SAPIENZA
UNIVERSITÀ DI ROMA

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

Network Algorithms

Lecture notes integrated with the book TODO

Author
Alessio Bandiera

February 21, 2025

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:

It is suggested that the "Progettazione di Algoritmi" and the "Automi: Calcolabilità e Complessità" courses were studied before approaching this course.

Licence:

1 The routing problem

1.1 Introduction on graphs

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

Definition 1.1: Graph

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

Graphs are usually represented through circles and lines, 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**.

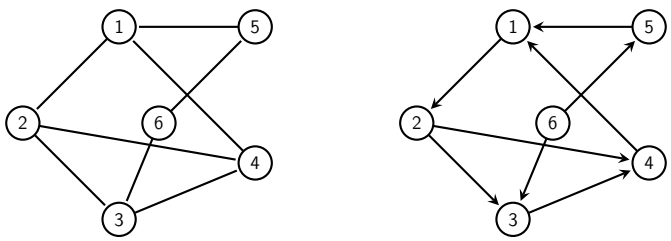


Figure 1.1: On the left: a simple graph. On the right: a simple digraph.

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?

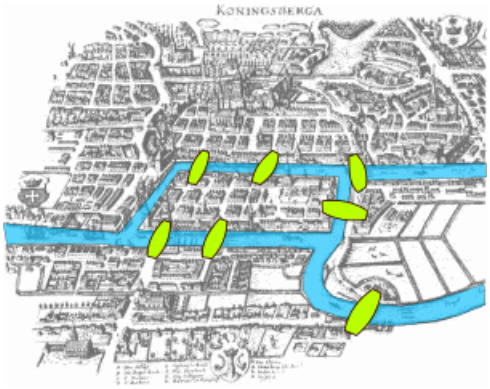


Figure 1.2: The city of Königsberg and its seven bridges.

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**.

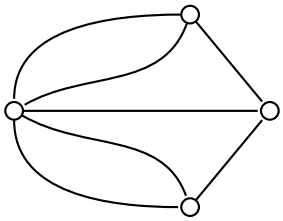


Figure 1.3: The multi-graph representing the *Seven Bridges of Königsberg* problem.

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.

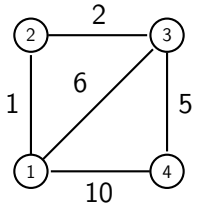


Figure 1.4: An undirected weighted graph.

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 $\text{dist}(u, v)$, is the minimum weight of all the paths $u \to v$ of G.

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

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 [42].

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 [4] and Ford [24] 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 [19] 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 [23] and Warshall [56], 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 × n matrix, initialized with every cell at +∞
3:      for u ∈ V(G) do
4:          dist[u][u] = 0
5:      end for
6:      for (u, v) ∈ E(G) do
7:          dist[u][v] = w(u, v)
8:      end for
9:      for k ∈ V(G) do
10:         for u ∈ V(G) do
11:             for v ∈ V(G) do
12:                 dist[u][v] = min(dist[u][v], dist[u][k] + dist[k][v])
13:             end for
14:         end for
15:     end for
16: end function
```

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

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

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

1.3 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.

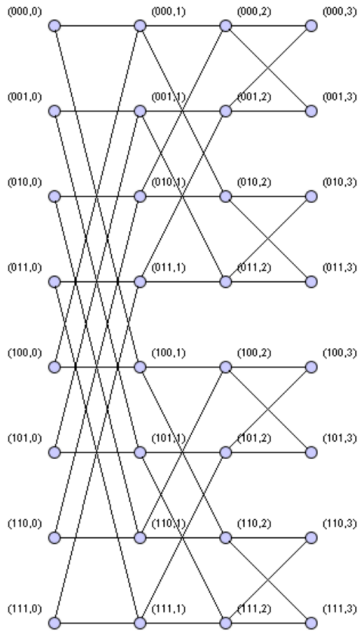


Figure 1.5: A butterfly network.

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.

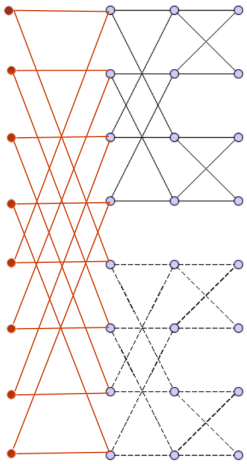


Figure 1.6: The recursive structure of butterfly networks.

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).

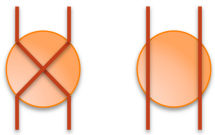


Figure 1.7: A butterfly network node.

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.

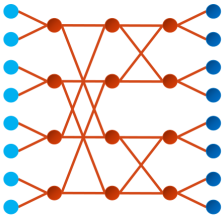


Figure 1.8: An extended butterfly network.

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 w_i == w'_i then
4:             Traverse a straight edge
5:         else
6:             Traverse a cross edge
7:         end if
8:     end for
9: end function
```

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

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

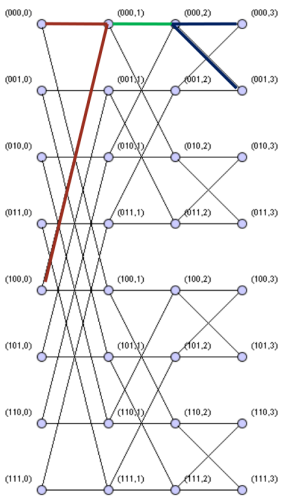
describes the permutation of the packet destinations. In a **greedy routing algorithm**, each packet follows its *greedy path*, meaning that at each intermediate layer, it makes progress toward its final destination by choosing the edges to cross through the algorithm described in Proposition 1.1.

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

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

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

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



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

of them may be delayed by $\sqrt{\frac{N}{2}} - 1$ steps. Since it takes $n = \log N$ steps to traverse the whole network, the greedy algorithm can take up to

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

steps to route a permutation.

The following theorem generalizes this result.

Theorem 1.1: Butterfly routing

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

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

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

Note that any packet that 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

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

□

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

1.3.2 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.

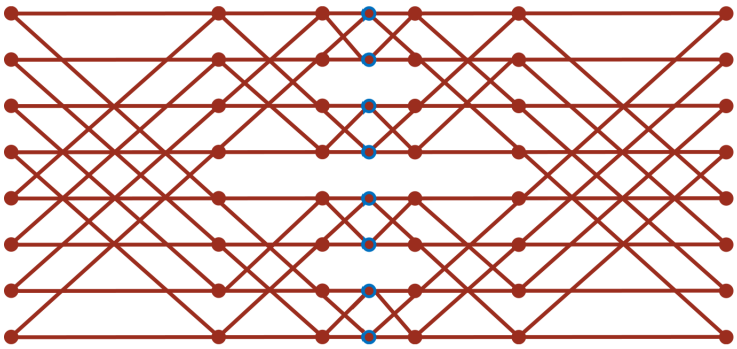


Figure 1.9: A Beneš network.

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.

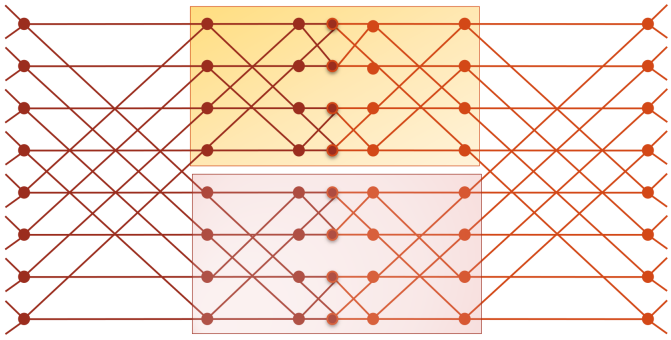


Figure 1.10: Subnetworks of a Beneš network.

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.

□

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

Theorem 1.3: Node-disjoint paths in Beneš networks

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

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

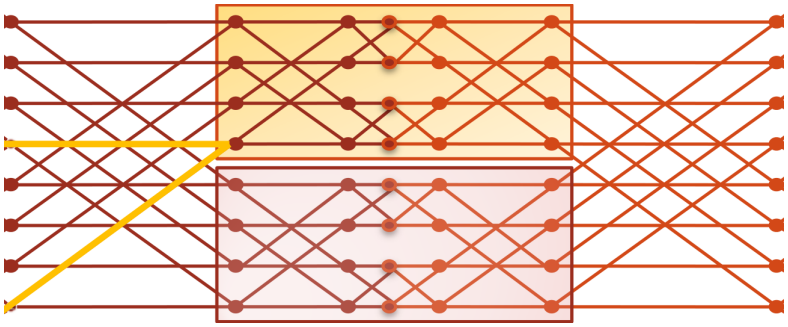


Figure 1.11: Mates in this type of Beneš network.

□

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

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

1.3.3 Mesh networks

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

Definition 1.6: Mesh network

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

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

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

- 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.

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 Very Large Scale Integration (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 [52], 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.

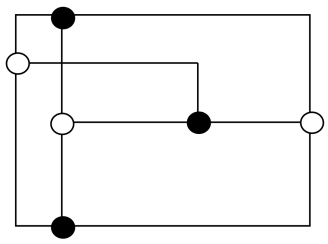


Figure 2.1: An orthogonal drawing.

- **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.

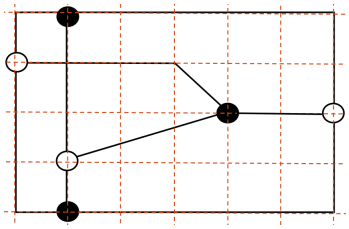


Figure 2.2: A grid drawing.

- **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;

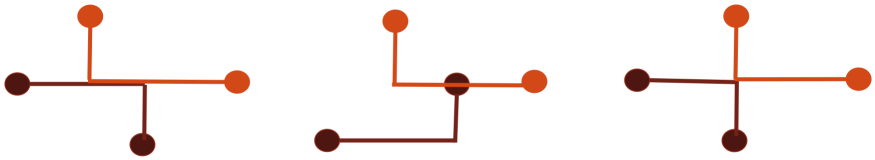


Figure 2.3: An overlapping, a node-edge cross, and a *knock-knee*.

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 [39] and Valiant [54], 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.

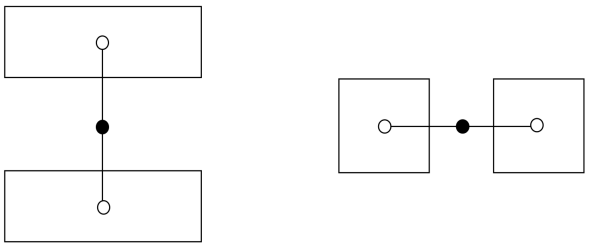


Figure 2.5: The inductive step of the inductive H tree construction.

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

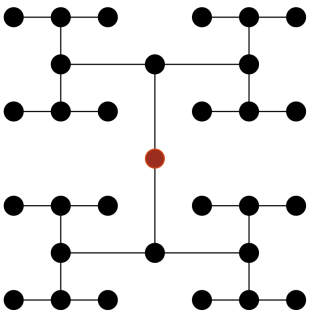


Figure 2.6: H tree of height $h = 4$.

Leiserson [39] and Valiant [54] 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. [7] 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.

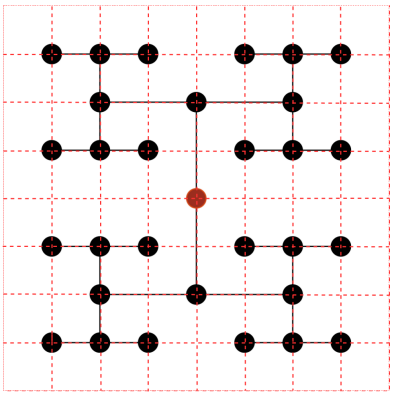


Figure 2.7: The grid of the H tree.

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.

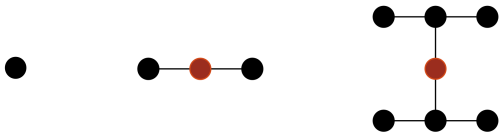


Figure 2.8: Cases for $h = 0$, $h = 1$ and $h = 2$.

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

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

and analogously

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

Hence, the area is

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

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

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

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

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

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

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

□

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. [59], 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*

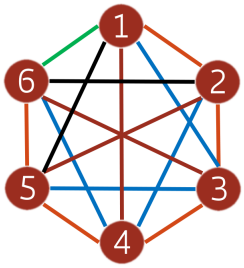


Figure 2.9: A 6-clique.

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.

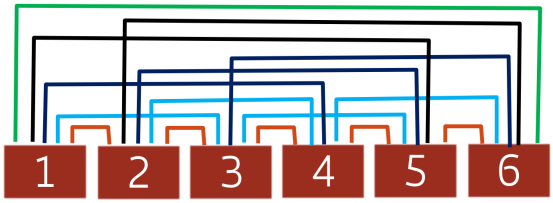


Figure 2.10: Arrangement of links in tracks

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:

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

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:

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

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

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

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 $\left\lfloor \frac{w}{2} \right\rfloor$ 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

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

□

2.1.3 The Wise layout

In a paper published by Wise [58], it was proposed a *different layout* for the butterfly network (discussed in Section 1.3.1), which is shown in the following figure.

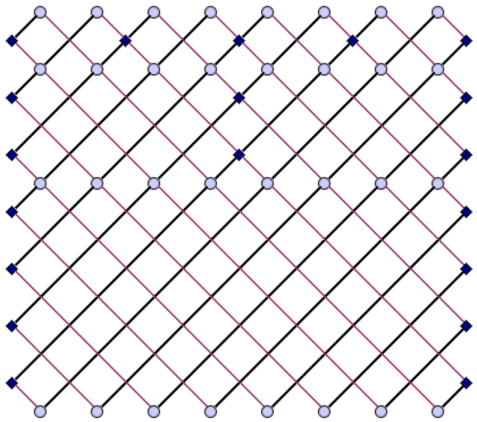


Figure 2.11: The alternative layout of the butterfly network.

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*.

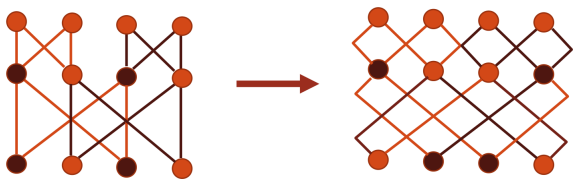


Figure 2.12: Rearrangement to get the Wise layout.

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 $\sqrt{2}(N-1)$, which is

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

- 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.

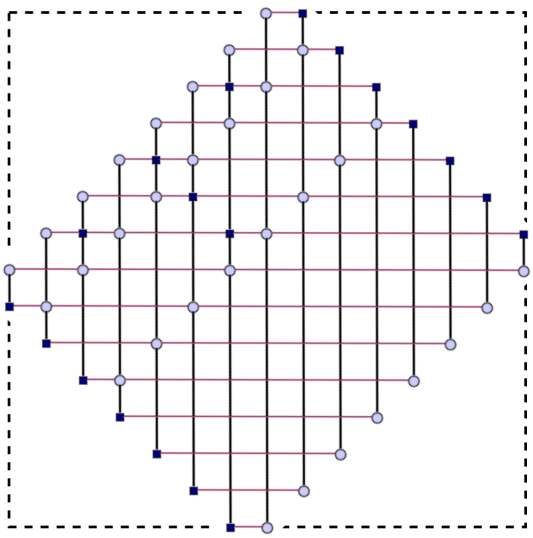


Figure 2.13: The *actual* Wise butterfly layout.

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. [21] 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.

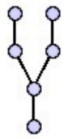


Figure 2.14: A layered graph.

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.

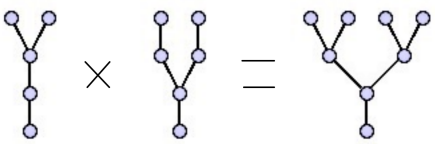


Figure 2.15: An LCP between two graphs.

LCPs are particularly useful because Even et al. [22] 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.

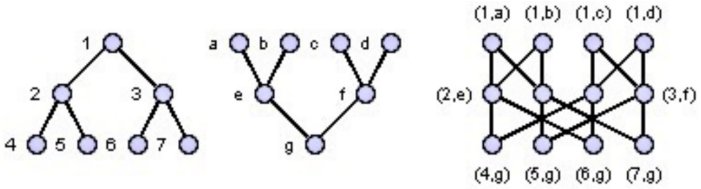


Figure 2.16: The LCP that defines the 2-dimensional butterfly network.

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

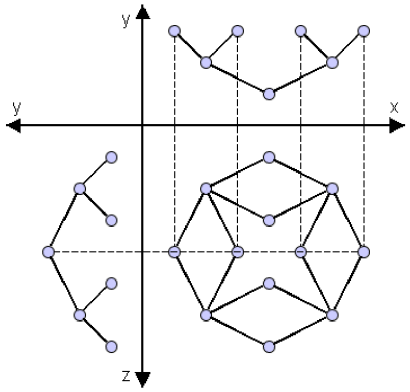


Figure 2.17: The LCP of the two graphs is obtained by this projection.

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.

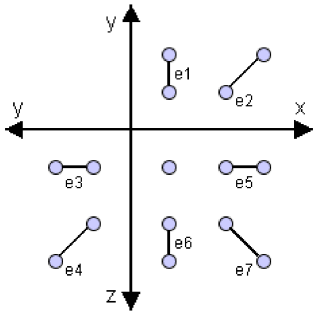


Figure 2.18: Possible edge cross products.

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:

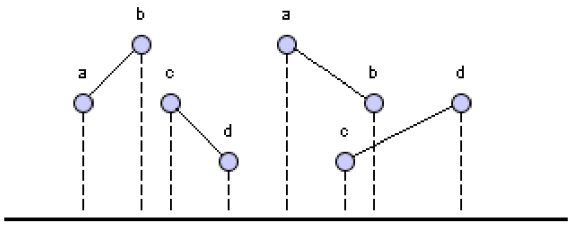


Figure 2.19: Consistent and inconsistent pair of edges, from left to right.

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.

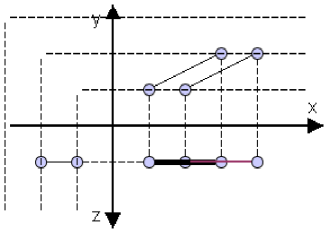


Figure 2.20: First inconsistency case

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.

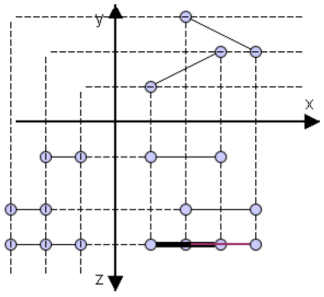


Figure 2.21: Second inconsistency case

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.

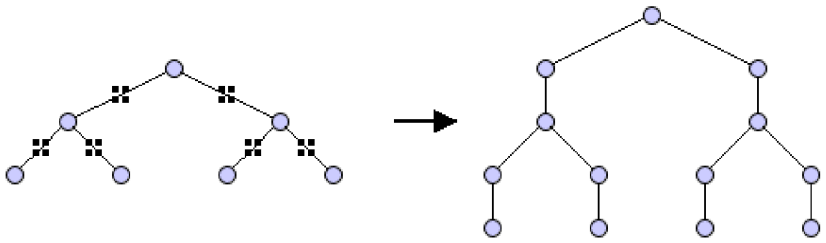


Figure 2.22: A complete layered binary tree with the nodes doubled.

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.

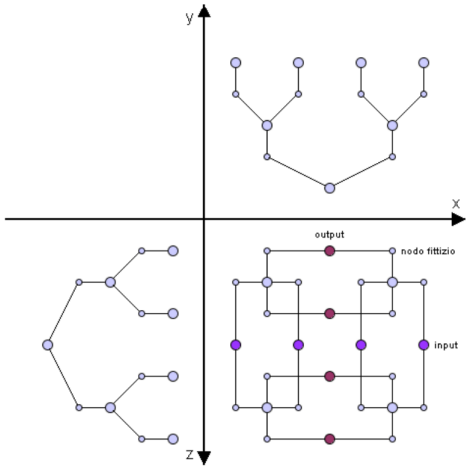


Figure 2.23: The LCP of two such trees.

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.

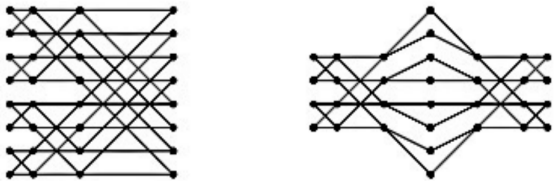
2.1.5 Optimal area of the butterfly network

In this section we will just show how it is possible to rearrange a butterfly network to get the optimal area, but we will not delve into the details of the proof.

Lemma 2.1

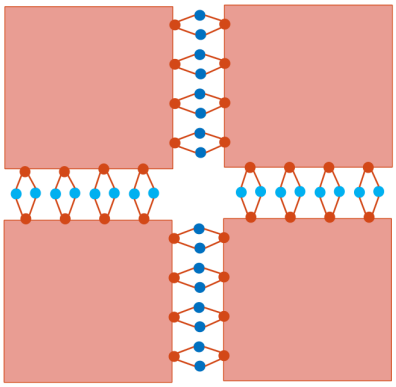
Given an n-dimensional butterfly network, for any non-negative integers $j, k > 0$ such that $0 \leq j \leq j + k \leq n$, the subgraph of the butterfly network induces by the nodes in levels $j, j+1, \ldots, j+k$ is the disjoint union of 2^{n-k} copies of k-dimensional butterfly networks.

In particular, if $j = 0$ and $k = n - 1$, we have the following:

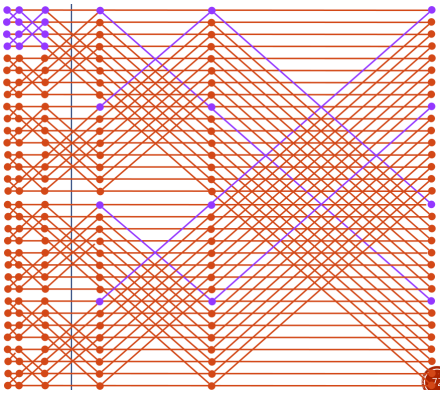


Therefore, an $(n-1)$-dimensional butterfly network can be built from a pair of $(n-2)$-dimensional butterfly networks connected by one node layer and one ege layer.

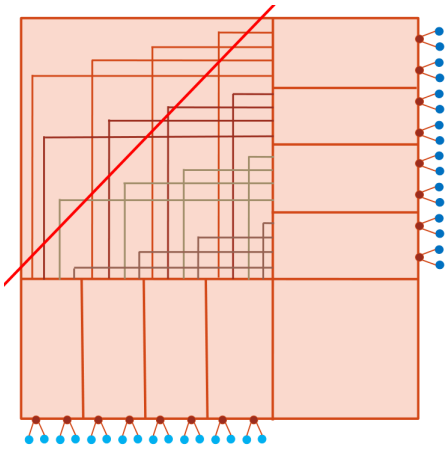
If we cut out the input and output nodes from an n-dimensional bufferfly network, we get the following:



and, in turn, each of the $(n-2)$-dimensional butterfly networks can be cut into many smaller butterfly networks, as shown below:



Lastly, although we will not cover the details of this result, it is possible to rearrange the butterfly such that the smaller butterfly networks can be connected as shown in the following image



and, for the case of the slanted layout, it possible to bend the cables along the *red line* shown in the previous image.

2.1.6 The hypercube network

The **hypercube network** layout is widely utilized in parallel computing due to its advantageous properties, including *high regularity*, a *logarithmic diameter*, and strong *fault tolerance*. These characteristics make it an efficient and reliable choice for various computational tasks.

Definition 2.8: n-dimensional hypercube

An **n-dimensional hypercube** Q_n has $N = 2^n$ nodes and $\frac{1}{2}n2^n$ edges. Each node is labeled with an n-bit binary string, and any two nodes are linked with an edge if and onlyh if the corresponding binary strings differ in precisely one bit.

Example 2.6 (Hypercubes)**.** The following are examples of hypercubes.

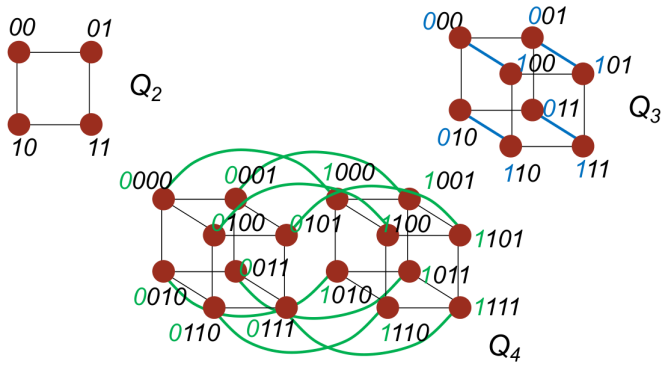


Figure 2.24: Some hypercube network, namely Q_2, Q_3 and Q_4.

In particular, from this example it is easy to see that Q_n can be built by joining, with additional edges, nodes in 2 different copies of Q_{n-1}, assuming they have the same label. Then, in the resulting structure, to obtain Q_n it is sufficient to prefix with a 0 and with a 1 different endpoints of the newly created edges. Note that the new edges will form a **perfect matching**.

Additionally, it can be proven that cutting the Q_n hypercube over these newly added edge is precisely the cut that yields the **bisection width** of Q_n.

Proposition 2.2: Bisection width of hypercubes

The bisection width of Q_n is $\frac{N}{2}$.

Together with Theorem 2.2, this property yields the following result.

Corollary 2.1

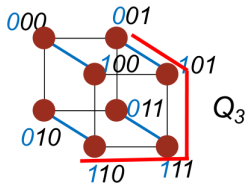
Each hypercube network Q_n must have area at least $\frac{N^2}{16}$.

Lastly, consider the following property.

Proposition 2.3: Diameter of hypercubes

Q_n has diameter n.

We will not cover the details of the proof of this property, but it can be intuitively seen from the following example of Q_3, as shown below.



Interestingly, it is possible to represent any hypercube network through a **collinear layout**. For instance, consider the following conversion between a Q_2 and the corresponding collinear layout having 2 tracks:

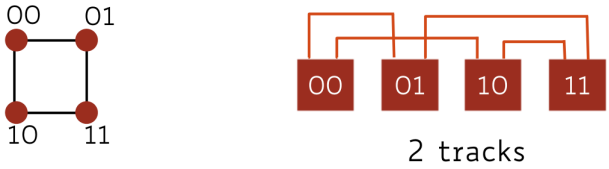


Figure 2.25: Conversion between a Q_2 and its collinear layout.

In general, if n is odd, and Q_{n-1} requires $f(n-1)$ tracks to be represented as a collinear layout, then Q_n requires the following number of tracks

$$n \text{ odd} \implies f(n) = 2 \cdot f(n-1) + 1$$

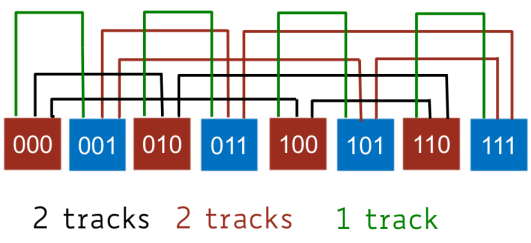


Figure 2.26: Conversion between a Q_3 and its collinear layout.

while if n is even, we just need to place two copies of Q_{n-2} next to each other in order to obtain Q_n, and

$$n \text{ even} \implies f(n) = 4 \cdot f(n-2) + 2$$

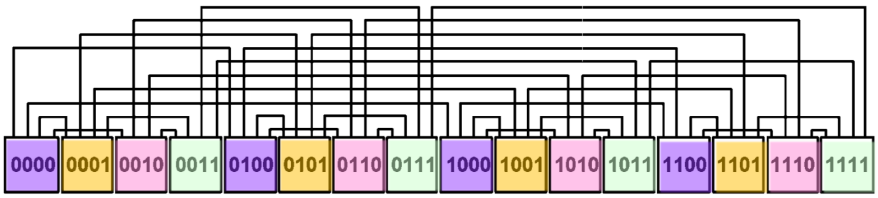


Figure 2.27: Conversion between a Q_4 and its collinear layout.

Solving the recurrence relations provided above, we derive the following result.

Theorem 2.3

The number of tracks required for the collinear layout of Q_n is $\frac{2}{3}N$.

Finally, although the details will be omitted in these notes, if can be proven the following theorem.

Theorem 2.4

Q_n can be laid out in $\frac{4}{9}N^2 + o(N^2)$.

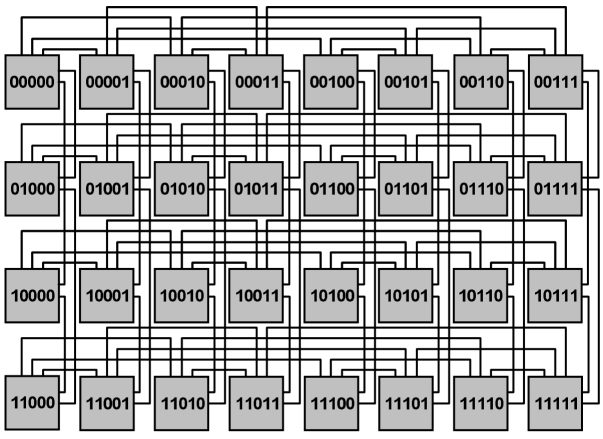


Figure 2.28: The arrangement of a Q_5's collinear layout.

3

The worm propagation prevention problem

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

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

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

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

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

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

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

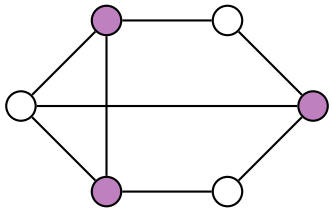


Figure 3.1: A vertex cover.

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 [36], which are a collection of well-known computational problems that were classified as NP-Complete shortly after the introduction of the Cook theorem [17].

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:

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

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

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

Despite the fact that the VC problem is NP-Complete, it is possible to find an *approximate solution* in polynomial time by leveraging its ILP formulation, by employing an ILP approximated solution.

3.1.1 Approximation algorithms

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

Algorithm 3.1: First greedy AVC

Given an undirected graph G, the algorithm finds an approximated 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

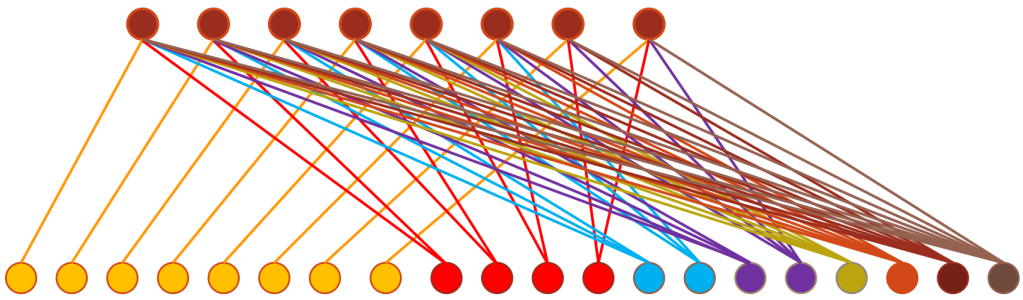
Given an undirected graph G, the algorithm finds an approximated minimum vertex cover for G.

1: **function** SECONDGREEDYAVC(G)
2: $V' := \varnothing$
3: **while** $E(G) \neq \varnothing$ **do**
4: $v \in \arg\max_{v \in V(G)} \deg(v)$
5: $V' = V' \cup \{v\}$
6: **for** $(u, v) \in E(G)$ **do** ▷ any edge having v as an endpoint
7: G.`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:



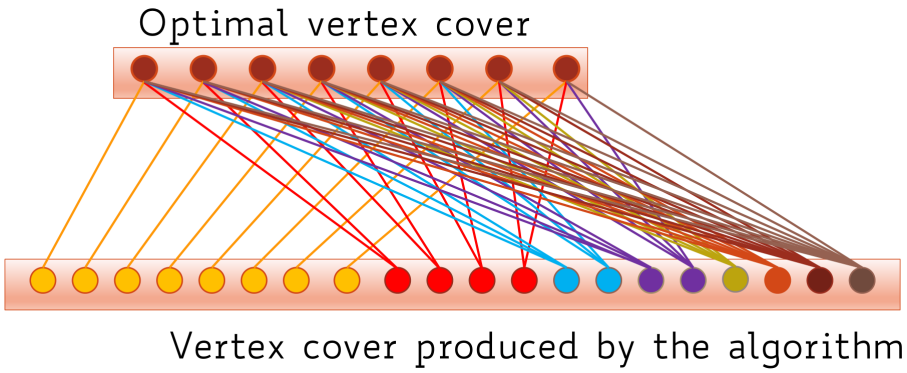
The upper row of nodes consists of r nodes, and the lower row consists of

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

meaning that the total number of nodes is

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

(note that the harmonic sum can be approximated by $\Theta(\log r)$). Although the **optimal** 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: $V' := \varnothing$
3: $E' := E(G)$
4: **while** $E' \neq \varnothing$ **do**
5: Choose $(i, j) \in E'$
6: $V' = V' \cup \{i, j\}$
7: **for** $(i, k) \in E(G)$ **do** ▷ any edge having i as an endpoint
8: $E' = E' - \{(i, k)\}$
9: **end for**
10: **for** $(j, h) \in E(G)$ **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

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

□

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 $|V'| \leq 2\,|V^*|$, 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** $x_i \geq \frac{1}{2}$ **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 $x_i^* \geq \frac{1}{2}$.

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

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

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 $x_i^* \leq 1$ for each $i \in [1,n]$, it must be that $Z \leq |V^*|$. 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 $y_i^* \leq 2x_i^*$, therefore

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

□

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 $V(G) - V'$.

□

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

$$\forall (x, y), (u, v) \in M \quad x, y \neq u, v$$

The nodes that are not covered by a matching are called **free nodes**.

Definition 3.5: Perfect matching

Given a graph G, a **perfect matching** is a matching that covers every vertex of 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 contrapositive 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

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

□

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 [50], 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. □

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. □

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. □

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. □

4

The frequency assignment problem

The introduction of new services, such as *data communication* (e.g., internet, emails and video conferencing), has led to **capacity shortages** in existing wired networks. This increasing demand highlights the need for alternative solutions in order to communicate over the internet effectively.

One such alternative is provided by **fixed wireless networks**, which consist of *wireless devices* positioned at fixed locations, such as on buildings or towers. These devices form a network through radio or other wireless connections, operating without reliance on established infrastructure or centralized control.

Fixed wireless networks offer a viable solution for *extending* the capacity of wired networks, providing flexibility and scalability while bypassing the limitations of traditional wired infrastructure. Moreover, they offer significant benefits, particularly in areas where traditional wired infrastructure is unavailable or impractical, for example

- **connecting remote areas**: they enable connectivity in rural or hard-to-reach locations without the need for costly and time-consuming cable installation
- **viable broadband solution**: in rural regions lacking wired infrastructure, fixed wireless broadband serves as a practical and efficient option for providing high-speed internet access

This technology bridges the digital divide by bringing connectivity to underserved communities, supporting activities such as communication, education, and business.

One of the most widely adopted applications of wireless communication is the creation of **fixed cellular telecommunication networks**. Unlike mobile cellular networks, where transmitters and receivers are mobile, fixed cellular networks feature *transmitters* and *receivers* positioned at fixed locations within the area of interest. These networks offer a cost-effective alternative to building traditional wired infrastructure. By eliminating the need for extensive cable installation, fixed cellular networks reduce deployment costs while maintaining reliable connectivity, making them an attractive solution for expanding

telecommunications, particularly in areas where wired networks are expensive or impractical to implement.

Fixed wireless services typically employ **directional radio antennas** at both ends of the signal to ensure reliable and efficient communication. These antennas are carefully aligned to maintain a stable connection.

Unlike mobile or portable wireless devices, which are often battery-powered, fixed wireless devices usually draw their electrical power from the *public utility mains*. This setup ensures continuous operation without the limitations of battery life, making fixed wireless an ideal solution for stable, long-term connectivity.

Wireless communication between two points is achieved using a **transmitter** and a **receiver**, and the wireless devices communicate over the air using *radio frequencies*. Therefore, if the radio frequencies are not utilized carefully, signals from different devices or networks could *overlap*, causing interference that degrades communication quality or renders it impossible. In fact, the **radio frequency spectrum** is a *finite* resource. Hence, it is necessary to develop a proper strategy to perform a **frequency assignment** in order to avoid overlapping problems.

Frequency assignment plays a crucial role in the operation of many different types of wireless networks. The specific requirements and methods for assigning frequencies can vary significantly depending on the nature of the network. As a result, the concept of frequency assignment is interpreted differently across various applications. This diversity has led to the development of multiple "flavors" of frequency assignment strategies, each tailored to address the unique challenges and constraints of specific network scenarios, as extensively discussed in the literature.

In a wireless communication, the *transmitter* generates electrical oscillations at a specific *radio frequency* (RF), which are then converted into electromagnetic waves for propagation through the air. The *receiver*, located at the destination point, detects these waves and converts them back into electrical signals, completing the communication process. This fundamental mechanism enables the transmission of data over varying distances without the need for physical connections. However, several challenges may affect the reliability and efficiency of wireless communication, some of which are listed below.

- In point-to-point connections, the transmitter and receiver must have a clear **line-of-sight**, meaning there should be no obstacles (e.g., buildings, trees, or terrain) blocking the signal path. To overcome this, transmitters and receivers are often installed at *elevated locations*, such as rooftops or towers, to reduce obstructions and improve signal reliability. The distance between the transmitter and receiver also plays a critical role, as signals weaken over long distances or in obstructed environments.
- As already mentioned, signals from multiple transmitters can **interfere** with one another, especially when operating on similar frequencies. Crossed or overlapping signals degrade communication quality and can lead to dropped connections or noise. To minimize interference, frequency reuse must be carefully planned. Frequencies can only be *reused* if transmitters are far enough apart to avoid signal overlap. Otherwise, the quality of the communication link may be compromised.

- The rapid expansion of new wireless services, such as digital cellular networks, has led to a *growing demand* for frequencies in the radio spectrum, which is a scarce and valuable resource, and the high cost of acquiring and licensing frequencies necessitates *efficient* use of the available spectrum. **Frequency reuse** strategies can help achieve significant cost savings by allowing the same frequency to be used by transmitters in non-overlapping or distant areas.

Therefore, it is crucial to develop efficient **frequency assignment strategies**, to mitigate the challenges that affect wireless communication. In particular, a solution to the frequency assignment problem seeks to balance two *competing objectives*:

- **economies of frequency reuse**: maximizing the efficient reuse of available frequencies to minimize costs and maximize capacity
- **quality of communication**: ensuring that frequency reuse does not degrade the quality of the network through interference or reduced performance

This balance is achieved by quantifying the trade-offs between these aspects, which transforms the problem into a mathematical optimization challenge. By applying optimization techniques, the goal is to *allocate frequencies* in a way that meets performance requirements while minimizing interference and resource usage. The class of problems aimed at solving frequency assignment challenges is called **Frequency Assignment Problems** (FAPs).

In general, FAPs consist of two primary components:

- **aim**: assign frequencies to a set of wireless communication connections; the frequencies must be selected from a predefined set, which may vary depending on the geographic location or other contextual factors.
- **constraint**: assigned frequencies should avoid interference; as we already discussed, if two connections use frequencies that are too close (or the same frequency in overlapping areas), *interference* occurs, degrading the quality of the communication signal, which must be avoided

But what is **interference** precisely? For *interference* to occur between two wireless communication signals, two primary conditions must be met.

- **Frequency proximity**: The frequencies of the two signals must be close to each other on the *electromagnetic spectrum*. This can happen either when the frequencies are in close proximity or when they are harmonics of one another (i.e., integer multiples of the same frequency). **Doppler effects** can also contribute to interference, especially if the relative speed between the transmitter and receiver causes a shift in the frequency, making it closer to another signal's frequency. **Harmonics** are generally less of a concern in modern systems since the frequency bands available for assignment are typically narrow, and there is usually not enough room for harmonic overlap.
- **Geographical proximity**: The two connections (transmitter and receiver pairs) must be geographically close to each other. This proximity increases the likelihood of *signal overlap*, especially if both connections are operating on similar or the same

frequencies. In close range, signals can interfere due to the strength and spread of the radio waves, especially in urban environments with high obstacle densities.

Both the frequency proximity and geographical proximity aspects of interference are modeled in various ways across the literature. As a result, various models of frequency assignment exist, each tailored to specific network requirements or deployment scenarios. Therefore, in this notes a *simplified model* will be introduced, that captures the essential aspects of the problem and provides a basic framework for understanding the trade-offs between frequency reuse and interference.

Definition 4.1: Interference graph

Given a set of wireless stations that need to communicate, an **interference graph** is defined as follows:

- the graph has one vertex per station
- a *directed* edge connects two nodes u and v if the station represented by u may need to communicate with the station represented by v; note that any communication — i.e. edge — may lead to an interference
- the *colors* of the nodes represent the *channels* that are assigned to each corresponding station

Example 4.1 (Interference graphs)**.** The following is an example of an interference graph.

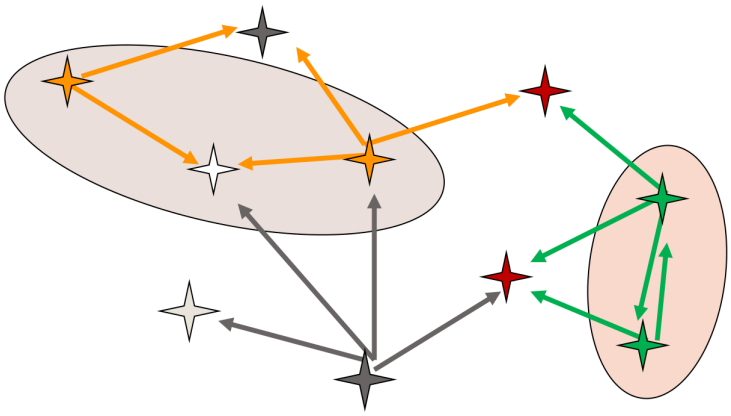


Figure 4.1: An interference graph.

Note that different colors represent different assigned channels for the various stations.

Moreover, note that in this example there are two highlighted regions. Those regions are **collision**, and this model is able to represent two different types of collisions.

- **Direct collisions**: these occur when two stations are located in *very close proximity* to each other and are assigned the *same frequency channel*. Because they are so close, their signals will overlap, causing interference. To avoid this, the frequencies assigned to these stations must be at least h apart, for some specified minimum

frequency separation h. In the example above, the rightmost ellipse represents a *direct collision*.

- **Hidden collisions**: these happen when two stations are located in *close proximity* to each other but are not directly within each other's signal range (i.e., they are not *line-of-sight* or are blocked by obstacles). However, their assigned frequencies still cause interference because they are close enough that their signals can interfere indirectly. In this case, the frequencies assigned to stations in these close locations must be at least k apart to avoid such interference, for some specified minimum frequency separation k. In the example above, the leftmost ellipse represents a *hidden collision*.

Therefore, given an interference graph, the problem of avoiding interference can be modeled as a **labeling** or **coloring** problem, which is discussed in the following section.

4.1 The $L(h,k)$-labeling problem

The labeling problem that will be considered in this chapter is defined as follows.

Definition 4.2: $L(h,k)$-labeling

Given a graph $G = (V, E)$, an **$L(h,k)$-labeling** of G is a *node coloring function* f that assigns colors to each node of G, such that:

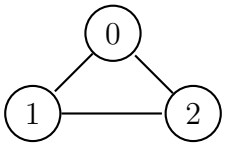
- $\forall (u,v) \in E(G) \quad |f(u) - f(v)| \geq h$
- $\forall u, v \in V(G) \quad \exists w \in V(G) \mid (u,w), (w,v) \in E(G) \implies |f(u) - f(v)| \geq k$

The words "labeling" and "coloring" will be used interchangeably in the following sections.

Note that in the literature two slightly different definitions of $L(h,k)$-labeling are usually utilized. In particular, the other definition is similar to the one provided above, except for the second constraint, which is often written as follows:

$$\forall u, v \in V(G) \quad \text{dist}_G(u,v) = 2 \implies |f(u) - f(v)| \geq k$$

Now, consider the following $L(1,2)$-labeling over a K_3



This is a valid $L(2,1)$-labeling w.r.t. this second definition, because there are no nodes at distance 2 in the graph, therefore the condition is vacuously true. However, this is not a valid $L(2,1)$-labeling w.r.t. the definition provided in Definition 4.2. This problem arises because $h < k$, but if $h \geq k$ the two definitions coincide.

Clearly, since **bandwidth** is expensive, a *good* coloring (i.e. frequency assignment) function f on the nodes is one that minimizes the total bandwidth $\sigma_{h,k}$ utilized in order to determine a valid $L(h,k)$-labeling. The minimum possible bandwidth will be noted with $\lambda_{h,k}$.

Observation 4.1: Notation

Usually the minimum color used for the labeling is 0, thus an $L(h,k)$-labeling having *span* (i.e. *bandwidth*) $\sigma_{h,k}(G)$ uses $\sigma_{h,k}(G)+1$ different colors, which may be counter-intuitive but this notation is used for historical reasons.

The problem of finding a **minimum bandwidth labeling** has been introduced in 1991 by Griggs et al. [30], in the context of a frequency assignment problem, for the case of $h=2$ and $k=1$.

However, the problem was already known in combinatorics for the case $h=k=1$, which was introduced by Wegner [57] in 1977. In particular, he introduced the concept in the context of coloring the **square** of a graph, which is defined as follows.

Definition 4.3: Graph square

Given a graph $G=(V,E)$, its **square** $G^2=(V',E')$ is a graph defined as follows:

- $V'(G^2) := V(G)$
- $(u,v) \in E'(G^2)$ if either $(u,v) \in E(G)$, or u and v are connected in G through a path of length 2

Example 4.2 (Graph square)**.** Given the following graph G, on the left, its square G^2 is the graph on its right.

Figure 4.2: A graph (on the left), and its square (on the right).

If $h=k=1$, the problem of finding an $L(1,1)$-labeling with minimum bandwidth on a graph G is equivalent to the problem of finding a *vertex coloring* (i.e. a labeling of the nodes such that adjacent nodes are assigned different labels, using the minimum number of colors) on G^2.

After the $L(h,k)$-labeling has been introduced, it has since been used to model multiple problems, for instance:

- the $L(0,1)$-labeling problem is used to model a type of integer *control code* assignment problem in radio networks, to avoid hidden collisions

- $L(0,1)$- and $L(1,1)$-labeling problems are used to model the channel assignment in optical cluster-based networks

Now consider the following two lemmas.

Lemma 4.1

Let $x, y, k \geq 0$ and $d > 0$; if $|x - y| \geq kd$ then $|x' - y'| \geq kd$, where $x' := d\left\lfloor \frac{x}{d} \right\rfloor$ and $y' := d\left\lfloor \frac{y}{d} \right\rfloor$.

Lemma 4.2

For any h, k, d, it holds that $\lambda_{dh,dk} = d \cdot \lambda_{h,k}$.

Proof. Let f be an $L(dh, dk)$-labeling over a given graph G, that realizes $\lambda_{dh,dk}$; thus, we can define f' to be a labeling function such that

$$f' : V(G) \to \mathbb{N} : x \mapsto \frac{f(x)}{d}$$

and it is clearly an $L(h,k)$-labeling of G, by definition. Hence, we have

$$\frac{\lambda_{dh,dk}}{d} = \sigma^{f'}_{h,k} \geq \lambda_{h,k}$$

Note that, if $d \nmid f(x)$, then we can still let $f'(x) := \left\lfloor \frac{x}{d} \right\rfloor$, since

$$|f(u) - f(v)| \geq dh \implies \left| d\left\lfloor \frac{f(u)}{d} \right\rfloor - d\left\lfloor \frac{f(v)}{d} \right\rfloor \right| \geq dh \implies \left| \left\lfloor \frac{f(u)}{d} \right\rfloor - \left\lfloor \frac{f(v)}{d} \right\rfloor \right| \geq h$$

which follows from Lemma 4.1.

Conversely, let f be an $L(h,k)$-labeling over a given graph G, that realizes $\lambda_{h,k}$; thus, we can define f' to be a labeling function such that

$$f' : V(G) \to \mathbb{N} : x \mapsto f(x) \cdot d$$

and it is clearly an $L(dh, dk)$-labeling of G, by definition. Hence, we have

$$d \cdot \lambda_{h,k} = \sigma^{f'}_{dh,dk} \geq \lambda_{dh,dk}$$

□

Because of this lemma, for any $L(a,b)$-labeling such that $a = dh$ and $b = dk$ for some h, k, d, we can restrict the problem into finding an $L(h,k)$-labeling, since $\lambda_{a,b} = d\lambda_{h,k}$. Therefore, we can always restrict the $L(h,k)$-labeling problem into the case in which h and k are **coprime numbers**.

Moreover, the case for $k = 0$, for any value of h, is not usually considered in this context, as it coincides with the classical vertex coloring (previously described). On the counter side, the case when $h = k$ is very studied in the literature, and the cases when $h = 2k$ are definitely the most studied $L(h,k)$-labeling problems.

It has been proven that both the $L(0,1)$-labeling of planar graphs, and the $L(1,1)$-labeling of general, planar, bounded degree and unit-disk graphs are all NP-Complete problems.

Now, let the **diameter** of a graph be the shortest path between each pair of vertices of the graph, and consider the following special form of the $L(2,1)$-labeling problem.

Definition 4.4: Distance 2 Labeling problem

Given a graph $G = (V, E)$, with diameter 2, the **Distance 2 Labeling** (DL) problem asks for the following: is there a $\lambda_{2,1}(G) \leq |V(G)|$?

In 1992, Griggs et al. [30] proved that this decisional problem is NP-Complete. However, before showing the proof of this theorem, it is necessary to introduce some definitions.

Definition 4.5: Hamiltonian path

Given a graph G, a **Hamiltonian path** (HP) on G is a *path* that visits each vertex of G exactly once.

Definition 4.6: Complement of a graph

Given a graph $G = (V, E)$, its **complement** $G^c = (V', E')$ is defined as follows:

- $V' := V(G)$
- $E' := \{(u, v) \in V \times V \mid (u, v) \notin E(G)\}$

In order to prove the theorem, we need to consider the following modified version of the DL problem first.

Definition 4.7: Injective Distance 2 Labeling

Given a graph $G = (V, E)$, with diameter 2, the **Injective Distance 2 Labeling** (IDL) problem, asks for the following: is there an injective function $f : V(G) \to [0, n-1]$ such that $|f(x) - f(y)| \geq 2$ whenever $(x, y) \in E(G)$?

We will prove the following theorem regarding the complexity of the IDL problem, which will be used to prove the complexity of the DL problem by reduction.

Theorem 4.1

IDL is NP-Complete.

Proof. We will prove that the IDL problem is equivalent to the problem of finding a HP on G^c.

First, consider an instance of the IDL problem, and its associated injective function f; since f is injective, its inverse f^{-1} exists. Now, order G's vertices as follows:

$$\forall i \in [0, n-1] \quad v_i = f^{-1}(i)$$

therefore the vertices are numbered after the color they are assigned to through f. Observe that, since $(x,y) \in E(G)$ implies that $|f(x) - f(y)| \geq 2$ by definition of f, we have that v_i and v_{i+1} cannot be adjacent in G. Hence, by definition, $(v_i, v_{i+1}) \in E(G^c)$ for all $i \in [0, n-2]$, implying that $v_0, \ldots, v_{n-1}$ is a Hamiltonian path of G^c.

Conversely, consider a Hamiltonian path $v_0, \ldots, v_{n-1}$ of G^c; thus, we can define the following function

$$f : V(G) \to [0, n-1] : v_i \mapsto i$$

such that

- it is trivially injective, by definition
- given an edge $(x,y) \in E(G)$, there will be i and j such that $x = v_i$ and $y = v_j$, and note that $(v_i, v_j) \notin E(G^c)$; therefore, we have that $f(x) = f(v_i) = i$ and $f(y) = f(v_j) = j$, and clearly $|i - j| \geq 2$ since x and y are not adjacent in G^c

proving that f is indeed an injective function that solves the IDL problem.

Since it is well-known that the HP problem is NP-Complete, and we just proved that HP and IDL are equivalent, we have that IDL is NP-Complete as well. □

We are now ready to prove the complexity of the DL problem.

Theorem 4.2

DL is NP-Complete.

Proof. Trivially it can be verified in polynomial time whether a labeling function f is a feasible $L(2,1)$-labeling for G, and whether $\lambda_{2,1}(G) \leq \max_{v \in V(G)} f(v) \leq n$.

Now, consider the graph $G = (V, E)$ of an instance of the IDL problem, and construct a graph $G' = (V', E')$ as follows:

- let x be a new vertex
- $V' := V \cup \{x\}$
- $E' := E \cup \{(x, v) \mid v \in V(G)\}$

therefore x is adjacent to all other nodes, implying that $|V'| = |V| + 1 = n + 1$ and G' has still diameter 2.

Now, assume that there exists an injection $f : V(G) \to [0, n-1]$ such that $(x,y) \in E(G) \implies |f(x) - f(y)| \geq 2$, and let $g : V'(G') \to [0, n+1]$ be a new function such that

$$g(u) := \begin{cases} f(v) & v \in V(G) \\ n+1 & u = x \end{cases}$$

Note that:

- $g(x)$ is defined to be $n+1$, and $f(v)$ is at most $n-1$, thus from the condition f satisfies by definition if follows that $(x,y) \in E'(G') \implies |g(x) - g(y)| \geq 2$
- since f is injective, and by the previous condition, it holds that

  $$|g(u) - g(v)| \neq 0 \implies |g(u) - g(v)| \geq 1$$

  for any pair of vertices of G' that are at distance 2

which implies that g is an $L(2,1)$-labeling for G', and clearly

$$\lambda_{2,1}(G') \leq \max_{v' \in V'(G')} g(v') \leq |V'|$$

Conversely, suppose that $\lambda_{2,1}(G') \leq |V'|$, i.e. there exists a feasible $L(2,1)$-labeling g such that $\max_{v' \in V'(G')} g(v') \leq |V'|$. Note that G' has diameter 2 because of x, therefore it must be that $u \neq v \implies g(u) \neq g(v)$ for each $u, v \in V'(G')$, otherwise g would not be a feasible $L(2,1)$-labeling; this implies that g must be injective.

Now, assume that $g(x) \neq |V| + 1 \wedge g(x) \neq 0$; since g is a $L(2,1)$-labeling of G', and x is connected to each vertex $v \in V(G)$, we have that

- no vertex $v \in V(G)$ has color $g(x)$, or $g(x) \pm 1$
- hence, the *lower* — w.r.t. $g(x)$ — range of possible colors for the other vertices is $[0, g(x) - 2]$, i.e.

  $$g(x) - 2 - 0 + 1 = g(x) - 1$$

  colors, and the *upper* range will be $[g(x) + 2, R]$, i.e.

  $$R - (g(x) + 2) + 1 = R - g(x) - 1$$

  colors, for some R
- to determine R, note that the number of vertices that have to be colored is n, therefore

  $$g(x) - 1 + R - g(x) - 1 = n \iff R = n + 2$$

- this implies that the range of possible colors of g is $[0, R+2]$, contradicting the hypothesis for which $\lambda_{2,1}(G') \leq n + 1$

Therefore, $g(x)$ is either 0 or $|V| + 1$, hence

- if $g(x) = |V|+1$, then the vertices in $V' - \{x\} = V$ must range between 0 and $n-1$, since g is a valid $L(2,1)$-labeling of G', and x is connected to all the other vertices of G; thus, let f be g restricted on V, hence

$$\forall v \in V(G) \quad f(v) := g(v)$$

- if $g(x) = 0$, then the vertices in $V' - \{x\} = V$ must range between 2 and $n+1$, since g is a valid $L(2,1)$-labeling of G', and x is connected to all the other vertices of G; thus, let f be defined as follows

$$\forall v \in V(G) \quad f(v) := g(v) - 2$$

and, in both cases, we have that f is both injective — since g was injective — and is a function defined on $f : V(G) \to [0, n-1]$ such that whenever $(x,y) \in E(G)$ we have that $|f(x) - f(y)| \geq 2$.

This proves that IDL is reducible to DL, therefore DL is NP-Complete. □

4.1.1 Known results

Research in this field has progressed in several directions, including:

- **bounds on** $\lambda_{h,k}$: investigating both *lower and upper bounds* for $\lambda_{h,k}$, providing theoretical insights into its limits and behavior across different scenarios
  - for instance, for what concerns *lower bounds*, it is known that $\lambda_{2,1} \geq \Delta + 1$ and that $h \geq k \implies \lambda_{h,k} \geq (\Delta - 1)k + h$
  - regarding *upper bounds*, it is known that $\lambda_{2,1} \leq \Delta^2 + 2\Delta$, proved by Griggs et al. [30], and it is conjectured that $\lambda_{2,1} \leq \Delta^2$
- limiting the coloring problem on **specialized graph classes**:
  - **Exact Labelings**: studies that focus on deriving precise labelings for specific graph structures
  - **Approximate Labelings**: approaches that aim for near-optimal solutions, particularly when exact computations are infeasible or overly complex

As for the previous section, we will focus on the $L(2,1)$-labeling problem specifically, which has been extensively studied in the literature.

To determine *upper bounds* for $\lambda_{2,1}$, we can consider the following **greedy approach** to label the nodes: given a graph $G = (V, E)$ with nodes $v_1, \ldots, v_n$, label its nodes in order, assigning to v_i the smallest color that *does not conflict* with the labels of its neighbourhood. Although this algorithm may lead to suboptimal solutions, it can be used to define upper bounds for $\lambda_{2,1}$ since it clearly yields a valid $L(2,1)$-labeling.

We will now cover *exact results* regarding $\lambda_{2,1}$. Consider a **clique graph** K_n; since all the nodes of a clique graph are pairwise adjacent, it must be that

$$\lambda_{2,1}(K_n) = 2(n-1)$$

Differently, consider a **star graph** $K_{1,t}$; we can prove that

$$\lambda_{2,1}(K_{1,t}) = t + 1$$

easily:

- clearly $\lambda_{2,1}(K_{1,t}) \leq t+1$ by using the *greedy algorithm* discussed previously, which would yield for example a labeling that assigns to the star's center the color $t+1$, and to any other vertex a color between 0 and $t-1$
- conversely, to prove that $\lambda_{2,1}(K_{1,t}) \geq t+1$, assume by way of contradiction that $\lambda_{2,1}(K_{1,t}) < t+1$; then, it must be that $\lambda_{2,1} \leq t$, but if the center of the star graph is labeled with any color between 0 and t, there would not be enough colors since there are t external nodes on the star graph

Additionally, for other graph topologies we have interesting results. For instance, the proof of the following result was first proposed by Griggs et al. [30].

Theorem 4.3: $\lambda_{2,1}$ on trees

Given a tree T_n, $\lambda_{2,1}(T_n)$ is either $\Delta+1$ or $\Delta+2$.

Proof. Trivially, we have that $\lambda_{2,1}(T_n) \geq \Delta+1$, since any tree T_n contains a $K_{1,\Delta}$ star graph.

To prove that $\lambda_{2,1}(T_n) \leq \Delta+2$, we need to consider a different *greedy labeling* algorithm, called **First-Fit labeling** which works as follows:

- order the nodes of T_n such that v_i is attached just once to the subgraph of T_n that contains $\{v_1, \ldots, v_{i-1}\}$; note that this ordering induces subgraphs $T_i := T_{i+1} - \{v_{i+1}\}$ where v_{i+1} is a leaf, for any $i \in [1, n-1]$
- label v_1 with 0
- label v_i with the first available color

Inductively, assume that we have already labeled all the nodes from v_1 to v_i, and consider v_{i+1} which has to be labeled. Let v_j be v_{i+1}'s parent w.r.t. T_n, and clearly $j \leq i+1$; moreover, note that, by definition of Δ, v_j has at most $\Delta-1$ adjacent nodes, not counting v_{i+1}. Therefore, since this is a $L(2,1)$-labeling, we have that in order to label v_{i+1}

- no more than 3 colors are forbidden because of v_j's color — namely, if v_j has color $f(v_j)$ for some coloring function f, then v_{i+1} cannot have color $f(v_j)-1$, $f(v_j)$ and $f(v_j)+1$
- at most $\Delta-1$ colors are forbidden due to v_j's other adjacent nodes

implying that, if we have at least $(\Delta-1)+3+1 = \Delta+3$ at our disposal — namely, from 0 to $\Delta+2$ — we are always able to label v_{i+1}. This proves that $\lambda_{2,1}(T_n) \leq \Delta+2$. □

In the original paper, it was conjectured that decide whether the correct value for a given T_n tree is $\Delta+1$ or $\Delta+2$ is NP-Complete, but this conjecture was later disproved

by Chang et al. [11] in 1996, which provided a polynomial $O(\Delta^{4.5}n)$ algorithm based on a *dynamic programming* approach. Multiple authors have proposed many various algorithms attempting to improve this time complexity, and finally in 2008 Hasunuma et al. [32] proposed a linear algorithm.

For what concerns **path graphs** P_n, it is immediate to see that $\lambda_{2,1}(P_2) = 2$ and $\lambda_{2,1}(P_3) = 3$ from the results for the star graphs. Moreover, by exhaustion it can be easily shown that $\lambda_{2,1}(P_4) = 3$, and for P_5 note that

- clearly $\lambda_{2,1}(P_5) \leq 4$, since we have 5 colors, from 0 to 4, and we just need to color each node with a different label
- since P_5 includes two P_4's, we have that $\lambda_{2,1}(P_5) \geq 3$
- this implies that $\lambda_{2,1}(P_5)$ is either 3 or 4, but it can be proved by exhaustion that the correct number is indeed 4

Finally, for any $n > 5$, we have that

- P_n contains a P_5, therefore $\lambda_{2,1}(P_n) \geq 4$
- in P_n we have that $\Delta = 2$, and for the Theorem 4.3 we know that

$$3 = \Delta + 1 \leq \lambda_{2,1}(P_n) \leq \Delta + 2 = 4$$

therefore, it must be that $\lambda_{2,1}(P_n) = 4$.

Lastly, if the graph considered is a **cycle graph**, we have the following result.

Theorem 4.4: $\lambda_{2,1}$ on cycle graphs

Given a cycle graph C_n, we have that $\lambda_{2,1}(C_n) = 4$.

Proof. For any $n \leq 4$, it is trivial to check the statement, as shown in the following picture

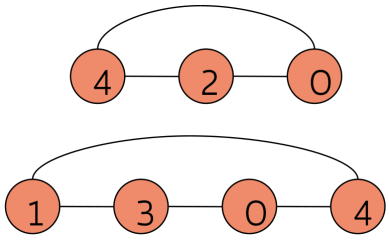


Figure 4.3: Cases for C_3 and C_4 (top to bottom).

For the case $n \geq 5$, note that C_n contains a P_n if we ignore one of the edges, hence $\lambda_{2,1}(C_n) \geq 4$ because of the previous result. To prove that $\lambda_{2,1}(C_n) \leq 4$, we just need to consider 3 possible cases:

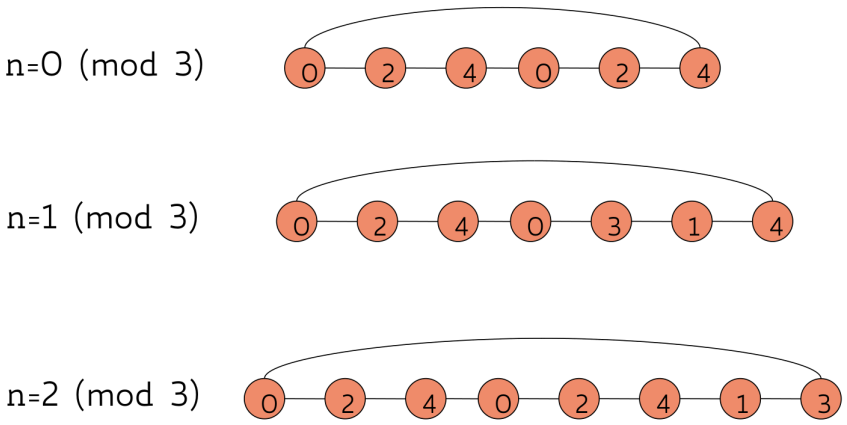


Figure 4.4: Cases for C_n when $n \geq 5$.

As it is shown in the figure:

- if $n \equiv 0 \pmod 3$, we just need to cycle 0, 2 and 4 repeatedly
- otherwise, if $n \equiv 1 \pmod 3$, we cycle 0, 2 and 4 for the first $n-4$ vertices of the cycle, and the last vertices will be colored with 0, 3, 1 and 4, respectively
- finally, if $n \equiv 2 \pmod 3$, we cycle 0, 2 and 4 for the first $n-2$ vertices of the cycle, and the last vertices will be colored with 1 and 3, respectively

□

As a final note, if the graph G considered is a **grid graph**, it is known that $\lambda_{2,1}(G) = \Delta+2$, and if it is an **outerplanar graph**, it has been proven that $\lambda_{2,1}(G) \leq 2\Delta + 2$ by Jonas [35] in 1993.

4.1.2 Variations of the problem

In the literature there are multiple variations of the $L(h,k)$-labeling problem. For instance, consider the following variation.

Definition 4.8: Oriented $L(h,k)$-labeling

Given a *directed* graph $G = (V, E)$, an **oriented $L(h,k)$-labeling** of G is a *node coloring function* f that assigns colors to each node of G, such that:

- $\forall (u,v) \in E(G) \quad |f(v) - f(u)| \geq h$
- $\forall u, v \in V(G) \quad \exists w \in V(G) \mid (u,w),(w,v) \in E(G) \implies |f(u) - f(v)| \geq k$

As in the undirected case, the objective of the problem is to minimize the *span* needed to define an oriented $L(h,k)$-labeling.

Note that $\lambda_{h,k}$ for the *directed* (i.e. oriented) case can vary greatly from the *undirected* case. For instance, recall that $\Delta + 1 \leq \lambda_{2,1}(T_n) \leq \Delta + 2$ for any *undirected* tree T_n, but

it has been shown by Chang et al. [12] that $\lambda_{2,1}(T_n) \leq 4$ for any *directed* tree.

Another type of variation to the $L(h,k)$-labeling problem is the following generalization.

Definition 4.9: $L(h_1,\ldots,h_k)$-labeling

Given a graph $G=(V,E)$, an **oriented $L(h_1,\ldots,h_k)$-labeling** of G is a *node coloring function* f that assigns colors to each node of G, such that

$$\forall u,v \in V(G) \mid \text{dist}_G(u,v) = i \in [1,k] \quad |f(u)-f(v)| \geq h_i$$

Clearly, in this context two vertices u and v are at distance $\text{dist}_G(u,v)=i$ in G if there is a path from u to v of length i.

The objective of this problem is analogous to the previous versions discussed, i.e. minimizing the *span*. On general graphs, it is known that $L(2,1,1)$- and $L(\delta,1,\ldots,1)$-labelings are both NP-Hard, therefore special classes of graphs are usually studied for this type of problems.

Differently, if we are interested in utilizing the coloring problem to solve a frequency assignment problem for a given network, and our network topology has a **backbone** structure such that the transmitting power of its nodes is higher that the rest of the network, we may exploit the following coloring problem, that has the same objective as the cases studied before.

Definition 4.10: Backbone coloring

Given a graph G, that has a *backbone* $H \subseteq G$, a **Backbone coloring** of G w.r.t H is a *node coloring function* f that assigns colors to each node of G, such that:

- $\forall (u,v) \in E(H) \quad |f(u)-f(v)| \geq h$
- $\forall (u,v) \in E(G)-E(H) \quad |f(u)-f(v)| \geq k$

As a final note, in real-world scenarios the transmitting stations are able to handle **multiple channels**, therefore usually a *set of channels* is assigned to each station of a given network. To exploit this capability of the stations, given two sets of integer values I and J, let

$$\text{dist}(I,J) := \min_{i\in I, j\in J} |i-j|$$

and consider the following extension of the $L(h,k)$-labeling problem that aims at minimizing the *span* as well, given an integer n.

Definition 4.11: n-multiple $L(h,k)$-labeling

Given a graph G, an **n-multiple $L(h,k)$-labeling** of G is a *node coloring function* f that assigns n colors to each node of G, such that:

- $\forall (u,v) \in E(G) \quad \text{dist}(f(u), f(v)) \geq h$
- $\forall (u,v) \in E(G) \mid \text{dist}_G(u,v) = 2 \quad \text{dist}(f(u), f(v)) \geq k$

4.1.3 Map coloring

A natural way to model a **map** of adjacent regions is with a **planar graph**, where each region is represented with a node, and there is an edge between each pair of nodes of the graph that represent adjacent regions of the map. Now, consider the following graph definition.

Definition 4.12: Dual graph

Given a planar graph $G = (V, E)$, its **dual graph** $G^* = (V', E')$ is defined as follows:

- the nodes of G^* are placed in the **faces** of G
- there is an edge between a pair of nodes u, v of G^* if and only if the faces of u and v are adjacent in G (i.e. they share an edge of G)

Example 4.3 (Dual graphs)**.** The following two graphs are one the dual of the other.

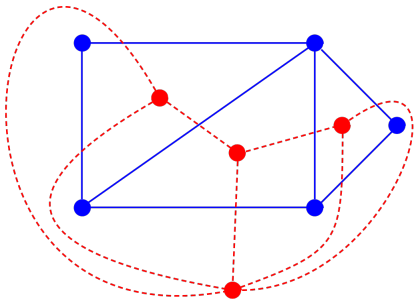


Figure 4.5: The red graph is the dual of the blue graph, and *vice versa*.

It can be shown that a **vertex coloring** of G^* corresponds to a **map coloring** of G.

Map coloring problems are valuable models for real-world scenarios. While cartographers have long known that four colors suffice to color any map such that no two adjacent regions share the same color, this was formally proven only in 1976 by Appel et al. [1]. The proof, which relied heavily on computer assistance, reduced the problem to more than 1700 configurations, each exhaustively verified by a machine.

Regarding classical **map coloring** problems, the following are some interesting results for small numbers, extensively studied in the literature:

- **2-coloring**: this problem is in P, and a polynomial time algorithm that solves it is the following
  - choose a starting region and assign one of the two available colors to it
  - assign the other color to its neighbouring regions
  - expand the colored region by alternating the 2 colors, until either all the regions have been colored, or there is a region that cannot be colored without conflicting with its neighbours, which implies that the map is 2-colorable from the beginning
- **3-coloring**: this problem is NP-Hard, however some techniques are available which can simplify the map before searching for a 3-coloring function, but such strategies do not change the time complexity of the problem in the worst case
- **4-coloring**: the 4-coloring problem is NP-Complete, as for the 3-coloring case
- **5-coloring**: it easy relatively easy to color a map using 5 colors, and there are algorithms that can simplify the input map

5

The broadcast's energy consumption problem

Wireless sensor networks (WSNs) are large-scale, multi-hop wireless systems composed of nodes with *limited resources*, such as energy, bandwidth, storage, and processing power. Designed primarily for continuous monitoring and data collection, WSNs play a crucial role in various applications by providing low-level surveillance and data gathering within a designated area.

Once deployed, Wireless Sensor Networks (WSNs) are designed to operate autonomously over extended periods without direct human intervention. Over their lifetimes, it is crucial to address software bugs, reconfigure system parameters, and perform software upgrades to maintain reliable system performance. For large-scale WSNs, manually retrieving and reconfiguring individual nodes is *impractical and inefficient*. As a result, **data dissemination**, i.e. **broadcasting**, is essential for efficiently disseminating updates and reconfigurations across the network.

Broadcasting enables the dissemination of data from a sink node to all nodes in the network using wireless communication. The transmitted data may include code for updating programs, system commands for configuration or control and updated system parameters to optimize performance.

There are three requirements for data dissemination in WSNs.

- **Reliability**: To ensure reliability, all nodes in the network must be *covered* during data dissemination. Since this process forms the foundation for critical services such as reprogramming and parameter distribution, failing to reach even a single node can lead to inconsistencies or a complete network crash.
- **Energy efficiency**: The data dissemination process must be conducted with *minimal energy consumption* due to the limited power resources of network nodes. Energy consumption primarily involves two components:
  - **read-write operations**, necessary for storing data blocks and thus unavoid-

able

  - **transmission activity**, the largest contributor to energy consumption and the aspect most amenable to optimization

- **Scalability**: In Wireless Sensor Networks, both the number of nodes and node density can vary significantly. A dissemination protocol is considered scalable if the completion time of data dissemination increases linearly with the size of the network.

A wireless ad-hoc network is composed of a set S of fixed radio stations connected via wireless links

- each node is equipped with **omnidirectional antennas**, enabling transmissions to be received by all neighboring nodes, resulting in a natural broadcast effect
- stations can communicate either directly (**single-hop communication**) if they are *sufficiently close*, or via **intermediate nodes**

Each station can dynamically *modulate* its own transmission power, in fact the **transmission radius** of a station depends on the energy power supplied to the station. In particular, the power P_s required by a station s to transmit data to another station t must satisfy the following inequality

$$P_s \geq \text{dist}^\alpha(s,t)$$

where $\alpha \geq 1$ is called **distance-power gradient**. Usually $2 \leq \alpha \leq 4$, and $\alpha = 2$ in empty space. This implies that, in order to have a communication from s to t, the power P_s must be *proportional* to $\text{dist}^\alpha(s,t)$.

The primary goal is to **minimize energy consumption**, as all devices rely on a common electricity source (e.g., when the stations are deployed in an ad-hoc fashion and connected to a centralized power supply). To achieve this goal, the stations collaborate to provide specific **connectivity properties**, by *dynamically adjusting* their transmission ranges.

Depending on the requirements of the network, the **transmission graph** may be required to

- be **strongly connected**: this problem is NP-Hard, and although there exists a 2-approximation algorithm in two-dimensional settings, found by Kirousis et al. [37], there exists values of $r > 1$ such that the problem is not r-approximable
- have a **bounded diameter**: this is non-trivial, and no approximation algorithms are currently known
- include a **spanning tree** rooted in a given source node s

The last requirement is the focus of this chapter, and it will be discussed in greater detail. In particular, consider a graph $G = (V, E)$, in which the vertices of the graph are the stations that need to communicate between one another, and E is the set of connections between the various stations. Moreover, consider a function $w : E(G) \to \mathbb{R}^+$ which assigns

weights to the edges of G such that $w(u, v)$ describes the cost of connecting u and v, due to either distance or other factors.

Since the main goal is to *minimize the energy consumption* of the various hops of the network, we need to decide the **power consumption** of each node on the graph G that we considered. Assume that there is a special *source node* s that is the starting point for any transmission that will be considered. In particular, we are interested in sending a **broadcast message** from s to any other node of G, while still minimizing the energy consumption.

Let a **power assignment** be a function $r : V(G) \to \mathbb{R}^+$, such that it induces a subgraph $G' = (V, E')$ of the graph G, where the edges are defined as follows

$$(i, j) \in E' \iff w(i, j) \leq r(i)$$

If G' is a *spanning tree* of G rooted in s, i.e. s can correctly broadcast a messsage to any other node of the network, we will call the power assignment r a **Broadcast Range Assignment**, or *broadcast* for short.

Definition 5.1: Min Broadcast problem

The **Min Broadcast** problem asks to find the power assignment r for a given network G that minimizes the energy consumption, i.e. that minimizes $\sum_{v \in V(G)} r(v)$.

Note that this problem *resambles* the problem of finding the Minimum Spanning Tree (MST) — which asks for the spanning tree of minimum weight of a given edge-weighted graph — but it is *not the same problem*. In fact, a power assignment is an assignment of the vertices which does "choose" edges, but if a vertex u has a power assignment $r(u)$, than *all the edges* (u, v) such that $w(u, v) \leq r(u)$ are automatically chosen, making the problem much harder to solve.

5.1 The Min Broadcast problem

Consider the following computational problem.

Definition 5.2: Set cover (SC)

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 [36] proved that the **set cover** problem is NP-Complete. Moreover, for the set cover problem, the following theoretical result is known.

Theorem 5.1: Inapproximability of Set Cover

The Set Cover problem is not approximable within a factor of $c \log n$, for some $c > 0$, where $n = |U|$ and U is the universe set of the set cover problem.

Given this theoretical result regarding the inapproximability of the set cover problem, we will prove that the Min Broadcast problem is not approximable by using the set cover problem.

Theorem 5.2: Inapproximability of Min Broadcast

The Min Broadcast problem is not approximable within any constant factor.

Proof. We will perform a reduction from an instance of the set cover problem to an instance of the Min Broadcast problem. Consider an instance of the set cover problem (U, C), where $U = \{s_1, \ldots, s_n\}$ is the unisverse set and $C = \{C_1, \ldots, C_m\}$ is the given collection of sets, and construct the following instance (G, w, s) of the Min Broadcast problem:

- let v_{s_i} be a vertex for each $s_i \in U$, and let v_{C_j} be a vertex for each $C_j \in C$; moreover, let s be a new node
- the nodes of the graph G are

$$V(G) := \{s\} \cup \{v_{s_i} \mid s_i \in U\} \cup \{v_{C_j} \mid C_j \in C\}$$

- the edges of the *directed* graph G are

$$E(G) := \{(s, v_{C_j}) \mid C_j \in C\} \cup \{(v_{C_j}, v_{s_i}) \mid C_j \in C, s_i \in C_j\}$$

  therefore, there is an edge between s and each vertex representing a set C_j, and an edge between a set C_j and its elements $s_i \in C_j$
- for any $e \in E(G)$, let $w(e) := 1$

Let C' be a solution for the instance of the set cover problem we considered; note that, by construction of the edges of G, any solution of the Min Broadcast instance we constructed assigns 1 to s and to all the nodes of the form v_{C_j} for the ones and only $C_j \in C'$, since the solution to the Min Broadcast must *minimize* the energy consumption. Moreover, the transmission graph induced by this power assignment must be a spanning tree rooted in s, since each element of U is contained in at least one set of C'. The cost of such a solution is $|C'| + 1$, for the source node s.

Conversely, assume that r is a feasible solution for the Min Broadcast problem, i.e. a power assignment that minimizes the energy consumption — note that, $r(s) = 1$, and w.l.o.g. $r(v)$ is 1 if v is of the form v_{C_j}, otherwise it is 0 if it is of the form v_{s_i}. Thus, a solution for the set cover problem C' can be derived by simply selecting all subsects $C_j \in C$ such that $r(C_j) = 1$ in the power assignment. The cardinality of such a solution is $|C'| = \text{cost}(r) - 1$, since $\text{cost}(r)$ must contain the cost of the source node s.

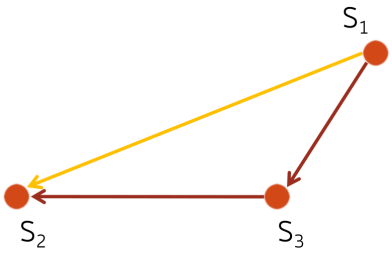
This proves that, given an instance of the set cover problem (U, C), an instance of the Min Broadcast problem (G, w, s) can be constructed such that there exists a solution for (U, C) of cardinality k if and only if there exists a solution for (G, w, s) of cost $k+1$. Therefore, if Min Broadcast is approximable within a constant factor, then SC is approximable within a constant factor as well, which is not possible by Theorem 5.1. □

As a final note, it can be proven that the Min Broadcast problem is NP-Complete [8], and in its general version — not discussed in this notes — it is not approximable within $(1 - \varepsilon)\Delta$, where Δ is the maximum degree of the spanning tree of the solution to the problem, and ε is an arbitrary constant.

5.1.1 Euclidean Min Broadcast

There exist special cases of the Min Broadcast problem, which are particularly interesting. An example is provided by the **Euclidean bidimensional Min Broadcast**, in which the problem is restricted to the Euclidean plane, and the weight between the edges of the networks considered will be the Euclidean distance between the hops, raised to some constant α.

In this special case, it is crucial to *collaborate* in order to minimize the overall energy consumption. For example, consider the following setting



S_1 needs to communicate with S_2, but it can only communicate directly to S_3, and S_3 can in turn communicate to S_2. Assume that $\alpha = 2$, hence the total cost of transmitting a message from S_1 to S_2 passing through S_3 is

$$\text{dist}(S_1, S_3)^2 + \text{dist}(S_3, S_2)^2$$

Note that if the angle $S_1S_3S_2$ is obtuse, then

$$\text{dist}(S_1, S_2)^2 > \text{dist}(S_1, S_3)^2 + \text{dist}(S_3, S_2)^2$$

for the triangle inequality.

In the Euclidean case, a power assignment r of a given network can be represented by the corresponding *family of disks* $D = \{D_1, \ldots, D_n\}$, and the overall energy consumption will be defined as

$$\text{cost}(D) := \sum_{i=1}^{n} r_i^{\alpha}$$

where r_i is the radius of D_i.

Differently from the Min Broadcast problem, nothing is known about the hardness of the Euclidean version of the problem, but there exists an approximation algorithm that is based on the computation of an MST of the given graph, which works as follows:

- compute the MST of the complete graph induced by $G^{(\alpha)}$, which is the complete and weighted graph where the weight of each edge (u, v) is precisely $\text{dist}(u, v)^\alpha$
- assign a direction to the edges, from s toward the leaves
- the power assignment is defined as follows: assign to each node i the radius equal to the length of the longest directed edge outgoing from i

Although it is easy to implement, the approximation ratio of this algorithm is very involved and outside the scope of these nodes. Nevertheless, the next section will focus on algorithms and theoretical results regarding the problem of finding an MST of a given graph.

5.2 The minimum spanning tree problem

The **Minimum Spanning Tree** (MST) problem is defined as follows.

Definition 5.3: Minimum Spanning Tree

Given a graph $G = (V, E)$, and a function $w : E(G) \to \mathbb{R}$, find the subtree of G that minimizes the total weight of its edges w.r.t. w.

In the general case there may be several minimum spanning trees of the given graph, but the following lemma can be proved for a special case of w.

Lemma 5.1: Uniqueness of the MST

Given a graph $G = (V, E)$, and a function $w : E(G) \to \mathbb{R}$, if w is such that all the edges have distinct weights, then G's MST is unique.

Proof. By way of contradiction, assume that there exist two distinct MSTs T and T' for G, such that

$$\sum_{e \in E(T)} w(e) = \sum_{e \in E(T')} w(e)$$

Without loss of generality, since $T \neq T'$ assume there exists $e_1 \in E(T) - E(T')$. Since T' is an MST, the graph induced by $E(T') \cup \{e_1\}$ must contain a cycle C, and there must be at least one edge $e_2 \in E(T') - E(T)$ which lies in C — if such an e_2 does not exist, then e_1 could not have existed in the first place.

Now, if $w(e_1) < w(e_2)$, replacing e_2 with e_1 in T' yields a spanning tree of G of less weight than the weight of T', which is a contradiction because we assumed that T' was an MST. The same reasoning can be applied in the case in which $w(e_2) < w(e_1)$, and the case in which $w(e_1) = w(e_2)$ does not hold by hypothesis on w. □

Lemma 5.2

Consider a graph $G = (V, E)$, a weight function $w : E(G) \to \mathbb{R}$, and a cycle C of G; if in C there exists an edge e such that its weight is larger than the weight of all the other edges of C, e does not belong to any MST of G.

Proof. By way of contradiction, assume that there exists an MST T of G such that $e \in E(T)$; hence, by definition of MST, the graph induced by $E(T) - \{e\}$ is described by two subtrees T_1 and T_2 of G, and the two endpoints of e belong two T_1 and T_2 respectively.

Note that the rest of the cycle C also connects the two subtrees, i.e. there exists at least another edge f of C such that the two subtrees T_1 and T_2 are connected, therefore we can define a new tree T' induced by $E(T_1) \cup E(T_2) \cup \{f\}$ which still spans G entirely. Finally, note that $w(e) > w(f)$ by hypothesis, therefore the total weight of T' is less than the weight of T, contradicting the assumption for which T was an MST. □

Lemma 5.3

Consider a graph $G = (V, E)$, and a weight function $w : E(G) \to \mathbb{R}$; if in G there exists a unique edge e of minimum weight, e is in any MST of G.

Proof. By way of contradiction, assume that there exists an MST T of G such that $e \notin E(T)$, and consider the graph induced by $E(T) \cup \{e\}$. Clearly, such a graph contains a cycle C that includes e, and by definition of T, removing any edge of C that is different from e itself, yields an MST of weight lower than the one of T, since

- e has minimum weight among all the edges of the graph
- the edge that was replaced with e must have *strictly greater* weight than e's weight, since e has unique minimum weight in G

and this contradicts the assumption of T being an MST of G. □

Lemma 5.4

Consider a graph $G = (V, E)$, a weight function $w : E(G) \to \mathbb{R}$, and a set of edges defined by a cut C on G; if the edge e with minimum weight in C is unique in C, then e is included in any MST of G.

Proof. By way of contradiction, assume that there exists an MST T of G such that $e \notin E(T)$; then, by definition, adding e to $E(T)$ would create a cycle that passes through the cut C of G at least twice. Therefore, if we remove any other edge that is both on the cut and in the cycle we just formed, we get a spanning tree of G with less weight then T's, by hypothesis on the weight of e. □

Corollary 5.1

Consider a graph $G = (V, E)$, a weight function $w : E(G) \to \mathbb{R}$, and a set of edges defined by a cut C on G; then, all the edges that have minimum weight in C are included in any MST of G.

There are three classical algorithms that are well-known in the literature which are able to compute the MST of a given graph, namely Kruskal's algorithm, Prim's algorithm and Borůvka's algorithm; all of the three algorithms employ **greedy** approaches, and they are presented down below.

Algorithm 5.1: Kruskal's algorithm

Given a graph G, and a weight function $w : E(G) \to \mathbb{R}^+$, the algorithm returns an MST of G.

1: **function** KRUSKAL(G, w)
2: $T := \varnothing$
3: **while** $V(T) \neq V(G)$ **do**
4: Let e be the edge that connects two different connected components of T that has the least weight w.r.t. w
5: $T = T \cup \{e\}$
6: **end while**
7: **return** T
8: **end function**

Idea. The algorithm repeatedly chosses the edge e that connects two different connected components of T which has the least weight w.r.t. w, until T spans G entirely.

Cost analysis. The algorithm is usually implemented using the **Union-Find** data structure, through which it is possible to achieve a cost of $O(m \log n)$ time (assuming that $\frac{n}{2} \leq m \leq n^2$, and $\log n$ and $\log m$ are within a constant factor to each other).

Algorithm 5.2: Prim's algorithm

Given a graph G, and a weight function $w : E(G) \to \mathbb{R}^+$, the algorithm returns an MST of G.

```
1:  function PRIM(G, w)
2:      v ∈ V(G)
3:      Sol := ∅
4:      R := {v}
5:      MinHeap H := []
6:      for u ∈ V(G) − {v} do
7:          H.insert(u, +∞)
8:      end for
9:      parents := [−1] ∗ n
10:     parents[v] = v
11:     for y ∈ V(G) : y ∼ v do
12:         parents[y] = v
13:         H.set_key(y, w(v, y))
14:     end for
15:     while R ≠ V(G) do
16:         y := H.extract_min()                    ▷ y is removed from H
17:         Sol = Sol ∪ {(parents[y], y)}
18:         R = R ∪ {y}
19:         for x ∈ V(G) − R : x ∼ y do
20:             if H.get_key(x) > w(x, y) then
21:                 parents[x] = y
22:                 H.set_key(x, w(x, y))
23:             end if
24:         end for
25:     end while
26:     return Sol
27: end function
```

Idea. The idea of the algorithm is very similar to the idea of Dijkstra's algorithm — proposed in Algorithm 1.2 — with the only difference being the *optimization function* of the algorithm:

- Dijkstra's algorithm aims at minimizing the distance between the starting node
- Prim's algorithm aims at minimizing the weight of the edge on the cut defined by the current set of visited vertices R — note that this algorithm relies on Corollary 5.1

Cost analysis. This implementation through a **MinHeap** has cost of $O(m \log n)$, but it is possible to achieve a cost of $O(m + n \log n)$ through a **Fibonacci Heap**.

Algorithm 5.3: Borůvka's algorithm

Given a graph G, and a weight function $w : E(G) \to \mathbb{R}^+$ that assigns *distinct weights* to the edges of G, the algorithm returns an MST of G.

```
1: function BORUVKA(G, w)
2:     T := ∅
3:     while V(T) ≠ V(G) do
4:         for each connected component C_i of T do
5:             Let e be the edge that connects C_i to another connected component of
   T that has the least weight w.r.t. w
6:             T = T ∪ {e}
7:         end for
8:     end while
9:     return T
10: end function
```

Idea. This algorithm repeatedly links different connected components of the tree it is generating, by always choosing the edge with minimum weight w.r.t w. Note that the hypothesis on w allows to introduce cycles inside the solution.

Cost analysis. As for the other algorithms, it is possible to implement this algorithm in $O(m \log n)$, and in planar graphs it is possible to provide an implementation of $O(m+n)$.

6

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, for instance:

- **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.

The problem of minimizing the time required for a data mule to collect data from all sensor nodes 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 control** subproblem is typically handled by engineers, and the **job scheduling** subproblem falls outside this chapter's scope, our focus will be 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 6.1: 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.

6.1 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 first 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 6.2: Hamiltonian cycle

Given a graph G, a **Hamiltonian cycle** (HC) is a cycle that passes through each node of 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 6.3: 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 6.1: 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 we will prove 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

□

In 1954 Dantzig et al. [18] showed that the TSP can be formulated as an ILP as well:

- given a complete graph $K_n = (V, E)$, assume that the TS tour is oriented
- define variables x_{ij} and w_{ij} for each $(i,j) \in E(K_n)$
- 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:

$$\min \sum_{(i,j)\in E(G)} w_{ij}x_{ij}$$

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

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 i 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. This constraint correctly avoids the possibility of forming cycles in the solution of size less than $V(K_n)$, because if a non-empty set of k vertices has k edges that connect them, then there must be a cycle between them. Additionally, 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 6.2: Inapproximability of the TSP

If there exists a polynomial time algorithm for the TSP with any approximation ratio $r > 1$, then P = 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 6.1, 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 P = NP because HC is NP-Complete. □

6.1.1 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 6.1: 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, by removing an 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

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

□

Consider the following special case of graphs.

Definition 6.4: 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 algorithms 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 6.1: 2-approximation TSP

Given a complete graph $K_n = (V, E')$, and a weight function $w : E'(K_n) \to \mathbb{R}^+$ 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(K_n, w)
2: Choose $r \in V(K_n)$ randomly
3: $T :=$ `findMST`(K_n, w, r) ▷ find an MST rooted in r w.r.t. w
4: `L` $:=$ `DFSpreorder`(T) ▷ a *preorder* DFS on T
5: $V := \varnothing$
6: `L'` $:=$ `[]`
7: **for** $v \in$ `L` **do**
8: **if** $v \notin V$ **then**
9: `L'.append`(v)
10: $V = V \cup \{v\}$
11: **end if**
12: **end for**
13: **return** L' ▷ 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.

Now, 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

$$\dots\ u\ v\ z\ \dots$$

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 6.1, we conclude that

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

□

In 1976 Christofides [15] 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 6.2: Handshaking lemma

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

Corollary 6.1

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, i.e. $\sum_{v \in O} \deg(v)$ is even. Finally, since this sum is even, and it only contains odd values, it must be that the sum is composed of an even number of addends, implying that $|O|$ is even. □

Definition 6.5: 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 6.3: Eulerian circuits

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

Algorithm 6.2: $\frac{3}{2}$-approximation TSP

Given a complete graph $K_n = (V, E')$, and a weight function $w : E'(K_n) \to \mathbb{R}^+$ 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** 3/2APPROXTSP(K_n, w)
2: Choose $r \in V(K_n)$ randomly
3: $T :=$ `findMST`(K_n, w, r) ▷ find an MST rooted in r w.r.t. w
4: $O := \{v \in V(T) \mid \deg_T(v) \text{ odd}\}$ ▷ $\deg_T(v)$ is the degree of v in T
5: $M :=$ `findMinWeightPM`(K_n^O, w) ▷ K_n^O is induced by O
6: $U := M \cup T$ ▷ U is a multi-graph
7: L := `findEulerianCircuit`(U)
8: $V := \varnothing$
9: L′ := []
10: **for** $v \in$ L **do**
11: **if** $v \notin V$ **then**
12: L′.`append`(v)
13: $V = V \cup \{v\}$
14: **end if**
15: **end for**
16: **return** L’ ▷ L’ is 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 K_n^O this set induces with the edges of K_n.

Now, consider the graph K_n^O, induced by O, and a minimum weight perfect matching M of K_n^O.

Note that, because of Corollary 6.1, $|O|$ is even, therefore K_n^O has a perfect matching; thus, let M be the minimum weight perfect matching of K_n^O. Moreover, let N^* be a TS tour on K_n^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 of K_n^O, but since M is the perfect matching of K_n^O with minimum weight, it must be that

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

Moreover, note that N^* is a TS tour on K_n^O, and H^* is a TS tour on G, therefore $w(N^*) \leq w(H^*)$ by the triangle inequality. Hence, we have that

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

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 multi-graph induced by U is connected, because T is a spanning tree
- since M is a minimum weight perfect matching of K_n^O, adding the edges of M in T 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 an Eulerian circuit by Theorem 6.3
- finally, by definition of U we have that

  $$w(U) = w(M \cup T) = w(M) + w(T) - w(M \cap T) + w(M \cap T) = w(M) + w(T)$$

  therefore, given a solution of the algorithm H, 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, and $w(T) \leq w(H^*)$ is a consequence of Lemma 6.1

□

Christofides [15] 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 [2] 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 Euclidean TSP. This means that, for any constant $c > 0$ in a d-dimensional Euclidean 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. [3].

7

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. [46] 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

7.1 The minimum connected dominating set problem

Definition 7.1: 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 7.1 (Dominating sets)**.** The following is an example of dominating set.

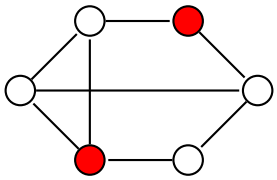


Figure 7.1: A dominating set.

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 7.2: 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 7.2 (Min-CDSs)**.** The following is a min-CDS for the graph illustrated in Example 7.1.

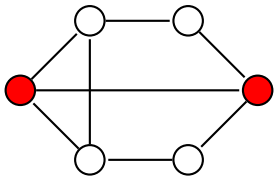


Figure 7.2: A min-CDS for the same graph as before.

Now, consider the following type of spanning trees.

Definition 7.3: 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 7.1: 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

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

□

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. [31] 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 [48] showed that the max-leaf ST problem can be approximated within a factor of 2;
- additionally, Ueno et al. [53] proved that, in graphs where $\Delta = 3$, both problems can be solved in polynomial time.

7.1.1 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, recall the NP-Complete Set Cover problem, introduced in Definition 5.2. 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 7.2: 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 $V = \{1, \dots, n\}$, and construct a set cover instance as follows:

- the universe is $U := V$
- the collection of subsets is $S := \{S_1, \dots, S_n\}$, where

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

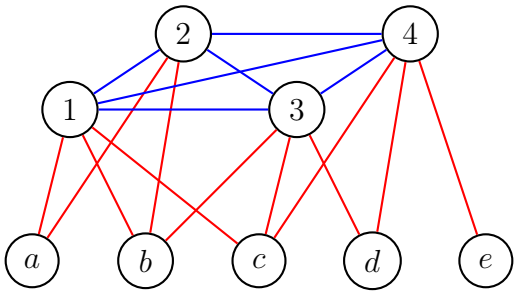
Consider a min-DS D for G; clearly, $C := \{S_v \mid v \in D\}$ is a feasible set cover, since each set $S_v \in C$ will cover v and all its neighbours. Conversely, if $C = \{S_v \mid v \in D\}$ is a set cover, is must be that D is a min-DS for G. Note that $|C| = |D|$.

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 replaceing 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 $|C| = |X| \leq |D|$. □

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.

7.1.2 Greedy approach for the min-CDS

In 1998 Guha et al. [31] 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. [45] 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 $\mathrm{B} := C$
- Gr (gray) nodes, defined as $\mathrm{Gr} := \{v \in V(G) \mid v \notin C \wedge \exists u \in C \mid (u, v) \in E(G)\}$, which is the set of nodes that are not in C but are adjacent to C
- W (white) noeds, defined as $\mathrm{W} := V(G) - \mathrm{B} - \mathrm{Gr}$, which is the set of nodes that are not in C and are not adjacent to C

Clearly, $\mathrm{B} \cup \mathrm{Gr} \cup \mathrm{W} = V(G)$, 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**:

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

Additionally, let $R(v)$ be equal to 1 if and only if $v \in \mathrm{W} \cup \mathrm{Gr}$ 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 7.1: 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.

7.1.3 Unit Disk Graphs

Definition 7.4: 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 [40] proved that, when restricing the min-CDS problem to UDGs, the problem is still NP-Hard; moreover, in 1990 Clark et al. [16] showed that the problem remains NP-Hard even if restricted to *grids*, a special type of UDGs. In 2003, Cheng et al. [14] 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.

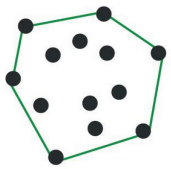
Finally, in 2010 Purohit et al. [44] provided a simple and distributed algorithm for UDGs that effectively reduces any given, even trivial, CDS to a smaller, more efficient structure.

Definition 7.5: 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 7.3 (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 7.2: Distributed reduction of CDS

Given a UDG G, and a CDS of G, the algorithm reduces the CDS.

---

function DISTRIBUTEDREDUCECDS(G, D)
  $V := D$
  do
    Choose $u \in \arg\min_{v \in V} \deg(v)$
    if $\texttt{CH}(\mathcal{N}'(u)) \subseteq \bigcup_{z \in \mathcal{N}(u)} \texttt{CH}(\mathcal{N}'(z))$ **then** ▷ CH computes the *convex hull*
      $D = D - \{u\}$
    end if
    $V = V - \{u\}$
  while $V \neq \varnothing$
  return D
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.

8

The centralized deployment of mobile sensors problem

To be effective, WSNs must ensure full coverage of the **area of interest** (AoI), with no internal sensing gaps, to accurately monitor the environment. Due to factors like human inaccessibility and constrained deployment budgets, careful sensor positioning within the AoI is *essential* to maximize coverage, extend network lifespan, and achieve operational goals. Therefore, **sensor mobility** is mandatory.

There are two main strategies for sensor deployment in wireless sensor networks using **mobility**, which are described below.

- **Carrier-Based Deployment**: In this method, *mobile robots* carry static sensors as payloads and navigate through the AoI. As they travel, these robots strategically place sensors at *designated points*, such as vertices of a geographic grid, to ensure optimal coverage. This approach allows precise, planned deployment in areas where direct human access might be limited.
- **Self-Deployment by Autonomous Sensors**: Here, sensors have *autonomous mobility* and can intelligently adjust their own geographic positions. This self-deployment allows sensors to actively modify their distribution, optimizing their positions to achieve a desired coverage pattern across the AoI. This method offers flexibility and adaptability, enabling dynamic responses to environmental changes and potential coverage gaps.

Mobile sensors are compact, low-cost units, capable of detecting and responding to changes in physical conditions. They are designed with several key components:

- **sensing unit**: detects and monitors environmental changes or specific conditions
- **communication unit**: facilitates data transmission to other devices or a central system
- **computing unit**: processes data and controls sensor operations

- **power supply**: a small battery that powers the device, often designed for energy efficiency
- **mobility system**: allows limited movement, enabling adjustments for optimal sensing or re-positioning

Mobile sensors collaborate to form an **ad-hoc network**, making them especially valuable in critical environments where rapid response and flexibility are essential, such as during pollutant leaks, gas plume detection, or fires. Each sensor operates under the following assumptions:

- **sensing range**: each sensor can monitor a circular area centered at its position, with a radius r_s, known as the *sensing range*, which represents the region in which the sensor can effectively detect changes in environmental conditions
- **communication range**: each sensor can communicate with nearby sensors within a circular area centered at its position, with a radius r_c, called the *communication range*, which defines the area in which sensors can reliably exchange data to coordinate their activities

These capabilities enable mobile sensors to dynamically adjust positions and share information, ensuring comprehensive coverage and effective monitoring across the area of interest.

The sensing and communication units in mobile sensors are distinct components, so the *ensing range* and *communication range* are not inherently linked from a hardware standpoint. However, they must be integrated at the protocol level to ensure both **connectivity** and **coverage** within the network. In particular, research shows that if $r_c \geq 2r_s$, protocols can guarantee coverage while inherently satisfying the connectivity requirement as well. This relationship allows the network to achieve *full monitoring* coverage of the area of interest while ensuring that all sensors remain connected, streamlining protocol design and enhancing network resilience.

Coverage is a critical consideration in environmental monitoring applications using WSNs. Broadly, coverage refers to the effectiveness and quality of the sensing function, measuring how well the sensor network can monitor and observe changes within a given area. High coverage ensures that the AoI is comprehensively monitored without gaps, allowing the WSN to detect and respond to environmental changes accurately.

To collect information from a target field, sensor nodes are strategically deployed at various locations throughout the area. Once deployed, these nodes form a wireless network, allowing them to transmit collected data to a **centralized sink node** for analysis. The quality and completeness of the information gathered are directly tied to the effectiveness of coverage across the AoI. *Adequate coverage* ensures that sensor nodes capture a comprehensive view of the field, minimizing gaps in data and providing a reliable basis for environmental monitoring and other applications.

Coverage requirements for wireless sensor networks vary based on the monitoring goals of the AoI and can be broadly classified into three types:

- **Point Coverage**: This type targets specific, discrete points within the AoI that

require continuous monitoring, often due to their critical importance. For example, in building security, each access point (such as doors and windows) is equipped with sensors to continuously monitor for unauthorized entry.

- **Area Coverage**: This form ensures that every location within a defined region is monitored, achieving complete surveillance of the area. An example is forest monitoring, where each point in the forest must be within the sensing range of at least one sensor. This approach helps detect events like forest fires or poaching activities as soon as they occur.
- **Barrier Coverage**: This type focuses on monitoring a specific path or the boundary of a region, creating a "*barrier*" that detects any movement across it. In forest protection, for instance, deploying sensors along the perimeter enables monitoring for poaching and unauthorized access at the boundary, allowing for immediate response.

For each of these coverage types, **continuous monitoring** using static sensors is essential to ensure reliable data collection and timely response to environmental changes or security breaches.

In certain applications, continuous monitoring of all points within an area is not essential. Instead, periodic *inspections* — referred to as **sweep coverage**, introduced by Cheng et al. [13] — are sufficient to monitor a specific set of points of interest. Unlike traditional coverage approaches that require constant sensor presence, sweep coverage employs sensors that move or are activated periodically to "patrol" these areas.

Definition 8.1: Sweep coverage

A *point of interest* (PoI) p is said to be **sweep covered** if and only if at least one *mobile sensor* visits p every t time periods, where t is called **sweep period** of p.

For example, in agricultural monitoring, regular patrols can be used to check on specific crop health indicators, or in environmental monitoring, periodic sweeps can detect pollutant levels at known hotspots. This approach reduces energy consumption and resource usage, making it well-suited for scenarios where frequent, but not continuous, data collection is adequate.

Definition 8.2: Sweep coverage problem

Given a set of PoIs P, find the minimum number of *mobile sensors* to guarantee *sweep coverage* for P.

In 2014 Gorain et al. [28] showed that, instead of using only mobile sensors, the use of both *static* and *mobile sensors* can be more effective in terms of total number of sensors used. Additionally, in 2015 Gorain et al. [29] showed an energy efficient sweep coverage problem, whose objective is to guarantee sweep coverage by employing both *mobile* and *static sensors*, such that the total energy consumption is minimized.

Moreover, the **location information** of sensors plays a crucial role in deployment protocols, as these protocols often rely on the *precise positioning* of sensors. Depending on the environment and application, different techniques for obtaining and utilizing location information are employed.

- **Outdoor Applications**: For applications in outdoor environments, GPS is the most widely used solution, since it provides accurate global positioning data, enabling sensors to autonomously determine their location for deployment or operation.
- **Indoor Centralized Applications**: For indoor environments where global positioning is needed but GPS may not be available, a *grid-based approach* is commonly used. In this method, the area is divided into a grid of predefined locations, known as *landmarks*, and sensors are deployed at specific grid points. In some cases, the sensors themselves may be used as *landmarks*, enabling them to assist in positioning other sensors within the network.
- **Indoor Distributed Applications**: In scenarios where sensors need to determine their location autonomously without a central system, techniques based on signal measurements are employed.

Definition 8.3: Deployment problem (for area coverage)

Given an AoI to cover, cover the AoI entirely while minimizing the number of used sensors, and maximize the covered area.

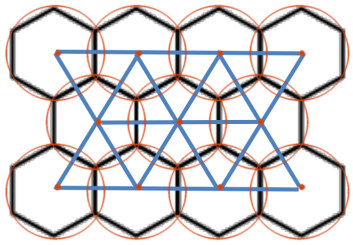
In order to design a **coordination algorithm** for this problem, given an initial configuration starting from either a *random configuration* or from a *safe location*, the goal is to achieve a configuration that ensures full coverage of the AoI, which may be defined through *regular tessellation* — where sensors are arranged in a structured pattern — or any other configuration that covers the AoI effectively, tailored to the specific coverage requirements, which may include:

- **traversed distance**: the distance traveled by sensors is the primary cost factor, as longer distances increase *energy* and *time requirements*
- **start/stop movements**: initiating or stopping sensor movement is *more costly* than continuous motion, so minimizing the number of start/stop actions can significantly reduce costs
- **communication cost**: this cost depends on the frequency of message exchanges and the size of each data packet transmitted
- **computation cost**: typically negligible, unless the sensors use highly sophisticated processors that require significant computational resources

Random deployment is the simplest method for placing sensors. It provides relatively satisfactory coverage, especially in situations where the target area undergoes rapid or unpredictable changes in conditions, or there is limited or no prior knowledge about the environment. It is particularly useful in *military applications*, where WSNs are often

established by dropping or scattering sensors over the area. However, random deployment may result in *uneven sensor distribution*, which can reduce the system's overall coverage efficiency and shorten the AoI lifetime. Therefore, random deployment is often used as an initial phase, followed by a *more strategic deployment* to optimize sensor placement and ensure uniform coverage.

It is well known that *optimal coverage* with *equally sized circles* is achieved by positioning sensor's centers at the vertices of a triangular grid, with an appropriately chosen grid size.



This configuration is used in the **carrier-based method**. Moreover, note that

- since physical movement consumes significant energy, the algorithm aims to minimze the number of robot movements
- to conserve bandwidth and energy, communication must be limited, therefore the approach relies on localized solutions, using only available local information rather than global network data

In the **self-deployment** method, two main approaches are used:

- **centralized (or global) approach**: this approach relies on global information about the network; while it can be effective, it is typically not scalable for large networks
- **distributed (or local) approach**: in this approach, sensors use only local information, with iterative exchanges between neighboring sensors; this method is more scalable and better suited for large, dynamic networks

If no prior information about the AoI is available, an **incremental deployment** strategy is used. This method follows a step-by-step approach, where each newly deployed node relies on data collected by the previously placed nodes to determine its optimal location. The deployment calculations are performed at a powerful *base station*, ensuring precise placement.

Each deployed node plays a crucial role in transmitting its local information back to the base station, which then uses this data for the next deployment iteration. As a result, every node must maintain reliable bidirectional communication with the sink. Notably, no specialized localization technique is required, as the nodes themselves act as landmarks to determine the placement of subsequent nodes.

This strategy has some advantages, for example:

- it ensures the *optimal placement* of each node at every step, leading to a well-structured and efficient network *o*nce deployed, sensors remain *fixed*, minimizing energy consumption and extending their operational lifespan

but it also has some downsides, in fact:

- deployment is *time-consuming*, which can significantly delay network initialization
- it is *computationally demanding*, as each new node placement requires extensive calculations to determine the ideal location

Differently, if prior information about the AoI is available, deployment can be **planned in advance** to ensure complete coverage. Each sensor is assigned a predefined position on a grid that spans the entire AoI, guaranteeing full network coverage and efficient placement.

To optimize performance, the *total energy consumption* should be minimized, ensuring long-term sustainability of the network. This optimization is achieved by modeling the deployment problem using the classical **minimum weight perfect matching approach** in bipartite graphs, which allows for an efficient and balanced assignment of sensor locations. This problem will be analyzed in the following chapter.

We can formalize the graph model of the problem as follows

Definition 8.4: Mobile sensor deployment problem

Given a set of n mobile sensors $S = \{s_1, \dots, s_n\}$, and a set of p locations of the AoI $L = \{l_1, \dots, l_p\}$, where $n \geq p$ — such that complete coverage can be guaranteed — for each $s_i \in S$ determine the location $l_j \in L$ that s_i will have to reach in order to minimize the total energy consumption.

To solve this problem, we will define a **weighted complete bipartite graph** $G = (S \cup L, E)$ weighted through w, described as follows:

- the graph contains one node for each sensor $s_i \in S$
- the graph contains one node for each location $l_j \in L$
- $E(G) = S \times L$, implying that there is an edge (s_i, l_j) for each possible sensor-location pair
- for each edge $e_{i,j}$, the weight $w(e_{i,j})$ is proportional to the energy that s_i has to spend to reach location l_j
- the objective is to determine a **matching** between the sensors and the locations such that the total consumed energy is minimized

8.1 Matchings on bipartite graphs

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 8.5: 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 8.6: 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.

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 8.7: 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

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

be S's **neighbourhood**.

Theorem 8.1: 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 $\forall S \subseteq V_1 \quad |S| \leq |\delta(S)|$. Let M^* be a maximum matching on G and, by way of contradiction, suppose that $|M^*| < |V|$, i.e. there exists at least a vertex $u_0 \in V_1$ not matched through M^*. Let $S_0 := \{u_0\}$; by hypothesis, it must be that $|S_0| \leq |\delta(S_0)|$, which means that there exists a vertex $v_0 \in \delta(S_0)$ such that $u_0 \sim v_1$.

By way of contradiction, assume that v_1 is not covered by the edges of M^*; thus, $u_0 \sim v_1$ would imply that $M^* \cup \{(u_0, v_1)\}$ is a matching that has more edges than M^* — note that u_0 is free w.r.t. M^*, by hypothesis — which is a contradiction by definition of M^*. This proves that v_1 must be matched through M^* with some vertex $u_1 \in V_1$, and clearly $u_1 \neq u_0$.

Let $S_1 := \{u_0, u_1\}$; since $|S_1| \leq |\delta(S_1)|$ and $u_0 \sim v_1 \sim u_1$, there must exist another vertex $v_2 \in \delta(S_1)$ such that $v_2 \neq v_1$. By the same reasoning as before, we have that v_2 must be matched through M^* to a node $u_2 \in V_1$, but note that $u_2 \notin S_1$, since

- $u_2 \neq u_0$ because we chose u_0 as a non-matched vertex w.r.t. M^* by hypothesis
- $u_2 \neq u_1$ since $(u_1, v_1) \in M^*$

Repeating this process, consider the iteration where $S_r = V_1$; since $|S_r| \leq |\delta(S_r)|$, by the same reasoning applied before, there exists a node $v_{r+1} \in V_2 - \{v_1, \ldots, v_r\}$ matched to a node $u_{r+1} \in V_1 - S_r = V_1 - V_1 = \varnothing$, which is clearly a contradiction.

This implies that $|M^*| \geq |V_1|$, which means that M^* is indeed a V_1-perfect matching, by definition.

□

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.

8.1.1 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 8.8: 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 8.1 (Flow networks)**.** The following is an example of a *flow network*.

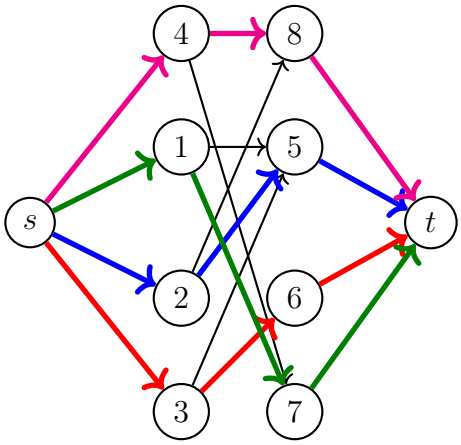


Figure 8.1: A flow network.

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

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

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 8.9: 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, the problem asks to find the flow f with maximum value $|f|$.

Theorem 8.2: 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.

Theorem 8.3

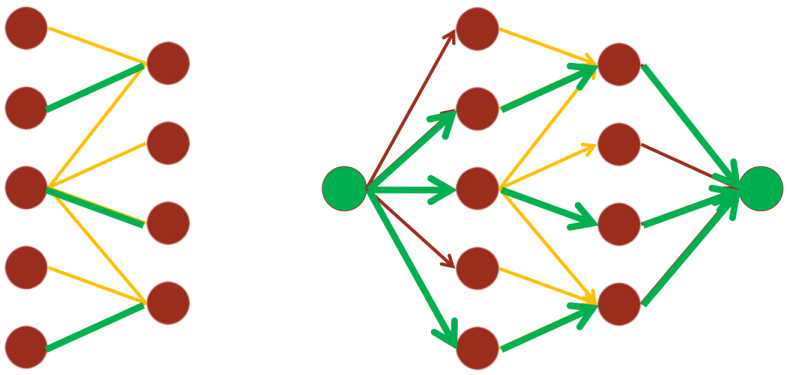
The maximum matching problem on bipartite graphs is reducible to the maximum flow problem on flow networks.

Proof sketch. 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 connecting each node in V_2 to the sink, respectively; moreover, let $E_e := E(G)$ 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

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

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.



□

Corollary 8.1

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 $O(m\mu)$.

8.1.2 Finding a maximum matching

In general, it is not trivial to find a **maximum matching** on a graph. First, we will discuss algorithms and theoretical results which can be leveraged in order to find a maximum matching on **bipartite graphs**.

Definition 8.10: Alternating path

Given a bipartite graph $G = (V, E)$, and a matching M of G, an **alternating path** w.r.t. M is a path that alternates the edges of M and $E(G) - M$.

Definition 8.11: Augmenting path (unweighted case)

Given a bipartite graph G, and a matching M of G, an **augmenting path** w.r.t. M is an alternating path starting and ending in two free nodes w.r.t. M.

In 1957, Berge [5] proved two important theorems in graph theory, one of which is discussed below.

Theorem 8.4: Augmenting paths

Given a graph G, M is a maximum matching of G if and only if there are no augmenting paths w.r.t. M.

Proof.

First implication. Consider the contrapositive: if there exists an augmenting path w.r.t. M, then M is not a maximum matching of G. Consider the edges of an augmenting path A w.r.t. M, and construct a new matching M' of G as follows:

$$M' := (M - A) \cup (A \cap (E(G) - M))$$

therefore, M' still contains the edges that were in M not considering the augmenting

path, but it also contains the edges that were in A and were *not* in M; in other words, M' is M that includes a swapped version of the augmenting path A. Note that, by definition, A starts and ends on free nodes w.r.t. M, which means that the first and last edges of A are not in M, which in turn implies that these two edges will appear in M', by construction. Moreover, note that M' is still a matching, by construction. This implies that $|M| < |M'|$, therefore M' is still a matching but of greater cardinality, hence M cannot be a maximum matching.

Second implication. Consider a graph G, a matching M for G, and assume that there are no augmenting paths in G w.r.t. M. Additionally, by way of contradiction, assume that M is not a maximum matching of G, thus there exists a matching M' for G such that $|M| < |M'|$.

Let H be the multi-graph induced by $M \cup M'$, where edges in $M \cap M'$ are counted twice. By construction, we have that for each $v \in V(H)$ it holds that $1 \leq \deg(v) \leq 2$, therefore each connected component of H is either a cycle or a path. Moreover, cycles in H can only have even length, otherwise an odd-length cycle would imply that there would be a vertex adjacent to two edges belonging to the same matching — since edges *must* alternate between M and M' in the connected compontens of H — which is not possible, by definition of matching. In particular, the connected compontents of H can be classified into 5 kinds:

1. a cycle of length 2 (recall that H is a multi-graph)
2. a cycle of length $2k$, for some $k > 1$
3. a path of length $2k$, for some $k > 1$
4. a path of length $2k + 1$, for some $k > 1$, whose endpoints are incident to M
5. a path of length $2k + 1$, for some $k > 1$, whose endpoints are incident to M'

Among all of these types of possible components, the last is the only type that has more edges that belong to M' than to M, and recall that we assumed, by way of contradiction, that $|M| < |M'|$; this implies that there must be at least one component of the last type, but this component is an augmenting path w.r.t. M, which is a contradiction.

□

This theorem can be leveraged in order to design an *iterative algorithm* which can find a *maximum matching* on a given bipartite graph. First, we need to define an algorithm to find an augmenting path on a bipartite graph.

Algorithm 8.1: Augmenting paths

Given a bipartite graph $G = (V = V_1 \cup V_2, E)$, and a matching M, the algorithm returns an augmenting path on G w.r.t. M, if possible.

1: **function** FINDAUGMENTINGPATH(G, M)
2: Construct a directed bipartite graph $G' = (V', E')$
3: $V_1' := V_1$
4: $V_2' := V_2$
5: $V' := V_1' \cup V_2'$
6: $E' := \{(v, u) \in V_2' \times V_1' \mid (u, v) \in E(G) - M\} \cup \{(u, v) \in V_1' \times V_2' \mid (u, v) \in M\}$
7: $F := \{v \in V_1' \mid v \text{ free w.r.t. } M\}$
8: **while** $F \neq \varnothing$ **do**
9: Choose $v \in F$
10: Continue a DFS starting on v on G', and if the DFS finds a node in $u \in V_2'$ free w.r.t. M, **return** the path $v \to u$
11: $F = F - \{v\}$
12: **end while**
13: **return** `None`
14: **end function**

Idea. By constructing a bipartite graph G', such that:

- an edge goes from V_1 to V_2 if it belongs to M
- an edge goes from V_2 to V_1 if it does not belong to M

it is sufficient to run a DFS starting from any node $v \in V_1'$ that is free w.r.t. M, and if the DFS finds a node $u \in V_2'$ that is still free w.r.t M, then the path $v \to u$ must be an augmenting path.

Cost analysis. Assuming that the various DFS visits on G' will not visit any edge more than once, the cost of the algorithm is the cost of performing 1 single DFS, which is precisely $O(n + m)$.

From this, we can define the following algorithm, which is able to find a *maximum matching* on a given bipartite graph.

Algorithm 8.2: Maximum matching (bipartite graphs)

Given a bipartite graph $G = (V, E)$, the algorithm returns a maximum matching for G.

1: **function** MAXIMUMMATCHINGBIPGRAPHS(G)
2: $M := \varnothing$
3: **do**
4: $p :=$ `findAugmentingPath`(G) ▷ defined in Algorithm 8.1
5: Swap the edges between M and $E(G) - M$ in p
6: **while** $p \neq$ `None`
7: **return** M
8: **end function**

Idea. When the algorithm returns M, there are no more augmenting paths w.r.t. M in G due to the `do-while` exiting condition, therefore M is a maximum matching by the Theorem 8.4.

Note that the algorithm can start from any matching for G, even an empty one $M := \varnothing$, since the first iteration of the algorithm will find an edge of $E(G)$ as augmenting path.

Cost analysis. Note that G cannot contain odd-length cycles, hence the number of free vertices of G w.r.t. M is at most $\frac{n}{2}$. Moreover, when the algorithm finishesm, a maximum matching on G has been found, which implies that there are no more free vertices in G w.r.t. M, thus the number of iterations is at most the number of free vertices of G w.r.t. M. Finally, since the cost of swapping the edges between M and $E(G) - M$ is $O(n)$, the final cost of the algorithm is

$$\frac{n}{2}\left[O(n+m) + O(n)\right] = \frac{n}{2}O(n^2 + nm) = O(nm)$$

Although this naïve solution still yields a polynomial time algorithm, in 1973 Hopcroft et al. [33] developed the so-called Hopcroft-Karp algorithm, which is able to find a maximum matching on a bipartite graphs in $O(m\sqrt{n})$ time, which is better than what this algorithm can achieve.

Algorithm 8.3: Hopcroft-Karp algorithm

Given a bipartite graph $G = (V = V_1 \cup V_2, E)$, the algorithm returns a maximum matching for G.

1: **function** HOPCROFTKARP(G)
2: $M := \varnothing$
3: Construct a directed bipartite graph $G' = (V', E')$
4: $V_1' := V_1$
5: $V_2' := V_2$
6: $V' := V_1' \cup V_2'$
7: $E' := \{(v, u) \in V_2' \times V_1' \mid (u, v) \in E(G) - M\} \cup \{(u, v) \in V_1' \times V_2' \mid (u, v) \in M\}$
8: $F_1 := \{v \in V_1' \mid v \text{ free w.r.t. } M\}$
9: **do**
10: Run a simultaneous BFS starting from all the vertices in F_1, until at least one free node in V_2' is found; let the set of free nodes found in V_2' be F_2
11: Run a DFS starting from the nodes in F_2 and climb the BFS trees up towards F_1; let P the set of all the paths found
12: **for** $p \in P$ **do**
13: Swap the edges between M and $E'(G') - M$ in p
14: **end for**
15: **while** $P \neq \varnothing$
16: **return** M
17: **end function**

Idea. The algorithm starts from a matching M of G, possibly empty, and then defines a graph G', which is constructed as we did in Algorithm 8.1. Moreover, it defines F_1 to be the set of free nodes in V_1' w.r.t. M.

Then, a `do-while` is initialized, in which the algorithm runs a *simultaneous* BFS, starting from all the vertices that are in F_1, which termintes when at least one free node in V_2 w.r.t. M is found; all the free nodes that were found (multiple free nodes can be found simultaneously) are put into F_2. Assume that the algorithm is at the k-th iteration of the `do-while`; inductively, we know that this procedure will reach the $(2k-1)$-th layer in the BFS tree, because all the previous layers must have been found in the previous iterations of the `do-while`. Moreover, note that every path starting from any root in F_1 of the BFS forest that ends in F_2 is

- an augmenting path, since it starts and ends on two free nodes w.r.t. M, by construction of G'
- *node-disjoint*, since we performed a BFS

This means that, by performing a DFS which starts from F_2 and climbs the various trees of the BFS forest towards F_1, the set of paths P encountered will contain *all* the augmenting paths of length $2k-1$. Therefore, updating M w.r.t. P, the algorithm is able to perfomrm the same idea of Algorithm 8.2 but with fewer iterations of the `do-while`.

Cost analysis. Assuming that $m > n$, the cost of the `do-while` is simply the cost of performing a BFS and a DFS, which has cost

$$O(n+m) + O(n+m) = O(n+m) = O(m)$$

Consider the first $\sqrt{n}$ steps of the algorithm; clearly, they take $O(m\sqrt{n})$ time. Note that, at each iteration of the `do-while`, the length of the augmenting paths found keeps increasing, since at the k-th iteration *all* the augmenting paths of length $2k-1$ are found. This means that, after the first $\sqrt{n}$ steps, the shortest augmenting path is at least $2(\sqrt{n}+1)-1 = 2\sqrt{n}+2-1 = 2\sqrt{n}+1$ long.

Moreover, note that the symmetric difference between a maximum matching M' and the partial matching M found after the first $\sqrt{n}$ steps is a set of

- vertex-disjoint alternating cycles
- alternating paths
- augmenting paths

Consider the augmenting paths: each of them must be at least $2\sqrt{n}$ long, therefore there can be at most

$$\frac{n}{2\sqrt{n}+1} = O(\sqrt{n})$$

such paths. This means that the maximum matching M' considered can be larger than M by at most $O(\sqrt{n})$ edges, since each augmenting path increases the cardinality of M by 1. Finally, in the worst case, each step of the algorithm finds one augmenting path per iteration, therefore at most $O(\sqrt{n})$ steps are required after the first $\sqrt{n}$ steps to find all the augmenting paths.

This shows that the algorithm runs in at most $O(\sqrt{n}) + O(\sqrt{n}) = O(\sqrt{n})$ iterations of the `do-while`, and since the cost of a single iteration is $O(m)$, we have that the total cost of the algorithm is

$$O(m) \cdot O(\sqrt{n}) = O(m\sqrt{n})$$

8.1.3 Finding a minimum-weight perfect matching

If the bipartite graph $G = (V, E)$ we are considering has a strictly-positive edge-assigning weight function associated, it is possible to define the following problem.

Definition 8.12: Minimum-weight perfect matching problem

Given a bipartite graph $G = (V, E)$, and a weight function $w : E(G) \to \mathbb{R}_{>0}$, find the perfect matching which minimizes the total weight.

Solving this problem might seem counterintuitive at first: a perfect matching is typically defined as *maximizing* the number of covered vertices, but at the same time this problem aims at *minimizing* the total weight of the edges included in the matching. To address this, it is often more practical to reformulate the problem as follows:

- swap sign of the weights assigned by w (therefore, all the weights are now strictly negative)
- find a **maximum weight perfect matching**

With this reduction, we can aim at maximizing both the weight and the number of covered vertices, simultaneously.

In order to develop an algorithm which is able to find a maximum weight perfect matching, we need to provide a new definition for *augmenting paths* — different from the Definition 8.11.

Definition 8.13: Augmenting path (weighted case)

Given a bipartite graph $G = (V, E)$, and a matching M of G, an **augmenting path** w.r.t. M is an alternating path such that the total weight of the edges in $E(G) - M$ is greater than the total weight of the edges in M.

The **weight** of such an augmenting path A is defined as follows:

$$w(A) := \sum_{e \in E(G)-M} w(e) - \sum_{e \in M} w(e)$$

Note that, differently from the previous definition, this type of augmenting path does not need to end on a free node.

Using this definition, we can construct the following algorithm, though its proof will be omitted due to its complexity.

Algorithm 8.4: Maximum weight perfect matching

Given a bipartite graph $G = (V, E)$, the algorithm returns a maximum weight perfect matching for G.

```
1: function MAXWEIGHTPERFECTMATCHINGBIPGRAPHS(G)
2:     M := ∅
3:     do
4:         p := findMaxWeightAugPath(G)          ▷ algorithm omitted
5:         if w(p) ≥ 0 then
6:             Swap the edges between M and E(G) − M in p
7:         else
8:             break
9:         end if
10:    while p ≠ None
11:    return M
12: end function
```

The cost of this algorithm depends on the implementation, but it is is at least $O(nm)$.

Lastly, in 1955 Kuhn [38] presented the so-called Hungarian method, through which it possible to transform the minimum weight perfect matching problem into an ILP, which can be solved in $O(n^3)$ time. Given a matching M of a bipartite graph $G = (V = V_1 \cup V_2, E)$, and a weight function $w : E(G) \to \mathbb{R}_{>0}$, define the following variables:

- let x_{ij} be a variable for each $(i, j) \in E(G)$, such that $x_{ij} = 1$ if and only if $(i, j) \in M$
- let c_{ij} be the weight $w(i, j)$ of the edge $(i, j) \in E(G)$

We define the following ILP:

$$\begin{aligned} \min \quad & \sum_{(i,j)\in E(G)} c_{ij}x_{ij} \\ & \sum_{j\in V_2} x_{ij} = 1 \qquad \forall i \in V_1 \\ & \sum_{i\in V_1} x_{ij} = 1 \qquad \forall j \in V_2 \\ & x \in \{0,1\}^n \end{aligned}$$

8.2 Maximum matchings in the general case

The previous section explored various algorithms capable of finding maximum matchings in *bipartite graphs*. However, all the algorithms and theorems discussed were based on the following *structural property* of bipartite graphs, which ensures the algorithms achieve favorable time complexity.

Theorem 8.5: Bipartite graphs

A graph is bipartite if and only if it does not contain cycles of odd length.

In fact, the algorithm for finding augmenting paths — in the unweighted version — Theorem 8.4 works by relying on the fact that the graph in input *cannot* contain odd-length cycles. For instance, consider the following scenario:

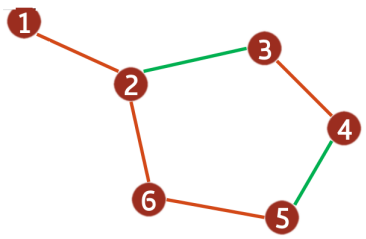


Figure 8.2: A cycle of odd length.

Any matching over this odd-length cycle will not cover every vertex of the cycle, by

definition of matching, otherwise there would be two edges of the matching sharing one vertex. This means that in any odd-length cycle there must be a vertex adjacent to two edges which cannot be part of the matching considered. In the situation illustrated in the figure, if the algorithm starts from 1 and tries to search for an augmenting path in the "wrong" direction (i.e. counterclockwise), the augmenting path that goes from 1 and ends in 6 going clockwise through the cycle will not be found.

In summary, problems arise when there are **blossoms** in the given graph, which are defined below.

Definition 8.14: Blossom

Given a graph G, and a matching M of G, a **blossom** w.r.t. M is a cycle of odd length which contains a maximal number of edges in M.

Example 8.2 (Blossoms)**.** The following is an example of a blossom.

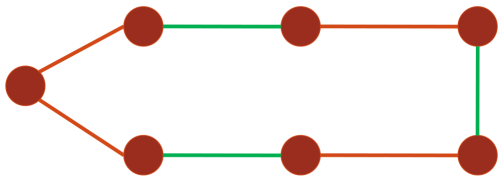


Figure 8.3: A blossom.

Although the presence of *blossoms* does not allow the use of the algorithms discussed earlier, the following lemma enables us to define an algorithm capable of solving the problem, which will be discussed later.

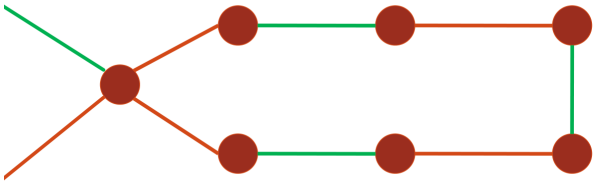
Lemma 8.1: Blossom lemma

Let G be a graph, let M be a matching of G, and let B be a blossom in G w.r.t. M that is *node-disjoint* from the rest of M. Let G' be the graph obtained by contracting B, and let M' be the matching induced by M on G'. Then M is a maximum matching of G if and only if M' is a maximum matching of G'.

Proof.

First implication. By way of contradiction, assume that M is maximum in G, but M' is not maximum in G'; therefore, by Theorem 8.4, there must exist an augmenting path P in G' w.r.t. M'.

Let B be a blossom in G w.r.t. M that is *node-disjoint* from the rest M, and let b be the node representing B in G'. In particular, note that the following situation cannot occur



since B is *node-disjoint* from the rest of M by hypothesis. Therefore, in G', two cases may occur.

- The first case occurs when P does not cross b. This implies that P is an augmenting path for both G' w.r.t. M' and G w.r.t. M, which would imply that M is not a maximum matching by Theorem 8.4, raising a contradiction
- Consider the case in which P crosses b. Note that any edge of $E(G) - E(B)$ that has an endpoint on B cannot be in M, because
  - the uncovered vertex of B is *node-disjoint* from the rest of M, by hypothesis
  - each other node of B is adjacent to an edge in M, by definition

  which implies that b must be free w.r.t. M'. This implies that b must be an endpoint of P. Now, consider B: clearly, we can define in G a path P' formed by $P' := P \cup P''$, since P must exist in G as well, and P'' is the portion of B which can *extend* P as augmenting path. This means that P' is an augmenting path in G w.r.t. M since the uncovered vertex of B is free w.r.t. M, which raises a contradiction by Theorem 8.4 as the previous case.

Second implication. By way of contradiction, assume that M' is maximum in G', but M is not maximum in G; therefore, by Theorem 8.4, there must exist an augmenting path P in G w.r.t. M.

Let B be a blossom in G w.r.t. M that is *node-disjoint* from the rest of M, and let b be the node representing B in G. Thus, as before, in G two cases may occur.

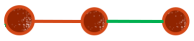
- The first case occurs when P does not cross B. This would imply that P is an augmenting path both for G and G', which would raise a contradiction by Theorem 8.4.
- The second case occurs when P crosses B. Note that, since B contains only 1 single free node w.r.t. M, *at least one* of the endpoints of P must lie outside B, and let this endpoint be w. Let P' be the portion of P that joins b and w in G'; we have that P' is an augmenting path in G' w.r.t. M', which raises a contradiction by Theorem 8.4.

□

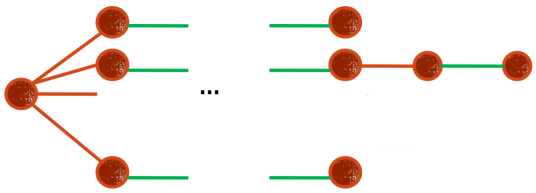
Finally, thanks to this lemma, in 1965 Edmonds [20] designed an algorithm, also known as Blossom algorithm. The idea of the algorithm is the following:

- start with a matching M of G, even empty

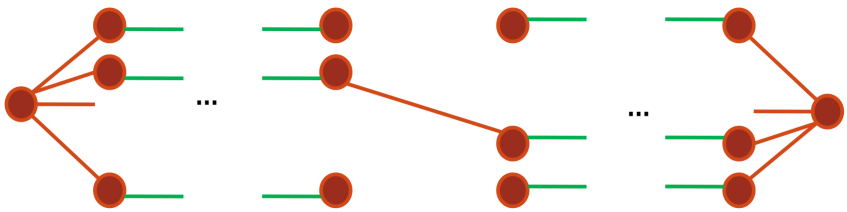
- repeatedly perform the following operations
  - let L be the set of free nodes of G w.r.t. M; F will be a forest of trees rooted on the vertices of L
  - expand F by adding pieces constructed as shown in the picture

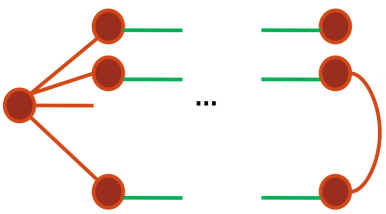
    note that this operation will define two type of nodes
    - **internal nodes**, the ones that are at odd distance from a node of L
    - **external nodes**, the remaining nodes
  - consider the neighbours of the *external nodes*: 4 possible cases can hold
    - there exists an external node x incident to a node y that is not in F; if this is the case, add (x, y) and (y, z) to the edges of F, for some edge $(y, z) \in M$

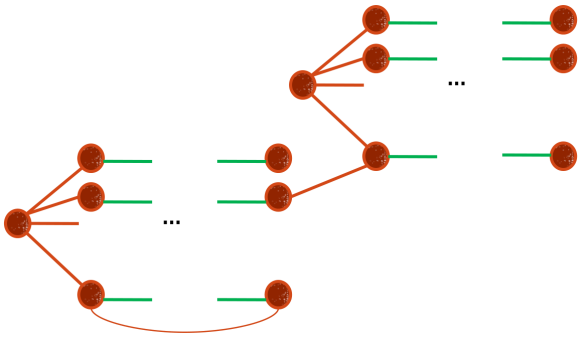
    - two external nodes that belong to two different components of F are adjacent; if this is the case, we found an augmenting path of G w.r.t. M, therefore swap the edges of the augmenting path and start the loop from the beginning

    - two external nodes x and y that belong to the same component of F are adjacent; it this is the case, such an edge defines a cycle on the component of F, which we can contract for the Lemma 8.1

- all the external nodes are adjacent to internal nodes;
- if this is the case, the algorithm ends because M is maximum



In particular, the time complexity of the algorithm depends on how blossoms are handled; in particular, depending on the data structure emplyoed, the cost of the algorithm can be either $O(n^3)$ or $O(mn^2)$. As a final note, the best-known implementation of this algorithms in terms of time complexity has been provided by Micali et al. [41], which showed an implementation that has a cost of $O(m\sqrt{n})$.

9

The sensor self-deployment problem

The previous chapter explored approaches to address the **deployment problem** through *centralized solutions*. These methods rely on a central server to compute and coordinate the deployment of sensors across the AoI.

However, centralized solutions are often *not ideal* due to several limitations:

- they require a continuous connection to a central server
- significant delays are likely to occur
- they lack fault tolerance, making them vulnerable to failures

In contrast, the *mobility of sensors* enables them to **self-deploy** autonomously. Starting from any initial configuration, they can move to a final distribution that ensures complete coverage of the AoI. This decentralized approach offers greater flexibility and resilience.

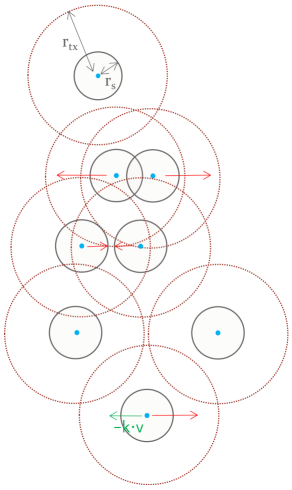
Self-deployment is crucial in "hostile" environments such as contaminated areas, fire zones, and battlefields. In these scenarios, sensors must autonomously position themselves and transmit the data they collect, as direct human intervention is often unsafe or impractical.

Typically, each sensor operates using a **look-compute-move** cycle, a process in which

- the sensor **observes** its surroundings to gather information about its immediate vicinity
- it then **computes** its next action based on this data, deciding where to move to optimize coverage or achieve its objective
- finally, it **moves** to its new position, repeating the cycle until the deployment is complete.

This iterative process enables sensors to *adapt dynamically* to their environment and achieve efficient coverage autonomously.

We can model sensors as *physical particles* influenced by forces such as magnetism and gravity, as shown in the following image:



- **repulsion**: when two sensors are too close to each other, they *repel*, mimicking the behavior of like-charged particles, ensuring that sensors spread out and avoid overcrowding in a given area
- **attraction**: if two sensors are far apart but still within communication range, they exert an *attractive force* on each other, similar to gravitational pull, encouraging sensors to maintain connectivity and fill coverage gaps
- **no interaction**: sensors ignore one another when they are too far apart to communicate, reflecting the absence of forces beyond their interaction range
- additionally, **friction** is introduced into the system to dampen *oscillatory movements*, helping sensors stabilize in their final positions and avoid perpetual motion

Nevertheless, this model inspired by physical particles presents several **weaknesses**:

- **manual parameter tuning**: the model requires *manual adjustment* of parameters such as the strength of repulsive and attractive forces, which can be time-consuming and may not generalize well to different environments
- **sensor oscillation**: sensors may experience oscillatory behavior, preventing them from stabilizing; potential solutions include
  - introducing **friction forces**, to dampen oscillations and promote stability
  - implementing **stopping conditions** that allow sensors to settle once an optimal configuration is reached
- **edge and obstacle effects**: in some implementations, only repulsive forces are considered, which can lead to *unintended consequences*, for instance sensors may be unintentionally drawn toward borders or obstacles due to an unbalanced repulsive force field

These challenges highlight areas for refinement to improve the model's robustness and adaptability in practical scenarios.

9.1 Voronoi diagrams

This chapter will discuss the details of a **deployment protocol** based on **Voronoi diagrams**, which partition the AoI into *distinct regions*, each assigned to a specific sensor. This ensures that every sensor is responsible for covering a well-defined portion of the AoI.

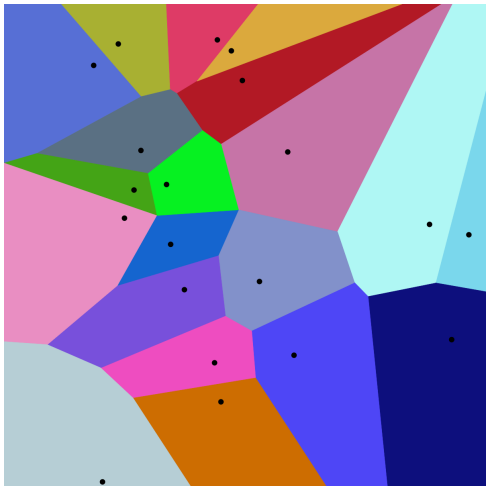


Figure 9.1: An example of a Voronoi diagram.

The protocol operates as follows.

- **Region assignment**. Each sensor is assigned a region of the AoI according to the Voronoi diagram. The region for each sensor consists of all points closer to that sensor than to any other.
- **Coverage responsibility**. Each sensor is tasked with covering its assigned region to the best of its ability.
- **Satisfaction criteria**. A sensor is considered *satisfied* if it completely covers its assigned portion of the AoI, or its entire sensing radius is fully utilized in attempting to cover its portion.
- **Adjustment for unsatisfied sensors**. If a sensor does not meet the *satisfaction criteria*, it moves to a new position to improve its coverage. Movement is guided by local information about its region and the positions of neighboring sensors.
- **Dynamic adaptation**. As sensors move, the Voronoi diagram updates dynamically, redistributing AoI regions to reflect the sensors' new positions.

This approach ensures efficient and adaptive coverage of the AoI while minimizing overlap and redundancy. By leveraging the geometric properties of Voronoi diagrams, the protocol provides a clear and logical framework for sensor deployment.

Definition 9.1: Voronoi diagram

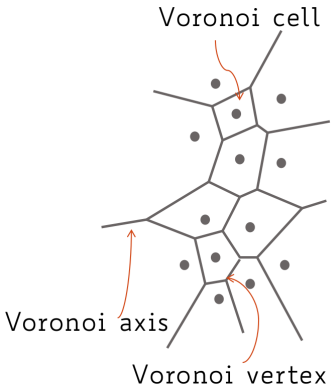
Given a set $\mathcal{P} = \{P_1, \ldots, P_n\}$ of n distinct points on the plane, the **Voronoi diagram** $\text{VD}(\mathcal{P})$ of the set of points is a *partition* of the plane into n cells $V_1, \ldots, V_n$, such that

- each V_i contains exactly 1 point $P_i \in \mathcal{P}$
- if a point Q on the plane lies in V_i, then

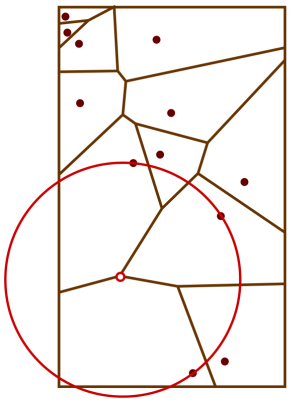
$$\forall P_j \in \mathcal{P}, j \neq i \quad \text{dist}(Q, P_i) < \text{dist}(Q, P_j)$$

In other words, a region V_i of the Voronoi diagram is the set of points that are closer to P_i than any other P_j on the plane.

Example 9.1 (Voronoi diagrams)**.** The following is an exmaple of a Voronoi diagram, along with some definitions of its components:

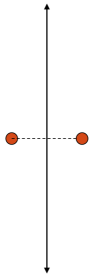


Clearly, by definition, Voronoi vertices are the points of the plane that are equidistant to its neighbouring points in $\mathcal{P}$.

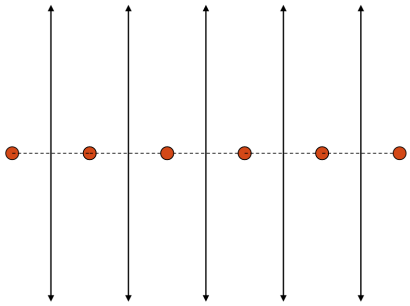


The Voronoi diagram of a single point is the following:

and the Voronoi diagram of two points is described by two half planes, as shown below:



which implies that the Voronoi diagram of *collinear points* is the following:



Note that 4 non-collinear points may generate two different Voronoi diagrams, as shown below:

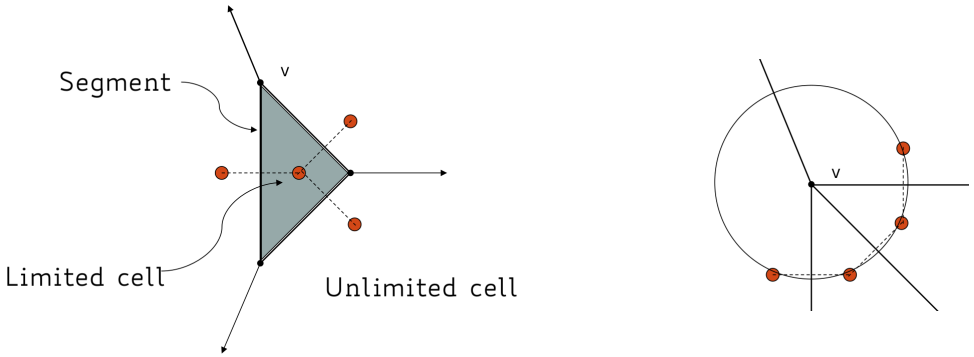


Figure 9.2: Different Voronoi diagrams for 4 points on the plane.

Therefore, we have that:

- a point Q on the plane lies on the **Voronoi segment** between two points P_i and P_j if and only if the largest empty circle centered in Q touches only P_i and P_j
- a point Q on the plane is a **Voronoi vertex** if and only if the largest empty circle centered in Q touches at least 3 sites of $\mathcal{P}$

Let V and E be respectively the sets of vertices and edges of a given Voronoi diagram — note that E also contains the edges that extend infinitely. Thus, it holds the following result.

Theorem 9.1

Consider a Voronoi diagram generated by n points, such that $n \geq 3$; it holds that $|V| \leq 2n - 5$ and $|E| \leq 3n - 6$.

Proof. If the Voronoi diagram is defined by *only* collinear points, then we have $|V| = 0 \leq 2n - 5$ for any $n \geq 3$, and $|E| = n - 1 \leq 3n - 6$ for any $n \geq 3$.

For the general case, consider the given Voronoi diagram; clearly, since it is not defined by *only* collinear points, it will contain some edges that extend infinitely, which implies that the Voronoi diagram does not describe a planar graph. Nevertheless, we can create a *dummy node* p_∞ to which we can connect all the infinite edges, turning the Voronoi diagram into a planar graph.

Now that the graph is planar, we can use Euler's formula which states that

$$|V| - |E| + F = 2$$

where F is the number of faces (i.e. Voronoi regions) of the graph we are considering, which is n, and since the number of nodes is $|V| + 1$ for the *dummy node*, we have that

$$|V| + 1 - |E| + n = 2$$

Moreover, for the Lemma 6.2 we know that

$$\sum_{v \in V \cup \{p_\infty\}} \deg(v) = 2\,|E|$$

and since $\deg(v) \geq 3$ it must be that

$$2\,|E| = \sum_{v \in V \cup \{p_\infty\}} \deg(v) \geq 3\,|V \cup \{p_\infty\}| = 3(|V| + 1)$$

Finally, by using this inequality and Euler's formula, we find the bounds of the statement. □

9.1.1 Algorithms for computing Voronoi diagrams

Voronoi diagrams can be computed by repeatedly intersecting half planes, as shown in the pictures below:

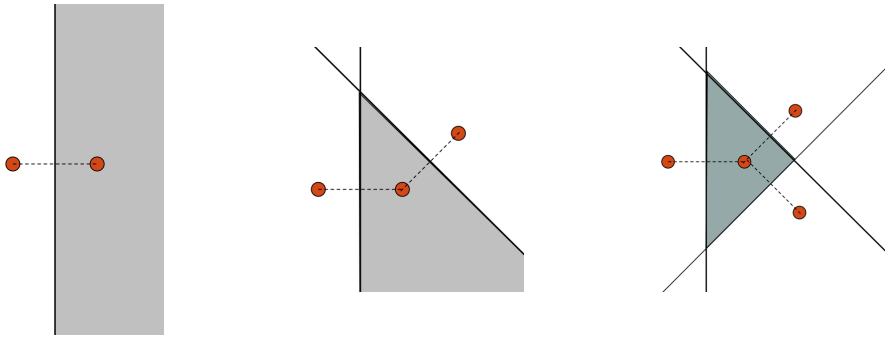


Figure 9.3: Different Voronoi diagrams for 4 points on the plane.

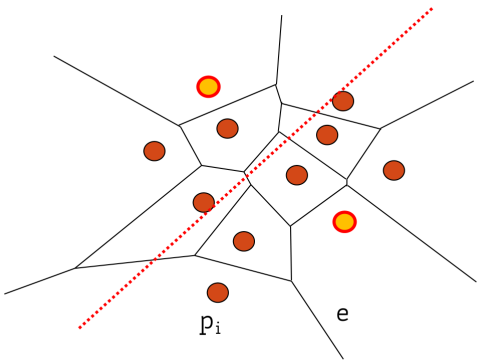
Each of the n Voronoi cells is obtained by intersecting $k = \Theta(n)$ half planes, and to determine the intersection of a certain number of half planes we can employ a *divide-et-impera* strategy:

- **divide**: the set of k half planes is recursively split until k single half planes are obtained, for instance through a tree-like structure
- **impera**: the half plane on each leaf is intersected with the whole search space, therefore each leaf now contains a polygon
- **combine**: recursively, bottom-up, compute the intersection of two sibling polygons and transfer the result on the father node

Two polyogns with p and p' vertices each can be intersected in $O(p + p')$ time, and it can be proven that the computational time of the whole algorithm is $O(k \log k)$. This result is optimal, because the sorting problem through comparisons can be reduced to the problem of intersecting half planes.

Hence, the cost of the algorithm that computes the Voronoi diagram through intersecting half planes is given by the cost of finding a single cell, which is $O(n \log n)$ since there are $O(n)$ half planes, times the number of cells, thus $O(n^2 \log n)$.

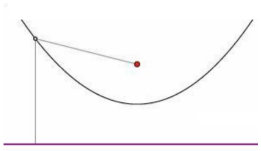
Note that we can actually do better: in fact, not every pair of points in $\mathcal{P}$ describes an axis of the Voronoi diagram, as shown below:



To improve the algorithm, we can employ a well-known technique in computational geometry, called **sweep line**, which is used to solve geometrical problems in two dimensions through a sequence of almost one-dimensional subproblems. When the sweep line moves — i.e. *sweeps* — the algorithm solves the single problem related to the object it is sweeping. Such an algorithm would not work for the Voronoi diagrams, since it would require to predict the position of the points before the sweep lines can sweep them.

Fortnuately, in 1986 Fortune [25] designed an algorithm based on a different type of line, called the **beach line**.

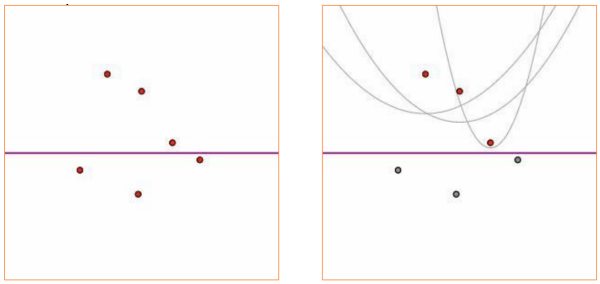
Consider a sweep line sweeping the given points in $\mathcal{P}$. By definition, each *site* — a point of $\mathcal{P}$ — will describe a parabola along with the sweep line that continuously goes down



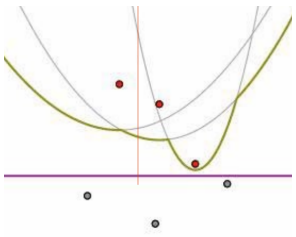
Consider a point $P \in \mathcal{P}$, let the sweep line be l, and consider any point Q on the plane; we have that

- $\text{dist}(P, Q) < l_y - Q_y$ if Q lies above the parabola
- $\text{dist}(P, Q) = l_y - Q_y$ if Q lies on the parabola
- $\text{dist}(P, Q) > l_y - Q_y$ if Q lies below the parabola

As the sweep line lowers its y-coordinate, multiple parabolas will be described.



We define the **beach line** to be the union of all the parabolas described.



If a point is above the beach line, it must be closer to one of the points in $\mathcal{P}$ above the sweep line than to the sweep line itself, by definition. Therefore, such a point will lie inside the Voronoi cell of the site that has already been swept by l, implying that the Voronoi diagram above the beach line is completely determined.

Now, consider a point Q; if Q is touched by the portion of the beach line generated by the point $P_i \in \mathcal{P}$, it will belong to the voronoi cell V_i, generated by P_i, implying that for all $j \neq i$ we have that

$$\text{dist}(Q, P_i) \leq \text{dist}(Q, P_j)$$

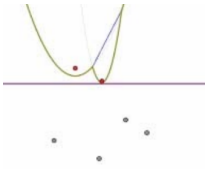
and, recalling that Q lies on the parabola generated by P_i and l if and only if $\text{dist}(Q, P_i) = l_y - Q_y$, we have that

$$\text{dist}(Q, P_j) \geq \text{dist}(Q, P_i) = l_y - Q_Y = \text{dist}(Q, l)$$

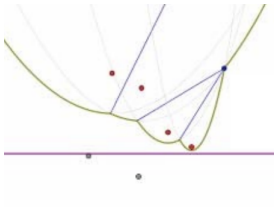
Therefore, when a point appears on the beach line, it is on the parabola associated to its closest site.

Points on the beach line lying at the intersection of two parabola arches are called **breakpoints**. Clearly, breakpoints are closest to two points of $\mathcal{P}$ at the same time, which means that they will lie on the *segments* of the Voronoi diagram that will be generated at the end of the algorithm.

A pair of breakpoints, corresponding to a segment of the Voronoi diagram, appears exactly when the sweep line encounters a new site, which is called **site event**.



Clearly, while the sweep line moves, breakpoints move too, because they will adjust depending on the arches of the parabolas determined with the lowering of the sweep line. But breakpoints move along a line, which turns into a vertex everytime a parabola arch disappears.



A new parabola arch appears everytime the sweep line encounters a new site, while the disappearing condition for the parabola arch — i.e. when it turns into a point, say x — occurs whenever an arch lies on 3 parabolas

- the one containing the disappearing arch
- the one to its right
- the one to its left

Therefore x is equally distant from 3 different points of $\mathcal{P}$, which implies that a circle centered at x passes through these 3 different points. Clearly, such an event determines a Voronoi vertex when the sweep line has finished to sweep this circle, indeed this events are called **circle events**.

Hence, if the next event encountered by the beach line is

- a **site event**, insert the new site encountered in a *list of sites*, in the order dictated by the appearing order of its parabola arch, and add a segment in the final Voronoi diagram
- a **circle event**, store both the new Voronoi vertex in the final diagram, and the information that it is the endpoint of segments corresponding to two breakpoints converging into a single point

and, in both cases, it must be checked whether a new triple of sites producing a next *circle event* has been discovered.

Finally, regarding the computational cost of the algorithm

- each event requires constant time to be detected, and a constant number of accesses to the data structures to be stored
- each data structure contains $O(n)$ information
- each access costs $O(\log n)$ time

therefore, the computational time of the Fortune's algorithm is $O(n \log n)$, and it requires $O(n)$ space, which is optimal since the sorting problem using comparisons can be reduced to the computation of the Voronoi diagrams.

9.2 Sensor heterogeneity

In sensor networks, it is often assumed that sensors are *uniform* in their capabilities and deployment. However, in many practical scenarios, sensors are **heterogeneous**, meaning

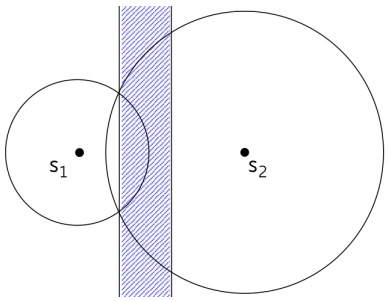
they *differ* in various ways. A heterogeneous sensor network can exhibit the following characteristics:

- **diverse devices**: sensors may vary in terms of their hardware capabilities, such as energy capacity, sensing range, or communication range
- **environment-dependent performance**: sensing and communication abilities are influenced by the physical environment

These differences introduce challenges for traditional methods that assume uniform sensor behavior. Specifically:

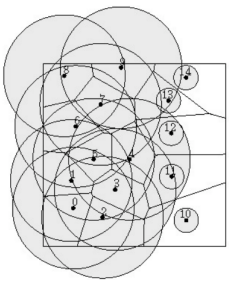
- **virtual force-based approaches**: they rely on *forces* derived from inter-sensor distances, but when sensors have varying coverage or communication ranges, distance alone is *insufficient* to optimize deployment or connectivity effectively
- **Voronoi cell-based approaches**: Voronoi diagrams divide the area into regions based on proximity to sensors but do not account for each sensor's *unique coverage* or *communication capabilities*, leading to suboptimal partitioning in heterogeneous networks

For instance, consider the following scenario:



In this context, a protocol based on the construction of Voronoi cells would determine the rightmost vertical line, which is clearly not the optimal partitioning in this context. In fact, the optimal partitioning is represented by the leftmost vertical line, which precisely divides the plane into two half planes at the intersection points of the circles centered in the two sites s_1 and s_2.

But this is not the only limitation of Voronoi based protocols if we consider networks with heterogeneous sensors. In fact, the following condition may occur



in which the (mobile) sensors on the left — that have the bigger radii — do not move since they completely cover their cells, and the sensors on the right — that have the smaller radii — do not move since their coverage capacity is maximized and cannot cover more than they already cover. This is a **stale situation**.

Because it is difficult to take it into account, in most existing algorithms sensor heterogeneity is typically overlooked, leading to *suboptimal performance* in heterogeneous sensor networks. To address this, Blaschke et al. [6] proposed a new notion of **distance**, which considers both the Euclidean distance and the device heterogeneity. Ideally, the resulting diagrams should satisfy the following properties:

- **straight-edged polygons**: the partitions should form *convex polygons*, which are computationally efficient to handle and simplify deployment planning
- **equidistant point sets**: the set of points equidistant from two sensors should include the intersection of their sensing circles, reflecting the range and coverage capabilities of heterogeneous sensors

The distance that has been introduced is the following, named after Edmond Laguerre.

Definition 9.2: Laguerre distance

Given two points $P = (x_P, y_P, z_P)$ and $Q = (x_Q, y_Q, z_Q)$ in $\mathbb{R}^3$, the **Laguerre distance** between P and Q is defined as follows

$$\text{dist}_L^2(P, Q) = (x_P - x_Q)^2 + (y_P - y_Q)^2 - (z_P - z_Q)^2$$

Given two circles $\mathscr{C}_1$ and $\mathscr{C}_2$, centered at C_1 and C_2 and having radii r_2 and r_2, respectively, the Laguerre distance between $\mathscr{C}_1$ and $\mathscr{C}_2$ is defined as follows

$$\text{dist}_L^2(\mathscr{C}_1, \mathscr{C}_2) = \text{dist}^2(C_1, C_2) - (r_1 - r_2)^2$$

Given two points $P = (x_P, y_P)$ and $C = (x_C, y_C)$ in $\mathbb{R}^2$, and a circle $\mathscr{C}$ centered at C having radius r, the Laguerre distance between P and $\mathscr{C}$ is defined as follows

$$\text{dist}_L^2(P, \mathscr{C}) = (x_P - x_C)^2 + (y_P - y_C)^2 - r^2$$

Lemma 9.1

Given two circles $\mathscr{C}_1$ and $\mathscr{C}_2$, centered at C_1 and C_2 — where $C_1 \neq C_2$ — and having radii r_1 and r_2, respectively, the set of points equally distant (in terms of Laguerre distance) from $\mathscr{C}_1$ and $\mathscr{C}_2$ is a vertical straight line, orthogonal to the segment $\overline{C_1C_2}$, and it is called **radical axis**

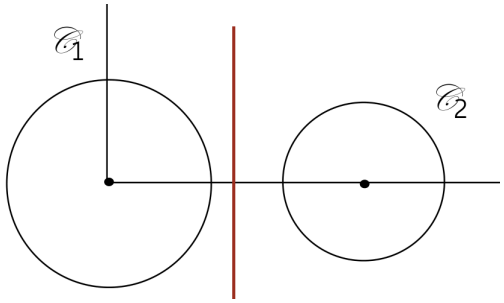


Figure 9.4: The radical axis between two circles.

Lemma 9.2

Given two circles $\mathscr{C}_1$ and $\mathscr{C}_2$, centered at C_1 and C_2 — where $C_1 \neq C_2$ — and having radii r_1 and r_2, respectively, C_1 and C_2 lie on the same side w.r.t. the radical axis if and only if

$$\text{dist}^2(C_1, C_2) < \left|r_1^2 - r_2^2\right|$$

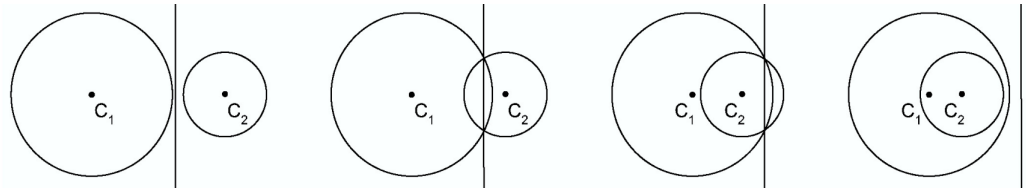


Figure 9.5: Possible positions of the radical axes between two circles.

By using this notion of distance, we can define the following diagrams, called **Voronoi-Laguerre** diagrams, introduced by Imai et al. [34], which are defined as follows.

Definition 9.3: Voronoi-Laguerre diagrams

Given a set of centers $C_1, \ldots, C_n$ of n circles $\mathscr{C}_1, \ldots, \mathscr{C}_n$, respectively, the i-th cell of the associated **Voronoi-Laguerre** diagram is defined as follows:

$$V_i := \bigcap_j \{P \in \mathbb{R}^2 \mid \text{dist}_L^2(C_i, P) \leq \text{dist}_L^2(C_j, P)\}$$

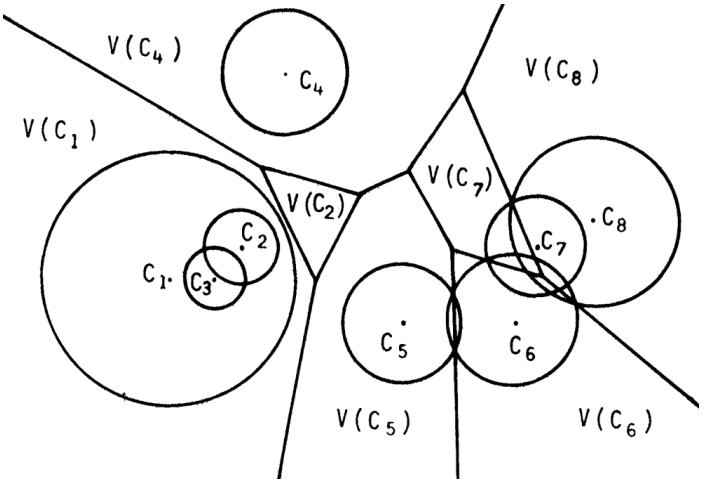


Figure 9.6: A Voronoi-Laguerre diagram.

There are some similarities with the classical Voronoi diagrams, in particular:

- Voronoi-Laguerre polygons partition the plane
- V_i is always convex, since it is generated by intersecting half planes
- clearly, if $r_i = 0$ for each $i \in [1, n]$, the Voronoi-Laguerre diargam degenerates into a classical Voronoi diagram

but it also presents some differences, in fact:

- $\mathscr{C}_i$ could be *external* to V_i (for instance, consider C_2's position w.r.t. V_2 in Figure 9.6)
- V_i could be *empty*, which happens if $\mathscr{C}_i$ is inside the union of other circles (for instance, consider $\mathscr{C}_3$, which has no corresponding V_3 cell in Figure 9.6)

Theorem 9.2

Given n circles $\mathscr{C}_1, \dots \mathscr{C}_n$, centered at $C_1, \dots, C_n$ and having radii $r_1, \dots r_n$, respectively, it holds that

$$\forall i, j \in [1, n] \quad V_i \cap \mathscr{C}_j \subseteq \mathscr{C}_i$$

10

The UAV monitoring problem

Unmanned Aerial Vehicles (UAVs) are advanced flying vehicles capable of *autonomously* determining their flight path, distinguishing them from drones, which are typically operated via remote control.

Initially, UAVs were predominantly employed in military applications. They were primarily deployed in hostile or high-risk areas to minimize the risk to human pilots. In recent years, UAVs have found innovative uses in civilian and commercial domains, including:

- **weather monitoring**: gathering meteorological data in real-time
- **forest fire detection**: identifying and tracking wildfires to aid rapid response
- **traffic control**: monitoring and managing traffic flow for urban planning and safety
- **emergency search and rescue**: locating and assisting individuals in distress in hard-to-reach areas

Suppose an AoI is provided, and assume that in the AoI there is a set $S = \{v_1, \ldots, v_n\}$ of sites that must be examined, and each site v_i requires a time t_i to be inspected. Moreover, consider a fleet of m UAVs that start from a **safe location** v_0, each of which have a battery capacity B. Each UAV must periodically fly back to v_0 in order to recharge its battery, which requires time R (which is often assumed to be from 2.5 to 5 times the value of B). It is crucial to determine the best strategy in order to overfly the sites in S *as efficiently as possible*. In a real-world scenarios, this could make the difference between saving a life and not being able to.

We can model this problem as a graph, as follows.

Figure 10.1: The graph model of the UAV monitoring problem.

In the image shown we have:

- the *AoI*, which is the rectangle enclosing the area
- the *safe location*, which is the red dot in the bottom-left corner
- the *sites*, which are the set of dots in the AoI different from the *safe location*

It must be possible to go from each node to every other node, therefore we assume there is an edge between each pair of nodes; therefore, if $n := |S|$, then the graph can be thought of a clique graph K_{n+1}, where the $(n+1)$-th node is the *safe location*.

Each UAV has a **flying-and-inspection** time that is bounded by B, the capacity of its own battery. We can assume that the graph is edge-weighted, for each pair of sites (v_i, v_j), the weight $w(v_i, v_j)$ represents the time a UAV needs to fly from u_i to u_j.

To distinguish between the UAVs, we will color each path a UAV traverses differently in the graph. In particular, each UAV flies along **cycles**, and in each ecycle it visits as many sites as it can w.r.t. its battery capacity B, passing through the *safe location* v_0 every time it has to recharge its battery (which requires time R).

But what does it mean to overfly the sites in S *as efficiently as possible*? In fact, we may have multiple optimization functions:

- minimize the **total completion time**
- minimize the **average waiting time**
- minimize the **number of cycles**

Let's try to find a problem which satisfies all the constraints of the **UAV monitoring problem**.

Definition 10.1: mTSP

The **multiple Traveling Salesmen** (mTSP) problem is formulated as follows: given a set of n cities, 1 depot, a *cost metric*, and m salesmen that must collectively cover all the cities, determine one tour for each of the m salesmen that minimize the total length w.r.t. the *cost metric*.

This problem does not model the UAV monitoring problem appropriately because it has no visiting time nor "battery capacity" constraint.

Definition 10.2: kTRPR

The **k-Traveling Repairperson Problem with Repairtimes** (kTRPR) is formulated as follows: given n customers, each with a repairtime, 1 depot, and k repairpersons that must visit all the n customers, let the *latency* of a site be the time elapsed before that site is visited by any repairperson; the problem asks to determine k cycles for each of the k repairpersons that minimize the sum of all the latencies of the sites.

This problem is a bit more similar to our problem, but it still does not model the UAV monitoring problem appropriately because it has no battery constraint.

Definition 10.3: mTRPD

The **multiple Traveling Repairperson Problem with Distance constraints** (mTRPD) is formulated as follows: given n customers, 1 depot, and k repairpersons that must visit all the n customers, but that are not allowed to travel a distance longer than a fixed limit, determine k cycles for each of the k repairperson that minimze the total waiting time for the customers.

This problem does not model our problem either, since it does not include repairtimes, and it is not trivial to extend a solution of the mTRPD to include repairtimes.

Definition 10.4: VRP class

The **Vehicle Routing Problem** (VRP) is a generic term for a class of problems focused on optimizing routes for a fleet of vehicles to efficiently serve a set of customers.

This class of problems do not model our problem correctly because there is usually a constraint on the number of visited customers per vehicle, which is not a constraint present in our problem.

Definition 10.5: TOP

The **Team Orienteering Problem** (TOP) is defined as follows: given n sites each with a profit, 1 depot, m vehicles each with a limited total duration for their routes, determine the cycles for each of the m vehicles that maximize the total profit.

Although the first round of this problem is equivalent to our problem, a solution to this problem may be not be best solution to our problem because it may be possible to take advantage of the capacities of the batteries of the UAVs and distributing the problem of covering all the sites throughout multple rounds.

In conclusion, no previously studied problem seems to model the UAV monitoring problem appropriately, which implies that we cannot exploit any know result.

10.1 Connection with the RMCCP

Given a set of locations V, a **cycle cover** $\mathcal{C} := \{C_1, \ldots, C_k\}$ for V is a set of cycles such that each location in V belongs to at least one cycle of $\mathcal{C}$. The **completion time** of a cycle cover is defined as the maximum *cost* among all the cycles of the cycle cover (for some given *cost* function).

Given a fixed value $x \geq 0$, and a *cost* function, an x-bounded cycle cover is a cycle cover in which each cycle of $\mathcal{C}$ has *cost* at most x. Finally, given a *safe location* v_0, a cycle C will be said to be **rooted** in v_0 if $v_0 \in C$, and a **rooted cycle cover** is a cycle cover made of cycles rooted on some shared vertex.

From these definitions, we can introduce the RMCCP, which is described as follows.

Definition 10.6: RMCCP

The **Rooted Minimum Cycle Cover Problem** (RMCCP) is defined as follows: given a graph $G = (V, E)$, where V is a set of locations, a root $v_0 \in V$, a distance function d defined on E, and a positive number $x \geq 0$, determine an x-bounded cycle cover, rooted in v_0, of minimum cardinality, if exists.

The RMCCP has been proved to be approximable, first by Nagarajan et al. [43] in 2012 within a factor of $O(\log n)$, then by Friggstad et al. [26] in 2014 within $O\left(\frac{\log x}{\log \log x}\right)$.

It can be shown that the RMCCP and the UAV monitoring problem are *tightly connected*.

Theorem 10.1

If the RMCCP can be approximated with an approximation ratio of α, the UAV monitoring problem can be approximated with an approximation ratio of $5\alpha + 1$. Moreover, if the UAV monitoring problem can be approximated with an approximation ratio of γ, the RMCCP can be approximated with an approximation ratio of $2\gamma + 1$.

This theorem implies that our problem *inherits* the hardness of the RMCCP. Nevertheless, it is not known whether the RMCCP admits a *constant* approximation algorithm.

Since previously known results do not appear to be exploitable, our problem must be approached as a **new problem**, specifically tailored to the unique requirements of the UAV monitoring context.

10.2 A new graph model

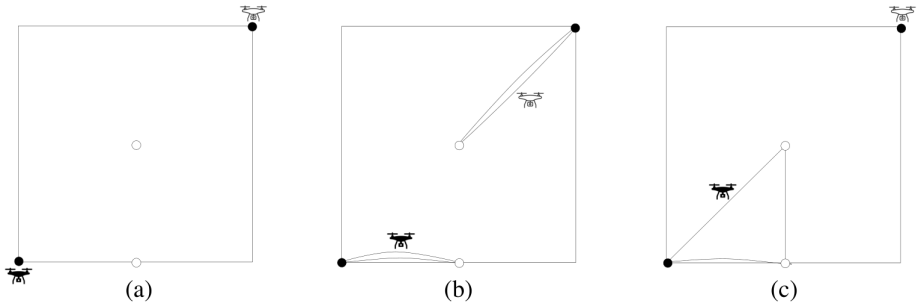
In 2023 Calamoneri et al. [10] introduced the following problem, aimed at modeling and extending the UAV monitoring problem.

Definition 10.7: MDMT-VRP-TCT

The **Multi-Depot Multi-Trip Vehicle Routing Problem with Total Completion Time minimization** (MDMT-VRP-TCT) is defined as follows: given the UAV monitoring problem, extended with multiple depots, determine the cycles for the UAVs that *minimize the total completion time*, subject to the constraints of battery capacity and flying-and-inspection time.

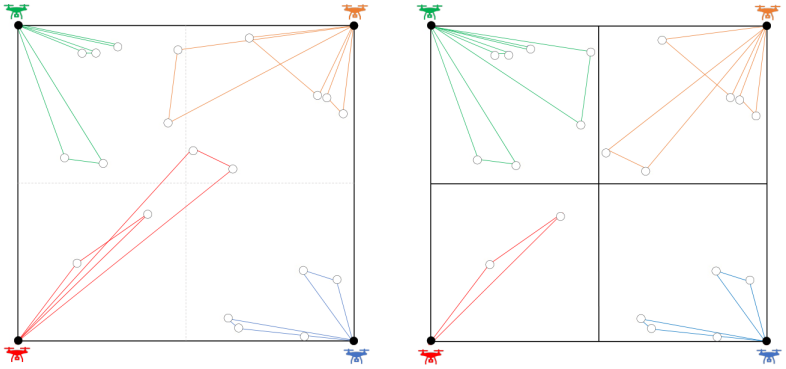
Note that in the formulation we described earlier the UAV monitoring problem had 1 safe location, but the MDMT-VRP-TCT generalizes this formulation by allowing the considered setting to admit multiple depots for the UAVs — note that 1 UAV is *tied* to 1 depot, in the context of this problem. This is particularly useful in real-world scenarios, such as natural disasters, where UAVs are deployed to scan affected areas and prioritize rescue efforts. In such cases, there may also be multiple depots coordinating multiple UAVs.

Note that, in a multi-depot context, minimizing the *completion time* and the total traversed distance is **different**, as shown in the following example.



Given the setting described in (a), where there are 2 UAVs, 2 depots and 2 sites, the figure in (b) describes a feasible solution for a *completion time minimization* problem, which clearly does not minimize the total traversed distance. In fact, in figure (c) there is an example which shows a feasible solution for a *total traversed distance minimization* problem.

Moreover, note that in a multi-depot context, we cannot try to solve the problem by partitioning the area into as many portions as the number of depots, since the sites that fall inside a region are automatically assigned to the UAV of the region, which may lead to suboptimal solutions, as shown in the following picture.



Additionally, the MDMT-VRP-TCT can be formulated as an ILP, by employing the following definitions:

- let a **sequence** be an ordered set of k target nodes (i.e. sites), and let the **duration** d_k of a k-long sequence be the sum between
  - all the traveling times between consecutive target nodes of the sequence
  - the service times of all the target nodes of the sequence
- given a UAV u, and its depot o_u, let a **trip** assigned to u be a k-long sequence that passes through o_u; the duration $d_{k,u}$ of such a $(k+1)$-long trip is defined as the sum between
  - the duration of the k-long sequence of the trip
  - the traveling distance between o_u and the first node of the sequence of the trip
  - the traveling distance between the last node of the sequence of the trip and o_u

Definition 10.8: Compatibility

Given a UAV u with a battery capacity of B, a k-long sequence is said to be **compatible** with u if the duration of its associated trip is upper bounded by B.

The core concept of such an ILP involves generating *all possible sequences* that are compatible with at least one UAV, and selecting the optimal solution. However, this approach may result in an *unmanageably large* number of sequences. To address this, a **matheuristic approach** may be employed, by generating only a *subset* of feasible cycles to input into the model. Notably, determining which sequences to generate becomes a critically important aspect of this method.

Finally, this problem may me extended in multiple ways:

- **priorities** for target locations may be introduced, for example hospitals and other important sites should be served first; this variant was discussed by Calamoneri et al. [9]
- if we assume that UAVs may be able to capture videos and other type of data, it could be possible to consider the problem with a additional constraints for the

memory capacity of the UAVs; this variant was explored by Sorbelli et al. [49]

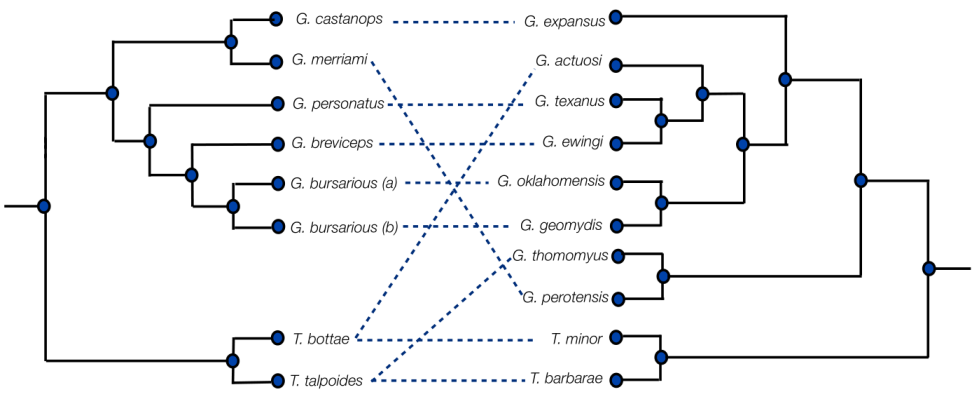
- **cooperation** between UAVs may be introduced
- it could be possilbe to study the problem in the situation in which the UAVs are allowed to recharge the battery (and the memory, optionally) in different depots, which is not allowed by the setting described by the MDMT-VRP-TCT
- the behaviour of the UAVs may be allowed to change dynamically w.r.t. some *emergency* situation

11

The reconciliation visualization problem

Various systems undergo a process called **coevolution**, where they evolve in response to one another's changes over time. This interconnected evolution is observed in multiple contexts:

- **hosts and their parasites or pathogens**: as hosts develop immune defenses, parasites and pathogens adapt to overcome these defenses, resulting in an ongoing evolutionary arms race
- **organisms and their genes**: genes within organisms coevolve, influencing traits and behaviors to ensure survival and reproduction, shaping the organism's overall fitness
- **geographical regions and their species**: ecosystems coevolve with the species they support, as environmental changes drive adaptations in species, which in turn impact the landscape and ecological balance
- **cultural traditions and populations**: cultures and the populations that uphold them coevolve, with traditions influencing societal behavior and values, while societal changes drive the adaptation and evolution of cultural practices



Let H (*host*) be the tree on the left, and P (*parasite*) be the tree on the right. These trees are called **phylogenetic trees**, and the ones in picure represent the evolution stages of ghopers — a family of rodents — and the ones of lice — a family of parasites — respectively. The leaves of the tree are the set of species that currently exist, and the mapping between the leaves of H and P is called a **leaf mapping function** which maps each parasite to the gopher they attack.

In biology, understanding the feasibility of mappings between phylogenetic trees is crucial for determining which parasite species may have attacked which hosts. To achieve this, biologists use a technique known as **reconciliation** of evolutionary trees. In particular, while computing the reconciliation of two trees it is important to follow the *leaf mapping function*, since those are the mappings that actually occur in nature.

Biologists aim to examine *all possible reconciliations* to determine which align with biological feasibility and which do not. Unfortunately, this requires to enumerate all optimal (according to some *cost* function) reconciliations, which could be enormous since it is exponential w.r.t. the size of the input trees. To circumvent this problem, we can either try to reduce the number of optimal reconciliation, or develop a strategy to visualize reconciliations cleverly.

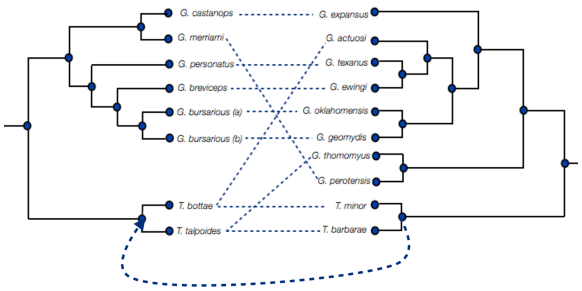
Additionally, some modern computational methods used to reconstruct phylogenetic trees produce **unrooted trees**, which represent the relationships between species without specifying the direction of evolutionary ancestry — i.e. it doesn't indicate which species is the "ancestor" and which are the "descendants".

To transform an unrooted tree into a **rooted tree** (which does indicate ancestry), scientists can employ **reconciliation** techniques, which leverage additional information to infer where the "root" — i.e. the *Least Common Ancestor* (LCA) — of the tree should be placed. By aligning the evolutionary histories of related systems, like hosts and parasites, reconciliations help establish a biologically consistent direction of evolution.

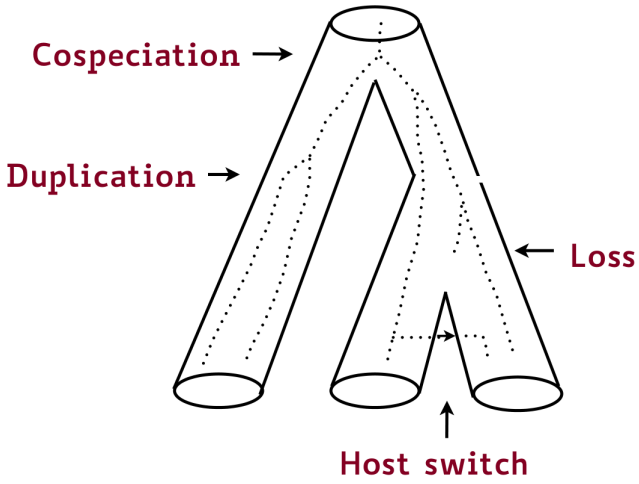
This shows how crucial it is to develop techniques in order to evaluate the feasibility of reconciliations, and to visualize them cleverly.

11.1 Reconciliation between phylogenetic trees

Informally, a **reconciliation** is a mapping from the nodes of the parasite tree P, to the nodes of the host tree H, such that the leaf mapping function betwen H and P is respected.



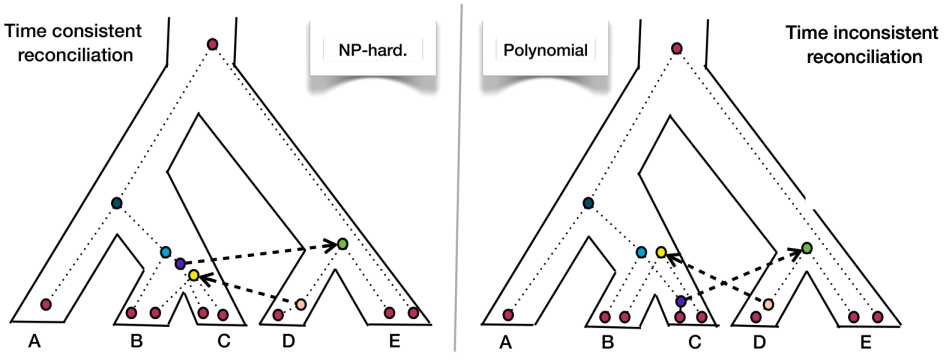
There are multiple ways to visualize the reconciliation between two trees, for example consider the following one shown in the picture, in which the host tree is represented by "tubes", and the parasite tree is contained by such tubes.



From this visualization, it is apparent that it is not straightforward to perform a mapping between the trees, and some phenomena may occur:

- **cospeciation**: in the case of cospeciation, the two trees match perfectly and both the host and the parasite speciated in the same fashion
- **duplication**: this occurs when the parasite speciated but the host did not, which graphically implies that one host tube will contain multiple branches of parasite species
- **loss**: this occurs when a species goes extinct, which graphically implies that there will be a truncated branch in the parasite tree
- **host switch** or **gene transfer**: this occurs when a parasite changed host, which graphically implies that a branch of the parasite tree changeshost tube

Another constraint that is important while computing reconciliations is the **time consistency** constraint, which ensures that the evolutionary events depicted in the reconciliation are temporally plausible. This means that the timing of these events must be consistent with the known or inferred evolutionary timelines of the species involved.



It turns out that computing time consistent reconciliations is NP-Hard, but if we drop this constraint we can compute **time inconsistent reconciliations** in polynomial time by employing a dynamic programming approach.

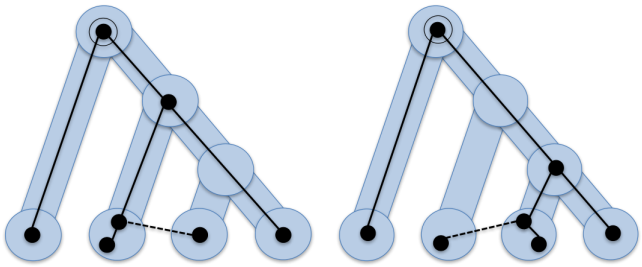
There are multiple *open problems* related to tree reconciliations:

- phylogenetic trees are generally assumed to be correct, but this may not be the case; this problem was explored by Sinaimeri et al. [47]
- if the *leaf mapping function* is *not a function* — which represents the general case and it is a more realistic model — the problem becomes much harder to solve
- is it possible to compute an optimal reconciliation in polynomial time if the distance of the host switches is bounded? (discussed by Tavernelli et al. [51])
- is it possible to compute an optimal time-consistent reconciliation in polynomial time for some particular topologies of the input trees?

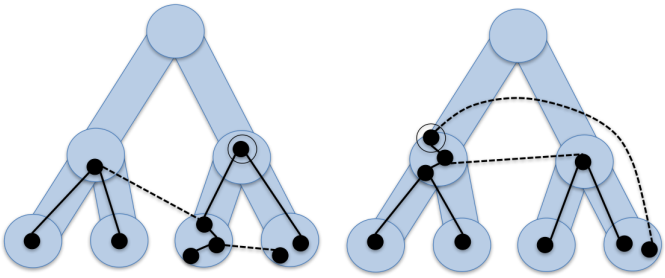
11.1.1 Reducing the number of optimal reconciliations

In order to reduce the number of optimal reconciliations two main approaches are employed:

- **similarity measure**: we can reduce the number by defining a *similarity measure* between reconciliations, i.e. by finding a subset S of all the reconciliations that represent the whole set, such that each of the optimal reconciliations is at a bounded distance from at least one of the reconciliations in S; a common *similarity measure* employed is the smallest number of operations needed to change one reconciliation into another, but this may not represent the concept of "similarity" well enough, as similar reconciliations may need a large number of operations to be changed from one to another



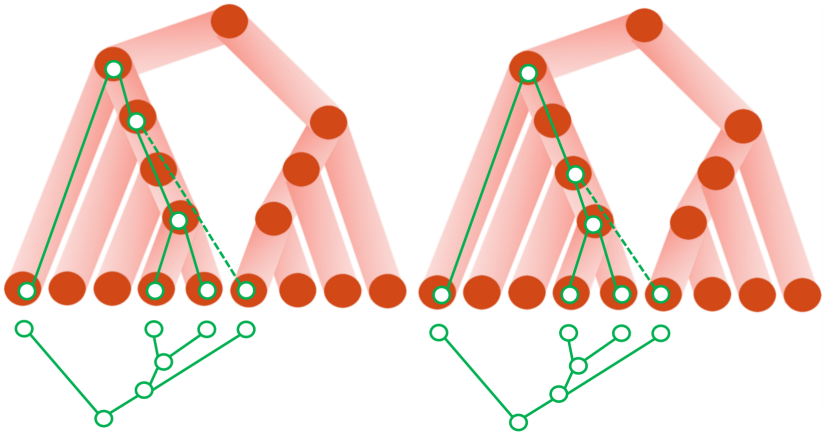
- **equivalence classes**: alternatively, the number may be reduced by using the definition of *equivalence classes* to group the reconciliations that may be considered biologically equivalent, and providing as output just one representative of the whole equivalence class; a commont *equivalence relation* employed is the event vector, which seems to be a good alternative since the number of event vectors is polynomial, but reconciliations with the same event vector may be every different from each other



In 2017 Gastaldello et al. [27] proved that, if the set of vertices of the parasite tree P that are associated to host switches is fixed, an optimal reconciliation can be easily identified by using the LCA mapping. This theoretical result lead to the following theorem.

Theorem 11.1: Equivalence of reconciliations

Two reconciliations are identical if and only if they have the same host switches.

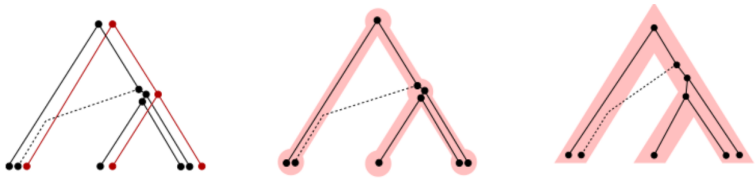


Although this is a good *similarity measure*, the problem is that establishing whether two reconciliations are equivalent requires to enumerate all the optimal solutions, and then cluster them according to this equivalence relation, which is unfeasible when the number of reconciliations is too large.

Is it possible to enumerate only one representative for each equivalence class, without the need of considering all of its elements? It has been proven to be possible for 3 distinct definitions of equivalence relations by Wang et al. [55], but the relations were rather artificial and they are not applicable in real-world contexts.

11.1.2 Visualizing reconciliations

Consider host tree H, a parasite tree P, a leaf mapping function f and a reconciliation R between H and P. There are three main **strategies** in which the reconciliation R is usually visualized:



- R is represented through two paired trees — the leftmost visualization
- P is drawn inside H — the visualization in the middle
- H is represented as pipes and P is drawn inside H's pipes — the rightmost visualization

Example 11.1 (Visualizations)**.** The following are some examples of visualizations generated through multiple programs that compute reconciliations.

Figure 11.1: A visualization produced by the CoRe-PA software (first strategy).

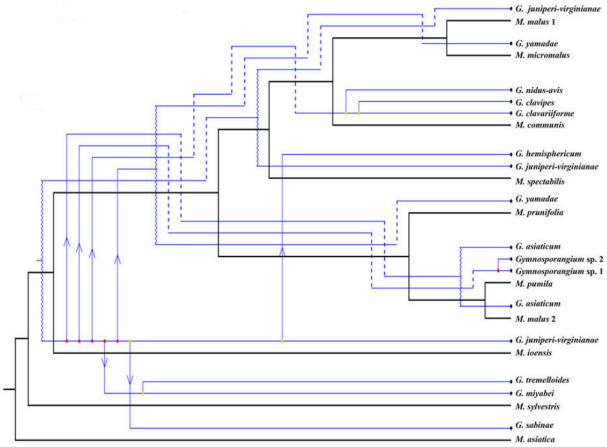


Figure 11.2: A visualization produced by the Jane 4 software (first strategy).

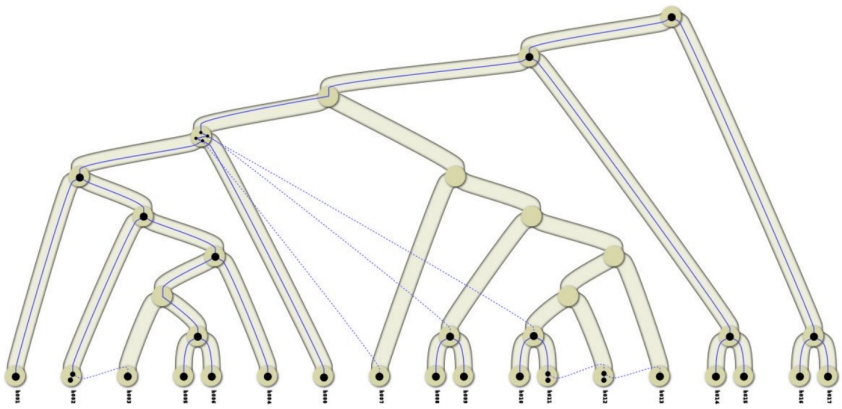


Figure 11.3: A visualization produced by the CophyTrees software (second strategy).

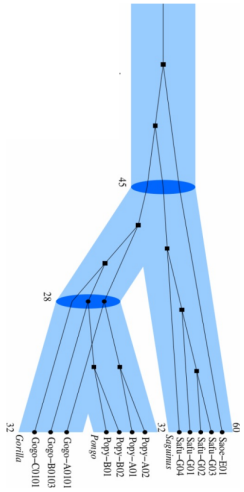


Figure 11.4: A visualization produced by the Primetv software (second/third strategy).

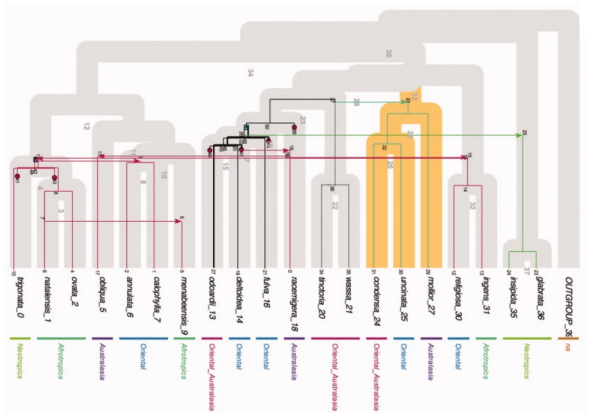
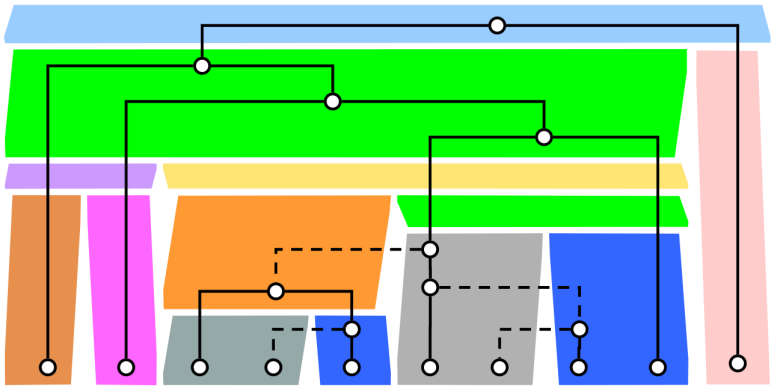
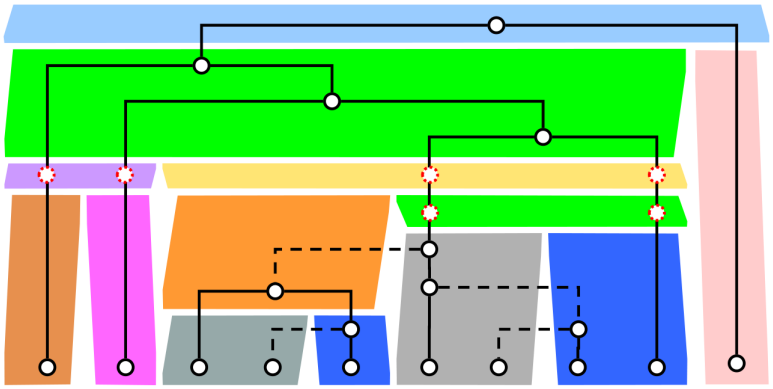


Figure 11.5: A visualization produced by the SylvX software (third strategy).

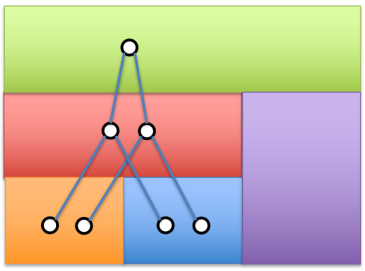
An interesting approach that was recently developed in order to visualize reconciliations is the following:



In this type of visualization, the host tree is represented through colored blocks (optionally slanted), and the parasite tree is drawn inside the colored blocks. Host switches are represented through dashed lines, and losses are represented as shown below



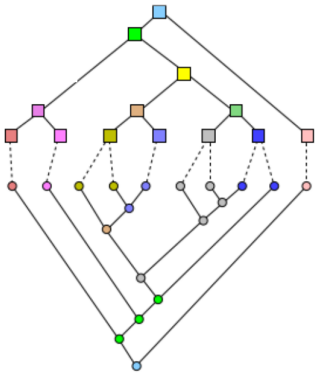
In general, in reconciliation visualizations it is ideal to minimize the number of crossings, since it enhances readability, but this is not always possible, as shown in the following example:



The following two theoretical results offer insight into the feasibility of reducing the number of crossings.

Theorem 11.2

A reconciliation admits a planar representation if and only if the associated tanglegram is planar.



Theorem 11.3

Deciding whether a time-consistent reconciliation admits a visualization drawing with at most k crossing is NP-Complete.

Bibliography

[1] Kenneth Appel et al. "Every planar map is four colorable". In: (1976).

[2] 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.

[3] 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.

[4] 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).

[5] Claude Berge. "TWO THEOREMS IN GRAPH THEORY". In: *Proceedings of the National Academy of Sciences* 43.9 (Sept. 1957), 842–844. ISSN: 1091-6490. DOI: 10.1073/pnas.43.9.842. URL: http://dx.doi.org/10.1073/pnas.43.9.842.

[6] Wilhelm Blaschke et al. "Vorlesungen über Differentialgeometrie und geometrische Grundlagen von Einsteins Relativitätstheorie: Elementare Differentialgeometrie". In: (1924).

[7] 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.

[8] Mario Čagalj et al. "Minimum-energy broadcast in all-wireless networks: NP-completeness and distribution issues". In: *Proceedings of the 8th annual international conference on Mobile computing and networking*. 2002, pp. 172–182.

[9] Tiziana Calamoneri et al. "A realistic model to support rescue operations after an earthquake via UAVs". In: *IEEE access* 10 (2022), pp. 6109–6125.

[10] Tiziana Calamoneri et al. "Management of a post-disaster emergency scenario through unmanned aerial vehicles: Multi-Depot Multi-Trip Vehicle Routing with Total Completion Time Minimization". In: *Expert Systems with Applications* 251 (Oct. 2024), p. 123766. ISSN: 0957-4174. DOI: 10.1016/j.eswa.2024.123766. URL: http://dx.doi.org/10.1016/j.eswa.2024.123766.

[11] Gerard J Chang et al. "The L(2,1)-labeling problem on graphs". In: *SIAM Journal on Discrete Mathematics* 9.2 (1996), pp. 309–316.

[12] Gerard J Chang et al. "The L (2, 1)-labeling problem on ditrees". In: *Ars Combinatoria* 66 (2003), pp. 23–31.

[13] Weifang Cheng et al. “Sweep coverage with mobile sensors”. In: *2008 IEEE International Symposium on Parallel and Distributed Processing*. IEEE, Apr. 2008, 1–9. DOI: 10.1109/ipdps.2008.4536245. URL: http://dx.doi.org/10.1109/IPDPS.2008.4536245.

[14] 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.

[15] 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.

[16] 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.

[17] 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.

[18] 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).

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

[20] Jack Edmonds. “Paths, Trees, and Flowers”. In: *Canadian Journal of Mathematics* 17 (1965), 449–467. ISSN: 1496-4279. DOI: 10.4153/cjm-1965-045-4. URL: http://dx.doi.org/10.4153/CJM-1965-045-4.

[21] 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.

[22] 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.

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

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

[25] Steven Fortune. “A sweepline algorithm for Voronoi diagrams”. In: *Proceedings of the second annual symposium on Computational geometry*. 1986, pp. 313–322.

[26] Zachary Friggstad et al. “Approximation algorithms for regret-bounded vehicle routing and applications to distance-constrained vehicle routing”. In: *Proceedings of the forty-sixth annual ACM symposium on Theory of computing*. STOC ’14. ACM, May 2014, 744–753. DOI: 10.1145/2591796.2591840. URL: http://dx.doi.org/10.1145/2591796.2591840.

[27] Mattia Gastaldello et al. "Extracting Few Representative Reconciliations with Host Switches". In: *Computational Intelligence Methods for Bioinformatics and Biostatistics*. Springer International Publishing, 2019, 9–18. ISBN: 9783030141608. DOI: 10.1007/978-3-030-14160-8_2. URL: http://dx.doi.org/10.1007/978-3-030-14160-8_2.

[28] Barun Gorain et al. "Approximation algorithms for sweep coverage in wireless sensor networks". In: *Journal of Parallel and Distributed Computing* 74.8 (Aug. 2014), 2699–2707. ISSN: 0743-7315. DOI: 10.1016/j.jpdc.2014.02.009. URL: http://dx.doi.org/10.1016/j.jpdc.2014.02.009.

[29] Barun Gorain et al. "Solving energy issues for sweep coverage in wireless sensor networks". In: *Discrete Applied Mathematics* 228 (Sept. 2017), 130–139. ISSN: 0166-218X. DOI: 10.1016/j.dam.2016.09.028. URL: http://dx.doi.org/10.1016/j.dam.2016.09.028.

[30] Jerrold R. Griggs et al. "Labelling Graphs with a Condition at Distance 2". In: *SIAM Journal on Discrete Mathematics* 5.4 (Nov. 1992), 586–595. ISSN: 1095-7146. DOI: 10.1137/0405048. URL: http://dx.doi.org/10.1137/0405048.

[31] 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.

[32] Toru Hasunuma et al. "An Algorithm for L (2, 1)-Labeling of Trees". In: *Scandinavian Workshop on Algorithm Theory*. Springer. 2008, pp. 185–197.

[33] John E. Hopcroft et al. "An $n^{5/2}$ Algorithm for Maximum Matchings in Bipartite Graphs". In: *SIAM Journal on Computing* 2.4 (Dec. 1973), 225–231. ISSN: 1095-7111. DOI: 10.1137/0202019. URL: http://dx.doi.org/10.1137/0202019.

[34] Hiroshi Imai et al. "Voronoi diagram in the Laguerre geometry and its applications". In: *SIAM Journal on Computing* 14.1 (1985), pp. 93–105.

[35] Theodore Kimball Jonas. *Graph coloring analogues with a condition at distance two: L (2, 1)-labellings and list lambda-labellings*. University of South Carolina, 1993.

[36] 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.

[37] Lefteris M Kirousis et al. "Power consumption in packet radio networks". In: *Theoretical Computer Science* 243.1-2 (2000), pp. 289–305.

[38] H. W. Kuhn. "The Hungarian method for the assignment problem". In: *Naval Research Logistics Quarterly* 2.1–2 (Mar. 1955), 83–97. ISSN: 1931-9193. DOI: 10.1002/nav.3800020109. URL: http://dx.doi.org/10.1002/nav.3800020109.

[39] 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.

[40] 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.

[41] Silvio Micali et al. "An O(v|v| c |E|) algoithm for finding maximum matching in general graphs". In: *21st Annual Symposium on Foundations of Computer Science*

(sfcs 1980). IEEE, Oct. 1980, 17–27. DOI: 10.1109/sfcs.1980.12. URL: http://dx.doi.org/10.1109/SFCS.1980.12.

[42] 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.

[43] Viswanath Nagarajan et al. "Approximation algorithms for distance constrained vehicle routing problems". In: *Networks* 59.2 (2012), pp. 209–214.

[44] 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.

[45] 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.

[46] 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.

[47] Blerina Sinaimeri et al. "1 The event-based model". In: ().

[48] 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.

[49] Francesco Betti Sorbelli et al. "Wireless IoT sensors data collection reward maximization by leveraging multiple energy-and storage-constrained UAVs". In: *Journal of Computer and System Sciences* 139 (2024), p. 103475.

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

[51] Daniele Tavernelli et al. "Linear Time Reconciliation With Bounded Transfers of Genes". In: *IEEE/ACM Transactions on Computational Biology and Bioinformatics* 19.2 (2020), pp. 1009–1017.

[52] 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.

[53] 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.

[54] 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.

[55] Yishu Wang et al. "Making Sense of a Cophylogeny Output: Efficient Listing of Representative Reconciliations". en. In: Schloss Dagstuhl – Leibniz-Zentrum für In-

formatik, 2021. DOI: 10.4230/LIPICS.WABI.2021.3. URL: https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.WABI.2021.3.

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

[57] Gerd Wegner. "Graphs with given diameter and a coloring problem". In: 1977. URL: https://api.semanticscholar.org/CorpusID:118873943.

[58] 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.

[59] 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.