieved.

Now let us prove that Bellman-Ford is $O(|V| \cdot |E|)$.

Proof: We proceed directly. We acknowledge that the initialization of the distance array for each vertex requires $|V|$ amount of work, thus $O(|V|)$. Then, the algorithm iterates over all the edges, $|V| - 1$ times. During each iteration, each edge is relaxed, thus updating the distance to each vertex. This process relaxes $|E|$ edges, $|V| - 1$ amount of times, giving us $O(|V| \cdot |E|)$ for this process. For the negative cycle check, the for loop is ran $|E| = |V| - 1$ times, thus giving $O(|V| - 1)$. Adding these up, we get $O(|V|) + O(|V| \cdot |E|) + O(|V| - 1)$. And by using properties of Big-O, we get that the Bellman-Ford Algorithm is $O(|V| \cdot |E|)$. ■

To calculate the time complexity of the Distance Routing Algorithm, some complications occur. In actual application, routers send their updated distance vector table periodically. The time complexity would depend on its periodic time, but is also affected by a slow convergence time. Another issue is that Bellman-Ford cannot prevent loops from occurring. Routing loops is a big problem and relates to the count to infinity problem. The routing algorithm is also vulnerable to errors. If the wrong path costs are made in the Distance Vector Tables, these errors can quickly propagate to other routers. In fact, this occurred to ISPs in the past, and redirected users to wrong pages.

VII. Conclusion

Computer routing algorithms are a fundamental piece on how computers and networks communicate with each other. Determining which routing algorithm to choose is a complex process, each with its pros and cons. We also see how graph theory and greedy algorithms come to play when designing computer routing algorithms.

References

[1] I. Ben-Aroya, D. D. Chinn, and A. Schuster, "A lower bound for nearly minimal adaptive and hot potato algorithms - algorithmica," SpringerLink, https://link.springer.com/article/10.1007/PL00009219

[2] Static and dynamic hot-potato packet routing in ..., http://www.mit.edu/ gamarnik/Papers/hot_potato.pdf

[3] "Shortest path algorithms tutorials & notes: Algorithms," HackerEarth, https://www.hackerearth.com/practice/algorithms/graphs/shortest-path-algorithms/tutorial

[4] Networks and graphs, https://kc.columbiasc.edu/ICS/icsfs/Networks.pdf?target=a96539a9-4cfd-44a5-baf2-fd798e1772ea

[5] T. Wolf and D. Serpanos, "Routers," Architecture of Network Systems, www.sciencedirect.com/science/article/pii/B9780123744944000074?via%3Dihub

[6] K. Ramasamy and D. Medhi, "Routing protocols: Framework and principles," Network Routing (Second edition), https://www.sciencedirect.com/science/article/pii/B9780128007372000041?ref=pdf_dow 2&rr=864edaa91cb561ab

[7] What is routing? - network routing explained - AWS, https://aws.amazon.com/what-is/routing/

[8] N. Kumari, "Routing algorithms in computer networks," Scaler Topics, https://www.scaler.com/topics/routing-algorithms-in-computer-networks/topic-challenge-container-444943