Chapter 4

Multipath Theory and Models

"In theory, theory and practice are the same.
In practice, they are not. "
–Albert Einstien

In this chapter we will start by detailing the various graph algorithms used to compute shortest paths both in the Link State and Distance Vector scenarios. Next, we will describe the concept of Selfish routing which will provide a formal definition of the consequences of congestion insensitive routing. Finally, we will give a brief introduction and present the most remarkable results of Queue Theory which will be used later in this document to model our Multipath routing protocol.

4.1 Graph Algorithms

The algorithms presented in this section do not provide multipath routes but only shortest paths. That said, they can be used, with some minor modifications, to build multipath routes as we will see in Chapter 5.

4.1.1 Dijkstra's Algorithm

Dijkstra's Algorithm was proposed by Edsger Dijkstra in 1959 [22], it is a graph search algorithm which given a graph composed of a set of edges, nodes and edge weights, finds the shortest path from a given source to all other nodes. It is therefore obvious that this algorithm is fundamental in Internet routing.

The idea behind this algorithm is simple, consider a city's road system with intersections. You wish to find the shortest path between two points in this city. At your start point, you build a list of intersections which are directly connected to it. Then, select the intersection that is closest to your destination and mark the road and the starting

intersection used. Next, repeat these steps considering the current intersection as your start point, considering one intersection at each iteration. Once the list of intersections is empty, the algorithm ends, and your shortest path is the the one consisting of all the marked intersections.

We will now give the algorithm description in textual form, the pseudo-code can be found in Algorithm ??, followed by a running example of the algorithm in Figure ??, the notation $\frac{x}{y|z}$ is used where x is the node name and y is the distance to the initial node and z the previous node.

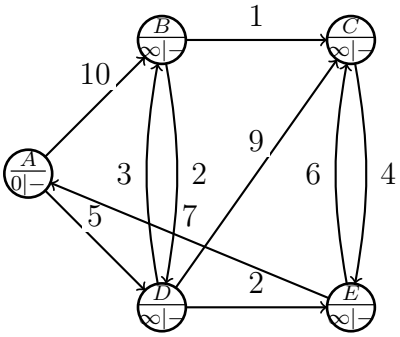
1. Set the distance of the initial node to zero and infinity for all the others.
2. Set all nodes to unvisited except the initial node (current node).
3. For unvisited neighbors of the current node, compute their distance from the initial node. If this distance is less than their current distance, replace their current distance with the computed one.
4. Once all the unvisited nodes have been visited, mark the current node as visited.
5. If no unvisited nodes remain, then the algorithm is finished. Otherwise pick the node with the smallest distance from the initial node and set it as the current node and repeat from step 3.

As is shown by Figure ??, Dijkstra's Algorithm grows a tree from a given source to all the other nodes for which the distance from the source to all the other nodes is minimal. We initially start with the original graph at node A which has a cost of zero. In the second step, we start at A and search for its closest neighbor (red arrows), we than remove A from the list of unvisited nodes (Q). Next, having found D as the closest neighbor, we can safely mark the edge from A to D as part of the shortest path (green arrow). We then repeat the search for D and find E and remove D from Q. At E we only consider the edge to C as the others link to nodes which are no longer in Q. At the end, we obtain the green graph which is the shortest path from A to all the other nodes. Table ?? shows the intermediate steps in the algorithm, the notation x/y is used where x represents the distance from the initial node and y is the precedent node in the shortest path.

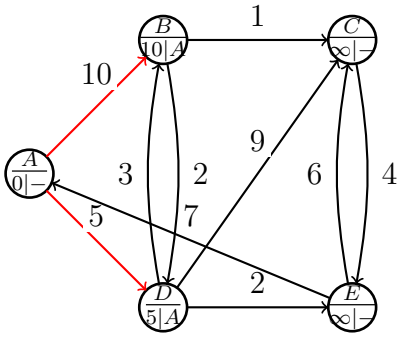
4.1.2 Bellman-Ford Algorithm

Bellman-Ford's algorithm [20] is the basis for Distance Vector routing protocols. It was proposed simultaneously by Bellman and Ford in 1958. It produces the same result as Dijkstra's algorithm but it is more flexible, namely it allows for negative edge weights[1].

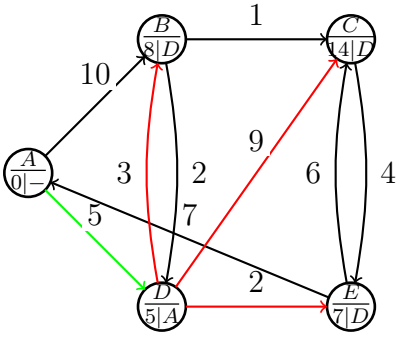
[1]While this is an interesting fact, it is useless for internet routing. Indeed, negative weights have no physical meaning.



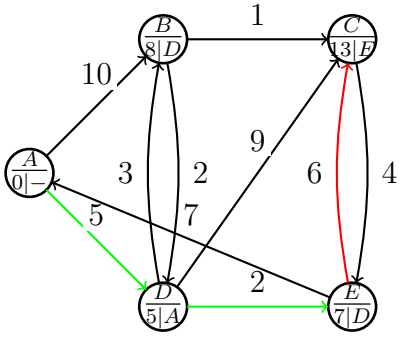
(a) Initial Graph.



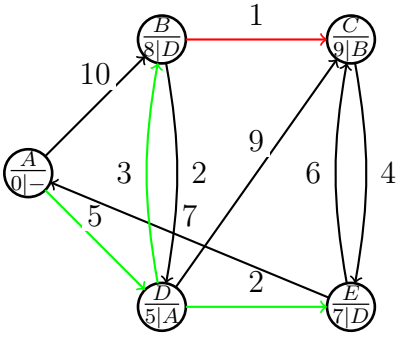
(b) Search for node closest to A.



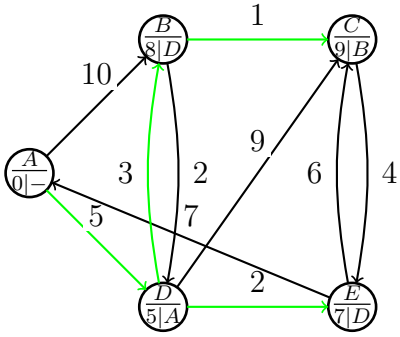
(c) D found, repeat search for closest node.



(d) Only consider node C, as others have already been found.



(e) Consider only C again, for the same reason.



(f) Green edges show the final shortest path graph.

Figure 4.1: Running Example of Dijkstra's Algorithm

I	A	B	C	D	E	Q
0	0/-	∞/−	∞/−	∞/−	∞/−	A-E
1	0/-	10/A	∞/−	5/A	∞/−	B-E
2		8/D	14/D	5/A	7/D	B,C,E
3		8/D	13/E		7/D	B,C
4		8/D	9/B			C
5			9/B			$\varnothing$

Table 4.1: Step by Step of Dijkstra's Algorithm

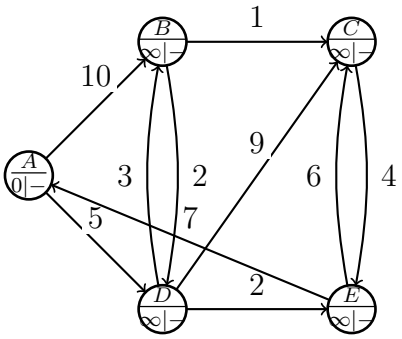
It is a decentralized algorithm which only requires nodes to inform its neighbors of its distances to other nodes. Then, each node receiving this information picks the shortest advertised weight.

In a network, routers which use this algorithm maintain a distance table containing an entry for each of the nodes in the network. By looking up in the distance table, a router will know where to send traffic intended for a particular destination. Below we give a textual description of the algorithm, whose pseudo-code can be found in Algorithm ?? followed by a running example of the algorithm.

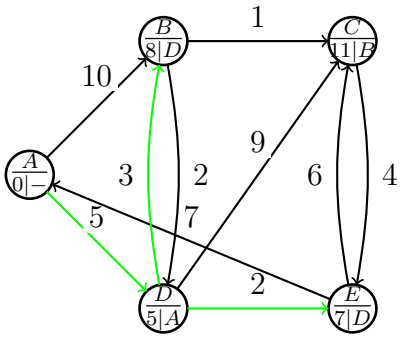
- Initialize the graph by setting all distances to ∞ and the source's distance to zero.
- If the distance from the source to a neighbor using a particular edge is shorter than the current distance, overwrite the current distance with the new one. Repeat this step for all nodes and reconsider all edges in the graph.
- Finally check for negative edge cycles by computing the distance from a node to its neighbor, if this distance is smaller than the stored distance then the graph contains negative edge weights.

The order in which the edges are visited is not specified by the algorithm, therefore we will opt to visit, from left to right, the top horizontal edges first, then the vertical ones, followed by the bottom horizontal ones, and leave the diagonal ones last. The notation is the same as in the case of Dijkstra.

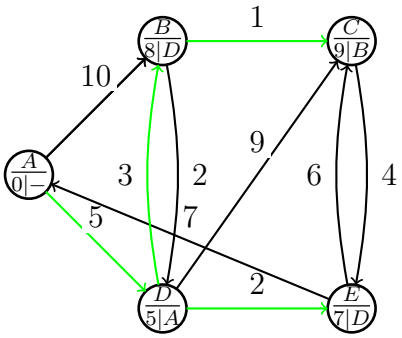
It may seem in Figure ??, that Bellman-Ford's algorithm is extremely efficient. This is not the case, because for each node it analyses every edge in the network, this is not shown for reasons of brevity and clarity. Bellman-Ford therefore requires $O(|V||E|)$ operations, whereas Dijkstra's algorithm requires $O(|E| + |V|log|V|)$, where $|V|$ is the number of vertices's and $|E|$ the number of edges.



(a) Initial Graph



(b) At node B, update distances according to edge traversal order



(c) Update distance for node C, since it was missed in the last iteration.

Figure 4.2: Running Example of Bellman-Ford's Algorithm

4.2 Selfish Routing

Selfish routing describes the overall negative effects on the performance of a network when routing decisions are taken selfishly. It is clear that the travel time between two points is highly dependant on the number of users using a given route. Yet, most of us will always opt for the shortest path that gets us to our destination fastest, regardless of the effect this decision has on other users. This is referred to as Selfish Routing [60].

Routing on the Internet is insensitive to congestion and therefore suffers from the phenomena explained by Selfish routing. Indeed, shortest path routing is analogous to commuters traveling to work each choosing the shortest path and thereby delaying everyone. Figure 4.3 shows a network (referred to as Pigou's Example [61]), in which the total traffic is represented by one, with two paths where the upper one has a delay of one time unit regardless of congestion and the lower path's delay is a function of the congestion. It is reasonable to assume that all traffic will be selfishly routed along the lower path, and therefore that it will be delayed by one time unit. On the other hand, if we consider that the traffic is split equally between the two paths, the overall delay is $\frac{3}{4}$. Indeed, the traffic on the upper path suffers a delay of one time unit but now the traffic on the lower path is only delayed by half a time unit. Therefore, we easily see that no traffic is worse off and moreover half of the traffic is delivered significantly faster. Clearly, the phenomenon illustrated in the first situation could be offset by increasing the link capacities but this is not a tenable nor scalable solution in the longterm.

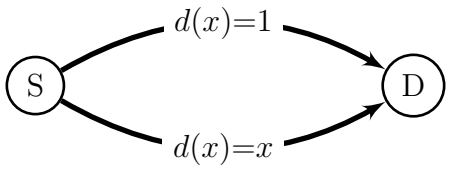


Figure 4.3: Pigou's Example

Selfish Routing also provides us with a further justification of deploying multipath routing protocols. Before we develop further this justification we must first proceed to the definitions of a few concepts which are more formally detailed in Section A.4. First, we must redefine the notion of a flow which will be limited to this section only.

Definition 3. *A* flow f *is interpreted as the aggregated routes chosen by the traffic, with* f_p *measuring the amount of traffic on route* p*.*

- *Nash Flow* can be seen as a flow routed selfishly, as in the first case given in Figure 4.3.

- *Optimal Flow* is the flow with minimum possible delay, as in the second case of Figure 4.3.

- *Price of Anarchy* is the *worst-possible* ratio between the delay of a Nash flow and that of an optimal flow.

With these definitions at hand and if we consider multipath networks with only linear delay functions, Selfish routing tells us that the upper bound[1] on the *Price of Anarchy* is $\frac{4}{3}$ (see proof in Section A.4). This result further justifies our exploration of multipath networks, because we would expect that in a multipath network with selfish users that additional problems in congestion would arise and therefore deteriorate the performance of the network. This result shows us that this is not the case, and that even in a multipath network with only selfish users there is "only" about a 33% loss in performance, which gives us hope that we may improve the performance of a multipath network significantly while paying a relatively small price in the worst-case scenario (ie. only shortest paths available). Indeed, as we will see in the next chapter, our protocol attempts to leverage all feasible alternative paths and defaults to the shortest path (selfish route) if the alternative paths are not available. As the delay functions increase in degree (ie. polynomial delay functions), the Price of Anarchy increases as well.

4.3 Analytical Models

In this section we present some analytical models based on Queuing Theory which will allow us to model multipath protocols. It should be stated that an exact analysis of multipath communication is too complex [62], however the independence assumption of Section 4.3.1.3 simplifies the calculations and provides a reasonable approximation.

4.3.1 Queue Theory

A queuing system [63], as shown in Figure 4.4 is characterized by the following parameters:

- *Arrival process (A)* is defined by the distribution of the inter-arrival time of clients into the queue. The only interesting arrival process of interest in this thesis is exponential (Markovian).

- *Service time distribution (B)* represents the time required to serve a client present at the top of the queue. We will only deal with Markovian distributions.

[1] An upper bound also exists for all polynomial delay functions, which is given in Section A.4, but for the sake of clarity we only refer to linear cost functions here.

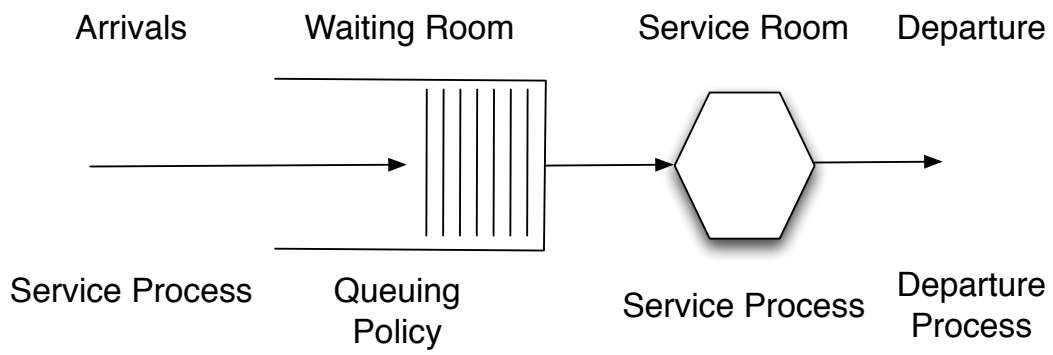


Figure 4.4: A standard queue.

- *Number of servers (m)* present in a single queue. We will deal with models containing one or more servers.
- *Capacity (K)* of the queue is the maximum number of clients which can be present in the system at any given moment.
- *Population size (N)* represents the potential number of clients whether finite or infinite. We will consider situations with infinite populations.
- *Queuing policy (D)* defines in which order the next client is selected from the queue, eg. first in, first out (FIFO).

Kendall's Notation is often used to identify the type of queue used which is denoted by A/B/m/K/N/D. The parameters K, N, and D are often omitted indicating that they are either infinite or that a FIFO policy is used.

The arrival and service time processes can be either of the following:

- *Markovian* - A Poisson distribution [64] where the inter-arrival times are independent and exponentially distributed.
- *Deterministic* - constant inter-arrival/service time.
- *General Distribution* where the underlying probability density function is arbitrary.

It is worth presenting some general results and notations which apply to all queuing systems:

- λ is the average inter arrival rate of clients into the systems.
- μ, the average service time per server.

- ρ is the utilization factor of the queue ($\frac{\lambda}{\mu}$ and $\frac{\lambda}{m\mu}$ in the multi-server case). For the queue to be stable, we should have $0 \leq \rho < 1$.
- W is the average waiting time in the system.
- T is the average time in the system.

$$T = W + \frac{1}{\mu} \tag{4.1}$$

- $\bar{N}$ is the average number of clients in the system given by Little's Result [65].

$$\bar{N} = \lambda T \tag{4.2}$$

- $\bar{N}_q$ is the average size of the queue.

$$\bar{N}_Q = \bar{N} - m\rho \tag{4.3}$$

- P_N is the probability that there are N clients in the system.

In the next sections we will present remarkable results for both the M/M/1 and M/M/m queues. We will also describe Kleinrock's Independence Assumption.

4.3.1.1 M/M/1 Queue

The M/M/1 queue is composed of an exponentially distributed inter-arrival process and service times along with a single server model. The M/M/1 queue is the simplest form a queue can take, while still remaining interesting. While being simple its behavior is similar to other more complex cases.

Since the inter arrival distribution is Poisson, the arrival rate is given by λ and the service time is $\frac{1}{\mu}$, which gives us directly the following relations:

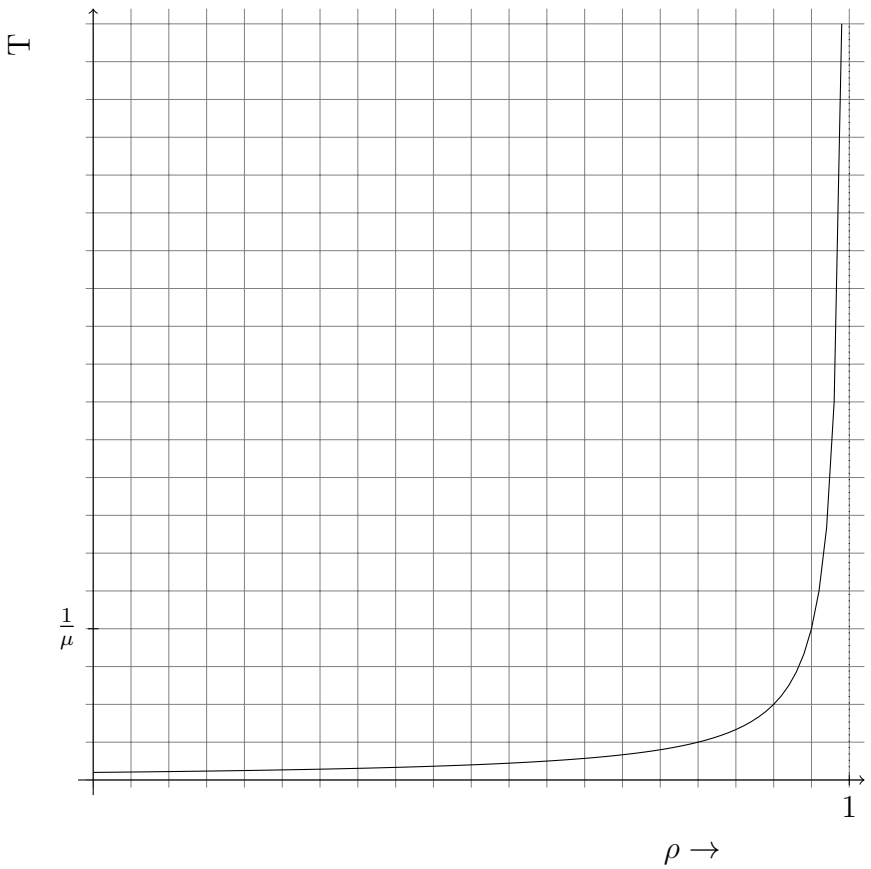
$$\bar{N} = \frac{\rho}{1-\rho} \tag{4.4}$$

Using Little's Result (Equation 4.2) and Equation 4.3, we obtain the following relations for W and T.

$$W = \frac{\rho}{\mu(1-\rho)} \tag{4.5}$$

and,

$$T = \frac{1}{\mu(1-\rho)} \tag{4.6}$$



All the relations given for $\bar{N}$, T, and W demonstrate common behavior with respect to the utilization factor ρ, more precisely they behave inversely to $1 - \rho$. Therefore, as ρ tends towards one the average delays and queue sizes tend towards infinity as shown in Figure **??**.

4.3.1.2 M/M/m Queue

We will now consider a generalization of the M/M/1 queue to m servers. In this model, a single queue forms the entry to the system and a collection of servers will handle the first client at the head of the queue. As previously, λ is the arrival rate and $\frac{1}{\mu}$ is the average service time. Also, we have here that $\rho = \frac{\lambda}{m\mu}$.

Before presenting the steady state relations relative to an M/M/m queue, we must first distinguish the situations where the queue is busy and when it is not, which help establish the steady state relations. This is given by the probability of finding that all servers are busy when a new client arrives:

$$P_Q = \frac{p_0(m\rho)^m}{m!(1-\rho)} \tag{4.7}$$

where,

$$p_0 = \left[\sum_{n=0}^{m-1} \frac{(m\rho)^n}{n!} + \frac{(m\rho)^m}{m!(1-\rho)} \right]^{-1} \tag{4.8}$$

P_Q is known a the *Erlang C formula* [66] and is widely using in telephony. Now we give the following relations for the number of clients in the system and in the queue, respectively:

$$\bar{N} = m\rho + \frac{\rho P_Q}{1-\rho} \tag{4.9}$$

and,

$$\bar{N}_Q = \frac{\rho P_Q}{1-\rho} \tag{4.10}$$

Using Little's Result (Equation 4.2) and Equation 4.3, we obtain the following relations for the average time (network delay) and the average waiting time (forwarding and processing time) spent in the system, respectively.

$$T = \frac{1}{\mu}\left(1 + \frac{P_Q}{m(1-\rho)}\right) \tag{4.11}$$

and,

$$W = \frac{P_Q}{m\mu\,(1-\rho)} \tag{4.12}$$

4.3.1.3 Kleinrock's Independence Assumption

In the two models, presented above, we have assumed that both the arrival and service times respect the Markovian property, ie. the future state of the system depends only on the present state. Moreover Burke's Theorem [67] states:

Theorem 1. ***Burke's Theorem.*** *The steady-state output of a queue with N channels in parallel, with Poisson arrivals and message lengths chosen independently from an exponential distribution is itself Poisson-distributed.*

Consider now the situation, which exists within data networks, where many queues interact in the sense that the output from one queue is the input of another (or possibly several others). Given Burke's Theorem and the models we have presented, one might think that delays in data networks are simple to obtain via such models. Unfortunately this is not the case, the message (or packet) inter-arrival time and the message lengths beyond the first queue in the network become strongly dependent. Due to the fact that

the service times at each queue are a function of the message length and therefore the arrival time at each subsequent queue is no longer Markovian [62].

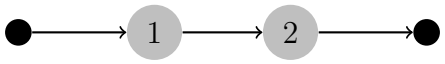


Figure 4.5: A tandem network.

Figure 4.5 illustrates two queues in tandem where packet lengths are exponentially distributed and independent of each other as well as the inter-arrival times at the first queue. We can therefore state that the first queue follows the M/M/1 model, but we cannot say the same about the second queue because the inter-arrival times at the second queue are strongly correlated with the packets lengths. Indeed the inter-arrival time at the second queue is **equal** to the transmission time at the first queue. As an analogy to this situation, consider a truck traveling on a narrow busy road along with several fast cars. Typically the truck will see an empty road ahead of it while it is being closely followed by a line of cars. The dependence between inter-arrival and message lengths (and therefore service time) is a source of great mathematical complexity for the analysis of queuing networks, in which even the simple case of the tandem network has no known exact solution [18]. A full mathematical demonstration of this dependence is given in Section A.3.

As previously shown the dependence appears for queues which are internal to the network. Therefore, one might ask: Why is there a difference between the initial (network entry) queue and internal queues? The answer is straightforward, the initial queue receives its messages from an external source consisting of many subscribers (in our case people or computers) which are all generating messages. The overall message generation by the subscribers exhibits an independence [62] since one message is different from one person to the next. In a general network a similar situation exists. Indeed, more than one queue can deliver messages to any given queue, similarly any given queue is receiving messages from many other queues. If we accept this observation then we can define the following assumption:

Theorem 2. ***Independence Assumption.*** *Each time a message is received at a queue within the net, a new length v is chosen for this message from the following distribution:*

$$P(v) = \mu e^{-\mu v}$$

Obviously, such an assumption does not correspond to the reality with a general network. Nevertheless, it results in a far simpler mathematical modeling of networks, while still maintaining an acceptable degree of accuracy [62].