"Sapienza" University of Rome
Faculty of Information Engineering,
Informatics and Statistics
Department of Computer Science

Network Algorithms

Lecture notes integrated with the book TODO

Author
Alessio Bandiera

September 30, 2024

Contents

Information and Contacts

Personal notes and summaries collected as part of the *Network Algorithms* course offered by the degree in Computer Science of the University of Rome "La Sapienza".

Further information and notes can be found at the following link: **https://github.com/aflaag-notes**. Anyone can feel free to report inaccuracies, improvements or requests through the Issue system provided by GitHub itself or by contacting the author privately:

- Email: **alessio.bandiera02@gmail.com**
- LinkedIn: **Alessio Bandiera**

The notes are constantly being updated, so please check if the changes have already been made in the most recent version.

Suggested prerequisites:

- Progettazione di Algorithmi

Licence:

1

TODO

1.1 TODO

1.1.1 Classical solutions

Algorithm 1.1.1.1 *Bellman-Ford*: TODO

```
function bellmanFord(G)
    TODO
end function
```

Algorithm 1.1.1.2 *Dijkstra*: TODO

```
function dijkstra(G)
    TODO
end function
```

Algorithm 1.1.1.1: Floyd-Warshall

Given a directed graph G, and an unconstrained weight function w for the edges, the algorithms returns a matrix `dist` such that `dist`$[u][v]$ is the weight of the least-cost path from u to v.

1: **function** FLOYDWARSHALL(G, w)
2:     Let `dist`$[n][n]$ be an $n \times n$ matrix, initialized with every cell at $+\infty$
3:     **for** $u \in V(G)$ **do**
4:         `dist`$[u][u] = 0$
5:     **end for**
6:     **for** $(u, v) \in E(G)$ **do**
7:         `dist`$[u][v] = w(u, v)$
8:     **end for**
9:     **for** $k \in V(G)$ **do**
10:         **for** $u \in V(G)$ **do**
11:             **for** $v \in V(G)$ **do**
12:                 `dist`$[u][v] = \min($`dist`$[u][k],$ `dist`$[k][v])$
13:             **end for**
14:         **end for**
15:     **end for**
16: **end function**

Idea. The core concept of the algorithm is to construct a matrix using a dynamic programming approach, that evaluates all possible paths between every pair of vertices. Specifically, to determine the shortest path from a vertex u to a vertex v, the algorithm considers two options: either traveling directly from u to v, or passing through an intermediate vertex k, potentially improving the path.

Cost analysis. The `for` loop in line 3 has cost $\Theta(n)$, the `for` loop in line 6 has cost $\Theta(m) = \Theta(n^2)$ and the cost of the triple nested `for` loop is simply $\Theta(n^3)$. Therefore, the cost of the algorithm is

$$\Theta(n) + \Theta(n^2) + \Theta(n^3) = \Theta(n^3)$$

1.2 Interconnection topologies

1.2.1 Butterfly network

Up to this point, the routing problem has considered the network as a graph where **the structure is not known to the nodes**, and can change over time due to factors like *faults* and *variable traffic*. However, when the network represents an **interconnection topology**, such as one connecting processors, the structure of the network is known and remains fixed. This characteristic can be leveraged in the packet-routing algorithms.

While the fixed nature of the network topology can be used to develop more efficient routing strategies, efficiency becomes a critical conecrn in interconnection topologies. As a result, solutions with stronger properties than basic shortest-path algorithms are required.

There are many types of routing models. In this notes, the focus will be on the store-and-forward model:

- aata is divided into *discrete packets*;
- each packet contains *control information* (such as source, destination, and sequence data) and is treated as an independent unit that is forwarded from node to node through the network;
- packets may be temporarily stored in **buffer queues** at intermediate nodes if necessary, due to link congestion or busy channels;
- each node makes a **local routing decision** based on the packet's destination address and the chosen routing algorithm;
- during each step of the routing process, **a single packet can cross each edge**;
- additionally, mechanisms for error detection and recovery may be employed to ensure reliable packet delivery, and flow control and congestion management may be applied to optimize network performance.

1.2.2 Bufferfly network

Definition 1.2.2.1: Bufferfly network

Let n be an integer, and let $N := 2^n$; an n-**bufferfly network** is a *layered graph* defined as follows:

- there are $n+1$ layers of N nodes each, for a total of $N(n+1)$ nodes;
- each node is labeled with a pair (w, i), where i is the *layer of the node*, and w is an n-bit binary number that denotes the *row of the node*;
- there are $2Nn = 2 \cdot 2^n \cdot n = n2^{n+1}$ edges;
- two nodes (w, i) and (w', i') are linked by an edge if and only if $i' = i + 1$ and either $w = w'$ (which is a *straight edge*) or w and w' differ in only the i-th bit (which is a *cross edge*).

Example 1.2.2.1 (Bufferfly network)**.** The following figure shows an example of a butterfly network.

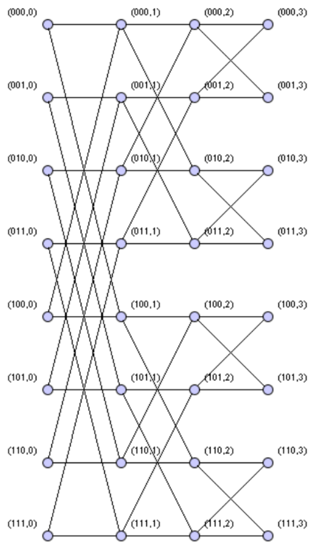


Figure 1.1: A butterfly network.

It can be shown that each node, except those in the first and the last layers, has degree 4 . Therefore, to perform the routing of the packets on a butterfly network, its nodes are prove it? **crossbar switches**, which have two input and two output ports and can operate in two states, namely *cross* and *bar* (shown below, respectively).

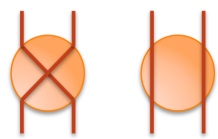


Figure 1.2: A butterfly network node.

Usually, $4N$ additional nodes are typically added ($2N$ for the input, and $2N$ for the output) such that $\deg(u) = 4$ for each $u \in V(G)$ — these nodes will not be considered in the networks analyzed in this notes.

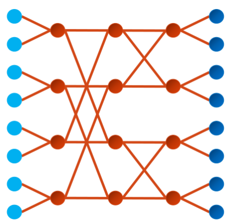


Figure 1.3: An extended butterfly network.

As a result, a butterfly network can be viewed as a *switching network* that connects $2N$

input units to $2N$ ouptut units, through a layered structure divided into $\log N + 1 = \log 2^n + 1 = n + 1$ layers, each consisting of N nodes.

Moreover, butterfly networks have a recursive structure, which is highlighted in the following figure. Specifically, one n-dimensional butterfly contains two $(n-1)$-dimensional butterfly networks as subgraphs.

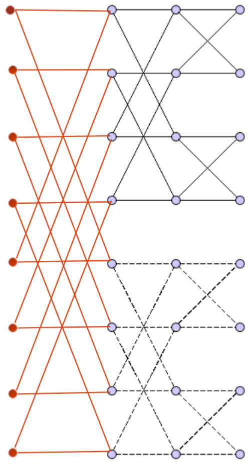


Figure 1.4: The recursive structure of butterfly networks.

The topology of the butterfly network can be leveraged as stated in the following proposition.

Proposition 1.2.2.1: Greedy path

Given a pair of rows w and w', there exists a *unique path of length* n, called **greedy path**, from node $(w, 0)$ to node (w', n). This path passes through each layer exactly once, and it can be found through the following procedure:

```
function GreedyPath(w, w')
    for i ∈ [1, n] do
        if w_i == w'_i then
            Traverse a straight edge
        else
            Traverse a cross edge
        end if
    end for
end function
```

1.2.3 Routing on a butterfly network

Packet-routing performed on a butterfly network can pose some challenges. Assume that each node $(u, 0)$ in the network on layer 0 of the butterfly contains a packet, which is destined for node $(\pi(u), n)$ in layer n — there are $n+1$ layers, ranging in $[0, n]$ —

where

$$\pi : [1, N] \to [1, N]$$

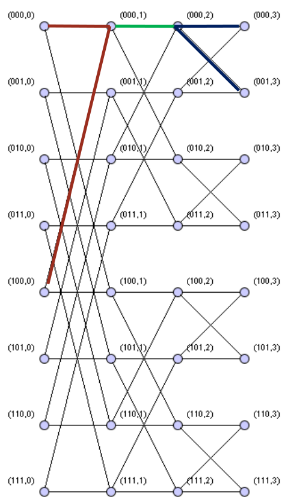
describes the permutation of the packet destinations. In a **greedy routing algorithm**, each packet follows its *greedy path*, meaning that at each intermediate layer, it makes progress toward its final destination by choosing the edges to cross through the algorithm described in Proposition 1.2.2.1.

When routing only a *single packet*, the greedy algorithm works efficiently, since there are no conflicts or competing resources along the path. However, when *multiple packets* are routed in parallel, conflicts can arise, especially when multiple packets attempt to traverse the same edge or node simultaneousl. n fact, *multiple greedy paths* may intersect at the same node or edge, and since only one packet can traverse a given edge at any moment, the other packets must be **delayed** until the edge becomes available. As a result, the butterfly network cannot route every permutation without delays, making it a **blocking network**.

For simplicity, assume that n is odd (though similar results hold for even values of n), and consider the following edge

$$e := \left(\left(0 \dots 0, \frac{n-1}{2} \right), \left(0 \dots 0, \frac{n+1}{2} \right) \right)$$

Note that e's endpoints are the roots of two complete binary trees, which have $2^{\frac{n-1}{2}}$ and $2^{\frac{n+1}{2}}$ nodes respectively.



In the worst case, π can be such that *each greedy path starting from a leaf on the left tree and ending on a leaf on the right tree traverses* e. Note that the number of such paths is precisely the number of leafs of the left complete binary tree, namely $2^{\frac{n-1}{2}} = \sqrt{\frac{N}{2}}$. Therefore, in the worst case $\sqrt{\frac{N}{2}}$ packets may need to traverse e, which means that one of them may be delayed by $\sqrt{\frac{N}{2}} - 1$ steps. Since it takes $n = \log N$ steps to traverse the whole network, the greedy algorithm can take up to

$$\sqrt{\frac{N}{2}} - 1 + \log N$$

steps to route a permutation.

The following theorem generalizes this result.

Theorem 1.2.3.1: Butterfly routing

Given any routing problem on a n-dimensional butterfly network, for which at most one packet starts at each 0-th layer node, and at most one packet is destined for each n-th layer node, the *greedy algorithm* will route all the packets to their destination in $O(\sqrt{N})$ steps.

Proof. For simplicity, assume that n is odd (though similar results can be proven for even values of n). Given $0 < i \leq n$, let e be any edge in the i-th layer, and let n_i be the number of greedy paths traversing e.

The number of greedy paths in the first half of the butterfly is bounded by the number of leaves of the left complete binary tree, namely $n_i \leq 2^{i-1}$. Analogously, on the second half of the butterfly, n_i is bounded by the number of leaves of the right complete binary tree, therefore $n_i \leq 2^{n-i}$. Note that both this results hold because n is odd.

scrivere che prendiamo il minimo, ma prima da capire il perché

Note that any packet that need to cross e can be delayed by *at most* the other $n_i - 1$ packets. Therefore, recalling that $\sum_{j=0}^{k} 2^j = 2^{k+1} - 1$, as a packet traverses layers 1 through n, the total delay it can encounter is at most

non capisco a che serva questa frase

$$\begin{aligned}
\sum_{i=1}^{n} (n_i - 1) &= \sum_{i=1}^{\frac{n+1}{2}} (n_1 - 1) + \sum_{\frac{n+1}{2}+1}^{n} (n_i - 1) \\
&\leq \sum_{i=1}^{\frac{n+1}{2}} \left(2^{i-1} - 1\right) + \sum_{i=\frac{n+3}{2}}^{n} \left(2^{n-i} - 1\right) \\
&= \sum_{j=0}^{\frac{n+1}{2}-1} \left(2^j - 1\right) + \sum_{j=0}^{\frac{n-3}{2}} \left(2^j - 1\right) \\
&= \sum_{j=0}^{\frac{n+1}{2}-1} 2^j + \sum_{j=0}^{\frac{n-3}{2}} 2^j - n \\
&= 2^{\frac{n+1}{2}} - 1 + 2^{\frac{n-1}{2}} - 1 - n \\
&\leq O(\sqrt{N}) - n \\
&\leq O(\sqrt{N})
\end{aligned}$$

□

Although such a greedy routing algorithm performs poorly in the worst case, it is **highly effective in practice**. In fact, for many practical classes of permutations, the greedy algorithm runs in n steps, which is optimal, and for most permutations the algorithm runs in $n + o(n)$ steps. Consequently, the greedy algorithm is widely used in real-world applications.

1.2.4 Beneš network

As shown in the previous section, the *butterfly network* can present efficiency problems due to packet delays caused by congestion when multiple packets are routed simultaneously. One way to *avoid routing delays* is by using a **non-blocking topology**.

Definition 1.2.4.1: Beneš network

An **n-dimensional Beneš network** is a network constructed by placing *two n-dimensional butterfly networks back-to-back.*

Example 1.2.4.1 (Beneš network)**.** The following is an example of a Beneš network. add pic

Note that an n-dimensional Beneš network has

$$2(n+1) - 1 = 2n + 2 - 1 = 2n + 1$$

layers, because the two n-dimensional butterfly network — which describe the first and last $n + 1$ layers — have an *overlapping layer*.

Consider the following property.

Definition 1.2.4.2: Rearrangeability

A network with N inputs and N outputs is said to be **rearrangeable** if, for any one-to-one mapping π of the inputs to the outputs, the mapping can be realized using exclusively *edge-disjoint paths.*

As for the case of the butterfly network, two inputs and two outputs are typically connected at both the beginning and end of the Beneš network, ensuring that each node has a degree of 4. Therefore, this type of Beneš network has $2N = 2 \cdot 2^n = 2^{n+1}$ inputs linked to the 0-th layer, and 2^{n+1} layers linked to the $2n$-th layer. add pic prob

However, in the case of the Beneš network, the following theorem will establish an important result that leverages these additional inputs and outputs.

Theorem 1.2.4.1: Rearrangeability of the Beneš network

Any n-dimensional Beneš network is rearrangeable.

Proof. The proof proceeds by induction on n.

Base case. When $n = 0$, the Beneš consists of a single node, the theorem is vacuously true, because there are no edges on the network.

Inductive hypothesis. Given any one-to-one mapping π of the 2^n inputs and outputs on a $(n-1)$-dimensional Beneš network, there exists *a set of edge-disjoint paths* from the inputs to the outputs connecting each input i to output $\pi(i)$, for each $1 \leq i \leq 2^n$.

Inductive step. TODO

$\square$