"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

November 13, 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:

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.

<div style="text-align: right; font-size: 3em;">1</div>

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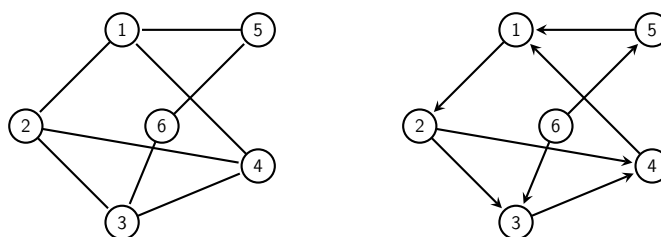
Figure 1.1: On the left: a simple graph. On the right: a simple digraph.

Graphs were born in 1736, when Euler used them to 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?
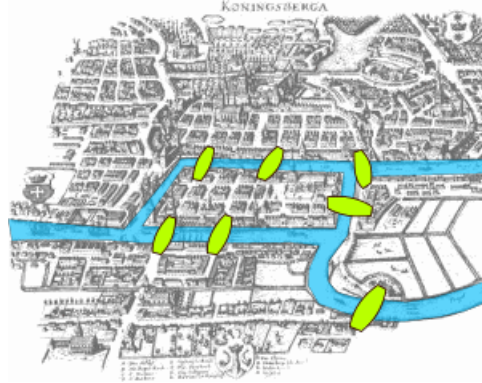
Figure 1.2: The city of Königsberg and its seven bridges.

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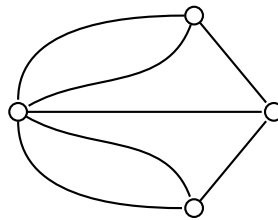
Figure 1.3: The multi-graph representing the *Seven Bridges of Königsberg* problem.

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 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 passing through it. 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* $w : E(G) \to \mathbb{R}$ 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 its edges.

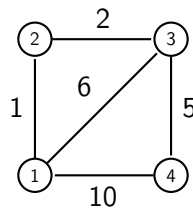**Example 1.1** (Weighted graphs)**.** The following is an example of a graph with weights on the edges.

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 $P = v_1, \ldots, v_k$ 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 $\mathrm{dist}(u, v)$, is the minimum weight of all the paths $u \to v$ of G.

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

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 [19].

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 $d : V(G) \to \mathbb{R}$ 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 [3] and Ford [14] independently, which described the following algorithm.

Algorithm 1.1: Bellman-Ford

Given a graph G, a weight function $w : E(G) \to \mathbb{R}$ 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 ∈ V(G) do
 3:         d(v) := +∞
 4:     end for
 5:     p := [NULL, . . . , NULL]
 6:     for i ∈ [1, n − 1] do
 7:         for (u, v) ∈ 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 does 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 the distance vector routing protocol, an iterative, asynchronous and distributed protocol.

One year later, Dijkstra [10] 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 $w : E(G) \to \mathbb{R}^+$ 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 ∈ V(G) do
 3:         d(v) := +∞
 4:     end for
 5:     p := [NULL, ..., NULL]
 6:     S := ∅
 7:     Q := V(G)                              ▷ Q is based on d
 8:     while Q ≠ ∅ do
 9:         u := Q.extract_min()
10:         S = S ∪ {u}
11:         for (u, v) ∈ 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.update()                 ▷ updating v's value in Q
16:             end if
17:         end for
18:     end while
19:     return p
20: 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 choosen, 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 independetly by Floyd [13] and Warshall [28], 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]` $= \min\left(\texttt{dist[u][v]}, \texttt{dist[u][k]} + \texttt{dist[k][v]}\right)$
> 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 the 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-**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.
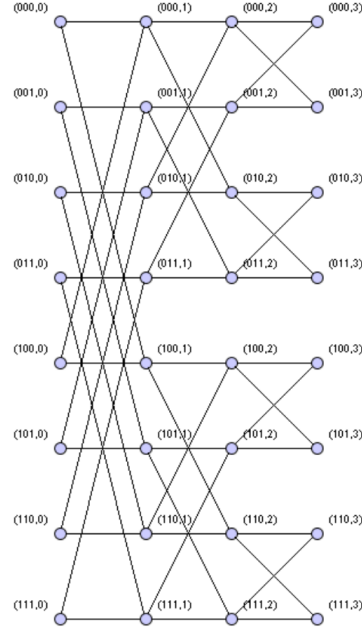
Figure 1.5: A butterfly network.

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.
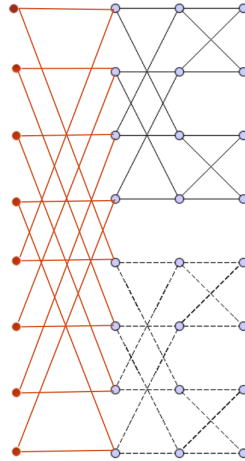

Figure 1.6: The recursive structure of butterfly networks.

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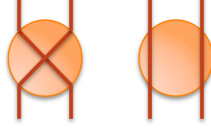

Figure 1.7: A butterfly network node.

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.
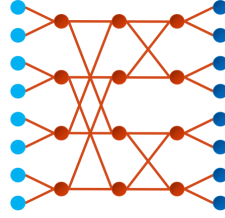
Figure 1.8: An extended butterfly network.

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 wᵢ == 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 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

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

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.
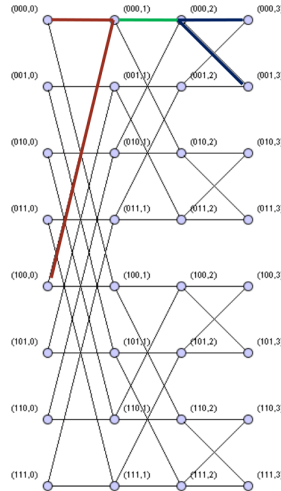
In the worst case, π can be such that *each greedy path starting from a leaf on the left tree and ending on a leaf on the right tree traverses e*. Note that the number of such paths is precisely the number of leaves of the left complete binary tree, namely $2^{\frac{n-1}{2}} = \sqrt{\frac{N}{2}}$.

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 $0 < i \le n$, 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 $n_i \le 2^{i-1}$. Analogously, on the second half of the butterfly, n_i is bounded by the number of leaves of the right complete binary tree, therefore $n_i \le 2^{n-i}$. Note that both this results hold because n is odd.

Note that any packet that needs 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

$$\sum_{i=1}^{n} (n_i - 1) = \sum_{i=1}^{\frac{n+1}{2}} (n_1 - 1) + \sum_{i=\frac{n+1}{2}+1}^{n} (n_i - 1)$$

$$\leq \sum_{i=1}^{\frac{n+1}{2}} \left(2^{i-1} - 1\right) + \sum_{i=\frac{n+3}{2}}^{n} \left(2^{n-i} - 1\right)$$

$$= \sum_{j=0}^{\frac{n+1}{2}-1} \left(2^j - 1\right) + \sum_{j=0}^{\frac{n-3}{2}} \left(2^j - 1\right)$$

$$= \sum_{j=0}^{\frac{n+1}{2}-1} 2^j + \sum_{j=0}^{\frac{n-3}{2}} 2^j - n$$

$$= 2^{\frac{n+1}{2}} - 1 + 2^{\frac{n-1}{2}} - 1 - n$$

$$\leq O(\sqrt{N}) - n$$

$$\leq O(\sqrt{N})$$

$\square$

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 packets' 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.*

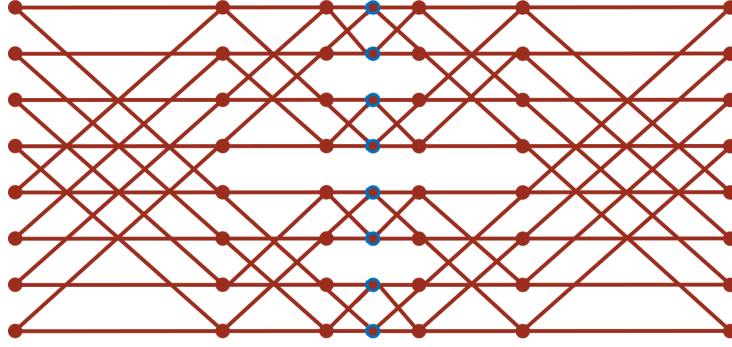**Example 1.3** (Beneš network)**.** The following is an example of a Beneš network.

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 networks — 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} outputs linked to the $2n$-th layer.

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, hence 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.
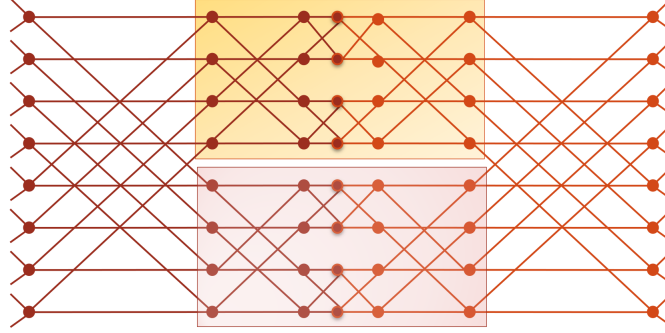

Figure 1.10: Subnetworks of a Beneš network.

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 to 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 subnetworks;

- 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.

$\square$

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 $1 \leq i \leq 2^{n-1}$.
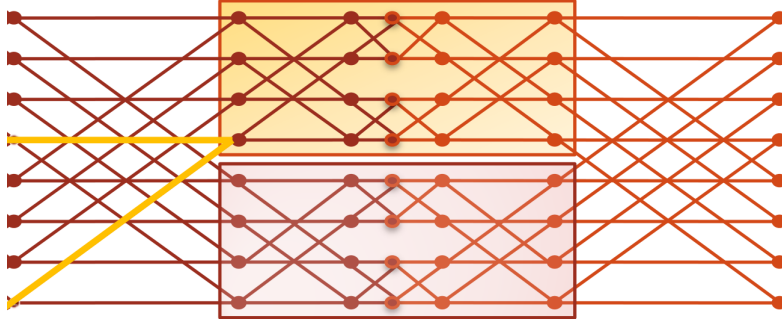
Figure 1.11: Mates in this type of Beneš network.

$\square$

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 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.

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 [25], 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 a 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.
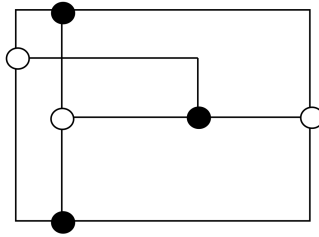
Figure 2.1: An orthogonal drawing.

- **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: A 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.
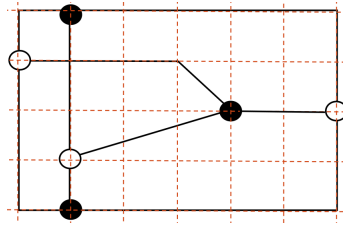
- **Area minimization**: silicon is a *costly material*, making it essential to minimize the layout area of integrated circuits. Compact layouts not only reduce material costs, but also enhance performance by shortening wire lengths, which decreases signal delay and power consumption. Therefore, **area 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 is 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.

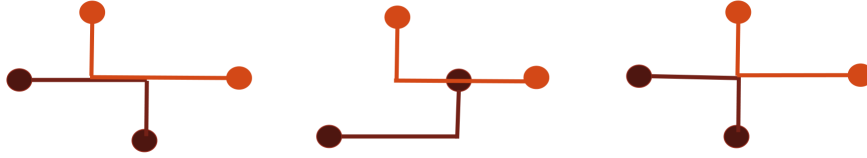- *overlappings*, *node-edge crosses* and *"knock-knees"* are not allowed;

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 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.

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 [17] and Valiant [27], which employs H trees, which are defined as follows.

> **Definition 2.4: H tree**
>
> An **H tree** organizes a complete binary 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 sufficient
>
>
>
> 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 is even, otherwise use the rightmost construction if h is odd.
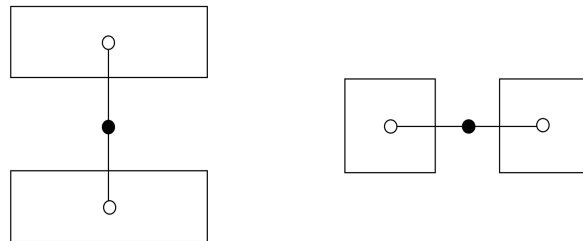>
>
>
> Figure 2.5: The inductive step of the inductive H tree construction.

Example 2.1 (H trees)**.** The following figure shows an example of an H tree of height $h = 4$.
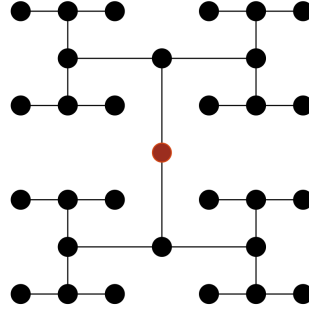
Figure 2.6: H tree of height $h = 4$.

Leiserson [17] and Valiant [27] 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. [4] 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
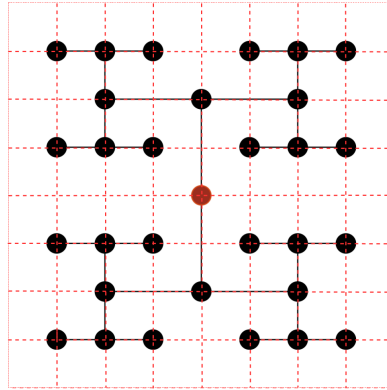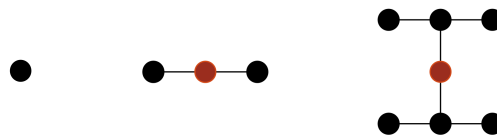
- for $h = 0$, $l_0 = w_0 = 2 \implies A_0 = l_0 \cdot w_0 = 2 \cdot 2 = 4 = 2(1 + 1)$

- 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* 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

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

and analogously

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

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

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

$\square$

2.1.2 The collinear layout

The Thompson's 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. [30], 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*
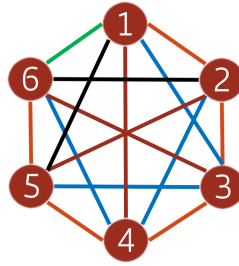
Figure 2.9: A 6-clique.

and let a **link of type-**i be any edge between two nodes whose labels differ by 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.
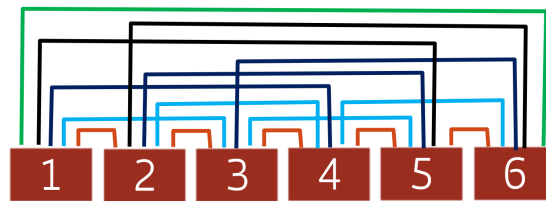
Figure 2.10: Arrangement of links in tracks

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

$$\sum_{i=1}^{n-1} \min(i, n-i) = \sum_{i=1}^{\frac{n}{2}} i + \sum_{i=\frac{n}{2}+1}^{n-1} (n-i)$$

$$= \sum_{i=1}^{\frac{n}{2}} i + \sum_{j=1}^{\frac{n}{2}-1} j \qquad (j = n-i)$$

$$= \frac{1}{2}\left[\frac{n}{2}\left(\frac{n}{2}+1\right) + \frac{n}{2}\left(\frac{n}{2}-1\right)\right]$$

$$= \frac{n^2}{4}$$

placeholder ⟨incomplete⟩

Theorem 2.2: Thompson's theorem

Given a VLSI design, and its corresponding graph G, the area occupied by the wires and the nodes of the design is at least $\frac{w^2}{4}$, where w is G's bisection width.

*Proof.* The bound of the area of the VLSI design will be proved by counting the minimum number of occupied squares of the grid.

Consider the VLSI design on a grid as a Cartesian plane, and vertical lines of the form $x = a$. Each possible $x = a$ split the vertices of the network into three separate subsets:

- L, which contains the vertices on the left of the vertical line

- R, which contains the vertices on the right of the vertical line

- S, which contains the vertices that lie right on the vertical line

In particular, by monotonicity, there exists an a such that $|L|+|S| \geq \frac{n}{2}$ and $|R|+|S| \geq \frac{n}{2}$. Clearly, if $|S| = 0$, the line $x = a$ cuts the graph into two sets of vertices L and R, which must cut at least w edges (by definition of *bisection width*). Otherwise, if $|S| \neq 0$, S can be split into two subsets S_1 and S_2, such that $|L \cup S_1| = |R \cup S_2| = \frac{n}{2}$, and the vertical line will still cut at least w edges.

The line $x = a$ is said to account for the w square units of area of wire and vertices that lie within $\frac{1}{2}$ unit distance of it.

Consider a "zig-zag" defined as follows:

$$Z_1(x) := \begin{cases} a - 1 & y \geq b_1 \\ a - 1 \leq x \leq a + 1 & y = b_1 \\ a + 1 & y \leq b_1 \end{cases}$$

where b_1 is such that Z_1 still bisects the graph, therefore it cuts at least w edges. Note that the horizontal segment of Z_1 may cut at most 2 wires, therefore its vertical sections will cross at least $w - 2$ wires.

Consider each possible zig-zag

$$Z_k(x) := \begin{cases} a - k & y \geq b_k \\ a_k \leq x \leq a + k & y = b_k \\ a + k & y \leq b_k \end{cases}$$

where b_k is such that Z_k still bisects the graph, therefore it cuts at least w edges. Since the horizontal segment will cut $2k$ edges, the vertical sections of Z_k will cut $w - 2k$ edges.

Finally, since $\left\lfloor \dfrac{w}{2} \right\rfloor$ zig-zags can be drawn on the graph (by definition of *bisection width*), the total area of wire and vertices of the VLSI design is at least

$$\sum_{k=0}^{\left\lfloor \frac{w}{2} \right\rfloor} > \frac{w^2}{4}$$

$\square$

2.1.3 The Wise layout

In a paper published by Wise [29], it was proposed a *different layout* for the butterfly network (discussed in Section 1.3.1), which is shown in the following figure.
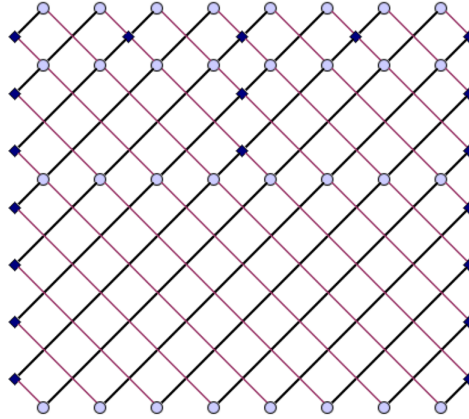

Figure 2.11: The alternative layout of the butterfly network.

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*.
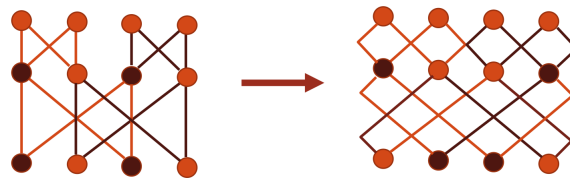
Figure 2.12: Rearrangement to get the Wise layout.

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

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

- placeholder

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

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

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 $2(N - 1)$:

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

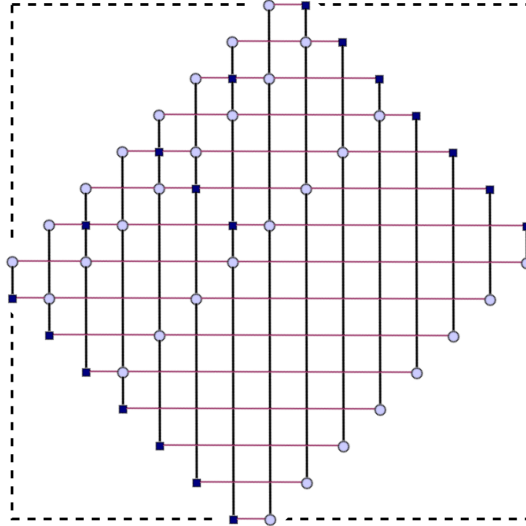Additionally, *knock-knees* are not avoided, but arranged in the layout thanks to devices that enlarge the layout area even more.

2.1.4 Layered layout

In 2000 Even et al. [11] 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 $G = (V_0, \ldots, V_l, E)$ consists of $l + 1$ layers of nodes, where V_i is the i-th node layer, and each edge $(u, v) \in E$ connects 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 $G_1 = (V_0^1, \ldots, V_l^1, E^1)$ and $G_2 = (V_0^2, \ldots, V_l^2, E^2)$ of $l + 1$ layers, the **layered cross product** (LCP) is a new *layered graph*

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

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;

- $((u^1, u^2), (v^1, v^2)) \in E \iff (u^1, u^2) \in E^1 \wedge (v^1, v^2) \in E_2$, i.e. there is an edge between two nodes of G if and only if the corresponding nodes were connected in the original graphs.

Note that the LCP is not commutative.

Example 2.4 (LCPs)**.** The following is an example of an LCP between two graphs.
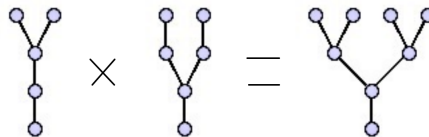
Figure 2.14: An LCP between two graphs.

LCPs are particularly useful because Even et al. [12] showed 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.
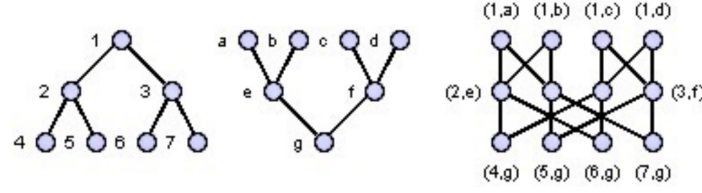
Figure 2.15: The LCP that defines the 2-dimensional butterfly network.

Interestingly, the LCP of two graphs can be evaluated through a method known as **Projection Methodology** (PM), as illustrated below.
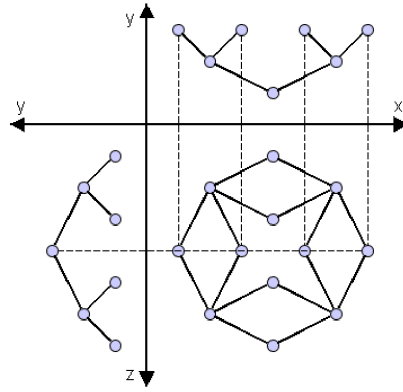
Figure 2.16: The LCP of the two graphs is obtained by this projection.

It is important to note that the PM may produce results that *do not align* with the requirements of the Thompson's 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.
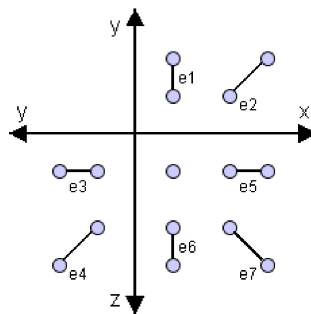
Figure 2.17: Possible edge cross products.

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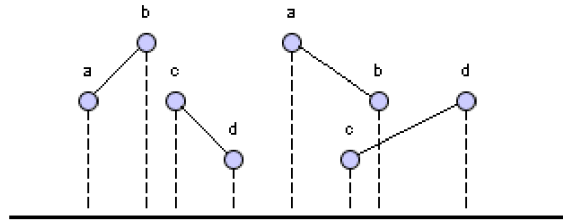
Figure 2.18: Consistent and inconsistent pair of edges, from left to right.

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.
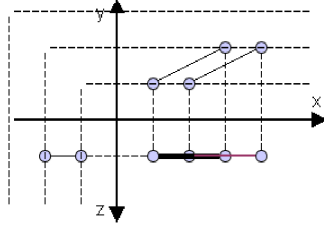
Figure 2.19: First inconsistency case

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.
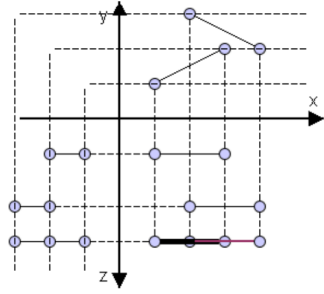
Figure 2.20: Second inconsistency case

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
>
> $$\bigcap_{i=0}^{l} \{\langle u_x, v_z \rangle \mid u \in V_i^1, v \in V_i^2\} = \varnothing$$
>
> where $\langle u_x, v_z \rangle$ is the node at the intersection of the projection lines of u and v along the x ans 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.
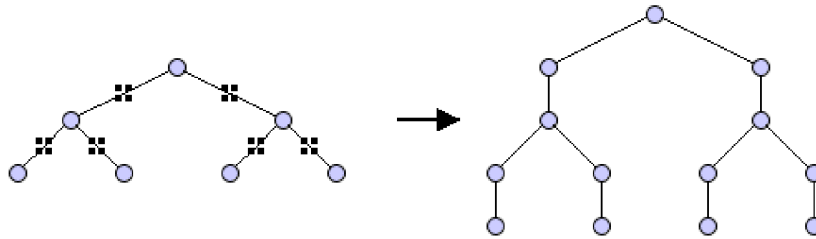
Figure 2.21: A complete layered binary tree with the nodes doubled.

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.

Lastly, the third constraint 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.
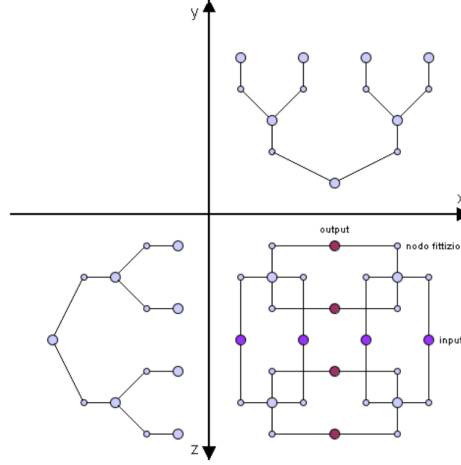
Figure 2.22: The LCP of two such trees.

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

Lemma 2.1

Given an n-dimensional butterfly network, for any non-negative integers $j, k > 0$ such that $0 \leq j \leq j + k \leq n$, the subgraph of the butterfly network induces by the nodes in levels $j, j+1, \ldots, j+k$ is the disjoint union of 2^{n-k} copies of k-dimensional butterfly networks.

In particular, if $j = 0$ and $k = n - 1$, we get have the following _____ add pic

Therefore, an $(n-1)$-dimensional butterfly network can be built from a pair of $(n-2)$-dimensional butterfly networks connected by one node layer and one ege layer. _____ what?

<div align="right">

3

</div>

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

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

this is wrong
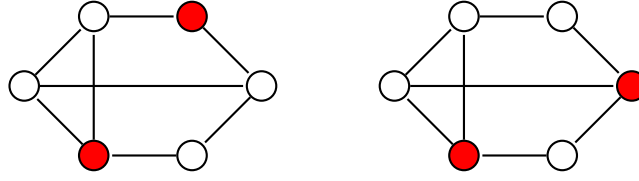
Figure 3.1: Two minimum vertex covers for a graph.

For the reasons mentioned, every minimum 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 **minimum 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 minimum 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 [16], which are a collection of well-known computational problems that were classified as NP-Complete shortly after the introduction of the Cook theorem [8].

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 variabile 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

Therefore, we get the following ILP:

$$\min \sum_{i=1}^{n} x_i$$

$$x_i + x_j \geq 1 \quad \forall (i, j) \in E(G)$$
$$x \in \{0, 1\}^n$$

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 minimum vertex cover for G.

```
 1: function FIRSTGREEDYAVC(G)
 2:     V' := ∅
 3:     E' := E(G)
 4:     while E' ≠ ∅ do
 5:         Choose (i, j) ∈ E'
 6:         V' = V' ∪ {i}
 7:         for (i, k) ∈ 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 minimum.

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

Consider a graph formed by two rows of nodes as follows: the upper row has r nodes, and the lower row has $k > r$ nodes. Each node of the upper row is connected to each node on the lower row. Note that the upper row is a *minimum* vertex cover for the graph.

Suppose that the algorithm always chooses the nodes from the lower row, then the resulting set of nodes is a vertex cover made of k nodes. Thus, the approximation ratio between this set of nodes and the minimum vertex cover is

$$\frac{k}{r} = \frac{n - r}{r}$$

Algorithm 3.2: Second greedy AVC

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

```
 1: function SECONDGREEDYAVC(G)
 2:     V' := ∅
 3:     while E(G) ≠ ∅ do
 4:         v ∈ arg max_{v∈V(G)} deg(v)
 5:         V' = V' ∪ {v}
 6:         for (u, v) ∈ E(G) do          ▷ any edge having v as an endpoint
 7:             G.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 choosen 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 minimum.

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 *vary 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
- $\left\lfloor \dfrac{r}{2} \right\rfloor$ nodes of degree 2 ...
- in general, $\left\lfloor \dfrac{r}{i} \right\rfloor$ 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** minimum 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 minimum covers V' for a given graph G such that $|V'| \leq 2\,|V^*|$, where V^* is a minimum vertex cover.

Algorithm 3.3: 2-approximation VC

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

```
 1: function 2APPROXVC(G)
 2:     V' := ∅
 3:     E' := E(G)
 4:     while E' ≠ ∅ do
 5:         Choose (i, j) ∈ E'
 6:         V' = V' ∪ {i, j}
 7:         for (i, k) ∈ E(G) do            ▷ any edge having i as an endpoint
 8:             E' = E' − {(i, k)}
 9:         end for
10:         for (j, h) ∈ E(G) do            ▷ 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 endpoint of the choosen edge are considered in the removal step.

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

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

Let A be the set of the edges *choosen* 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 minimum vertex cover V^* such that
$$|A| \leq |V^*|$$

Finally, we have that

$$|V'| = 2\,|A| \le 2\,|V^*|$$

$\square$

Algorithm 3.4: 2-approximation VC (ILP)

Given an undirected graph G, the algorithm returns a vector that represents an approximated minimum vertex cover V' for G, such that $|V'| \le 2\,|V^*|$, where V^* is a minimum vertex cover.

1: **function** 2APPROXVCILP(G)
2:     Consider the ILP formulation of the VC problem on G
3:     Relax the ILP by replacing the $x \in \{0,1\}^n$ constraint into $x \in [0,1]^n \subseteq \mathbb{R}^n$
4:     $x^* :=$ `polyLPSolver()`
5:     $y^* \in \{0,1\}^n$
6:     **for** $i \in [1,n]$ **do**
7:         **if** $x_i \ge \frac{1}{2}$ **then**
8:             $y_i^* := 1$
9:         **else**
10:             $y_i^* := 0$
11:         **end if**
12:     **end for**
13:     **return** y^*
14: **end function**

Idea. By relaxing the ILP formulation of the VC problem on G to a LP problem, we can use any polynomial LP solver to get a fractional solution $x^* \in [0,1]^n$. Thus, to get a valid vertex cover, it is sufficient to consider only the x_i^*'s such that $x_i^* \ge \frac{1}{2}$.

Proof. Let V' be the set of vertices described by y^*. It is easy to see that V' is a vertex cover for G, because the constraint of the ILP

$$x_i^* + x_j^* \ge 1$$

forces at least one between x_i^* and x_j^* to be greater or equal than $\frac{1}{2}$, therefore at least one between y_i^* and y_j^* will be set to 1.

Now we will prove that the algorithm returns a 2-approximation of an optimal solution V^*. Let

$$Z := \sum_{i=1}^{n} x_i^*$$

Since $x_i^* \le 1$ for each $i \in [1,n]$, it must be that $Z \le |V^*|$. Let $y^* \in \{0,1\}^n$ be the integer solution obtained from $x^* \in [0,1]^n$ by the rounding procedure; clearly, for each $i \in [1,n]$, we have that $y_i^* \le 2x_i^*$, therefore

$$|V'| = y_1^* + \ldots + y_n^* \le 2(x_1^* + \ldots + x_n^*) = 2Z \le 2\,|V^*|$$

$\square$

The vertex cover problem is related to many other graph theory problems.

> **Definition 3.3: Independent set**
>
> Given an undirected graph G, $S \subseteq V(G)$ is an **independent set** if and only if
>
> $$\forall v, v' \in S \quad \nexists (v, v') \in E(G)$$

> **Theorem 3.1**
>
> A set of nodes V' is a vertex cover over a graph G if and only if its complement $V(G) - V'$ is an independent set.

Proof.

> *First implication.* By way of contradiction, assume that there exist $x, y \in V(G) - V'$ such that $(x, y) \in E(G)$; note that neither x nor y are in V' because they are in its complement, therefore (x, y) is not covered by the vertex cover V'.
>
> *Second implication.* Analogously, by way of contradiction, assume that there exists an edge $(x, y) \in E(G)$ that is not covered by any node in V', then both $x, y \in V(G) - V'$, therefore there exist two adjacent nodes in $V(G) - V'$.

$\square$

> **Corollary 3.1**
>
> The number of nodes of a graph is equal to the size of its minimum vertex cover, plus the size of a maximum independent set.

> **Definition 3.4: Matching**
>
> Given a graph G, $M \subseteq E(G)$ is a **matching** of G if and only if
>
> $$\forall (x, y), (u, v) \in M \quad x, y \neq u, v$$

> **Definition 3.5: Perfect matching**
>
> Given a graph G, a **perfect matching** is a matching that covers every vertex of the G.

Note that every perfect matching is a *maximum matching*. Additionally, note that a perfect matching exists only if the number of vertices n of G is even, since each edge of the perfect matching covers exactly 2 vertices. In particular, the cardinality of a perfect

matching is always $\frac{n}{2}$, since each node is adjacent to exactly one edge of the perfect matching.

Theorem 3.2

Let M be a matching of G and C a vertex cover for G; then $|M| \leq |C|$.

Proof. C is a vertex cover, thus it must cover all edges in $E(G)$, and in particular it covers all edges in M. By definition of vertex cover, for each edge in M, at least one of its endpoints must be in C, therefore $|C|$ must be at least $|M|$. □

Corollary 3.2

Let M be a matching of G and C a vertex cover for G. If $|M| = |C|$ then M is a maximum matching and C is a minimum vertex cover.

Note that, although a maximum matching can be found in polynomial time, the contra-positive of this corollary is not true in general, therefore it is not possible to try to solve the minimum vertex cover through the maximum matching problem.

Algorithm 3.5: 2-approximation VC (matching)

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

```
1: function 2APPROXVCMATCHING(G)
2:     M := findMaximalMatching(G)
3:     V' := ∅
4:     for (u, v) ∈ M do
5:         V' = V' ∪ {u, v}
6:     end for
7:     return V'
8: end function
```

Idea. By computing a maximum matching M on G, all the endpoints of the edges in M will form a vertex cover for G, by definition of matching.

Cost analysis. Note that the time complexity of the algorithm depends directly on the algorithm used to compute the maximum matching M of G.

Proof. Consider an edge $(u, v) \in E(G)$; this edge is either in M, and therefore both endpoints are inserted into V' by the algorithm, or $(u, v) \in E(G) - M$, but at least one of its endpoints must be in V', otherwise it could have been put into M, but this is not possible because M is maximum. This proves that V' is a vertex cover.

Let V^* be a minimum vertex cover for G; to prove that V' is a 2-approximation of the VC problem, note that by Theorem 3.2 we have that $|M| \leq |V^*|$, therefore

$$|V'| = 2\,|M| \leq 2\,|V^*|$$

$\square$

> **Definition 3.6: Bipartite graph**
>
> A graph G is said to be **bipartite** if and only if there is a partition U, W of $V(G)$ such that both U and V are independent sets.

Bipartite graphs are very important, because the following theorem, proved by König [24], allows to compute a maximum VC on bipartite graphs in polynomial time.

> **Theorem 3.3: König's theorem**
>
> In any bipartite graph, the number of edges in a maximum matching is equal to the number of vertices in a minimum vertex cover.

3.1.2 The eternal vertex cover problem

In **Dynamic network security**, the *fault-tolerance model*'s objective is to deploy a *minimum* set of guards across network nodes to provide continuous protection against attacks or faults on any single network link at any time. When an attack or fault occurs on a link, a guard stationed at one of the adjacent nodes detects it and *immediately* moves across the link to defend or repair the issue. Meanwhile, the remaining guards reconfigure by repositioning themselves to adjacent nodes. This reconfiguration ensures that the system remains **protected** from future attacks or failures, maintaining a dynamic, adaptive defense mechanism.

This process ensures that protection is not only *instantaneous* but can be maintained *indefinitely*. The guards adjust their positions in response to each new incident, guaranteeing continuous defense against single-link attacks or failures in an *ad infinitum* manner, adapting dynamically to evolving threats or faults without compromising the network's resilience.

This model can be translated into the **eternal vertex cover** problem, in which

- the network is modeled as a graph

- at most one defender is located at each node

- an attacker can target edges

- a defender *can* protect all the edges incident to the nodes where the guard is located

- to defend an attacked edge, a guard must move along the attacked edge

- any guard can traverse one edge at a time

Given the set of nodes where guards are deployed at any moment, if these nodes do not form a **vertex cover**, the attacker can exploit any uncovered edge to bypass the defense and successfully breach the network. Therefore, to ensure continuous protection, the defender must *dynamically reconfigure* the guard positions so that the current set of guarded nodes always forms a valid vertex cover. This reconfiguration is crucial after any attack, transforming *one vertex cover into another* in response to the attack, ensuring that no edge remains exposed.

If $\alpha(G)$ is the cardinality of a minimum VC on a graph G, and $\alpha^\infty(G)$ is the cardinality of a minimum eternal VC, then

$$\alpha(G) \leq \alpha^\infty(G)$$

> **Theorem 3.4: Shadow guard**
>
> Let G be a connected graph, and let V' be a vertex cover for G that induces a connected subgraph of G. Then $\alpha^\infty(G) \leq |V'| + 1$.

Proof. Choose a vertex $d \in V(G) - V'$, and place a new guard on it; we will refer to this guard as *shadow guard*. Let P be the following path

$$P = d, v_1, \ldots, v_k, x$$

where $v_1, \ldots, v_k \in V'$, and (v_k, x) is the attacked edge. To defend (v_k, x), it is sufficient to slide each guard over P, towards x, therefore the *shadow guard* will now be on v_1, and the attacked edge will be defended by the guard that slid from v_k to x.

Finally, note that the new set of vertices on which the guards now stand on still form a vertex cover, hence $V' \cup \{d\}$ is an eternal vertex cover. $\square$

> **Lemma 3.1**
>
> Let G be a connected graph, and let V' be a vertex cover inducing a subgraph of G, with k connected components. Then $\alpha^\infty(G) \leq |V'| + k$.

Proof. Considering each connected component of V' as a separate connected subgraph, we get the result of the lemma, by the same reasoning of the previous theorem. $\square$

Note that V' can induce at most $|V'|$ connected components, therefore we get the following theorem.

> **Theorem 3.5**
>
> Given a graph G, we have that
>
> $$\alpha(G) \leq \alpha^\infty(G) \leq 2\alpha(G)$$

Theorem 3.6: Eternal VC on cycles

For any $n \geq 3$, we have that

$$\alpha^\infty(C_n) = \alpha(C_n) = \left\lceil \frac{n}{2} \right\rceil$$

where C_n is a cycle graph of n nodes.

Proof. By placing a guard on alternated nodes of the cycle graph C_n, we get an eternal vertex cover: if an edge is attacked, it is sufficient to rotate each guard by 1. $\square$

Theorem 3.7: Eternal VC on paths

For any $n \geq 1$, we have that
$$\alpha^\infty(P_n) = n - 1$$
where P_n is a path graph of n nodes.

Proof. Consider a vertex cover of less than $n - 1$ nodes of P_n. It is always possible to design an attack strategy such that all the guards form a connected path, therefore an edge will be unprotected because there are less than $n - 1$ guards. $\square$

4

The data mule scheduling problem

A **sensor** is a device that detects and responds to specific *environmental inputs*, such as light, heat, motion, moisture, or pressure, among many other physical phenomena. Upon sensing these inputs, the sensor generates an *output signal*, which can either be displayed locally in a human-readable format or transmitted electronically over a network for further analysis or processing. This process allows sensors to play a crucial role in monitoring and interpreting real-world conditions across a wide range of applications.

Sensor networks are wireless networks made up of numerous small sensors, often low-cost, designed to collect *environmental data*. These networks are rapidly expanding due to their broad range of applications, enabling effective monitoring and data collection for diverse purposes across industries.

We assume that the type of networks discussed in this chapter are **fixed**, therefore the sensor cannot change position over time. Moreover, we assume that the sensors need to send all the gathered information to a **base station**.

From an engineering standpoint, one of the most critical challenges in fixed sensor networks is **energy management**, for the following reasons:

- Sensor networks are often deployed in remote or inaccessible areas where direct access to power sources is impractical. As a result, sensors typically rely on *batteries*, which are difficult to replace or recharge due to the network's location and density.

- Many sensor network applications require *continuous, long-term data collection* over extended periods, often months or even years, which demands *efficient energy use* to prolong network operation without frequent maintenance.

Addressing this energy constraints is essential for maintaining reliable and cost-effective sensor network functionality in the field. In particular, since sensors remain *stationary* in this setup, **wireless communication** becomes one of the most energy-intensive operations on each sensor node. To conserve energy, it's critical to **minimize communication** where possible.

A possible approach to the problem may be **multi-hop communication to the base station**, but this solution is less than ideal due to several potential drawbacks:

- **unstable communication infrastructure**: connections between nodes in a multi-hop setup can be prone to *instability*, especially in changing environments where interference or physical obstructions disrupt communication paths

- **high energy costs with sparse deployment**: in areas where nodes are *sparsely* deployed, the distance to the nearest node or the base station can be considerable, hence nodes must increase their transmission range to maintain connectivity, consuming significantly more energy and reducing overall network lifespan

- **energy depletion in dense networks**: in *densely* packed networks, nodes near the base station bear a heavy communication load, often forwarding data from distant nodes, which can lead to rapid energy depletion for these nodes, creating bottlenecks in the network and reducing its effectiveness over time.

Another strategy, which is significantly more *energy-efficient*, is to leverage **mobile data mules**. A *data mule* is a mobile node equipped with:

- **wireless communication**, enabling data collection from stationary sensor nodes

- **ample storage capacity**, allowing it to store data collected from multiple sensors

The data mule traverses the sensing area, collecting data as it comes within close range of each sensor node. Later, it returns to the base station to deposit all gathered data.

There are multiple advantages to this approach.

- **energy savings for sensor nodes**: each sensor only needs to transmit data over short distances, conserving substantial energy as it eliminates the need for long-range or multi-hop communication; additionally, sensor nodes avoid the energy cost of forwarding data from other nodes, reducing their workload

- **reduced energy constraints for the data mule**: since data mules typically return to the base station to recharge, their energy limitations are less critical than those of stationary sensors, allowing for more flexible and longer collection periods

- **simplified network management**: this approach reduces the need for complex routing among nodes, which minimizes processing demands on sensors and further extends their battery life

Although data mules offer greater flexibility in sensor networks, it remains important to ensure they traverse sensor nodes in an *optimized manner*. Efficient traversal minimizes the data mule's battery usage, helping to extend its operational life and reducing associated costs.

To minimize the time required for a data mule to collect data from all sensor nodes, the problem can be framed as a **scheduling problem**, where each sensor node's communication represents a *job*. The goal is to control both the movement (i.e. *path* and *speed*) of the data mule and its communication schedule with each node, similar to job allocation in classical scheduling problems.

However, data mule scheduling presents additional complexity due to unique location and time constraints:

- **location constraints**: each data transfer becomes available only when the mule is within the *wireless communication range* of a node, thus proximity to each node is essential, which imposes spatial constraints that affect when data can be collected

- **time constraints**: given a fixed bandwidth and a continuously moving mule, each node requires a specific *time window* to transmit its data successfully; note that the data mule *cannot stop*, meaning the timing of its arrival and departure relative to each node must be precise to ensure successful data transfer

To address these constraints, optimization techniques can be applied to schedule the data mule's path and communication windows effectively:

- **path optimization**: plan a path that *minimizes travel distance*, while ensuring each node is visited within its communication range

- **speed control**: adjust the mule's speed dynamically based on *node density* and *transmission time* needs, allowing it to linger in areas where longer data transfer times are required

- **adaptive scheduling**: incorporate *adaptive scheduling algorithms* to allocate data transfer time based on each node's data volume and communication range requirements

More specifically, the problem of efficiently managing a data mule's traversal can be broken down into three interrelated subproblems.

- **Path selection**: This step involves determining the *optimal trajectory* for the data mule within the sensor field. The goal is to ensure that the data mule comes within the *communication range* of each sensor node at least once to collect data effectively. This subproblem can be approached using *shortest-path algorithms* or *traveling salesman-like* methods, ensuring that the mule visits each required location while minimizing travel distance.

- **Speed control**: After selecting the path, the next challenge is to adjust the data mule's *speed* along this trajectory. The mule must stay within each node's *communication range* just long enough to complete data transfer without stopping. This requires fine-tuning speed based on the data volume and bandwidth limitations at each node, allowing efficient data collection while conserving battery life by avoiding unnecessary idling.

- **Job scheduling**: When the data mule is within range of multiple sensors, it needs a strategy to *decide* the sequence of data collection. This scheduling can be framed as a *job allocation problem*, where each sensor's data transfer represents a *job* with specific time intervals during which it can be completed. The task is to allocate time slots for each job to ensure all data is collected in the shortest possible time. This problem closely aligns with classical job scheduling, where the objective is to assign time slots efficiently to maximize throughput and minimize the overall collection time.

Since the **speed contro** subproblem is typically handled by engineers, and the **job scheduling** subproblem falls outside this chapter's scope, our focus here is on the first subproblem, the **path selection** for the data mule.

Consider a scenario where sensor nodes operate at *varying sampling rates* (e.g., pollution sensors that adjust to real-time conditions). Each sensor has a *limited buffer* for storing its data until a mobile data mule—serving as a mobile base station—arrives to offload this information. Once the data mule reaches a sensor node, it transfers the collected data to its own storage, freeing up the sensor's buffer for new data. This setup has several practical implications:

- **varying sampling rates**: sensors sampling data at different rates may fill their buffers at different speeds, impacting the urgency and frequency of data mule visits, in fact sensors with higher sampling rates (e.g., in highly polluted areas) might require more frequent visits to prevent data loss

- **finite buffer constraints**: since each sensor has a limited data storage capacity, efficient scheduling and path planning for the data mule become crucial to avoid data overflow at any sensor, thus path selection must account for these storage limitations and prioritize nodes based on buffer status and sampling frequency

The problem of scheduling the visits of a mobile data mule to ensure that none of the sensor nodes' buffers overflow can be summarized as the **Mobile Element Scheduling (MES)** problem. This problem involves planning the optimal sequence and timing of visits by the data mule to collect data from sensor nodes, considering the limited buffer capacity of each sensor node and the varying data collection rates.

Now, consider the following well-known computational problem.

> ### Definition 4.1: Traveling Salesman Problem
>
> The Traveling Salesman Problem (TSP) is defined as follows: a salesman has to visit a given set of cities, such that his tour ends on the same city on which he started, while minimizing the total length of the trip.

Note that the MES problem differs from the TSP in key ways:

- **objective**: TSP searches for the shortest path visiting each city (i.e. node) exactly once, while in MES a node may need multiple visits due to varying sampling rate and buffer state

- **deadlines**: in TSP the costs are fixed (the total length of the trip) and there are no time constraints, while in MES deadlines dynamically update after each visit, requiring the data mule to adjust its path in real time

Despite these differences, TSP-based approaches can still be useful for solving MES, particularly in path planning. In fact, TSP can serve as a useful approximation for MES, by considering the optimal route between nodes, and then adjusting that route dynamically to accommodate nodes that require more frequent visits.

4.1 The Traveling Salesman Problem

The origins of the TSP are somewhat ambiguous:

- an 1832 handbook for traveling salesmen references the problem, presenting example routes through Germany and Switzerland, though without any mathematical formulation

- in the mid-1800s, mathematicians W. R. Hamilton and T. Kirkman introduced the first formal mathematical formulation of the problem

- the TSP in its general form was studied in the 1930s, when researchers analyzed the limitations of the brute-force algorithm and noted the non-optimality of simpler heuristics like the nearest neighbour approach

---

Definition 4.2: Hamiltonian cycle

Given a graph $G = (V, E)$, a **Hamiltonian cycle** (HC) is a cycle that passes through each node in $V(G)$ exactly once.

It can be proven that determining whether a graph G contains a HC is NP-Complete— HC will be used interchangeably for "Hamiltonian cycle" and the associated decision problem. Now, consider the following decisional version of the TSP.

Definition 4.3: TSP (decisional version)

Let $K_n = (V, E)$ be a complete graph having n nodes, $w : E(G) \to \mathbb{R}^+$ be a non-negative edge-weight function, and $t \geq 0$; does K_n contain a Hamiltonian cycle with total cost at most t?

Note that any complete graph K_n trivially contains a HC, but the problem aims at minimizing the cost of the HC. The following proof shows that the TSP is NP-Complete as well.

Theorem 4.1: TSP $\in$ NP-Complete

The TSP is NP-Complete.

Proof. It can be easily proved that TSP is in NP: given a complete graph $K_n = (V, E)$, and a walk over K_n, it can be checked in polynomial time if the walk is a Hamiltonian cycle, and its total weight is bounded by t.

Now it will be proven that TSP is NP-Hard, by reducing HC to TSP as follows:

- consider a graph $G = (V, E)$, and construct the complete graph $K_n = (V, E')$, where $E(G) \subseteq E'(K_n)$ and the remaining edges are the ones added to make K_n a complete graph — note that $n := |V(G)|$

- let $t := n$ and define $w : E'(K_n) \to \mathbb{R}^+$ as follows:

$$
\begin{cases}
w(i,j) = 1 & (i,j) \in E(G) \\
w(i,j) = 2 & (i,j) \in E'(K_n) - E(G)
\end{cases}
$$

- assume that there exists a Hamiltonian cycle C in G; by definition of w, all edges of C in K_n have weight 1, since they are all in G; this shows that if G has a HC, then K_n has a traveling salesman tour of cost $n = t$

- conversely, if K_n has a traveling salesman (TS) tour of cost n, all edges of the tour necessarily have weight 1, because edges with weight 2 would not minimize the cost; therefore, by definition of w, this tour describes a HC in G

$\square$

In 1954 Dantzig et al. [9] showed that the TSP can be formulated as an ILP as well:

- given a complete graph $K_n = (V, E)$, and assume that the TS tour is oriented

- define variables x_{ij} and w_{ij} for each $(i,j) \in E(G)$

- let $x_{ij} = 1$ if and only if the TS tour traverses the oriented edge (i,j)

- let $w_{ij} = w(i,j)$

Then, the TSP can be formulated as an ILP as follows:

$$
\min \sum_{i=1}^{n} \sum_{j=1}^{n} w_{ij} x_{ij}
$$

$$
\sum_{i=1}^{n} x_{ij} = 1, \quad j \in V(K_n)
$$
$$
\sum_{j=1}^{n} x_{ij} = 1, \quad i \in V(K_n)
$$
$$
\sum_{i,j \in S} x_{ij} < |S|, \quad S \subsetneq V(K_n), S \neq \varnothing
$$
$$
x \in \{0,1\}^n
$$

The first two constraints force the tour to be Hamiltonian, by imposing that for any vertex j there must be at most 1 incoming edge, and for each vertex j there must be at most 1 outgoing edge. However, this does not imply that the solution of the ILP is a cycle: in fact, without the third constraint, a valid solution could involve multiple unconnected cycles of G. Therefore, the last constraint imposes that any *proper subset* of vertices S of $V(K_n)$ must cover a number of edges that is strictly less than the number of vertices of S itself; in fact, this constraint avoids the possibility of forming cycles in the solution because a cycle has the same number of edges and vertices. Note that, in reality, the last constraint hides $|\mathcal{P}(V(K_n))| - 2 = 2^n - 2$ constraints, one for each possible *proper subset* S of $V(K_n)$.

The next theorem shows that, in addition to being NP-Complete, the TSP is also non-approximable, making it an especially challenging problem.

> **Theorem 4.2: Inapproximability of the TSP**
>
> If there exists a polynomial time algorithm for the TSP with any approximation ratio $r > 1$, then $\mathsf{P} = \mathsf{NP}$.

Proof. Let $G = (V, E)$ be an instance of HC, and construct a complete graph $K_{|V|} = (V, E')$ starting from G by adding edges; moreover, define $w : E(K_n) \to \mathbb{R}^+$ as follows:

$$\begin{cases} w(i,j) = 1 & (i,j) \in E(G) \\ w(i,j) = 2 + (r-1)n & (i,j) \in E'(K_n) - E(G) \end{cases}$$

Note that, for the same reasoning applied in the proof of Theorem 4.1, a TS tour with total weight n exists in K_n if and only if G has a HC.

Assume there exists an r-approximation polynomial algorithm A for the TSP; therefore, because the total weight of the TS tour is n, A ran on K_n would find a solution H such that

$$\sum_{(i,j) \in H} w(i,j) \leq rn$$

Now, assume that there exists an edge $(\hat{i}, \hat{j})$ in H such that $(\hat{i}, \hat{j}) \in E'(K_n) - E(G)$; hence, by definition of w, the total weight of H must be at least

$$\sum_{(i,j) \in H} w(i,j) = (n-1) \cdot 1 + 2 + (r-1)n = n - 1 + 2 + rn - n = rn + 1 > rn$$

because H would be a cycle containing $n - 1$ edges from $E(G)$ and 1 edge from $E'(K_n) - E(G)$. Since A is an r-approximation algorithm, this implies that any solution H for A must contain only edges inside G, otherwise H would not be an r-approximation of an optimal solution for the TSP. However, if any of A's solutions H lie entirely in G, then H is a HC for G as previously discussed, which means that A can find a HC in G in polynomial time, which would imply that $\mathsf{P} = \mathsf{NP}$ because HC is $\mathsf{NP\text{-}Complete}$. $\qquad\square$

4.1.1 Special cases for the TSP

Despite this result showing that the TSP is not generally approximable, it is still possible to find effective approximation algorithms for certain special cases. Note that, for any set of edges E, the following notation will be used

$$w(E) := \sum_{(i,j) \in E} w(i,j)$$

> **Lemma 4.1: Lower bound on TS tours**
>
> Given a graph G and a weight function $w : E(G) \to \mathbb{R}^+$, the weight of any TS tour on G is at least the weight of any MST of G.

Proof. Consider a graph G, an MST T of G, and an optimal TS tour H^* on G. Clearly, if by removing and edge from H^*, we obtain a path P, which has weight strictly less than H^*'s weight — note that this is true because $w(i,j) \in \mathbb{R}^+$ for any $(i,j) \in E(G)$. Moreover, note that a path is a special case of a tree, therefore P's weight must be at least T's weight, by definition of MST. Thus, we have that

$$w(T) \leq w(P) \leq w(H^*)$$

$\square$

Consider the following special case of graphs.

Definition 4.4: Metric graphs

Given a graph G, and a weight function $w : E(G) \to \mathbb{R}^+$, G is said to be a **metric graph** if and only if

$$\forall u, v, z \in V(G) \quad w(u, z) \leq w(u, v) + w(v, z)$$

which means that w satisfies the triangle inequality.

For metric graphs, there exist algorithm that can approximate solutions for the TSP. In particular, the following algorithm leverages the triangle inequality property of w to obtain a 2-approximation for the TSP.

Algorithm 4.1: 2-approximation TSP

Given a complete graph $K_n = (V, E')$, and a weight function $w : E'(G) \to \mathbb{R}^+$ such that K_n is a metric graph, the algorithm finds a TS tour H such that $w(H) \leq 2w(H^*)$, where H^* is an optimal TS tour.

1: **function** 2APPROXTSP(G, w)
2:      Choose $r \in V(K_n)$ randomly
3:      $T := \texttt{findMST}(K_n, r)$ $\triangleright$ find an MST rooted in r
4:      $\texttt{L} := \texttt{DFSpreorder}(T)$ $\triangleright$ a *preorder* DFS on T
5:      $V := \varnothing$
6:      $\texttt{L}' := \texttt{[]}$
7:      **for** $v \in \texttt{L}$ **do**
8:          **if** $v \notin V$ **then**
9:              $\texttt{L}'.\texttt{append}(v)$
10:              $V = V \cup \{v\}$
11:          **end if**
12:      **end for**
13:      **return** L' $\triangleright$ L' is L without repetitions
14: **end function**

Proof. We will prove that any solution H of the algorithm is a 2-approximation of an optimal solution H^* for the TSP.

Consider an optimal TS tour H^*, and an MST T of the complete graph K_n in input, rooted in some $r \in V(K_n)$. The list L computed by the algorithm is obtained from a *preorder* DFS visit T, therefore each *edge* of the visit L will appear exactly *twice*. This means that the tour C described by the edges between the vertices of L is such that $w(C) = 2w(T)$.

Note that C is not a TS tour, since there nodes are repeated. However, by leveraging the triangle inequality of w, we can prove that the weight of the final list L' — which is L without repetitions of the vertices — is bounded by the weight of L. In fact, for any instance L in which

$$\ldots \; u \; v \; z \; \ldots$$

where v is repeated, by removing v and passing through (u, z) directly — which always exists because K_n is a complete graph — will not worsen the total weight of the tour, because

$$w(u, z) \leq w(u, v) + w(v, z)$$

by hypothesis. Let H be the tour described by the edges between the vertices of L'; hence, we have that $w(H) \leq w(C)$.

Finally, because of Lemma 4.1, we conclude that

$$w(H) \leq w(C) = 2w(T) \leq 2w(H^*)$$

$\square$

In 1976 Christofides [6] showed that it is possible to obtain a better approximation of the TSP problem, because the algorithm previously discussed does not exploit all the available edges on the graph.

Lemma 4.2: Handshaking lemma

Given a graph G, the sum of all the degrees of the vertices in $V(G)$ is $2\,|E|$.

Corollary 4.1

The number of vertices that have an odd degree in a graph is even.

Proof. Consider a graph G; for the handshaking lemma, we have that

$$\sum_{v \in V(G)} \deg(v) = 2m$$

Let $O := \{v \in V(G) \mid \deg(v) \text{ odd}\}$ and $E := \{v \in V(G) \mid \deg(v) \text{ even}\}$ then, we have that

$$\sum_{v \in O} \deg(v) + \sum_{v \in E} \deg(v) = \sum_{v \in V(G)} \deg(v) = 2m$$

because O and E describe a partition on $V(G)$. Therefore, because each degree of nodes in E is even by definition, $\sum_{v \in E} \deg(v)$ is even, which means that the handshaking lemma is satisfied only if $\sum_{v \in O} \deg(v)$ is even as well. However, since each degree of nodes in O is odd by definition, it must be that the entire sum is even. $\square$

Definition 4.5: Eulerian circuit

An **Eulerian circuit** is a walk through a graph, which uses every edge exactly once, and starts and ends at the same vertex.

Theorem 4.3: Eulerian circuits

A graph has an Eulerian circuit if and only if the degree of every vertex is even.

Algorithm 4.2: $\frac{3}{2}$-approximation TSP

Given a complete graph $K_n = (V, E')$, and a weight function $w : E'(G) \to \mathbb{R}^+$ such that K_n is a metric graph, the algorithm finds a TS tour H such that $w(H) \leq \frac{3}{2} w(H^*)$, where H^* is an optimal TS tour.

1: **function** $3/2\textsc{approxTSP}(G, w)$
2:     Choose $r \in V(K_n)$ randomly
3:     $T := \texttt{findMST}(K_n, r)$ $\triangleright$ find an MST rooted in r
4:     $O := \{v \in V(T) \mid \deg_T(v) \text{ odd}\}$ $\triangleright \deg_T(v)$ is the degree of v in T
5:     $M := \texttt{findMinWeightPM}(G^O)$ $\triangleright G^O$ is induced by O
6:     $U := M \cup T$ $\triangleright U$ is a multi-graph
7:     $\texttt{L} := \texttt{findEulerianCircuit}(U)$
8:     $V := \varnothing$
9:     $\texttt{L}' := \texttt{[]}$
10:     **for** $v \in \texttt{L}$ **do**
11:         **if** $v \notin V$ **then**
12:             $\texttt{L}'.\texttt{append}(v)$
13:             $V = V \cup \{v\}$
14:         **end if**
15:     **end for**
16:     **return** $\texttt{L}'$ $\triangleright \texttt{L}'$ is $\texttt{L}$ without repetitions
17: **end function**

Proof. We will prove that any solution H of the algorithm is a $\frac{3}{2}$-approximation of an optimal solution H^* for the TSP.

Consider an MST T of K_n rooted in some node $r \in V(K_n)$, and consider the set of vertices O that have odd degree in T, and the subgraph G^O this set induces. Now, consider the graph G^O, induced by O, and a minimum weight perfect matching M of G^O.

Note that, because of Corollary 4.1, $|O|$ is even, therefore G^O has a perfect matching;

---

thus, let M be the minimum weight perfect matching of G^O. Moreover, let N^* be a TS tour on G^O, and let N_O and N_E be the two subsets of edges obtained by taking the edges of N^* alternately. Note that N_O and N_E describe a partition of N^*, which implies that $w(N_E) + w(N_O) = w(N^*)$. Additionally, note that both N_O and N_E are perfect matchings G^O, but since M is the perfect matching of G^O with minimum weight, it must be that

$$w(M) \leq \min(w(N_O), w(N_E)) \leq \frac{w(N_O) + w(N_E)}{2} = \frac{w(N^*)}{2}$$

Consider a TS tour A, obtained from H^*, by skipping the even-degree nodes:

- clearly, this is a TS tour on O, because O is derived from T, which is a spanning tree, therefore it will contain all the odd-degree vertices of K_n; by triangle inequality, this implies that $w(A) \leq w(H^*)$

- lastly, since N^* is an optimal TS tour on O by definition, we have that $w(N^*) \leq w(A)$

- this means that $w(N^*) \leq w(A) \leq w(H^*)$, therefore

$$w(M) \leq \frac{w(N^*)}{2} \leq \frac{w(H^*)}{2}$$

Now consider the multi-graph described by the edges in $U := M \cup T$, where the edges that appear both in M and in T are *counted twice*:

- clearly, the graph induced by U is connected, because T is a spanning tree

- since M is a minimum weight perfect matching of G^O, adding the edges from M to T into U will turn the degree of the vertices that were in O into even degrees, therefore all nodes in U are of even degree; this means that, since U is connected, it is always possible to find a Eulerian circuit by Theorem 4.3

- because $w(M) + w(T) = w(U)$, we have that

$$w(H) \leq w(U) = w(M) + w(T) \leq \frac{w(H^*)}{2} + w(H^*) \leq \frac{3}{2}w(H^*)$$

where $w(H) \leq w(U)$ by definition of H — H is the solution provided by the algorithm — and $w(T) \leq w(H^*)$ is a consequence of Lemma 4.1

$\square$

Christofides [6] originally developed his algorithm to solve the Euclidean TSP, a specific type of metric TSP. However, this algorithm is more general and can be applied to any metric TSP, not limited to Euclidean cases. Despite its general applicability, there are certain inputs that push the performance of this algorithm close to its worst-case approximation ratio of $\frac{3}{2}$.

In fact, it is actually possible to achieve a better approximation, and a solution was provided by Arora [1] in 1996, for which he was awarded with the Gödel Prize in 2010. In particular, they proved that there exists a polynomial-time approximation scheme (PTAS) for the Euclidian TSP. This means that, for any constant $c > 0$ in a d-dimensional

Euclidian space, there is a polynomial-time algorithm that can find a TS tour with a length at most $1 + \frac{1}{c}$ times the optimal length for geometric instances of TSP, in time

$$O\left(n(\log n)^{\left(O(c\sqrt{d})\right)^{d-1}}\right)$$

which was later improved in 2012 by Bartal et al. [2].

placeholder

dk if i'm doing this

5

The data collection problem

Consider the same setting discussed in the previous chapter. When a data mule is not actively used, sensor nodes operate in a **low-energy mode**, monitoring and sensing their environment with *minimal energy consumption*. Therefore, the primary energy consumption occurs during data collection, which is the focus of the **data collection problem**.

In particular, the goal of this problem is to efficiently *transfer* all the periodically sensed data from the sensor nodes to the **base station**, using one or more hops, while maximizing the overall network lifetime. This requires optimizing energy usage by minimizing communication costs (e.g., transmission power) and efficiently routing the data.

There are several approaches in literature to addressing this problem, each with its benefits and trade-offs.

- **Naive approach**: In this approach, each sensor node increases its transmission range to send data directly to the base station. While simple, this strategy leads to *enormous energy consumption*, because nodes that are farther away from the base station require more power to transmit data. This reduces the network's overall lifetime, as nodes quickly deplete their energy reserves.

- **Multi-hop data routing**: In this method, sensor nodes send data to the nearest node on the shortest path toward the base station, using multiple hops for data transmission. This approach helps reduce the energy spent on long-distance transmissions. However, a significant issue arises near the base station, where nodes close to it end up handling a *large volume of data* from other nodes, causing them to drain energy faster. This leads to an *uneven load distribution* and may cause these nodes to fail prematurely, reducing the network lifetime.

- **Clustering**: Sensor nodes are grouped into *clusters*, with each cluster having a *cluster head* that aggregates the data from its members and forwards it to the base station. This helps in reducing the energy consumption of individual sensor nodes by localizing communication within clusters. However, minimizing both the intra-cluster and inter-cluster distances is critical, as energy dissipation is proportional to

---

the distance a signal travels. This requires *careful planning* of the cluster structure to ensure that energy consumption is evenly distributed and does not lead to premature failure of certain nodes.

- **Duty cycle based mode**: In this strategy, sensor nodes alternate between *active* and *sleep* modes, only transmitting data during their active periods. This approach helps conserve energy by ensuring that sensors remain inactive during periods without significant events. The downside is that it introduces *delays*, as the data sender must wait for a neighbour to wake up in order to transmit. The duty cycle mode increases transmission delay, which can affect network performance, particularly in time-sensitive applications.

Each of these approaches aims to balance energy consumption, transmission delay, and network lifetime, but each has its own limitations, which must be considered depending on the specific application and environment of the sensor network.

In 2019 Shi et al. [22] presented an approach that employs a **duty cycle mode** to improve energy efficiency in sensor networks. The method involves constructing a *connected subnetwork*, which will be referred to as **backbone**, from a selected subset of nodes, and the approach works as follows.

- **Backbone formation**: A part of the nodes forms a *backbone*, where these nodes operate in a *sleep/awake* mode at fixed intervals. These nodes only wake up periodically to transmit data, while the rest of the nodes in the network turn off their radios when not transmitting, conserving energy as they continue sensing the environment.

- **Data transmission**: When a node needs to send data, it activates its radio and communicates directly with the backbone nodes. The data is then routed through the backbone network to the base station. The backbone nodes thus handle the more energy-consuming tasks of data forwarding.

- **Energy conservation**: The majority of nodes in the network spend most of their time in *sleep mode*, drastically reducing energy consumption. However, this results in higher energy use for the nodes in the backbone, as they are responsible for routing data.

- **Dynamic backbone reconstruction**: To balance the energy consumption across the network, after a certain period, the nodes with higher residual energy are selected to form a *new backbone*. This ensures that no single node or group of nodes is overly taxed, thus extending the network's lifetime.

The **backbone** nodes must meet the following criteria:

- **minimal size**: the number of nodes in the backbone should be as small as possible to save energy while maintaining network efficiency

- **connectivity**: every node in the backbone must be able to route data to the base station, ensuring that there is *at least one path* from each backbone node to the base station

- **dominating set**: every other node in the network must communicate directly with at least one node in the backbone, meaning the backbone forms a so called

minimum connected dominating set, ensuring that all nodes in the network are covered by the backbone

5.1 The minimum connected dominating set problem

Definition 5.1: Dominating set

Given a graph $G = (V, E)$, a **dominating set** (DS) for G is a subset $D \subseteq V(G)$ such that every node not in D is adjacent to at least one member of D. Formally

$$\forall v \in V(G) - D \quad \exists d \in D \mid (d, v) \in E(G)$$

Example 5.1 (Dominating sets)**.** The following are two examples of dominating sets.

To solve the data collection problem, we require that the backbone is **connected**, and we want to minimize the size of the backbone, therefore we are interested in finding the **minimum connected dominating set** (min-CDS).

Definition 5.2: Min-CDS problem

Given a graph G, find a dominating set D with the smallest possible cardinality, such that D still induces a connected graph in G.

Example 5.2 (Min-CDSs)**.** The following is a min-CDS for the graph illustrated in Example 5.1.

Now, consider the following type of spanning trees.

Definition 5.3: Max-leaf spanning tree

Given a graph G, a **max-leaf spanning tree** (max-leaf ST) for G is a spanning tree that has the largest possible number of leaves among all spanning trees of G.

The following result is crucial in order to solve the min-CDS problem.

Theorem 5.1: Equivalence of min-CDSs and max-leaf STs

Given a graph G, such that $n > 2$, a max-leaf ST T for G having l leaves, and a min-CDS D for G of cardinality d. We have that

$$n = l + d$$

Proof. Consider a min-CDS D for G; a ST T' for G can be constructed by starting from D, by simply considering D itself with the nodes in $V(G) - D$ as leaves of T'. Moreover, T' spans G completely, because D is a DS, which means that all the nodes in $V(G) - D$

are covered by definition of D. This proves that T' is both connected, and spans G completely. Moreover, we can assume that T' is acyclic, since D is a min-CDS, and in particular it is connected, hence it is always possible to remove cycles from D such that the resulting graph T' is still connected but acyclic. Therefore, T' is a spanning tree of G, and because T is a max-leaf ST for G, it follows that

$$l \geq l' = |V(G)| - |D| = n - d$$

where l' is the number of leaves of T', which is $n - d$ by construction.

Conversely, consider max-leaf ST T of G; since T spans G, and since T is connected, it must be that the nodes D' comprising the nodes in T that are not leaves form a CDS for G. Moreover, since D is a min-CDS, it follows that

$$n - l = |V(G)| - |T| = d' \geq d$$

where d' is the number of vertices of D', which is $n - l$ by construction.

In particular, we have that

$$n - l \geq d \iff -l \geq d - n \iff l \leq n - d$$

which, combined with the previous inequality, it must imply that

$$l = n - d \iff n = l + d$$

$\square$

Computationally, this theorem implies that determining the connected domination number is as difficult as finding the maximum leaf number in a graph. Specifically, the associated decision problem of finding the min-CDS in a graph is NP-Complete, which implies that the decision problem of finding the max-leaf ST is NP-Complete as well.

In terms of **approximation algorithms**, the connected domination number and the maximum leaf spanning tree problems differ significantly:

- for the min-CDS problem, Guha et al. [15] proved that there exists an approximation algorithm that achieves a factor of $2 \ln \Delta + O(1)$, where $\Delta := \max_{v \in V(G)} \deg(v)$ for the graph G;

- on the other hand, Solis-Oba [23] showed that the max-leaf ST problem can be approximated within a factor of 2;

- additionally, Ueno et al. [26] proved that, in graphs where $\Delta = 3$, both problems can be solved in polynomial time.

5.1.1 Minimum Dominating set

If we remove the requirement that the dominating set forms a connected subgraph, the problem becomes that of finding a **minimum dominating set** (min-DS). This problem has been a central topic in graph theory since the 1950s, with interest in its complexity, applications, and approximation approaches significantly increasing in the mid-1970s.

Now, consider the following computational problem.

> **Definition 5.4: Set cover**
>
> Given a *universe* set $U = \{1, \ldots, n\}$, and a collection of sets $S \in \mathcal{P}(U)$ such that $\bigcup_{X \in S} X = U$, find the *smallest* sub-collection of S whose union still equals U.

In 1972 Karp [16] proved that the **set cover** problem is NP-Complete. The following theorem proves that finding a min-DS is NP-Complete as well, and will show a close relation between the set cover and the min-DS problem.

> **Theorem 5.2: Equivalence of min-DS and set cover**
>
> There is a bijection between the solutions of the min-DS and the set cover problem.

Proof. Given an instance of the min-DS problem, we will construct an instance of the set cover problem, and vice versa.

Consider an instance of the min-DS, consisting of a graph $G = (V, E)$ with $V = \{1, \ldots, n\}$, and construct a set cover instance as follows:

- the universe is $U := V$

- the collection of subsets is $S := \{S_1, \ldots, S_n\}$, where

$$S_i := \{j \in V(G) \mid (i, j) \in E(G)\} \cup \{i\}$$

Consider a min-DS D for G; clearly, $C := \{S_v \mid v \in D\}$ is a feasible set cover, since each set $S_v \in C$ will cover v and all its neighbours. Conversely, if $C = \{S_v \mid v \in D\}$ is a set cover, is must be that D is a min-DS for G. Note that $|C| = |D|$.

Now, consider a instance of the set cover problem, where U is the *universe* set, and $S = \{S_i \mid i \in I\}$ for some set of indices I such that $U \cap I = \varnothing$; we can construct an instance of the min-DS problem as follows:

- construct a graph $G = (V, E)$ such that $V := I \cup U$

- construct the set of edges as follows

$$E := \{(i, j) \mid i, j \in I\} \cup \{(i, u) \mid i \in I, u \in S_i\}$$

For instance, consider the following instance of the set cover problem, where the *universe* set is $U = \{a, b, c, d, e\}$, $I = \{1, 2, 3, 4\}$ and $S = \{S_1, S_2, S_3, S_4\}$, where $S_1 = \{a, b, c\}$, $S_2 = \{a, b\}$, $S_3 = \{b, c, d\}$ and $S_4 = \{c, d, e\}$. The graph G that will be constructed from the reduction is the following:

Note that G is a split graph, a graph in which the vertices can be partitioned into a *clique* and an *independent set*. In fact, in this construction, the endpoints of the edges derived

from I form a clique, and the endpoints of the edges derived from (i, u) with $i \in I$ and $u \in S_i$ must form an independent set, by construction.

Now, consider a set $D \subseteq I$ of indices; if $C := \{S_i \mid i \in D\}$ is a set cover for U, then for each $u \in U$ there is an $i \in D$ such that $u \in S_i$, and by construction, $(i, u) \in E(G)$, therefore u is dominated by i. Additionally, since the endpoints of the edges constructed from I form a clique, any non-empty subset D of I will dominate all the vertices of the clique, and D cannot be empty otherwise C would not be a solution to the set cover. This implies that D is a min-DS for G.

Conversely, consider a DS D; a min-DS X from D can be constructed, such that $|X| \leq |D|$ and $X \subseteq I$, by simply replacing each $u \in D \cap U$ with a neighbour $i \in I$ of u, which must exist by construction because for any $u \in U$ there exists an $i \in I$ such that $S_i \in I$. Therefore $C = \{S_i \mid i \in X\}$ is a set cover for U, with $|C| = |X| \leq |D|$. $\qquad\square$

Given the established equivalence between the min-DS and set cover problems, we conclude that:

- not only is the min-DS problem NP-Complete, but an efficient algorithm for finding a min-DS would yield an efficient algorithm for the set cover problem, and vice versa

- the reductions between min-DS and the set cover problem preserve the **approximation ratio**, which means that any α-approximation algorithm for the min-DS would also serve as an α-approximation the set cover problem, and vice versa.

5.1.2 Greedy approach for the min-CDS

In 1998 Guha et al. [15] provided a two-step greedy algorithm which is able to find a min-CDS of a graph with an approximation ratio of $3 + \ln \Delta$, where Δ is the maximum degree of G. In 2004 Ruan et al. [21] proved that it is possible to design a single-step greedy algorithm and get a better approximation ratio of $2 + \ln \Delta$, but its implementation is much more complex. The following section will illustrate the two-step greedy approach.

Consider a graph $G = (V, E)$, and a subset $C \subseteq V(G)$ of its nodes; all the nodes in $V(G)$ can be divided into three classes:

- B (black) nodes, defined as B $:= C$

- Gr (gray) nodes, defined as Gr $:= \{v \in V(G) \mid v \notin C \land \exists u \in C \mid (u, v) \in E(G)\}$, which is the set of nodes that are not in C but are adjacent to C

- W (white) noeds, defined as W $:= V(G) - \text{B} - \text{Gr}$, which is the set of nodes that are not in C and are not adjacent to C

Clearly, $\text{B} \cup \text{Gr} \cup \text{W} = V(G)$, and C is a CDS if and only if there is no white node *and* the subgraph induced by B is connected. Let CC be the number of connected components in the subgraph induced by B; we want to define a two-step algorithm based on the following **potential function**:

$$|\text{W}| + \text{CC} = 1$$

Additionally, let $R(v)$ be equal to 1 if and only if $v \in W \cup Gr$ is such that coloring it in black, and its adjacent white nodes in gray, reduces the value of the potential function, 0 otherwise.

The algorithm computes as follows.

Algorithm 5.1: Two-step greedy min-CDS

Given a graph G, the algorithm returns a CDS for G with an approximation ratio of $3 + \ln \Delta$, where Δ is the maximum degree of G.

```
 1: function GREEDYMINCDS(G, B, Gr, W)
 2:     while True do
 3:         if ∃v ∈ W ∪ Gr | R(v) == 1 then
 4:             Color v in black and color its adjacent white nodes in gray
 5:         else
 6:             break
 7:         end if
 8:     end while
 9:     do
10:         Color either one or two gray nodes in black to reduce CC
11:     while CC == 1
12: end function
```

Idea. The first step loop of the algorithm is the first step of the procedure, which removes any white node from G, which means that at the end of the first loop B is a DS for G. However, it may not be connected, therefore the second loop changes the coloring of the vertices such that B is forced to form a CDS.

5.1.3 Unit Disk Graphs

Definition 5.5: Unit Disk graph

Consider a set of n circles of equal radius on the plane; the **unit disk graph** (UDG) of the circles if constructed as follows:

- the nodes are the centers of the circles

- there is an edge between two nodes if the circles they are center of intersect — note that tangent circles intersect

The set of the n circles is called **intersection model**.

UDGs are useful for modeling **wireless networks**: each circle's center represents a *transceiver*, and the radius represents its *transmission range*. In a homogeneous network, all circles are approximately the same size, and, without loss of generality, we assume each radius is equal to 1.

In 1982, Lichtenstein [18] proved that, when restricting the min-CDS problem to UDGs, the problem is still NP-Hard; moreover, in 1990 Clark et al. [7] showed that the problem remains NP-Hard even if restricted to *grids*, a special type of UDGs. In 2003, Cheng et al. [5] proposed a PTAS for the min-CDS problem in UDGs, which guarantees that for any arbitrarily small $\varepsilon > 0$, it provides a $(1 + \varepsilon)$-approximation. Importantly, while the algorithm's runtime remains polynomial in the input size for each fixed ε, the polynomial's degree depends on ε itself, meaning the complexity may vary as ε changes. Nevertheless, this algorithm is not used in practice because the implementation is fairly complex.

Finally, in 2010 Purohit et al. [20] provided a simple and distributed algorithm for UDGs that effectively reduces any given, even trivial, CDS to a smaller, more efficient structure.

> **Definition 5.6: Convex hull**
>
> Given a set of points X on a plane, the **convex hull** in the 2D space is the minimum convex set containing X.

Example 5.3 (Convex hulls)**.** Given the following set of points

the following is its associated *convex hull*.

Given an undirected graph G, let $\mathcal{N}(v) := \{u \in V(G) \mid (u,v) \in E(G)\}$ be the **neighbourhood** of v, and let $\mathcal{N}'(v) := \mathcal{N}(v) \cup \{v\}$ be the **closed neighbourhood** of v. The algorithm which employs convex hulls is defined as follows.

> **Algorithm 5.2: Distributed reduction of CDS**
>
> Given a UDG G, and a CDS of G, the algorithm reduces the CDS.
>
> 1: **function** DISTRIBUTEDREDUCECDS(G, D)
> 2: $V := D$
> 3:   **do**
> 4:   Choose $u \in \arg\min_{v \in V} \deg(v)$
> 5:   **if** CH($\mathcal{N}'(u)$) $\subseteq \bigcup_{z \in \mathcal{N}(u)}$ CH($\mathcal{N}'(z)$) **then**   ▷ CH computes the *convex hull*
> 6: $D = D - \{u\}$
> 7:   **end if**
> 8: $V = V - \{u\}$
> 9:   **while** $V \neq \varnothing$
> 10:   **return** D
> 11: **end function**

This algorithm has several practical advantages, because it is able to reduce the size of an initial CDS, streamlining the structure without complex calculations, and it works without requiring global network knowledge, making it adaptable for *decentralized environments*. However, it's worth noting that **no approximation ratio** is provided or guaranteed. This trade-off makes it a useful heuristic for practical scenarios but limits its theoretical

guarantees.

TODO

placeholder ──

6.1   TODO

Recalling the definition discussed in previous chapters, given a graph $G = (V, E)$, a **matching** is a set of edges $M \subseteq E(G)$ such that every node in $V(G)$ is adjacent to at most one edge in M, thus no two edges in M share common vertices. We will now use the following definitions.

> **Definition 6.1: Maximal matching**
>
> Given a graph $G = (V, E)$, and a matching M of G, M is a **maximal** matching if there exists no $e \in E(G) - M$ such that $M \cup \{e\}$ is still a matching.

> **Definition 6.2: Maximum matching**
>
> Given a graph G, and a matching M of G, M is a **maximum** matching if it has the maximum possible cardinality $|M|$ for any matching of G.

Example 6.1 (Maximal and maximum). placeholder ──────────────────

These concepts of *maximal* (or *minimal*) and *maximum* (or *minimum*) will be applied accordingly to every other structure discussed.

Moreover, consider the following definition on bipartite graphs.

> **Definition 6.3: X-perfect matching**
>
> Given a bipartite graph G, bipartite into two sets X and Y, an X-**perfect matching** is a matching which covers every vertex in X.

Given a bipartite graph $G = (V = X \cup Y, E)$, in this notes, when referring to *any type of matching* on bipartite graphs, unless explicited, it will be assumed that we are reffering to an X-type of matching, where X is the set of smaller cardinality between X and Y.

A *maximal matching* can be found using a **greedy algorithm**, which is simple and efficient but does not necessarily yield the largest possible matching. Finding a *maximum matching* is a *polynomial-time problem*, but it requires more sophisticated algorithms and is more complex than finding a maximal matching.

As we already discussed, not all graphs have a *perfect matching*, but when one exists, certain theorems for bipartite graphs — like the following, proved by Philip Hall in 1935 — can help determine its existence and structure.

Given a set of vertices $S \subseteq V(G)$ of a graph G, let

$$\delta(S) := \{v \in V(G) \mid \exists x \in V(G) : (v, x) \in E(G)\}$$

be S's **neighbourhood**.

> **Theorem 6.1: Hall's marriage theorem**
>
> Given a bipartite graph G, bipartite into V_1 and V_2, where $|V_1| \leq |V_2|$, G has a V_1-perfect matching if and only if for each set S of k nodes in V_1 there are at least k nodes in V_2 adjacent to some node in S. Using symbols
>
> $$\forall S \subseteq V_1 \quad |S| \leq |\delta(S)|$$

Proof.

First implication. Consider a V_1-perfect matching M of G, and let $S \subseteq V_1$ be a set of vertices; by definition of M, each node $s \in S$ is matched through M with a different node in $\delta(S)$, therefore $|S| \leq |\delta(S)|$.

Second implication. Assume that $\forall S \subseteq V_1 \quad |S| \leq |\delta(S)|$ and, by way of contradiction, assume that the cardinality of the V_1-maximum matching M is such that $|M| < |V_1|$, therefore M is not a V_1-perfect matching. Thus, there must at least one vertex $u_0 \in V_1$ which is not covered by M. Let $S = \{u_0\}$; by hypothesis, we have that $1 = |S| \leq |\delta(S)|$, therefore there must exist at least 1 vertex in $\delta(S)$, which means that u_0 is adjacent to at least one vertex. Let this vertex be $v_1 \in V_2$. Now, two cases may occur:

- if v_1 is not covered by M, then placeholder _____ what?

- instead, if v_1 is covered by M, consider the vertex matched with v_1 through M, and let this node be $u_1 \in V_1$

Now, consider a second set $S = \{u_0, u_1\}$; by hypothesis, we have that $2 = |S| \leq |\delta(S)|$, which implies that there must exist at least one other node in $\delta(S)$, other than v_1, which must be adjacent either to u_0 or u_1. Let this vertex be $v_2 \in V_2$.

At this point, the same argument as before can be applied, but since G is finite, we will eventually reach a node v_r TODO _____ WHAT

□

This theorem establishes a criterion for the *existence* of a perfect matching, but does not provide an efficient, algorithmic method to construct one — finding the matching by enumerating all subsets of V_1 requires exponential time, making it impractical for large sets.

6.1.1 Flow networks

A **flow network** is a directed graph where each edge has a **capacity**, and each edge receives a *flow*. The amount of flow on an edge cannot exceed the capacity of the edge.

> **Definition 6.4: Flow network**
>
> A **flow network** is a directed simple graph $G = (V, E)$ with a non-negative **capacity** function $c : E(G) \to \mathbb{R}^+$, in which two nodes are distinguished, namely the **source** — commonly indicated with s — and a **sink** — usually denoted with t.

Example 6.2 (Flow networks)**.** The following is an example of a *flow network*. placeholder _____ add pic

A **flow** is a map $f : E(G) \to \mathbb{R}$ that satisfies the following two constraints:

- **skew symmetry** the flow on an arc from u to v is equivalent to the negation of the flow on the arc from v to u; the sign of the flow indicates the flow's direction

$$f(u, v) = -f(v, u)$$

- **capacity constraint**: the flow of an edge cannot exceed its capacity

$$\forall (u, v) \in E(G) \quad f(u, v) \leq c(u, v)$$

- **conservation of flows**: the sum of the flows entering a node must equal the sum of the flows exiting that node (except for s and t)

$$\forall v \in V(G) - \{s, t\} \quad \sum_{\substack{u:(u,v)\in E(G) \\ f(u,v)>0}} f(u, v) = \sum_{\substack{u(u,v)\in E(G) \\ f(u,v)>0}} f(v, u)$$

The **value of flow** is defined as the amount of flow passing from s to t

$$|f| = \sum_{v:(s,v)\in E(G)} f(s, v) = \sum_{u:(u,t)\in E(G)} f(u, t)$$

Definition 6.5: Maximum flow problem

Given a flow network G, the **maximum flow problem** is to route as much flow as possible from G's source to G's sink, in other words find the flow f with maximum value $|f|$.

Theorem 6.2

The perfect matching problem on bipartite graphs is equivalent to the maximum flow problem on flow networks.

Proof. Consider a bipartite graph $G = (V = V_1 \cup V_2, E)$, and construct a flow network $G' = (V', E')$ as follows:

- let s and t be new nodes, i.e. the source and the sink, respectively; then $V' := V \cup \{s, t\}$

- let $E_1 := \{(s, u) \mid u \in V_1\}$, $E_2 := \{(v, t) \mid v \in V_2\}$ be the set of edges connecting the source to each node in V_1, and the one connecting each node in V_2 to the sink, respectively; moreover, let

$$E_e := \{(u, v) \mid u \in V_1, v \in V(G), (u, v) \in E(G)\}$$

  be the set of edges that connects V_1 and V_2 exactly as they were connected in G; finally $E' := E_1 \cup E_e \cup E_2$

- let the capacity be a function $c : E(G) \to \mathbb{R}^+$ such that

$$\forall u, v \in V'(G') \quad c(u, v) = 1$$

placeholder ⌐ add pic

Consider a perfect matching M on the bipartite graph G, by defining an integer-valued flow f on the flow network G', it is easy to see that f is a maximum flow, since the capacity of each edge is 1; the other implication can be proved with a similar reasoning. $\square$

Theorem 6.3: Integral flow theorem

Given a flow network $G = (V, E)$, and a capacity function c, if c only assumes integer values, there exists a max flow f for G such that $|f|$, and $f(u, v)$ for any $u, v \in V(G)$, are integers.

Corollary 6.1

The cardinality of a maximum matching M on a bipartite graph G is equal to the value of the maximum flow f in the associated flow network G', therefore $|M| = |f|$.

The Ford-Fulkerson algorithm for computing the *maximum flow* in a flow network has a time complexity of $O(m\,|f|)$; however, in our specific case, the maximum flow in G' is bounded above by $\mu := \min(|X|, |Y|)$, where $V' = X \cup Y$ — recall that this is a X-perfect matching, where $|X| \leq |Y|$. Consequently, an algorithm that leverages the maximum flow to find a maximum matching has an overall complexity of $O(m\mu)$.

Bibliography

[1]   S. Arora. "Polynomial time approximation schemes for Euclidean TSP and other geometric problems". In: *Proceedings of 37th Conference on Foundations of Computer Science*. SFCS-96. IEEE Comput. Soc. Press, 2–11. DOI: 10.1109/sfcs.1996.548458. URL: http://dx.doi.org/10.1109/SFCS.1996.548458.

[2]   Yair Bartal et al. "The traveling salesman problem: low-dimensionality implies a polynomial time approximation scheme". In: *Proceedings of the forty-fourth annual ACM symposium on Theory of computing*. STOC'12. ACM, May 2012, 663–672. DOI: 10.1145/2213977.2214038. URL: http://dx.doi.org/10.1145/2213977.2214038.

[3]   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).

[4]   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.

[5]   Xiuzhen Cheng et al. "A polynomial-time approximation scheme for the minimum-connected dominating set in ad hoc wireless networks". In: *Networks* 42.4 (Sept. 2003), 202–208. ISSN: 1097-0037. DOI: 10.1002/net.10097. URL: http://dx.doi.org/10.1002/net.10097.

[6]   Nicos Christofides. "Worst-Case Analysis of a New Heuristic for the Travelling Salesman Problem". In: *Operations Research Forum* 3.1 (Mar. 2022). ISSN: 2662-2556. DOI: 10.1007/s43069-021-00101-z. URL: http://dx.doi.org/10.1007/s43069-021-00101-z.

[7]   Brent N. Clark et al. "Unit disk graphs". In: *Discrete Mathematics* 86.1–3 (Dec. 1990), 165–177. ISSN: 0012-365X. DOI: 10.1016/0012-365x(90)90358-o. URL: http://dx.doi.org/10.1016/0012-365X(90)90358-O.

[8]   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. URL: http://dx.doi.org/10.1145/800157.805047.

[9]   G. Dantzig et al. "Solution of a Large-Scale Traveling-Salesman Problem". In: *Journal of the Operations Research Society of America* 2.4 (1954), pp. 393–410. ISSN: 00963984. URL: http://www.jstor.org/stable/166695 (visited on 11/04/2024).

[10]   E.W. Dijkstra. "A Note on Two Problems in Connexion with Graphs." In: *Numerische Mathematik* 1 (1959), pp. 269–271.

[11]   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. URL: http://dx.doi.org/10.1002/1097-0037(200009)36:2<91::AID-NET3>3.0.CO;2-4.

[12] 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. URL: http://dx.doi.org/10.1002/(SICI)1097-0037(199707)29:4<219::AID-NET5>3.0.CO;2-I.

[13] Robert W. Floyd. "Algorithm 97: Shortest path". In: *Commun. ACM* 5.6 (June 1962), p. 345. ISSN: 0001-0782. DOI: 10.1145/367766.368168.

[14] L. R. Ford. *Network Flow Theory.* Santa Monica, CA: RAND Corporation, 1956.

[15] S. Guha et al. "Approximation Algorithms for Connected Dominating Sets". In: *Algorithmica* 20.4 (Apr. 1998), 374–387. ISSN: 1432-0541. DOI: 10.1007/pl00009201. URL: http://dx.doi.org/10.1007/PL00009201.

[16] 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. URL: http://dx.doi.org/10.1007/978-1-4684-2001-2_9.

[17] 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.

[18] David Lichtenstein. "Planar Formulae and Their Uses". In: *SIAM Journal on Computing* 11.2 (May 1982), 329–343. ISSN: 1095-7111. DOI: 10.1137/0211025. URL: http://dx.doi.org/10.1137/0211025.

[19] 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.

[20] G N Purohit et al. "Constructing Minimum Connected Dominating Set: Algorithmic Approach". In: *International Journal on Applications of Graph Theory In wireless Ad Hoc Networks And sensor Networks* 2.3 (Sept. 2010), 59–66. ISSN: 0975-7260. DOI: 10.5121/jgraphoc.2010.2305. URL: http://dx.doi.org/10.5121/jgraphoc.2010.2305.

[21] Lu Ruan et al. "A greedy approximation for minimum connected dominating sets". In: *Theoretical Computer Science* 329.1–3 (Dec. 2004), 325–330. ISSN: 0304-3975. DOI: 10.1016/j.tcs.2004.08.013. URL: http://dx.doi.org/10.1016/j.tcs.2004.08.013.

[22] Weiwei Shi et al. "Adding Duty Cycle Only in Connected Dominating Sets for Energy Efficient and Fast Data Collection". In: *IEEE Access* 7 (2019), 120475–120499. ISSN: 2169-3536. DOI: 10.1109/access.2019.2937626. URL: http://dx.doi.org/10.1109/ACCESS.2019.2937626.

[23] Roberto Solis-Oba. "2-Approximation Algorithm for Finding a Spanning Tree with Maximum Number of Leaves". In: *Algorithms — ESA' 98.* Springer Berlin Heidelberg, 1998, 441–452. ISBN: 9783540685302. DOI: 10.1007/3-540-68530-8_37. URL: http://dx.doi.org/10.1007/3-540-68530-8_37.

[24] Gábor Szárnyas. *Graphs and matrices: A translation of "Graphok 'es matrixok" by D'enes Kőnig (1931).* Sept. 2020. DOI: 10.48550/arXiv.2009.03780.

[25] 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.

[26] Shuichi Ueno et al. "On the nonseparating independent set problem and feedback set problem for graphs with no vertex degree exceeding three". In: *Discrete Mathematics* 72.1–3 (Dec. 1988), 355–360. ISSN: 0012-365X. DOI: 10.1016/0012-365x(88)90226-9. URL: http://dx.doi.org/10.1016/0012-365X(88)90226-9.

[27] 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.

[28] Stephen Warshall. "A Theorem on Boolean Matrices". In: *J. ACM* 9.1 (Jan. 1962), 11–12. ISSN: 0004-5411. DOI: 10.1145/321105.321107.

[29] 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. URL: http://dx.doi.org/10.1007/978-3-642-68402-9_21.

[30] 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. URL: http://dx.doi.org/10.1109/FMPC.1999.750589.