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

October 5, 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 Introduction on graphs

In many network applications, graphs are used as a natural model. In other applications, the graph model may be less obvious, but appears to be anyway very useful. Graph algorithms are useful instruments to solve important and living problems. We will see a number of advanced techniques for efficient algorithm design to solve problems from networks and graphs.

Definition 1.1: Graph

A **graph** is a mathematical structure $G = (V, E)$ made of a set V called the *vertex set* (or *node set*), and a set $E \subseteq V \times V$ called *edge set*.

Graphs are usually represented through circles and lines, were each line between two vertices u, v represents the edge (u, v). We will assume to be working with *simple graphs*, a type of graph that doesn't allow loop edges, i.e. edges from a node to itself, or a multiple number of edges between two vertices.

The edges of a graph can also be *directed* or *undirected*. In the former, the two edges (u, v) and (v, u) are considered two distinct edges while in the latter they are considered as the same edge. A directed graph is usually also referred to as **digraph**.

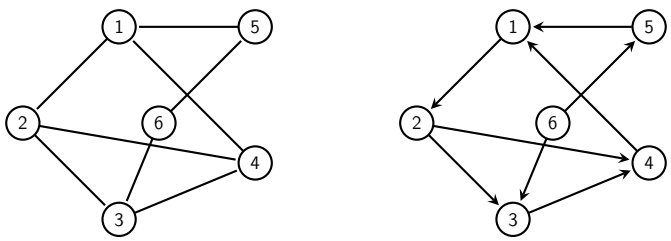


Figure 1.1: On the left: a simple graph. On the right: a simple digraph

Graphs were born in 1736, when Euler used them formalize and solve the famous *Seven Bridges of Königsberg* problem: is there a way to walk through all the bridges of the town and end up on the starting point?

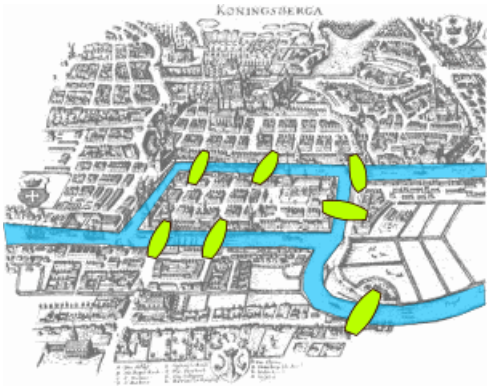


Figure 1.2: The city of Königsberg and its seven bridges

To solve the problem, Euler represented the problem as the following *multi-graph*, i.e. a non-simple graph that allows multiple edges between two vertices. Euler proved that the answer to the question is negative: a walk that passes through all the edges of such graph while also returning to the starting node cannot exist.

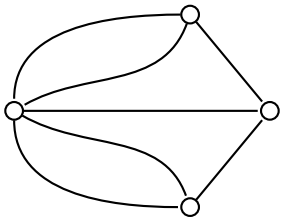


Figure 1.3: The multi-graph representing the Seven Bridges of Königsberg problem

In general, a **walk** on a graph G is given by a sequence of nodes $v_1, \ldots, v_k$ such that $(v_i, v_{i+1}) \in E(G)$. A **path** is walk whose vertices are all distinct. As we'll see in the following sections, walks and paths are the basis of graph theory.

1.2 The routing problem

When packets are sent from a computer to another through a network, each computer has to route data on a path passing through intermediate computers. This problem is usually referred to as the **routing problem**.

By modelling the network as a graph whose vertices correspond to the computers and its edges correspond to the links between them, such problem is reduced to the concept of a path from an initial node to an arrival node.

Based on the required conditions, the routing reduces to a specific type of path problem:

1. In **non-adaptive routing**, the routing algorithm must minimize the number of intermediate computers on the route. This problem reduces to the *shortest path problem*, i.e. finding the path that passes through the lowest amount of edges from node s to node t. This type of routing gives good results with consistent topology and traffic conditions, but performs poorly in case of congestion.
2. In **adaptive routing**, the routing algorithm must take into account the traffic conditions: if a route is congested, we want to avoid it in. This problem reduces to the *least cost path problem*, i.e. finding the path with the least cost from node d to node t. This type of routing gives good results with high network workload, but routes must be computed frequently in order to perform well.
3. In **fault-sensitive routing**, the routing algorithm must consider the possibility of a link failing: we want the route with the highest probability of working.

Each of these problems can be modeled as a graph. In particular, adaptive routing and fault-sensitive routing need an additional *weight function* $w : E(G) \to \mathbb{R}$ such that $w(e)$ represents the weight of an edge $e \in E(G)$. The **weight (or cost) of a path** P, written as $w(P)$, is the sum of the edges that compose it.

Example 1.1 (Weighted graphs)**.** The following is an example of a graph with weights on the edges.

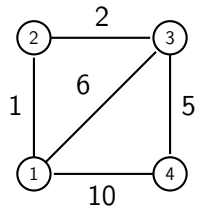


Figure 1.4: An indirected weighted graph.

For instance, the path $1, 2, 3, 4$ has weight $1 + 2 + 5 = 8$.

The weight measure varies based on the context. In adaptive routing the traffic acts as the weight, while in fault-sensitive routing the probability acts as the weight. In particular,

let $p(u, v)$ be the probability that an edge $(u, v) \in E(G)$ doesn't fail. Under the not-so-realistic assumption that edge failures occur independently of each other, we get that the probability that a path $P = v_1, \ldots, v_k$ doesn't fail is given by $p(v_1, v_2) \cdot \ldots \cdot p(v_{k-1}, v_k)$.

By setting each weight $w(u, v)$ equal to $-\log(p(u, v))$, we get that the product $p(v_1, v_2) \cdot \ldots p(v_{k-1}, v_k)$ reaches its maximum when the sum $w(v_1, v_2) + \ldots + w(v_{k-1}, v_k)$ reaches its minimum. Through this weight function, fault-sensitive routing is also reduces to the least cost path problem.

Similarly, the shortest path problem can also be reduced to the least cost path problem by setting $w(u, v)$ equal to 1 for each edge. One problem to rule them all!

Definition 1.2: Distance

Let $G = (V, E)$ be a graph. Given two nodes $u, v \in V(G)$, the **distance** between u and v, written as $\text{dist}(u, v)$, is the minimum weight of all the paths $u \to v$ of G.

On digraphs the concept of distance is non-symmetrical: the distance $\text{dist}(u, v)$ may be different from the distance $\text{dist}(v, u)$. Moreover, when there is no path $u \to v$, we assume that $\text{dist}(u, v) = +\infty$.

1.3 TODO

1.3.1 Classical solutions

Algorithm 1.1 *Bellman-Ford*: TODO

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

Algorithm 1.2 *Dijkstra*: TODO

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

Algorithm 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`][`v`]`$,$ `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

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

1.4 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:

- 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.

1.4.1 Butterfly networks

Definition 1.3: Butterfly 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 (Butterfly network)**.** The following figure shows an example of a butterfly network.

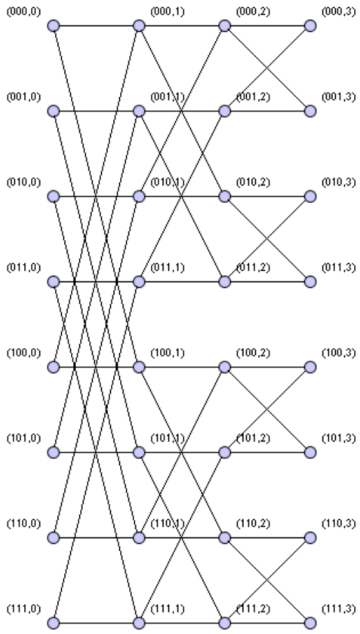


Figure 1.5: A butterfly network.

Note that the nodes of a butterfly network can be **rearranged** to form a mirror image of the original network.

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.

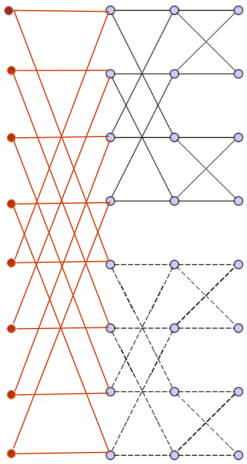


Figure 1.6: The recursive structure of butterfly networks.

Through the recursive structure of the butterfly network it can be easily shown, by structural induction, that each node of the network has degree 4, except for the ones in the first and last layer. Therefore, to perform the routing of the packets on a butterfly network,

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

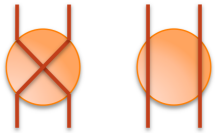


Figure 1.7: A butterfly network node.

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.

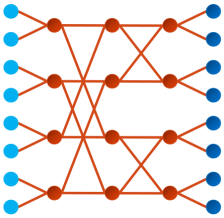


Figure 1.8: An extended butterfly network.

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.

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

Proposition 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 ∈ [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
```

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

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

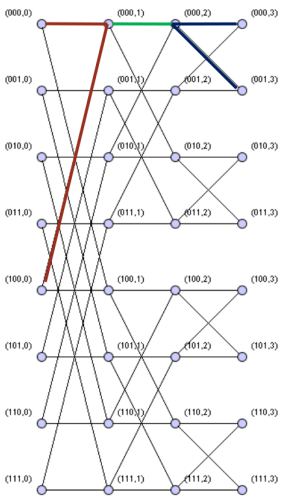
describes the permutation of the packet destinations. In a **greedy routing algorithm**, each packet follows its *greedy path*, meaning that at each intermediate layer, it makes progress toward its final destination by choosing the edges to cross through the algorithm described in Proposition 1.1.

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 1.1: 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

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

□

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.

1.4.2 Beneš networks

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 1.4: Beneš network

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

Example 1.3 (Beneš network)**.** The following is an example of a Beneš network.

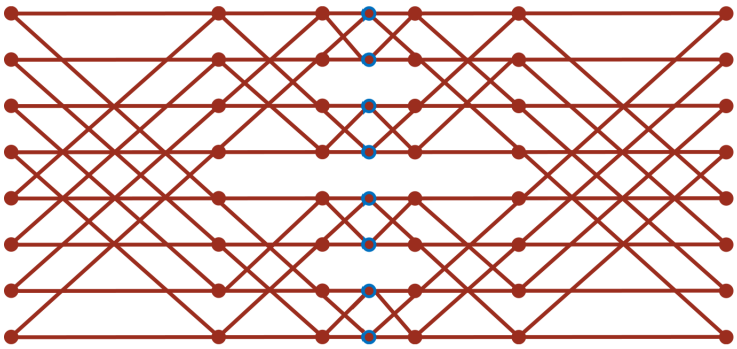


Figure 1.9: A Beneš network.

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 1.5: 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.

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: 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 of 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. Consider an n-dimensional Beneš network, with 2^{n+1} inputs and outputs; note that its middle $2n-1$ layers describe two $(n-1)$-dimensionl Beneš networks, as shown in figure.

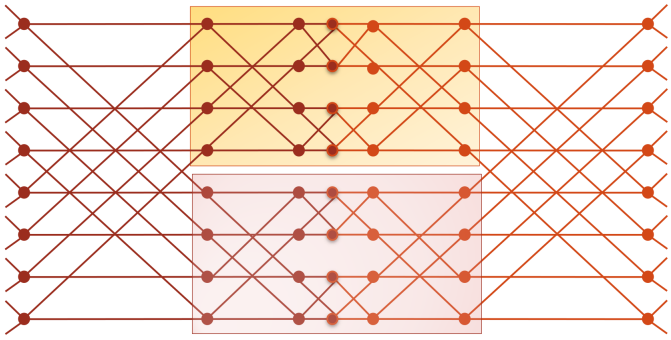


Figure 1.10: Subnetworks of a Beneš network.

Note that each *starting node* — those in layer 0 — has degree 4, and 2 of the links connect each starting node to the inputs, external to the Beneš network. Therefore, by definition of the Beneš network, the remaining two edges must connect each starting node with the two separate $(n-1)$-dimensional Beneš networks. Formally, each input $2i-1$ and $2i$ must use different Beneš subnetworks, for each $1 \leq i \leq 2n$.

The proof is constructive, and involves a so called **looping algorithm**, which proceeds as follows:

- let two inupts connected to the same starting node be referred to as *mates*;
- without loss of generality, start by routing input 1 to its destination, defined by $\pi(1)$; note that, as stated previously, this node will traverse only one of the two unconnected $(n-1)$-dimensional Beneš networks;
- route $\pi(1)$'s mate to its input, by traversing the Beneš subnetwork that *was not* traversed by the path $1 \to \pi(1)$;
- keep routing back and forth packets through the n-dimensional Beneš network; eventually, it will be routed the first input's *mate*, which closes a routing loop;
- open another loop and continue routing packets as described.

Finally, note that routing within the $(n-1)$-dimensional Beneš networks is assumed to be achievable with edge-disjoint pahts inductively.

□

If the Beneš network has *1 single input and output connected to layers 0 and* $2n$ *respectively*, the following *stronger* theorem can be proven.

Theorem 1.3: Node-disjoint paths in Beneš networks

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

Proof. Details are omitted, because it is analogous to the proof of the previous theorem, but since there is a single input and a single ouptut connected to layer 0 and $2n$ respectively, the *mate* of an input i is input $i + 2^{n-1}$, for each $1 \leq i \leq 2^{n-1}$.

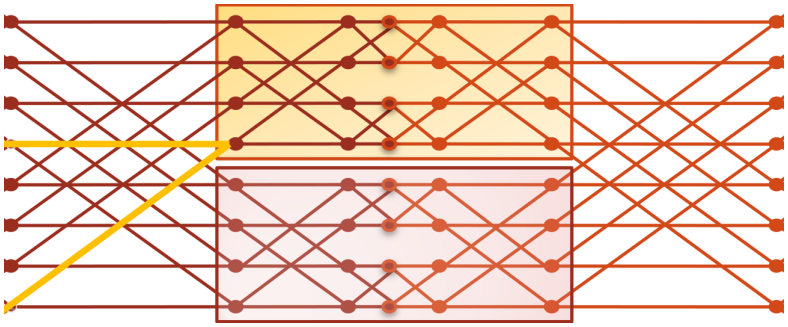


Figure 1.11: Mates in this type of Beneš network.

□

Although rearrangeability can be achieved, and even node-disjoint paths can be employed to route packets on Beneš networks, both versions of the **looping algorithm** have notable drawbacks:

- a **global controller** is *required* to manage the network, determining the routing for each packet, knowing the permutation π of the packets;
- every time a new permutation π needs to be routed, it takes $\Theta(NlogN)$ time to reconfigure all the switches.

1.4.3 Mesh network

Another important and widely used interconnection topology is the **mesh network**, which is described as follows.

Definition 1.6: Mesh network

Given two integers $m, n \geq 1$, an $m \times n$ **mesh network** $M_{m,n}$ is defines as follows:

- the nodes of the network are labeled by the following cartesian product

$$\{1, \ldots, m\} \times \{1, \ldots, n\}$$

- there is an edge between nodes $\langle i, j \rangle$ and $\langle i', j' \rangle$ if and only if

$$|i - i'| + |j - j'| = 1$$

- the path comprising the nodes labeled with $\{i\} \times \{1, \ldots n\}$ define the i-th row of the network; analogously, the set $\{1, \ldots, m\} \times \{j\}$ define the j-th column.

Example 1.4 (Mesh network)**.** placeholder add pic

For the convenience of physical layout, mesh networks are the most used topologies in Network-on-Chip (NoC) design; however, this network will not be explored in these notes.

1.5 The interconnection topology layout problem

The **interconnection topology layout problem** is a crucial challenge in (VLSI) design, the process of creating an integrated circuit (IC) by combining billions of MOS transistors onto a single chip. It involves finding the most efficient way to place and connect various components (such as transistors, resistors, and other circuit elements) on a silicon chip. The goal is to optimize several factors, including *space*, *power consumption*, *signal delay*, and *manufacturing cost*. This problem becomes particularly important as modern chips contain billions of transistors and require complex interconnections between components.

The problem originated in the 1940s, during the early stages of digital computing. However, at that time, the technology was not advanced enough to implement complex circuit layouts in an efficient manner. Physical constraints, costs, and the lack of sophisticated computational methods limited the practical application of these ideas.

In recent decades, as technology advanced, VLSI design has evolved to allow highly dense and intricate circuits in both 2D and 3D layouts. This made the **interconnection topology layout problem** a crucial area of study, particularly for *optimizing performance*, *reducing power consumption*, and *controlling costs* in increasingly smaller chip designs.

1.5.1 Thompson's Model

To addresses the challenge of finding efficient ways to place and route the components of a VLSI circuit, while maintaining certain spatial constraints, Clark Duncan Thompson developed the Thompson's Model [3], which involves representing the circuit as a graph

drawing, and analyzing how the layout corresponds to graph drawing principles.

Definition 1.7: Graph drawing

Given a graph G, its **drawing** Γ is a function that

- maps each node $v \in V(G)$ to a distinct point $\Gamma(v)$ in the drawing
- maps each edge $(u, v) \in E(G)$ in an open Jordan curve $\Gamma(u, v)$, that starts from $\Gamma(u)$ and ends in $\Gamma(v)$, such that it does not cross any point that is the mapping of a node.

Thompson performed the following mapping, between *VLSI circuits* and *graphs*:

- the *various components* of the VLSI circuit, such as *ports*, *switches* and other electronic elements, are represented by **nodes** in a graph;
- the *wires*, or connections, between the components are represented by **edges** in a graph.

However, due to the following spatial constraints imposed by VLSI technology manufacturing, this simple model requires further refinement in order to define a good **drawing** of this graph.

- **Orthogonal drawing**: *slanting lines* (diagonal connections) between components can only be *approximated*, using small horizontal and vertical segments, because of the limitations in how the VLSI fabrication process manufactures the connections onto the silicon wafer. This forces the drawing to be **orthogonal**, which means that *edges are represented as broken lines*, whose segments are horizontal or vertical, parallel to the coordinate axes.

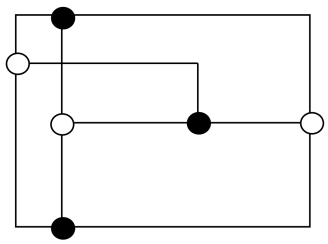


Figure 1.12: An orthogonal drawing.

- **Grid drawing**: maintaining *adequate spacing* between wires is crucial to *prevent interference*, which can degrade signal integrity. Proper spacing reduces parasitic capacitance and inductance, ensuring faster signal transmission and lower power consumption. Therefore, the graph drawing must be a **grid drawing**, such that all nodes, and crosses and bends of all the edges are put on grid points, on a grid plane, where the *grid unit* is the minimum distance allowed between two wires.

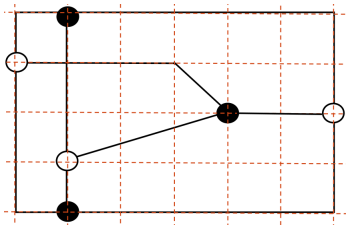


Figure 1.13: An grid drawing.

- **Crossing number minimization**: wires *must not cross*, to avoid interference and signal integrity issues. To manage this constraint, designers often route wires on opposite sides of the circuit board, utilizing small "holes" that create vertical connections between layers. While this technique helps prevent crossings, it is essential to **minimize** the number of such holes, as their fabrication can be *expensive* and may complicate the manufacturing process.
- **Area minimization**: silicon is a *costly material*, making it essential to minimize the layout area of integrated circuits. Compact layouts not only reduce material costs, but also enhance performance by shortening wire lengths, which decreases signal delay and power consumption. Therefore, **area minimization** is a critical objective in the design process, as efficient use of silicon can lead to functional advantages in the final product.
- **Edge length minimization**: wire lengths must be kept *short*, because propagation delay increases with wire length, negatively impacting circuit performance. In layered topologies, it's particularly important that wires within the same layer are approximately equal in length to *prevent synchronization issues* between signals. Thus, **edge length minimization** is crucial, as it helps ensure faster signal transmission and consistent timing across the circuit.

In 1980, Thompson introduced the following model, which describes how to draw the graph of a circuit to comply with the aforementioned constraints of VLSI design.

Definition 1.8: Thompson's Model

Given a graph of a topology G, the **Thompson's Model** is defines its layout drawing as a *plane representation*, composed of a multitude of *unit-distance horizontal and vertical traces*. This layout adheres to the following criteria:

- every *node* in $V(G)$ is mapped to the *intersection points* of the traces;
- every *edge* in $E(G)$ is represented by *disjoint paths*, formed by horizontal and vertical segments along the traces; these paths *must not* intersect nodes that are not their endpoints, and they can only cross each other at designated trace intersection points.
- *overlappings*, *node-edge crosses* and *"knock-knees"* are not allowed.

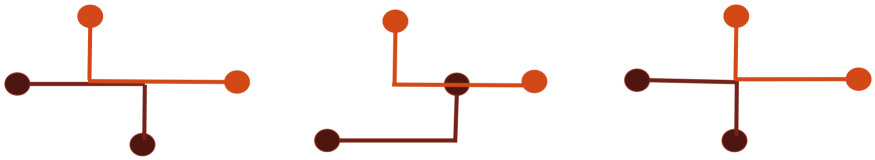


Figure 1.14: An overlapping, a node-edge cross, and a knock-knee.

In other words, this definition states that the layout of the graph of a circuit should be drawn through an **orthogonal grid drawing**, which is defined as follows.

Definition 1.9: Orthogonal grid drawing

An **orthogonal grid drawing** of a given graph G is a bijection, such that:

- each node $v \in V(G)$ is mapped to *plane points* $\Gamma(v)$ at *integer coordinates*;
- each edge $(u, v) \in E(G)$ is mapped to *non-overlapping paths*, such that the images of the endpoints $\Gamma(u)$ and $\Gamma(v)$ are connected by the corresponding paths;
- each path is constituted by *horizontal and vertical segments*, and each possible bend lies on *integer coordinates*.

Observation 1.1: Orthogonal grid drawings

Note that only graphs with $\deg(v) \leq 4$ for each $v \in V(G)$ can be correctly drawn.

Hence, the **interconnection topology layout** is an **orthogonal grid drawing** of the corresopnding graph, aimed at *minimize* the *area*, the *number of crossings* and the *wire length*.

The literature on graph drawing is extensive, but it is *not possible* to apply **existing algorithms** for orthogonal grid drawing to address the layout problem. In fact, while these algorithms provide *certain bounds* on optimization functions, for any input graph meeting

specified criteria, interconnection topologies are typically **highly structured graphs**, often regular, symmetric, or recursively built. By leveraging these unique properties, it is possible to achieve *significantly better results*. General graph drawing algorithms take a graph as input and create a planar representation; in contrast, **layout algorithms** are *specifically designed* for *particular interconnection topologies*, and require only the dimensions of the topology as input. This implies that each interconnection topology will necessitate its **own tailored algorithm**.

It's also noteworthy that improving an optimization function by even a *constant* factor can have **substantial implications**, particularly concerning area optimization. For example, if one layout occupies half the area of another, it effectively *reduces costs by half*, making such optimizations critically important.

The following sections will explore some interconnection topologies and their own orthogonal grid drawing algorithms.

1.5.2 H trees

An efficient algorithm for generating an orthogonal grid drawing of a **n-node complete binary tree** has been found independently by Leiserson [2] and Valiant [4], which employs H tree, defined as follows.

Definition 1.10: H tree

An **H tree** organizes the tree such that *only horizontal and vertical lines* connect the nodes. It can be defined inductively from its height h as follows:

- if $h = 0$ then a single node is needed

Figure 1.15: An H tree of height $h = 0$.

- otherwise, given two H trees of height $h - 1$, connect them as shown in the left drawing if h even, otherwise use the right construction if h is odd.

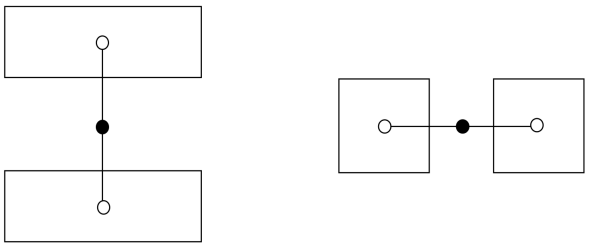


Figure 1.16: The inductive step of the inductive H tree construction.

Example 1.5 (H trees)**.** The following figure shows an example of an H tree of height $h = 4$.

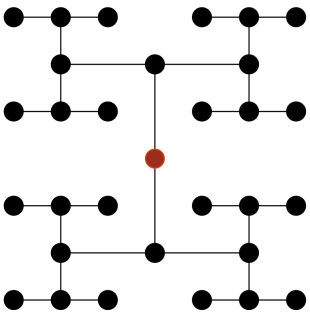


Figure 1.17: H tree of height $h = 4$.

Leiserson [2] and Valiant [4] showed that an H tree can be represented in an area of $O(n)$, where n is the number of nodes of the H tree — trivially, the area must be $\Omega(n)$. However, $O(n)$ is not sufficient, and the constant factor concealed by the big O notation must also be considered. Additionally, Brent et al. [1] proved that, if the leaves of a binary tree are required to be positioned along the borders of the rectangular area, the layout must occupy $\Omega(n \log n)$ area instead.

Note that the area of the grid we are considering is the following.

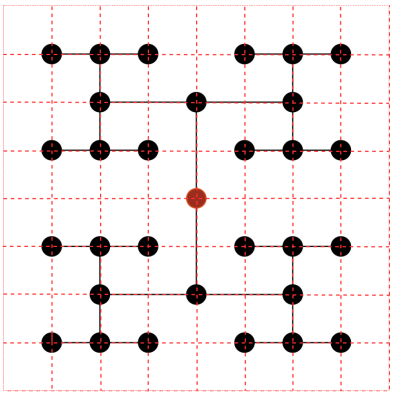


Figure 1.18: The grid of the H tree

Theorem 1.4: Area of an H tree

The area occupied by an n-node H tree is $2(n + 1) + o(n)$.

Proof. The proof proceeds by induction on the height of h the H tree

Base case. There are 3 base cases, namely when $h = 0$, $h = 1$ and $h = 2$, respectively shown in the figure below.

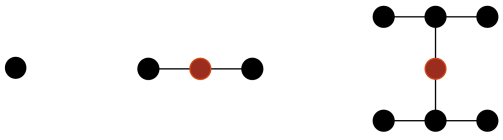


Figure 1.19: Cases for $h = 0$, $h = 1$ and $h = 2$.

Let l_h and w_h be the two sides of the rectangle enclosing the H tree of height h, respectively; thus, we have that

- for $h = 0$, $l_0 = w_0 = 2 \implies A_0 = l_0 \cdot w_0 = 2 \cdot 2 = 4 = 2(1 + 1)$
- for $h = 1$, $l_1 = 2$ and $w_1 = 4$, therefore

$$A_1 = l_1 \cdot w_1 = 2 \cdot 4 = 8 = 2(3 + 1)$$

- for $h = 2$, $l_2 = w_2 = 4 \implies A_2 = l_2 \cdot w_2 = 4 \cdot 4 = 16 = 2(7 + 1)$

Inductive hypothesis. Assume the result is true for an H tree of height $h - 1$.

Inductive step. Two different cases must be analyzed, specifically when h is *odd* an h is *even*.

- For the *odd* case, the sides of the rectangle are defined as follows:

$$\begin{cases} l_h = l_{h-1} = 2l_{h-2} \\ w_h = 2w_{h-1} = 2w_{h-2} \end{cases}$$

(note that $l_{h-1} = 2l_{h-2}$ and $w_{h-1} = w_{h-2}$). Therefore

$$\begin{aligned} l_h &= 2l_{h-2} \\ &= \dots \\ &= 2^k \cdot l_{h-2k} \qquad \left(h - 2k = 1 \implies k = \frac{h-1}{2}\right) \\ &= 2^{\frac{h-1}{2}} \cdot l_1 \\ &= 2^{\frac{h-1}{2}} \cdot 2 \\ &= 2^{\frac{h-1}{2}+1} \\ &= 2^{\frac{h+1}{2}} \end{aligned}$$

and analogously

$$\begin{aligned} w_h &= 2w_{h-2} \\ &= \dots \\ &= 2^k \cdot w_{h-2k} \qquad \left(h - 2k = 1 \implies k = \frac{h-1}{2}\right) \\ &= 2^{\frac{h-1}{2}} \cdot w_1 \\ &= 2^{\frac{h-1}{2}} \cdot 4 \\ &= 2^{\frac{h-1}{2}+2} \\ &= 2^{\frac{h+3}{2}} \end{aligned}$$

Hence, the area is

$$\begin{aligned} A_h &= l_h \cdot w_h \\ &= 2^{\frac{h+1}{2}} \cdot 2^{\frac{h+3}{2}} \\ &= 2^{\frac{2h+4}{2}} \\ &= 2^{h+2} \qquad (h = \log(n+1) - 1) \\ &= 2^{\log(n+1)-1+2} \\ &= 2^{\log(n+1)+1} \\ &= 2(n+1) \end{aligned}$$

- For the *even* case, the sides of the rectangle are defined as follows:

$$\begin{cases} l_h = 2l_{h-1} = 2l_{h-2} \\ w_h = w_{h-1} = 2w_{h-2} \end{cases}$$

(note that $l_{h-1} = l_{h-2}$ and $w_{h-1} = 2w_{h-2}$) Therefore, the calculations are analogous, but $h - 2k = 0 \implies k = \dfrac{h}{2}$ which leads to

$$l_h = w_h = 2^{\frac{h+2}{2}}$$

(recall that $l_0 = w_0 = 2$), hence

$$\begin{aligned} A_h &= l_h \cdot w_h \\ &= 2^{\frac{h+2}{2}} \cdot 2^{\frac{h+2}{2}} \\ &= 2^{\frac{2h+4}{2}} \\ &= 2^{h+2} \\ &= \dots \\ &= 2(n+1) \end{aligned}$$

□

Bibliography

[1] R.P. Brent et al. "On the area of binary tree layouts". In: *Information Processing Letters* 11.1 (Aug. 1980), 46–48. ISSN: 0020-0190. DOI: 10.1016/0020-0190(80)90034-4. URL: http://dx.doi.org/10.1016/0020-0190(80)90034-4.

[2] Charles E. Leiserson. "Area-efficient graph layouts". In: *21st Annual Symposium on Foundations of Computer Science (sfcs 1980)*. IEEE, Oct. 1980. DOI: 10.1109/sfcs.1980.13. URL: http://dx.doi.org/10.1109/SFCS.1980.13.

[3] C. D. Thompson. "Area-time complexity for VLSI". In: *Proceedings of the eleventh annual ACM symposium on Theory of computing - STOC '79*. STOC '79. ACM Press, 1979, 81–88. DOI: 10.1145/800135.804401. URL: http://dx.doi.org/10.1145/800135.804401.

[4] Leslie G. Valiant. "Universality considerations in VLSI circuits". In: *IEEE Transactions on Computers* C–30.2 (Feb. 1981), 135–140. ISSN: 0018-9340. DOI: 10.1109/tc.1981.6312176. URL: http://dx.doi.org/10.1109/TC.1981.6312176.