



SAPIENZA
UNIVERSITÀ DI ROMA

“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 29, 2024

Contents

Information and Contacts	1
1 TODO	2
1.1 TODO	2
1.1.1 Classical solutions	2
1.2 Interconnection topologies	3
1.2.1 Bufferfly network	4
1.2.2 Routing on a butterfly network	6

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 Algoritmi

Licence:

These documents are distributed under the [GNU Free Documentation License](#), a form of copyleft intended for use on a manual, textbook or other documents. Material licensed under the current version of the license can be used for any purpose, as long as the use meets certain conditions:

- All previous authors of the work must be **attributed**.
- All changes to the work must be **logged**.
- All derivative works must be **licensed under the same license**.
- The full text of the license, unmodified invariant sections as defined by the author if any, and any other added warranty disclaimers (such as a general disclaimer alerting readers that the document may not be accurate for example) and copyright notices from previous versions must be maintained.
- Technical measures such as DRM may not be used to control or obstruct distribution or editing of the document.

1

TODO

1.1 TODO

1.1.1 Classical solutions

Algorithm 1.1.1.1 *Bellman-Ford*: TODO

```
1: function BELLMANFORD( $G$ )  
2:   TODO  
3: end function
```

Algorithm 1.1.1.2 *Dijkstra*: TODO

```
1: function DIJKSTRA( $G$ )  
2:   TODO  
3: end function
```

Algorithm 1.1.1.1: Floyd-Warshall

Given a directed graph G , and an unconstrained weight function w for the edges, the algorithm 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

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 concern 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:

- data 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.

1.2.1 Butterfly network

Definition 1.2.1.1: Butterfly network

Let n be an integer, and let $N := 2^n$; an n -**butterfly 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 1.2.1.1 (Butterfly network). The following figure shows an example of a butterfly network.

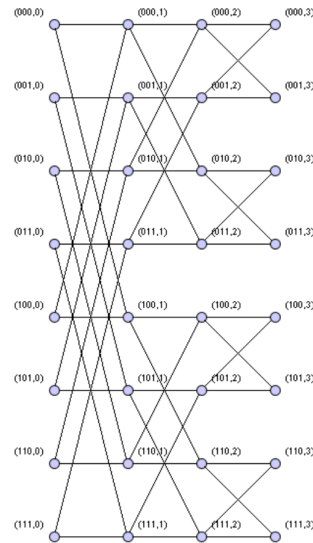


Figure 1.1: A butterfly network.

The nodes of a butterfly are **crossbar switches**, which have two input and two output ports and can operate in two states, namely *cross* and *bar* (shown below, respectively).

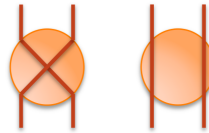


Figure 1.2: A butterfly network node.

It can be shown that each node, except those in the first and the last layers, has degree 4 . Therefore, $4N$ 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.

prove
it?

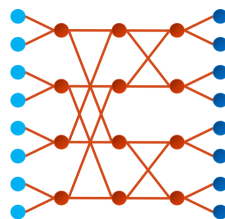


Figure 1.3: An extended butterfly network.

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

input units to $2N$ output 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.

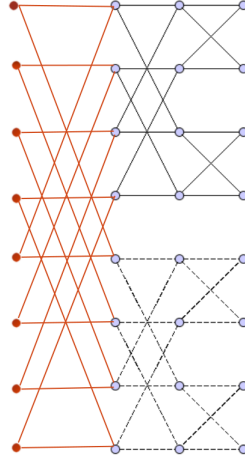


Figure 1.4: The recursive structure of butterfly networks.

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

Proposition 1.2.1.1: 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:

```

1: function GREEDYPATH( $w, w'$ )
2:   for  $i \in [1, n]$  do
3:     if  $w_i == w'_i$  then
4:       Traverse a straight edge
5:     else
6:       Traverse a cross edge
7:     end if
8:   end for
9: end function

```

1.2.2 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

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

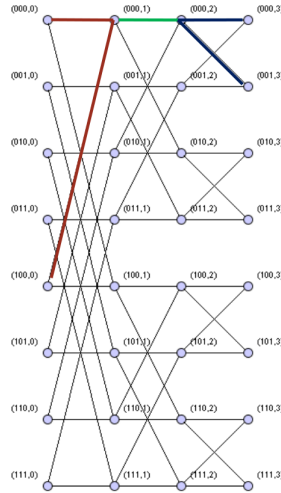
describes the permutation of the packet destinations. In a **greedy routing algorithm**, each packet follows its *greedy path*, meaning that at each intermediate layer, it makes progress toward its final destination by choosing the edges to cross through the algorithm described in [Proposition 1.2.1.1](#).

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 simultaneously. In 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 leaves 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 1.2.2.1: 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.

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

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

scrivere
che
pren-
diamo
il min-
imo,
ma
prima
da
capire
il per-
ché

non
capisco
a che
serva
questa
frase

□

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.