



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 20, 2024

Contents

Information and Contacts	1
1 The routing problem	2
1.1 Introduction on graphs	2
1.2 The least cost path problem	4
1.2.1 Classical algorithms	6
1.3 Interconnection topologies	8
1.3.1 Butterfly networks	9
1.3.2 Beneš networks	14
1.3.3 Mesh networks	17
2 The interconnection topology layout problem	19
2.1 The orthogonal grid drawing problem	19
2.1.1 H trees	23
2.1.2 The collinear layout	26
2.1.3 The Wise layout	28
2.1.4 Layered layout	30
2.1.5 Optimal area of the butterfly network	35
3 The worm propagation prevention problem	36
3.1 The vertex cover problem	37
3.1.1 Approximation algorithms	39

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

The routing problem

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 still 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, where 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**.

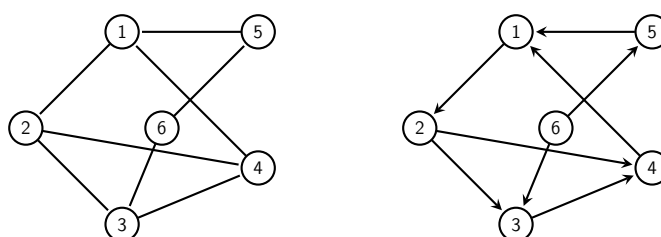


Figure 1.1: On the left: a simple graph. On the right: a simple digraph

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?

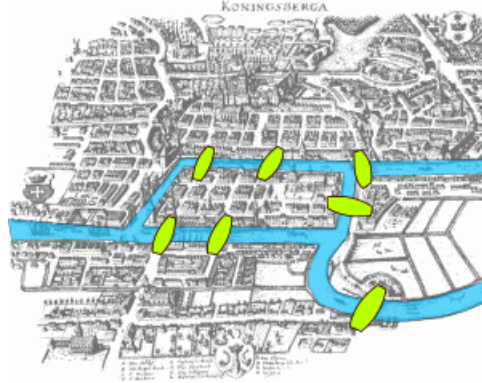


Figure 1.2: The city of Königsberg and its seven bridges

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.

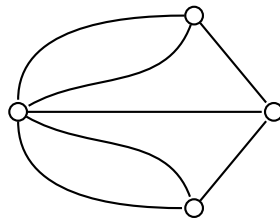


Figure 1.3: The multi-graph representing the Seven Bridges of Königsberg problem

In general, a **walk** on a graph G is given by a sequence of nodes v_1, \dots, 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 least cost path 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. Usually this type of routing is implemented through one global *routing table*.
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. In this type of routing, each router creates its own *routing table*.
3. In **half-adaptive routing**, depending on traffic and workload on the network, the routing type can switch between *adaptive* and *non-adaptive*.
4. 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* $w : E(G) \rightarrow \mathbb{R}$ 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.

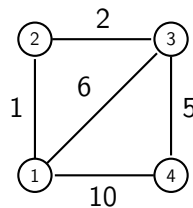


Figure 1.4: An undirected weighted graph.

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 $P = v_1, \dots, v_k$ doesn't fail is given by $p(v_1, v_2) \dots p(v_{k-1}, v_k)$.

By setting each weight $w(u, v)$ equal to $-\log(p(u, v))$, we get that the product $p(v_1, v_2) \dots p(v_{k-1}, v_k)$ reaches its maximum when the sum $w(v_1, v_2) + \dots + w(v_{k-1}, v_k)$ reaches its minimum. Through this weight function, fault-sensitive routing is also reduced 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 $u \rightarrow v$ 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 $u \rightarrow v$, we assume that $\text{dist}(u, v) = +\infty$.

Note that for the **one-to-all** shortest path problem where each edge has unit weight, the problem can be solved through a simple *Breadth-First-Search* (BFS) algorithm, invented by Moore [11].

Moreover, all known algorithm for finding the least cost path between any two nodes on a graph are based on **graph exploration**, which is based on multiple *walks* (i.e. paths where vertices may be repeated) on the graph. This raises a problem when there are **negative-weight cycles**, because the walk could take such a cycle infinitely many times, and the weight of the path between the two nodes can be lowered infinitely, without halting. Therefore, only networks without negative-weight cycles will be discussed.

Therefore, in any solution of the least cost path problem:

- cycles having **negative weight** cannot exist, by hypothesis;
- cycles having **positive weight** cannot exist, by contradiction: if there is such a cycle in a solution, then a solution without the cycle would yield a lower-weight path;
- cycles having **null length** cannot exist, by assumption: if there is such a cycle in a solution, then a solution without the cycle would yield a path of the same weight;

Hence, we can assume that there exists at least one solution **without any cycle**.

1.2.1 Classical algorithms

All the classical algorithms that will be described are based on the **relaxation** principle. Given a graph G , a weight function w , and a starting vertex $s \in V(G)$, let $d : V(G) \rightarrow \mathbb{R}$ be a function that represents the current *estimate* of the distance from s to any other node. At the beginning, $d(v) := +\infty$ for each $v \in V(G)$. Then, a **relaxation step** is performed as follows: given an edge $(u, v) \in E(G)$, if $d(u) + w(u, v) < d(v)$, then we set $d(v) = d(u) + w(u, v)$.

The first papers that presented a solution to the least cost path problem were published by Bellman [1] and Ford [8] independently, which described the following algorithm.

Algorithm 1.1: Bellman-Ford

Given a graph G , a weight function $w : E(G) \rightarrow \mathbb{R}$ on the edges, and an input node s , the algorithm returns the minimum distance tree rooted in s as a parent array, based on w .

```

1: function BELLMANFORD( $G, w, s$ )
2:   for  $v \in V(G)$  do
3:      $d(v) := +\infty$ 
4:   end for
5:    $p := [\text{NULL}, \dots, \text{NULL}]$ 
6:   for  $i \in [1, n - 1]$  do
7:     for  $(u, v) \in E(G)$  do
8:       if  $d(u) + w(u, v) < d(v)$  then                                ▷ relaxation step
9:          $d(v) = d(u) + w(u, v)$ 
10:         $p[v] = u$ 
11:      end if
12:    end for
13:  end for
14:  return  $p$ 
15: end function

```

Idea. The algorithm updates each distance $\text{dist}(s, v)$ for any $v \in V(G)$ progressively: for instance, in the first iteration of the **for** loop in line 6, since each distance is set to $+\infty$, only s 's neighbours will be updated. This will be repeated by “expanding” the updated vertices progressively at each iteration, exactly $n - 1$ times, because a path has at most $n - 1$ nodes since we are assuming that our solution do not contain cycles.

Cost analysis. The cost of the algorithm is simply given by

$$O(n) + O((n - 1) \cdot m) = O(nm)$$

The Bellman-Ford algorithm is used for [distance vector routing protocol](#), an iterative, asynchronous and distributed protocol.

One year later, Dijkstra [4] presented the following algorithm, which lowered the computational cost of the Bellman-Ford algorithm. In fact, each step of the latter iterates on all the nodes in G , even when the majority will not be updated.

Note that the following algorithm lowers the time complexity, at the cost of reducing the generality of the algorithm, because the weight function w can only be defined on \mathbb{R}^+ .

Algorithm 1.2: Dijkstra

Given a graph G , a weight function $w : E(G) \rightarrow \mathbb{R}^+$ on the edges, and an input node s , the algorithm returns the minimum distance tree rooted in s as a parent array, based on w .

```

1: function DIJKSTRA( $G, w, s$ )
2:   for  $v \in V(G)$  do
3:      $d(v) := +\infty$ 
4:   end for
5:    $p := [\text{NULL}, \dots, \text{NULL}]$ 
6:    $S := \emptyset$ 
7:    $Q := V(G)$  ▷  $Q$  is based on  $d$ 
8:   while  $Q \neq \emptyset$  do
9:      $u := Q.\text{extract\_min}()$ 
10:     $S = S \cup \{u\}$ 
11:    for  $(u, v) \in E(G)$  do
12:      if  $d(u) + w(u, v) < d(v)$  then ▷ relaxation step
13:         $d(v) = d(u) + w(u, v)$ 
14:         $p[v] = u$ 
15:         $Q.\text{update}()$ 
16:      end if
17:    end for
18:  end while
19: end function

```

Idea. The algorithm expands S , i.e. the set of visited nodes, iteratively, and at each iteration:

- the closest node u to s is chosen, based on $d(u)$;
- for each outgoing edge (u, v) from u , v is relaxed w.r.t (u, v) , and Q is updated based on $d(v)$.

Cost analysis. The cost of the algorithm depends on the implementation:

- if Q is implemented through a *queue*, then the time complexity is $O(n^2)$
- if Q is implemented through a *heap*, then the time complexity is $O(m \log n)$

- if Q is implemented through a *fibonacci heap*, then the time complexity is $O(m + n \log n)$

The last algorithm that will be discussed was discovered independently by Floyd [7] and Warshall [14], which solves the **all-to-all** version of the least cost path problem.

Algorithm 1.3: 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[ $u$ ][ $v$ ] =  $\min(\text{dist}[u][v], \text{dist}[u][k] + \text{dist}[k][v])$ 
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

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

1.3 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.

1.3.1 Butterfly networks

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

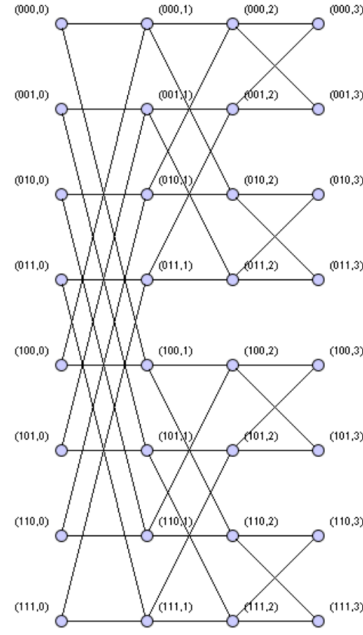


Figure 1.5: A butterfly network.

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.

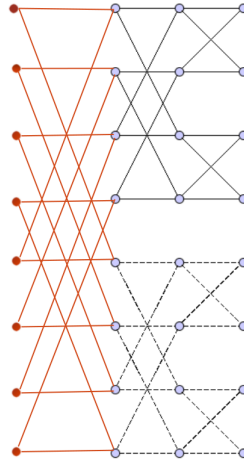


Figure 1.6: The recursive structure of butterfly networks.

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).

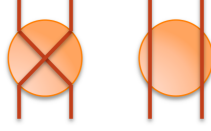


Figure 1.7: A butterfly network node.

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.

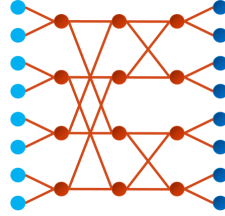


Figure 1.8: An extended butterfly network.

As a result, a butterfly network can be viewed as a *switching network* that connects $2N$ input units to $2N$ output 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  $i \in [1, n]$  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

```

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

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

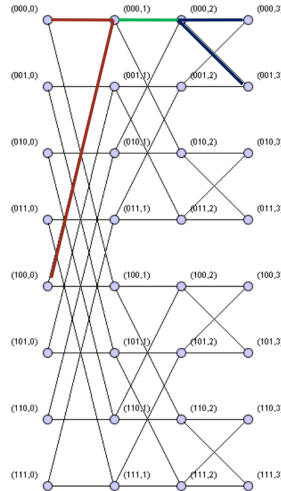
describes the permutation of the packet destinations. In a **greedy routing algorithm**, each packet follows its *greedy path*, meaning that at each intermediate layer, it makes progress toward its final destination by choosing the edges to cross through the algorithm described in [Proposition 1.1](#).

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 1.1: 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

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

□

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.

1.3.2 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.

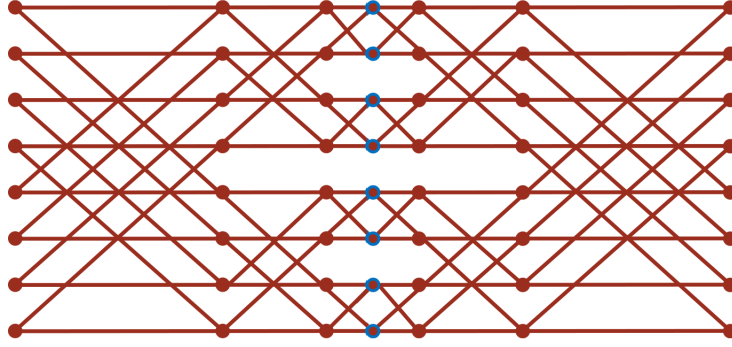


Figure 1.9: A Beneš network.

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 $(n - 1)$ -dimensional Beneš networks, as shown in figure.

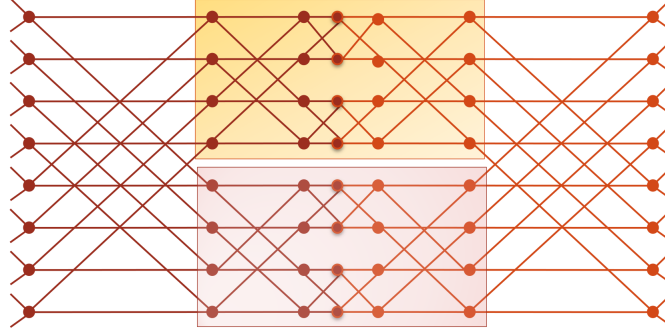


Figure 1.10: Subnetworks of a Beneš network.

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 inputs 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 $1 \rightarrow \pi(1)$;
- 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 paths 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 output connected to layer 0 and $2n$ respectively, the *mate* of an input i is input $i + 2^{n-1}$, for each $1 \leq i \leq 2^{n-1}$.

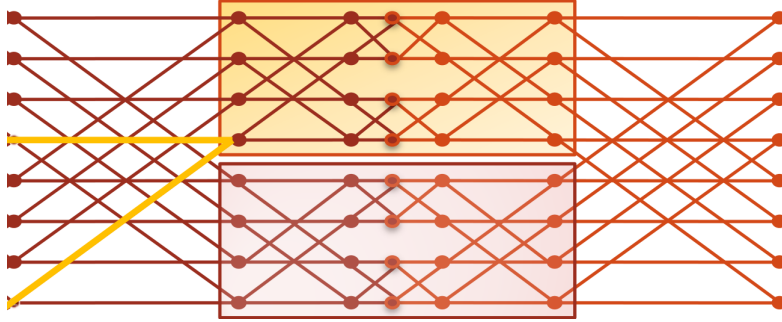


Figure 1.11: Mates in this type of Beneš network.

□

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 $\Theta(N \log N)$ time to reconfigure all the switches.

1.3.3 Mesh networks

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 defined as follows:

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

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

- 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 $\{i\} \times \{1, \dots, n\}$ define the i -th row of the network; analogously, the set $\{1, \dots, m\} \times \{j\}$ 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.

2

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.

2.1 The orthogonal grid drawing problem

To address 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 [12], which involves representing the circuit as a **graph drawing**, and analyzing how the layout corresponds to graph drawing principles.

Definition 2.1: 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.

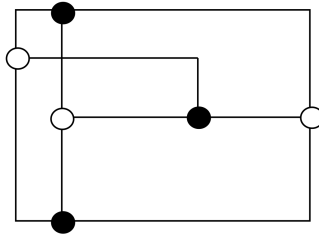


Figure 2.1: An orthogonal drawing.

- **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.

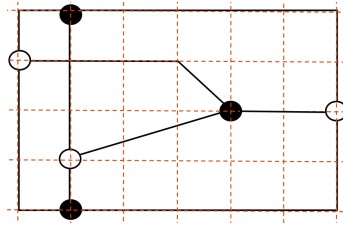


Figure 2.2: An grid drawing.

- **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 2.2: Thompson's Model

Given a graph of a topology G , the **Thompson's Model** 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.

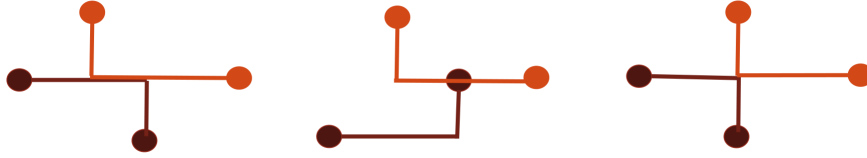


Figure 2.3: An overlapping, a node-edge cross, and a knock-knee.

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 2.3: 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 2.1: 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 corresponding 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.

2.1.1 H trees

An efficient algorithm for generating an orthogonal grid drawing of a n -**node complete binary tree** has been found independently by Leiserson [10] and Valiant [13], which employs **H tree**, defined as follows.

Definition 2.4: 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 2.4: 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.

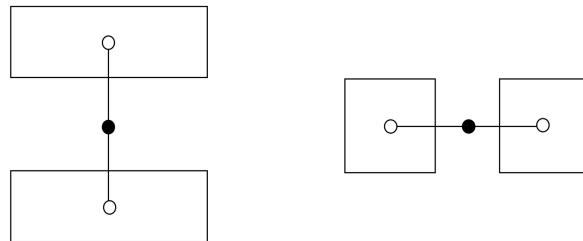
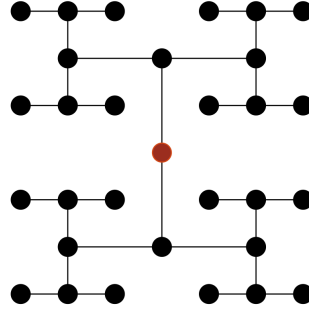


Figure 2.5: The inductive step of the inductive H tree construction.

Example 2.1 (H trees). The following figure shows an example of an H tree of height $h = 4$.


 Figure 2.6: H tree of height $h = 4$.

Leiserson [10] and Valiant [13] 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. [2] 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.

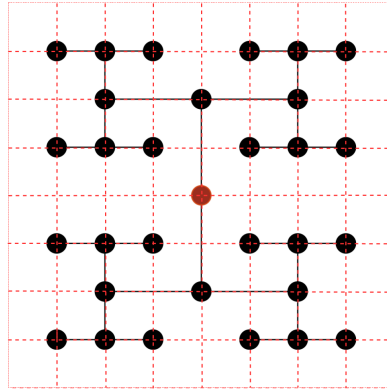


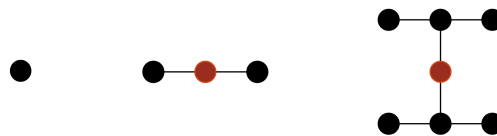
Figure 2.7: The grid of the H tree

Theorem 2.1: 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 2.8: 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* and 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

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

and analogously

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

Hence, the area is

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

- 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 $h - 2k = 0 \implies k = \frac{h}{2}$ which leads to

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

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

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

□

2.1.2 The collinear layout

The Thompson model imposes the restriction that each *processing element* (i.e. node) can have at most **4 wires** coming out of it in 2D (and 6 in 3D). This constraint ensures that nodes have *manageable connectivity*, which is crucial for simplifying VLSI layouts.

However, when nodes with **higher degrees** are *required*, especially in more complex designs, this limitation becomes problematic. By the late 1990s, researchers proposed the following **non-constant node degree model** as a solution:

- a node with degree d occupies a square with side length proportional to $\Theta(d)$;
- the wires connecting these nodes follow **horizontal or vertical paths** along *grid lines*, similar to how connections are handled in lower-degree models.

This adaptation maintains simplicity while accommodating *more complex topologies*. This evolution in layout strategies allows for more scalable VLSI design, making it possible to handle larger, more interconnected networks without overly restrictive node degree constraints.

In particular, this section will focus on a layout proposed by Yeh et al. [16], called the **collinear layout**, in which all the nodes of the network are placed *on the same line*.

The following example will show how to get a *collinear layout* from a **complete graph**.

Example 2.2 (Collinear layouts). Consider the following *labeled complete graph*

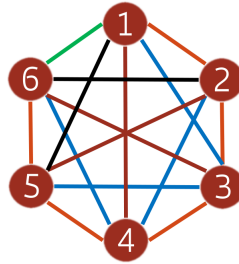


Figure 2.9: A clique of degree 6.

and let a **link of type- i** be any edge between two nodes whose labels differ of exactly i . To obtain the corresponding *collinear layout*, place the 6 nodes on the same line in order, and connect them by placing type- i links in the least possible number of **tracks** — in this context a *track* is a horizontal line on which links can be placed. For instance, type-1 links can be placed in 1 track, type-2 links can be placed in 2 tracks — by placing links between odd numbers on one track and links connecting even numbers on the other — and so on.

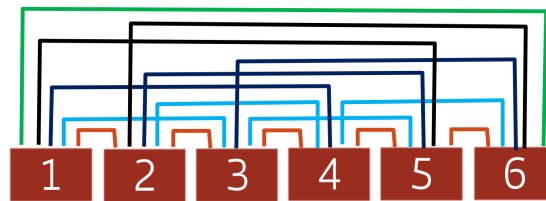


Figure 2.10: Arrangement of links in tracks

This example shows that type- i links of a collinear layout occupy at most $\min(i, n - i)$. In fact, the following property can be derived. Thus, the total number of tracks of this layout can be obtained by evaluating the following sum:

$$\begin{aligned}
\sum_{i=1}^{n-1} \min(i, n-i) &= \sum_{i=1}^{\frac{n}{2}} i + \sum_{i=\frac{n}{2}+1}^{n-1} (n-i) \\
&= \sum_{i=1}^{\frac{n}{2}} i + \sum_{j=1}^{\frac{n}{2}-1} j \quad (j = n-i) \\
&= \frac{1}{2} \left[\frac{n}{2} \left(\frac{n}{2} + 1 \right) + \frac{n}{2} \left(\frac{n}{2} - 1 \right) \right] \\
&= \frac{n^2}{4}
\end{aligned}$$

placeholder

incomplete

2.1.3 The Wise layout

In a paper published by Wise [15], it was proposed a *different layout* for the butterfly network (discussed in Section 1.3.1), which is shown in the following figure.

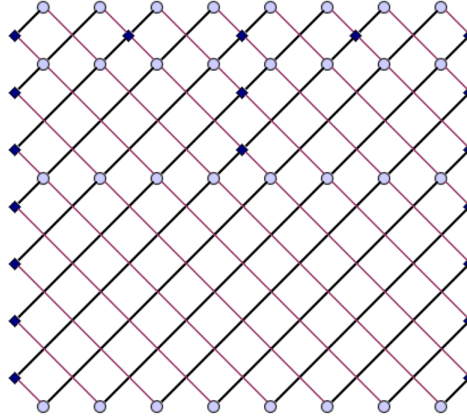


Figure 2.11: The alternative layout of the butterfly network.

Note that inputs are placed in the upper layer, and outputs in the lower layer; also, the *blue circles* represent nodes of the butterfly network, and *black squares* represent **devices** that allows to avoid interference, since those wire conjunctions are *knock-knees*.

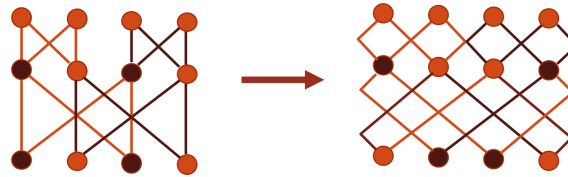


Figure 2.12: Rearrangement to get the Wise layout

This rearrangement of the wires allows to have some important *properties*.

- All the wires in the same layer have **equal length**. Note that this is *not true* for every layout; for instance, in the *classical drawing* of the butterfly network, the *straight edges* in the last layer have **unit length**, while the *cross-edges* in the same layer have lengths that **scale** linearly with the input size N . This disparity is problematic, as it causes a *loss of synchronization* in the information flow. Nevertheless, this length grows exponentially.
- The length of the **longest path** from any input to any output is linear in N , namely $2(N - 1)$, and it can be computed by evaluating the diagonal of the square having side length equal to $\sqrt{2}(N - 1)$, shown in figure [which is](#)

$$\sqrt{2 \cdot (\sqrt{2}(N - 1))^2} = \sqrt{2 \cdot 2(N - 1)^2} = \sqrt{4 \cdot (N - 1)^2} = 2(N - 1)$$

- placeholder
- The value of the **area** of this layout is good, which is

$$(\sqrt{2}(N - 1))^2 + o(N^2) = 2N^2 + o(N^2)$$

Later studies found that the **area** of this layout is *inaccurate* because the **slanted** lines — rotated by 45° — that define this layout cannot be produced by machines of the fabrication process. In fact, machines can only create *horizontal and vertical lines*, which means that the actual layout on a board would occupy significantly more area.

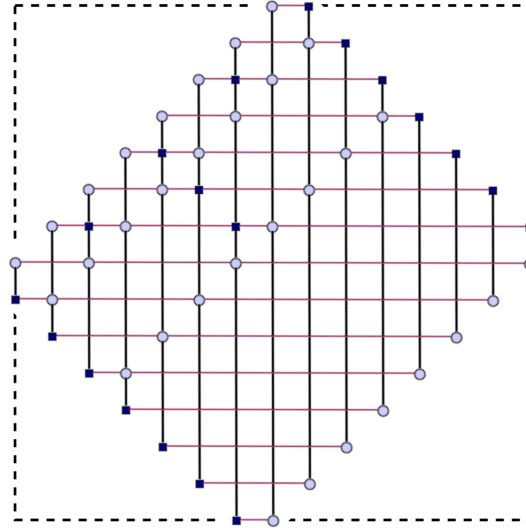


Figure 2.13: The *actual* Wise butterfly layout.

The area of this layout can be evaluated by calculating the area of this *bigger* square, which has side length $2(N - 1)$:

$$(2(N - 1))^2 = 4N^2 + o(N^2)$$

Additionally, *knock-knees* are not avoided, but arranged in the layout thanks to devices that enlarge the layout area even more.

2.1.4 Layered layout

In 2000 Even et al. [5] presented a new layout, based on the **layered cross product** between graphs, which is described below.

Definition 2.5: Layered graph

A **layered graph** of $l + 1$ layers $G = (V_0, \dots, V_l, E)$ consists of $l + 1$ layers of nodes, where V_i is the i -th node layer, and each edge $(u, v) \in E$ connect u and v if and only if $u \in V_i$ and $v \in V_{i+1}$ — i.e. they belong to adjacent layers.

Example 2.3 (Layered graphs). The following is an example of a layered graph.

placeholder

add pic

Definition 2.6: Layered cross product

Given two layered graphs $G_1 = (V_0^1, \dots, V_l^1, E^1)$ and $G_2 = (V_0^2, \dots, V_l^2, E^2)$ of $l + 1$ layers, the **layered cross product** (LCP) is a new *layered graph*

$$G = G_1 \times G_2 := (V_0, \dots, V_l, E)$$

defined as follows:

- for each $i \in [0, l]$, $V_i := V_i^1 \times V_i^2$, i.e. each layer in G is the *cartesian product* of the corresponding two layers in G_1 and G_2 ;
- $((u^1, u^2), (v^1, v^2)) \in E \iff (u^1, u^2) \in E^1 \wedge (v^1, v^2) \in E^2$, there is an edge between two nodes of G if and only if the corresponding nodes are connected in the original graphs.

Note that the LCP is not commutative.

Example 2.4 (LCPs). The following is an example of LCP between two graphs.

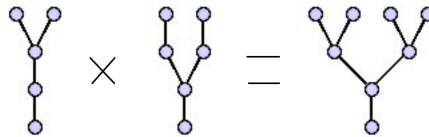


Figure 2.14: An LCP between two graphs.

LCPs are particularly useful because Even et al. [6] demonstrated that *various topologies* can be defined as LCPs of *simpler structures*, such as trees. Specifically, it can be shown that butterfly networks are LCPs of **two complete binary trees**, one oriented upward and the other downward.

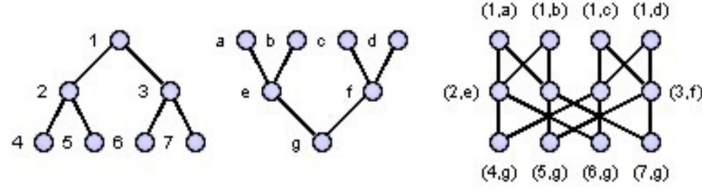


Figure 2.15: The LCP that defines the 2-dimensional butterfly network.

Interestingly, the LCP of two graphs can be evaluated through a method known as **Projection Methodology** (PM), as illustrated below.

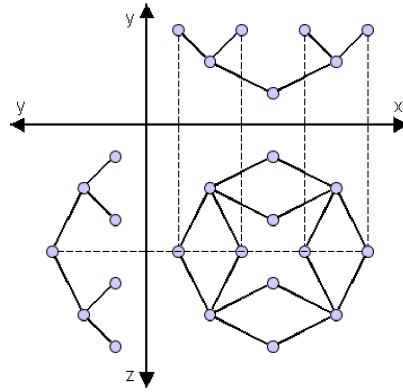


Figure 2.16: The LCP of the two graphs is obtained by this projection.

It is important to note that the PM may produce results that *do not align* with the requirements of the Thompson model. For instance, while the projection above still represents the same butterfly network as before, it is not an **orthogonal drawing**.

Consider the following projection plane.

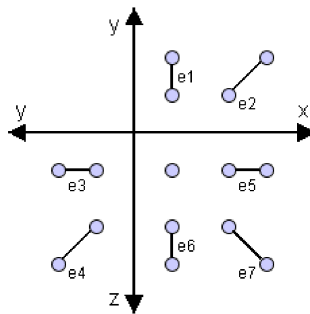


Figure 2.17: Possible edge cross products.

From these projections, it is evident that:

- the product of *two diagonal edges* yields a *diagonal edge*, which is *not allowed*;

- the product of a *vertical edge* and a *diagonal edge* yields a *vertical edge*, which is allowed;
- the product of a *diagonal edge* and a *vertical edge* yields a *horizontal edge*, which is allowed;
- the product of *vertical edges* yields *two overlapping points*, which is *not allowed*.

Therefore, to achieve a *valid layout* using the PM, it is essential to ensure that the product of *two diagonal edges* or *two vertical edges* **never occurs**.

Note that this is not the only problem that may occur in layouts generated through the PM.

Definition 2.7: Consistent edges

Two edges e_1 and e_2 are said to be **consistent** if the open intervals of their projections along the same axis are *disjoint*.

Example 2.5 (Consistent edges). Consider the following edges and their projections on the x -axis:

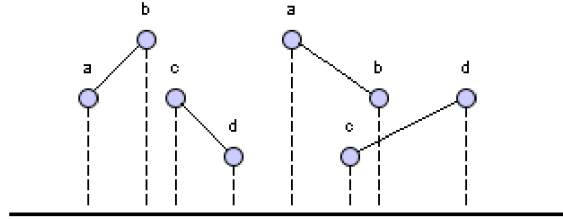


Figure 2.18: Consistent and inconsistent pair of edges, from left to right.

the first two edges are *consistent*, while the other two are not.

Consistency of edges in the input graphs must be checked, to avoid overlapping wires in the resulting graph. In particular, *two cases* must be avoided.

In this first scenario, in G_1 there are two inconsistent edges in the same layer i , and there is an edge in G_2 in layer i as well. This produces two overlapping edges in the projection.

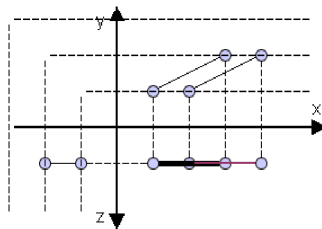


Figure 2.19: First inconsistency case

Note that this situation arises only when the edge in G_2 is parallel to the x -axis, as it cannot be drawn diagonally, since the inconsistent edges in G_1 are already diagonal, and — as previously discussed — the cross product between two diagonal edges must be avoided.

The second scenario occurs when in G_1 there are two inconsistent edges in different layers i_1 and i_2 , and there are collinear edges in G_2 in layers i_1 and i_2 as well. This produces two overlapping edges in the projection.

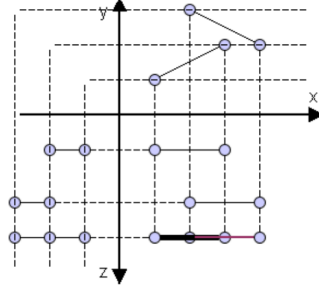


Figure 2.20: Second inconsistency case

All the required constraints are summarized in the next proposition.

Proposition 2.1: Valid PM layouts

Given two graphs G_1 and G_2 , the PM between them generates **valid layouts**, i.e. in the resulting graph

1. every edge lies on grid lines
2. at most one node is mapped to each grid point
3. no pair of edges overlap

if and only if

1. the cross product of any pair of edges $e_1 \in E_1$ and $e_2 \in E_2$ is such that exactly one between e_1 and e_2 is drawn diagonally
2. for each $i \in [0, l]$, it holds that

$$\bigcap_{i=0}^l \{ \langle u_x, v_z \rangle \mid u \in V_i^1, v \in V_i^2 \} = \emptyset$$

where $\langle u_x, v_z \rangle$ is the node at the intersection of the projection lines of u and v along the x and z axes, respectively

3. there are no edges in G_2 on layers that contain inconsistent edges in G_1 , and no collinear edges in different layers of G_2 in which G_1 has inconsistent edges

respectively.

For the first constraint, a solution is to **double** the number of layers, such that the edges in the drawing of G_1 are diagonal in *odd* layers, and straight in the *even* layers, while the edges in the drawing of G_2 are straight in the *odd* layers, and diagonal in the *even* layers.

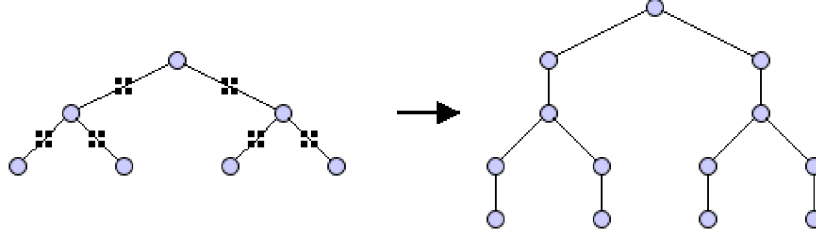


Figure 2.21: A complete layered binary tree with the nodes doubled.

The second constraint can be addressed by ensuring that no pair of nodes in the drawing of the first (or second) graph, except for the two endpoints of the same straight edge, share the same x -coordinate (or z -coordinate). This is always achievable by appropriately enlarging the drawings of the two graphs.

This condition is more difficult to enforce and presents a significant limitation of this technique. For this reason, the focus of this work is restricted to networks where each LCP is calculated from two complete layered binary trees, with double the number of nodes.

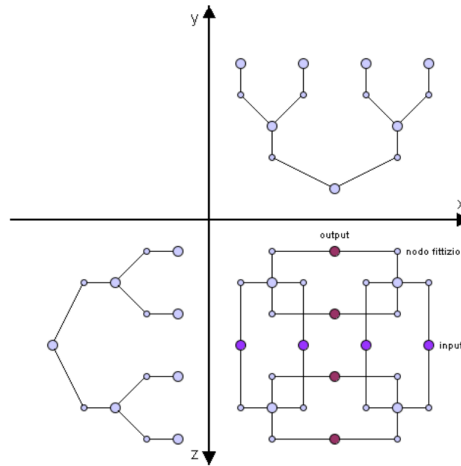


Figure 2.22: The LCP of two such trees.

This figure represents the **planar layout** of a butterfly network, which adheres to all the previously outlined constraints. Moreover

- it is symmetric;
- it is a square with side length $2(N - 1)$, therefore the area is $4N^2 + o(N^2)$ — note that this area is worse than the Wise layout (discussed in [Section 2.1.3](#)) because in

this case the whole area is filled, whereas the Wise layout only uses a portion of such a big area;

- all the edges on the same layer have the same length;
- unfortunately, input and output nodes do not lie on the boundaries.

2.1.5 Optimal area of the butterfly network

3

The worm propagation prevention problem

A [computer worm](#) is a type of malware designed to self-replicate and spread across networks without needing a host program. Unlike *viruses*, which require human action to propagate, **worms** use computer networks to exploit security vulnerabilities in target systems, infiltrating and duplicating themselves automatically.

Once a worm gains access, it can spread rapidly to other devices, causing network slowdowns, data breaches, or system damage, depending on the worm's purpose. Additionally, worms can carry payloads that steal sensitive data, install other forms of malware, or create backdoors for unauthorized access. Effective network security measures, such as patching vulnerabilities and monitoring traffic, are essential to prevent worm attacks.

The harmful effects of a worm can be broadly classified into *two categories*:

- **Direct damage.** These are caused by the worm's execution on the victim's system. It may lead to system instability, data corruption, file deletion, or even the theft of sensitive information. The worm might consume significant system resources, slowing down performance or rendering the machine unusable. They consist solely of instructions to replicate themselves, and typically do not cause severe direct damage beyond consuming computational resources, which can degrade system performance. However, more advanced *direct damage* worms often disrupt the proper functioning of security software, such as antivirus programs and firewalls, making it harder to detect and remove the malware. This interference can severely hinder the normal operation of the infected machine. In many cases, they also act as *carriers* for the automatic installation of [backdoors](#) or [keyloggers](#), which can later be exploited by attackers or other forms of malware, further compromising the system's security.
- **Indirect damage.** These arise from the methods the worm uses to spread. For example, worms can generate a large volume of traffic while replicating, which can overwhelm networks, disrupt email systems, and lead to costly downtime or loss of productivity. Additionally, their use of social engineering tactics might result

in reputational damage or further security breaches. Their damages result from the widespread infection of many computers across a network, creating cascading effects. These type of worms send numerous email messages during replication, flooding inboxes and contributing to email spam, which wastes valuable bandwidth and user attention. They exploit known vulnerabilities in certain software which can lead to software malfunctions, causing instability in the operating system. This often results in system crashes, forced reboots, or even shutdowns, further disrupting normal operations..

Assume that the time required to transmit information over any connection in a network is constant and denoted as T . If a worm successfully infects a set of nodes C , and the worm can spread to all other nodes in the network in a single step, then the entire network will be **infected** within time T . Therefore, we are interested in finding the set of nodes C that can lead to a fully infected network.

The property that every edge in the network is incident to at least one node in the infected set C ensures that the entire network can be infected after the first propagation step. This condition is *sufficient* (though not *necessary*) to guarantee that the worm will spread to all nodes in the next step.

From a network manager's perspective, any filter implemented to protect against first-order worm attacks typically reduces communication efficiency. Therefore, **minimizing** the number of filters is crucial to strike a balance between security and maintaining communication speed.

Note that, in reality, the situation is more complicated because large-scale networks tend to have *dynamic connections*, meaning the structure of the network changes over time, which can affect the speed and manner of the worm's propagation.

3.1 The vertex cover problem

The problem of finding the set C of vertices discussed earlier, where each edge in the network is incident to at least one vertex in C , can be reduced to finding the **minimal vertex cover** of the network graph.

Definition 3.1: Vertex cover

Given a graph G , a **vertex cover** for G is a set of vertices $C \subseteq V(G)$ such that every edge in G is incident to at least one vertex in C . Using symbols

$$\forall (u, v) \in E(G) \quad u \in C \vee v \in C$$

Note that the minimal vertex cover is not unique. Moreover, for any graph G , $V(G)$ is trivially a vertex cover for G . To find the minimal vertex cover is not trivial, because there are $\mathcal{P}(V(G)) = 2^{V(G)}$ possible subsets of vertices to check.

Example 3.1 (Vertex covers). The following are two examples of minimal vertex covers for the given graph:

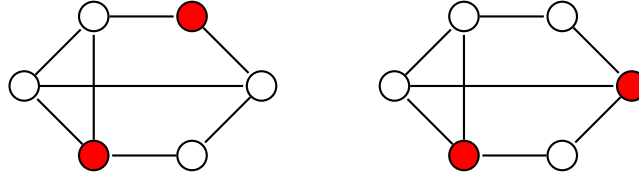


Figure 3.1: Two minimal vertex covers for a graph.

For the reasons mentioned, every minimal vertex cover serves as an excellent starting point for a worm's propagation within a network. Protecting the computers corresponding to the nodes in the **minimal vertex cover** of the communication graph is crucial. This strategy ensures that every edge in the graph is monitored, thus preventing a worm from exploiting vulnerabilities in unprotected nodes.

Moreover, if the graph has multiple minimum vertex covers, it is essential to identify and protect *at least* the computers that lie in the **intersection** of all these covers. This intersection represents the most secure nodes, as they are critical in preventing the spread of the worm across all potential configurations of the network.

The following definition provides the **decisional version** of the minimal vertex cover problem.

Definition 3.2: Minimal Vertex Cover (VC) problem

Given a graph G , and an integer $k \geq 0$, is there a vertex cover C for G such that $|C| \leq k$?

The VC problem is one of Karp's 21 **NP-Complete** problems [9], which are a collection of well-known computational problems that were classified as **NP-Complete** shortly after the introduction of the Cook theorem [3].

The VC problem can be also formulated as a 0-1 **integer linear program** (ILP):

- For every node $v \in V(G)$, we define the variable x_v .
- For every edge $(i, j) \in E(G)$ we add the constraint $x_i + x_j \geq 1$. This constraint enforces that at least one between x_i and x_j has to be set to 1.

We get that:

$$\begin{aligned} \min \quad & \sum_{i=1}^n x_i \\ & x_i + x_j \geq 1 \quad \forall (i, j) \in E(G) \\ & x \in \{0, 1\}^n \end{aligned}$$

This formulation is a reduction from VC to ILP, implying that the latter is **NP-Complete** as well.

Despite the fact that the VC problem is **NP-Complete**, it is possible to find an *approximate*

solution in polynomial time by leveraging its ILP formulation, by employing an ILP approximated solution.

3.1.1 Approximation algorithms

There are multiple algorithms for finding approximated solutions for the VC problem. The first algorithm presented is a naïve solution, based on a **greedy** approach.

Algorithm 3.1: First greedy AVC

Given an undirected graph G , the algorithm finds an approximated minimal vertex cover for G .

```

1: function FIRSTGREEDYAVC( $G$ )
2:    $V' := \emptyset$ 
3:    $E' := E(G)$ 
4:   while  $E' \neq \emptyset$  do
5:     Choose  $(i, j) \in E'$ 
6:      $V' = V' \cup \{i\}$ 
7:     for  $(i, k) \in E(G)$  do                                ▷ any edge having  $i$  as an endpoint
8:        $E' = E' - \{(i, k)\}$ 
9:     end for
10:  end while
11:  return  $V'$ 
12: end function

```

Idea. At each iteration of the algorithm, an edge $(i, j) \in E'$ is chosen randomly, then i is added to the vertex cover set V' , and any edge having i as an endpoint is removed from E' . This ensures that V' is a vertex cover, though it may not be minimal.

Cost analysis. Edge has to be explored or removed by the algorithm, therefore the cost is $O(n + m)$.

Algorithm 3.2: Second greedy AVC

Given an undirected graph G , the algorithm finds an approximated minimal vertex cover for G .


```

1: function SECONDGREEDYAVC( $G$ )
2:    $V' := \emptyset$ 
3:   while  $E(G) \neq \emptyset$  do
4:      $v \in \arg \max_{v \in V(G)} \deg(v)$ 
5:      $V' = V' \cup \{v\}$ 
6:     for  $(u, v) \in E(G)$  do                                ▷ any edge having  $v$  as an endpoint
7:        $G.\text{remove\_edge}(u, v)$ 
8:     end for
9:   end while
10:  return  $V'$ 
11: end function

```

Idea. At each step of the algorithm, the vertex with the highest degree v is chosen from the current set of vertices $V(G)$, then v is added to the vertex cover V' , and any edge having v as an endpoint is removed from G . This ensures that V' is a vertex cover, though it may not be minimal.

Cost analysis. At each iteration, the cost of finding v is $O(m)$, and because the cost of removing an edge from G is $O(n + m)$, we have that the cost of the algorithm is $O(m(n + m))$.


Note that algorithm can produce a vertex cover V' whose cardinality is *very far* from optimum. In fact, consider the following graph: 

The upper row of nodes consists of r nodes, and the lower row consists of

- r nodes of degree 1
- $\left\lfloor \frac{r}{2} \right\rfloor$ nodes of degree 2 ...
- in general, $\left\lfloor \frac{r}{i} \right\rfloor$ nodes of degree i

meaning that the total number of nodes is

$$n = r + \sum_{i=1}^r \left\lfloor \frac{r}{i} \right\rfloor \leq r + r \sum_{i=1}^r \frac{1}{i} = \Theta(r \log r)$$

(note that the [harmonic sum](#) can be approximated by $\Theta(\log r)$). Although the **optimal** minimal vertex cover is the *upper row* itself, consisting of r nodes, it may happen that the algorithm chooses the *lower row* as vertex cover, as shown in the following figure 

This means that there is an approximation ratio of

$$\frac{\Theta(r \log r)}{r} = \Theta(\log r)$$

However, there are better algorithms that can find approximated minimal covers V' for a given graph G such that $|V'| \leq 2|V^*|$, where V^* is a minimal vertex cover.

Algorithm 3.3: 2-approximation VC

Given an undirected graph G , the algorithm finds an approximated minimal vertex cover V' for G , such that $|V'| \leq 2|V^*|$, where V^* is a minimal vertex cover.

```

1: function 2APPROXVC( $G$ )
2:    $V' := \emptyset$ 
3:    $E' := E(G)$ 
4:   while  $E' \neq \emptyset$  do
5:     Choose  $(i, j) \in E'$ 
6:      $V' = V' \cup \{i, j\}$ 
7:     for  $(i, k) \in E(G)$  do                                 $\triangleright$  any edge having  $i$  as an endpoint
8:        $E' = E' - \{(i, k)\}$ 
9:     end for
10:    for  $(j, h) \in E(G)$  do                                 $\triangleright$  any edge having  $j$  as an endpoint
11:       $E' = E' - \{(j, h)\}$ 
12:    end for
13:  end while
14:  return  $V'$ 
15: end function

```

Idea. The algorithm computes as the Algorithm 3.1, but both endpoints of the chosen edge are considered in the removal step.

Cost analysis. The cost of the algorithm is the same as the Algorithm 3.1, which is $O(n + m)$.

Proof. We will prove that any V' returned from the algorithm is such that $|V'| \leq 2|V^*|$ for some minimal vertex cover V^* . Note that, by construction, V' is a vertex cover for G .

Let A be the set of the edges *chosen* from E' . For each edge $(i, j) \in A$, i and j are added to V' by the algorithm, therefore

$$|V'| = 2|A|$$

Moreover, all the edges having either i or j as endpoint are removed from E' , thus edges in A cannot be incident, which means that there exists a minimal vertex cover V^* such that

$$|A| \leq |V^*|$$

Finally, we have that

$$|V'| = 2|A| \leq 2|V^*|$$

□

Bibliography

- [1] Richard Bellman. “On a Routing Problem”. In: *Quarterly of Applied Mathematics* 16.1 (1958), pp. 87–90. ISSN: 0033569X, 15524485. (Visited on 10/19/2024).
- [2] 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](https://doi.org/10.1016/0020-0190(80)90034-4). URL: [http://dx.doi.org/10.1016/0020-0190\(80\)90034-4](http://dx.doi.org/10.1016/0020-0190(80)90034-4).
- [3] Stephen A. Cook. “The complexity of theorem-proving procedures”. In: *Proceedings of the third annual ACM symposium on Theory of computing - STOC '71*. STOC '71. ACM Press, 1971, 151–158. DOI: [10.1145/800157.805047](https://doi.org/10.1145/800157.805047). URL: <http://dx.doi.org/10.1145/800157.805047>.
- [4] E.W. Dijkstra. “A Note on Two Problems in Connexion with Graphs.” In: *Numerische Mathematik* 1 (1959), pp. 269–271.
- [5] Guy Even et al. “Embedding interconnection networks in grids via the layered cross product”. In: *Networks* 36.2 (2000), 91–95. ISSN: 1097-0037. DOI: [10.1002/1097-0037\(200009\)36:2<91::aid-net3>3.0.co;2-4](https://doi.org/10.1002/1097-0037(200009)36:2<91::aid-net3>3.0.co;2-4). URL: [http://dx.doi.org/10.1002/1097-0037\(200009\)36:2<91::AID-NET3>3.0.CO;2-4](http://dx.doi.org/10.1002/1097-0037(200009)36:2<91::AID-NET3>3.0.CO;2-4).
- [6] Shimon Even et al. “Layered cross product - A technique to construct interconnection networks”. In: *Networks* 29.4 (July 1997), 219–223. ISSN: 1097-0037. DOI: [10.1002/\(sici\)1097-0037\(199707\)29:4<219::aid-net5>3.0.co;2-i](https://doi.org/10.1002/(sici)1097-0037(199707)29:4<219::aid-net5>3.0.co;2-i). URL: [http://dx.doi.org/10.1002/\(SICI\)1097-0037\(199707\)29:4<219::AID-NET5>3.0.CO;2-I](http://dx.doi.org/10.1002/(SICI)1097-0037(199707)29:4<219::AID-NET5>3.0.CO;2-I).
- [7] Robert W. Floyd. “Algorithm 97: Shortest path”. In: *Commun. ACM* 5.6 (June 1962), p. 345. ISSN: 0001-0782. DOI: [10.1145/367766.368168](https://doi.org/10.1145/367766.368168).
- [8] L. R. Ford. *Network Flow Theory*. Santa Monica, CA: RAND Corporation, 1956.
- [9] Richard M. Karp. “Reducibility among Combinatorial Problems”. In: *Complexity of Computer Computations*. Springer US, 1972, 85–103. ISBN: 9781468420012. DOI: [10.1007/978-1-4684-2001-2_9](https://doi.org/10.1007/978-1-4684-2001-2_9). URL: http://dx.doi.org/10.1007/978-1-4684-2001-2_9.
- [10] 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](https://doi.org/10.1109/SFCS.1980.13). URL: <http://dx.doi.org/10.1109/SFCS.1980.13>.
- [11] E.F. Moore. *The Shortest Path Through a Maze*. Bell Telephone System. Technical publications. monograph. Bell Telephone System., 1959. URL: <https://books.google.it/books?id=IVZBHAAACAAJ>.
- [12] 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](https://doi.org/10.1145/800135.804401). URL: <http://dx.doi.org/10.1145/800135.804401>.
- [13] 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](https://doi.org/10.1109/tc.1981.6312176). URL: <http://dx.doi.org/10.1109/TC.1981.6312176>.
- [14] Stephen Warshall. “A Theorem on Boolean Matrices”. In: *J. ACM* 9.1 (Jan. 1962), 11–12. ISSN: 0004-5411. DOI: [10.1145/321105.321107](https://doi.org/10.1145/321105.321107).
- [15] David S. Wise. “Compact Layouts of Banyan/FFT Networks”. In: *VLSI Systems and Computations*. Springer Berlin Heidelberg, 1981, 186–195. ISBN: 9783642684029. DOI: [10.1007/978-3-642-68402-9_21](https://doi.org/10.1007/978-3-642-68402-9_21). URL: http://dx.doi.org/10.1007/978-3-642-68402-9_21.
- [16] Chi-Hsiang Yeh et al. “Efficient VLSI layouts of hypercubic networks”. In: *Proceedings. Frontiers '99. Seventh Symposium on the Frontiers of Massively Parallel Computation*. Vol. 9. IEEE, 1999, 98–105. DOI: [10.1109/fmpc.1999.750589](https://doi.org/10.1109/fmpc.1999.750589). URL: <http://dx.doi.org/10.1109/FMPC.1999.750589>.