Network Algorithms' Notes

Daniele Bertagnoli

2022/2023

Contents

1 Introduction

1.1 Routing Problem

In general, a routing problem is about sending packets from a computer to another through the network. Each node in the network must compute the route for that packet, in order to reach the destination. We can have two main types of routing algorithm:

- **Non-adaptive routing**: a routing algorithm could try to send packets through a network so that the length the used path is minimized. Such length can be measured in terms of number of hops between pairs of computers. When the algorithm is represented using a graph, it is modeled as **shortest path problem**. This is a good solution with a consistent topology and traffic, the performances decrease when the traffic volume changes over time. We typically use **routing tables**.
- **Adaptive routing**: it takes account of the traffic conditions in the network, so the packet is sent to the network zones that are less congestioned. The algorithm can be represented using a **edge-weighted graph** that models the **least cost path problem**. This could be a good solution when the network's workload is unbalanced or high. Each node must compute its own routing table so these algorithms have an overhead due to the table creation and update.

We can also model an **half-adaptive routing algorithm** that switches between the two solutions based on the network traffic' state.

Routing with faults

When the problem is modeled as a graph, the weights can be also interpreted as the **probability of failure of that edge** (e.g. used for telephone lines). So in all these cases, we will look for the route with the highest probability of success. In detail, let $p(e)$ the probability that the edge e does not fail, if we assume that each edge can fail independently with respect to the other edges, the probability that the path $P = (e_1, e_2, ..., e_n)$ does not fail is given by $p(e_1) * ... * p(e_n)$. We can build a least cost path problem starting from there, since the logarithm is monotonic increasing, $p(e_1) * ... * p(e_n)$ is maximized when $\log p(e_1) * ... * \log p(e_n)$ is maximized. Since $p(e) \geq 1$ then $\log p(e) \leq 0$, therefore if we consider a function $w(e) = -\log p(e)$ where $w(e) \geq 0$, we can try to minimize $w(e_1) * ... * w(e_n)$.

1.2 Shortest Path Problem

Let $G = (V, E)$ be a graph and $w(e)$ the length (weight) of each edge e. We can model different versions of this problem:

- **All to all**
- **All to one**
- **One to all**
- **One to one**

Where the lengths can be:

- **All equal (unit length)**
- **Non negative**
- **Possibly negative but without negate cycles**
- **Creating negative cycles**

1.3 Least Cost Path Problem

G is a graph (either directed or not) and $w : E \to R0$ is the edge-weight function. So, (G, w) is called **network**, for each path $P = (e_1, e_2, ..., e_n)$ the length of P is defined as $w(P) = w(e_1) + ... + w(e_n)$. Given two nodes a and b, the **distance between** a **and** b, s $d(a, b)$ is defined as the minimum cost among the possible path. When b cannot be reached from a, then $d(a, b) = \infty$. If the minimum does not exist (there are negative cycles), then **only networks without negative cycles are feasible**. In general, **in any solution we want to avoid ANY cycle, the best path can contain at most** $n - 1$ **nodes:**

- **Avoid cycles with negative length**, avoided by hypothesis.
- **Avoid cycles with positive length**, cannot exist (by contradiction) otherwise exists a new optimal solution without that cycle.
- **Avoid cycles with null length**, otherwise we could remove that cycle from the solution without affecting the solution.

To determine univocally the path, is sufficient that each node stores the predecessor.

These are some classical algorithms to solve the least cost path problem:

- **Relaxiation**: is used to estimate the weigth of the shortest path from a to b.

- At the beginning $d(v)=\infty$ for each v
- One relaxation step is performed as follows:
  - Given an edge (u,v)
  - If $d(u)+w(u,v)<d(v)$

    $d(v)=d(u)+w(u,w)$

    $pt(v)=u$
- Time complexity of one relaxation step: $O(1)$

- **Bellman-Ford Algorithm**: it solves the shortest path one-to-all. This solution works when G has **no cycles of negative length**.

For each v initialize $d(v)$ and $p(v)$ $O(n)+$
For $i=1$ to $n-1$ do $|n-1$ times
  For each (u,v) relax v w.r.t. (u,v) $|\ mO(1)$

- Time Complexity: $O(nm)$

This algorithm is used for **distance vector routing**, an iterative and asynchronous protocol.

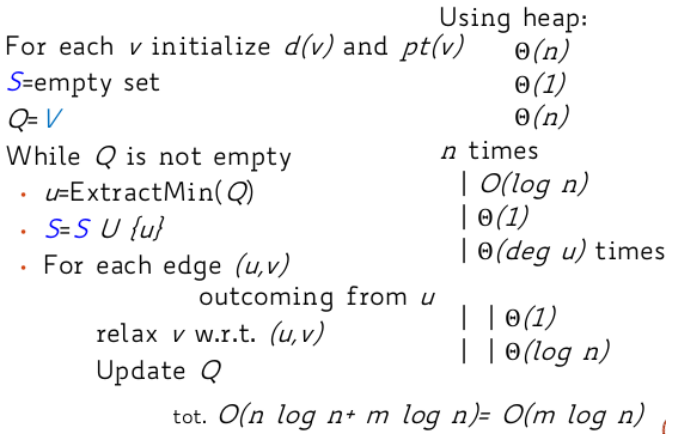
- **Dijkstra Algorithm**: it assumes that G has **non-negative edge weights**. The idea is to partition the nodes of the graph, S all the nodes that have already a shortest path to the destination and $V-S$ the remaining nodes. We keep $V-S$ in a priority queue (e.g. min heap). At each step, we take u as the node with the minimum distance d from the source and relax all the edges outcoming from u.

- For each v initialize $d(v)$ and $pt(v)$
- S=empty set
- $Q=V$
- While Q is not empty
  - u=ExtractMin(Q)
  - $S=S\ U\ \{u\}$
  - For each edge (u,v) outcoming from u
    relax v w.r.t. (u,v)
    Update Q

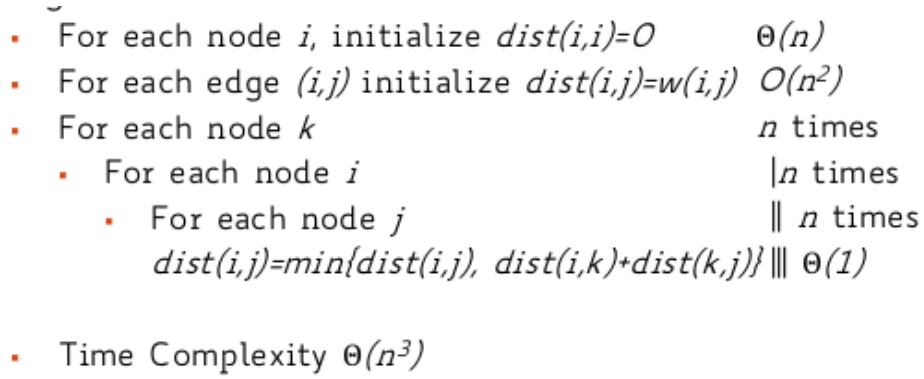
The time complexity depends on the data structure used to implement Q:

Queue: $O(n^2)$
Heap: $O(m\ log\ n)$
Fibonacci Heap: $O(m+n\ log\ n)$

If we use the best structure, so the heap:



- **Floyd-Warshall Algorithm**: is used to solve the all-to-all problem. We modify the graph in a way that all the edges exits, if the edge does not exist in the original graph, the distance between the two node is ∞. We can build a matrix in which each cell $[i, j]$ is the minimum distance between i and j.



2 Interconnection Network

In the previous cases, was not necessary that every node was knowing the whole graph (since we were assuming that each node was a signle computer for instance). However, **interconnected topologies** connect different parts of the same computer. In this case, the **efficiency is the main issue that we want to optimize** since the algorithm determines the throughput of the machine. There are several routing models, we will focus on **store-and-forward** (also known as *packet switching*). In this model:

- **Data are splitted into packets**.

- **Only one packet can cross the edge during a routing step**.
- **Each packet is mantained by one node**.
- **In some algorithms, nodes have queues where the packets are stored if the node is busy**.

Since we are a in single machine, we have a global controller that controls where the packets must be forward to (in the previous cases the algorithms were distributed). The one-to-one packet is very easy to solve since we have the whole network in the controller, we have to focus on **many-to-one** and **one-to-many** problems.

2.1 Interconnection Network Topologies

2.1.1 Butterfly Network

They are layered graph (where the layer refers to the vertical layer), in which let $N = 2^n$ the the n-dimension Butterfly is composed by:

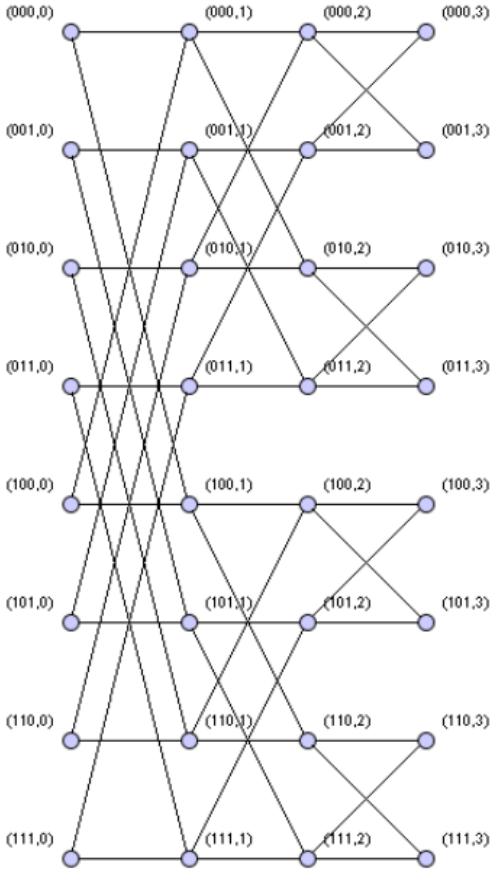
- $N(n+1)$ nodes, we have $n+1$ layers composed by 2^n nodes.
- $2 * N * n$ edges.

Each node corresponds to a pair (w, i) where:

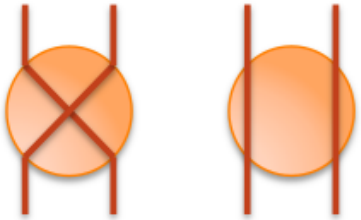
- i is the layer of the node.
- w is a n-binary number that denotes the row of the node.

Two nodes (w, i) and (w', i') are linked by and iff $i' = i + 1$ and either

- $w' = w$.
- w and w' differ in the $i - th$ bit (called **cross edge**).



Input and outputs are usually processors and are often omitted in the draw. We add the inputs and outputs in this way to ensure that **each node is exactly a 4-degree node**. Nodes of the butterfly networks are **crossbar switches** so each of those have 2 inputs and 2 outputs, hence they can assume one of these two states: **corss** and **bar**. **The butterfly network can be seen as a switching network** that connects $2N$ inputs with $2N$ outputs using $n+1$ layers (each of those with N nodes)



The **butterfly networks are recursive structures**, the n-dimensional BN contains two $(n-1)$-dimensional BN.

For each pair each pair of rows w and w' exists a unique path of length n known as **gready path** from $(w, 0)$ and w', n built as follows:

- We take the crossing edge from i to $i+1$ if the w and w' differs on i-th bit.
- We take the straight edge otherwise.

This gready algorithm is perfect when we have to transfer only one packet, but when we have to pass multiple packets, it could happen that more than one packet arrives to the same node. This means that we need a **queue**, in fact, the BN cannot ensure that all the packets are delivered without delays so we call these cases as **blocking network**. In general, **given any routing problem on a n-dimensional BN for which at most one packet starts at each layer-0 node and at most one packet is destinationated to each layer-n node, the packet are delivered in $O(\sqrt{N})$.**

Proof. Let's assume that n is odd for simplicity (the even case is similar). Let e be any edge in layer i, where $0 \leq i \leq n$. Let n_i be the number of paths that traverse e. Now we can define the two trees, the left one and the right one with respect to e: $n_i \leq 2^{i-1}$ **left tree** and $n_i \leq 2^{n-1}$ **right tree**. Any packet crossing e can be delayed by at most $n_i - 1$ packets. So the total delay is:

$$\sum_{i=1}^{n}(n_i-1) = \sum_{i=1}^{(n+1)/2}(n_i-1) + \sum_{i=(n+3)/2}^{n}(n_i-1) \leq \sum_{i=1}^{(n+1)/2}(2^{i-1}-1) + \sum_{i=(n+3)/2}^{n}(2^{n-i}-1) \leq$$

recalling that $\sum_{j=0}^{k} 2^j = 2^{k+1}-1$

$=(n+1)/2+1$

$= \sum_{j=0}^{(n+1)/2-1}(2^j-1)$

$= \sum_{j=0}^{(n-3)/2}(2^j-1)$

$$\leq 2^{(n+1)/2} + 2^{(n-1)/2} - n = O(\sqrt{N}) - n = O(\sqrt{N})$$ ∎

However, **even if the algorithm seems to perform poorly in the worst case, BN are very useful in practice**.

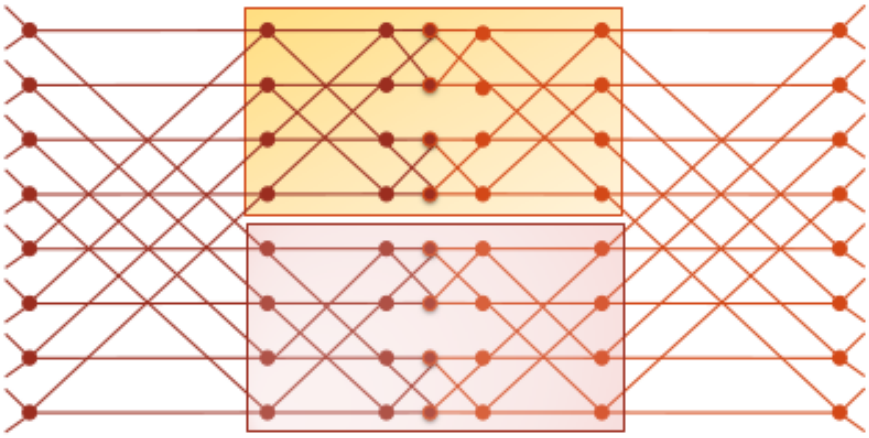
2.1.2 Beneš Network

They are **non-blocking topologies** obtained starting from the butterfly networks. In detail, by taking two butterfly networks, one is flipped and then merged to the other one. The n-dimensional

Beneš network has $2n + 1$ layers, each of those with 2^n nodes. The **Beneš networks are non-blocking** even if we try to deliver $2N$ inputs to $2N$ outputs, so **for any permutation we can always define edge-disjoint paths for every node**.

Proof This can be proved by induction on n.

- Case $n = 0$: the Beneš network consists of a single node, so a 2x2 switch. Is simple to proof that is non-blocking.
- Inductive case: assume that is proved for $n - 1$ dimensional networks. Since the Beneš networks are recursive structures, the n-dimensional network is composed by two $n - 1$ dimensional networks. For each input, we can choose which of the two inputs can go to the first and which must go through the second network. Since for those two networks we have proved that they are non-blocking by induction, we can state that also the n-dimensional network is not blocking since the two inputs are travelling over two different networks.

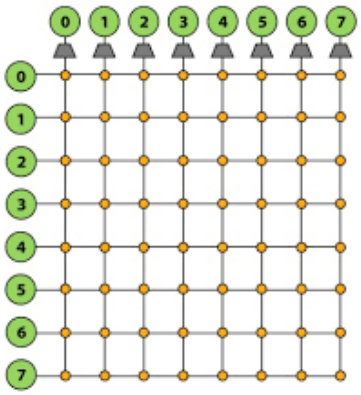


This proof gives us also the routing algorithm for the Beneš networks, in fact, for each couple of inputs, we start by sending the input to the upper-sub network and then route the other input to the lower-sub network. We can iterate this steps until the inputs reach their destinations. This is called **looping algorithm**.

Moreover, with the Beneš networks, in the case that we have at most one packet for each input node that must be delivered, at most one for each output node, we can **always build a node-disjoint path**. In this case, the couple of inputs are not $2i$ and $2i + 1$ but i and $i + 2^{(n-1)}$. The main drawback of this looping algorithms (both versions), is that we need a global controller that knows all the permutations and that runs this algorithm for every input. Every time that a new permutation must be routed, $\theta(N \log n)$ time is necessary to reset the switches.

2.1.3 Mesh Networks

These networks are built as $m * n$ matrix in which we have one edge that connects every input node to every output node. This type of network is very used since the routing algorithm is very easy and intuitive.



2.2 Interconnection Topology Layout Problem

The interconnection topology layout problem arises from the problem of producing efficient VLSI (Very Large Scale Integration) layouts on a silicon board. The main idea is to represent the circuit as a graph where the nodes are the ports, the swithces are the wires and so on.

The VLSI technology imposes many constraints:

- The device that prints the connections can only draw orthogonal lines (**orthogonal drawing**).
- In order to avoid interferences between the connections, we must keep them far enough (**grid drawing**).
- Since the wires cannot touch with each other, each cross must be implemented as a hole in the board. We must **minimize the number of crosses in the graph**.
- The silicon is very expensive, so the layout must have a small area (**area minimization**).
- Wires should not be too long to avoid propagation delay. Moreover, we should try to print wires with the same length in order to avoid synchronization problems (**edge length minimization**).

There exists many algorithms to draw these graphs, but they only guarantee some optimization bounds, **the optimal solution cannot be computed using an algorithm**.

2.3 Thomposon Model

This model was proposed to satisfy all the previous listed constraints, the **layout of the topology G is a plane representation on a bunch of unit distance horizontal and vertical traces** that maps:

- Nodes into the intersections of the traces that have only integer coordinates.
- Edges into disjoint paths that does not violate the constraints.

The Thomposon models is an **orthogonal grid drawing** of the graph $G = (V, E)$. Using this model we can draw graph in which **each node can be up to degree 4**.

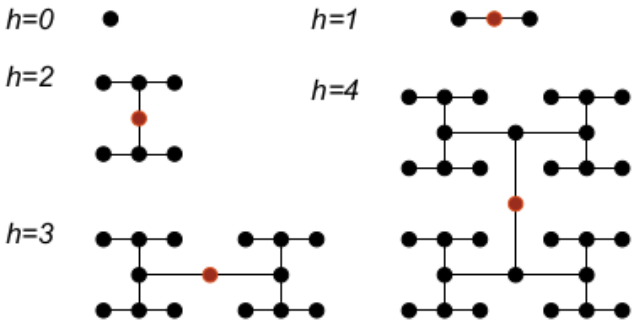
2.3.1 Tree Layout (H-Trees)

Also knows as **complete binary tree layout**, is a grid orthogonal rectilinear (each edge is represent by a single continous segment) plane representation of an n node complete binary tree in area $O(n)$. If the leaves are constrained to lay on the board, the area is $\Omega(n \log n)$. If we have **to represent a non-complete binary tree, we can represent it as complete tree and delete the missing parts, the area in this case is not optimal anymore**.

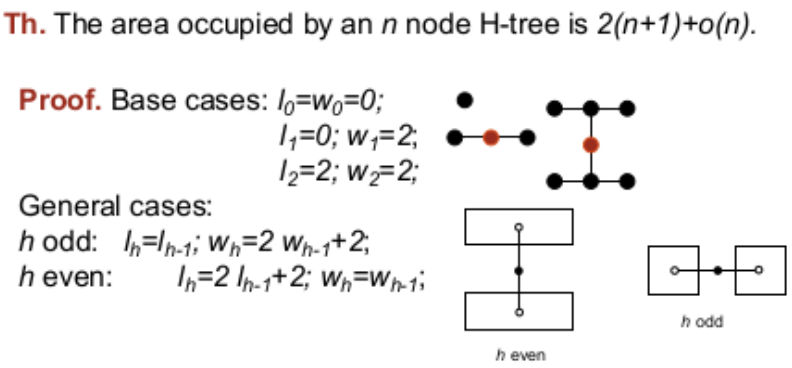
The construction is induction-based with respect to the height coordinate h:

- Base case $h = 0$:
- Inductive case:
  - **If h is even**, the two roots of the two trees are connected vertically.
  - **If h is odd**,the two roots of the two trees are connected horizontally.

The nodes are connected to a new node that is the root of the tree.



The resulting area of a H-tree made up by n nodes is $2(n + 1) + O(n)$. The proof is induction-based:

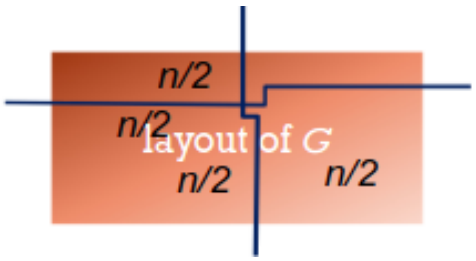


2.4 Complete Graph Layouts

These layouts are very big in size since **each node is directly connected to every node**. However, this ensure that **we do not need any routing algorithm** since each node has a direct connection with any possible destination. In this type of layouts, nodes can have a degree that is greater than the 4-th nodes degree obtained in a Thomposon representation. In particular, since the nodes are always represented through a grid, we cannot represent them anymore using dot. In fact, in this layouts, **a d-degree node is represented as a square $d * d$.**

Bisection Width The bisection width of a network indicates the minimum number of edges to be cut to obtain two equally sized sub-networks. The bisection width of a complete graph is $\frac{n^2}{4} + O(n^2)$. A lower bound on a layout area is the square of its bisection width, thus making the lower bound area equal to $\frac{n^4}{16} + O(n^4)$.

Proof: Let's draw two bisection lines in this way:



If we move the corner of the each bisection line one cell on the grid at time, we will find the step in which the nodes are perfectly splitted in two halfs (with respect to each bisection line). In this way the line is cutting AT LEAST "bisection width"-number of columns horizontally and "bisection

width"-number of rows vertically, so **width and height must be at least "bisection width" large** so the area must be at least $\min(width) * \min(height) = (bisection\_width)^2$.

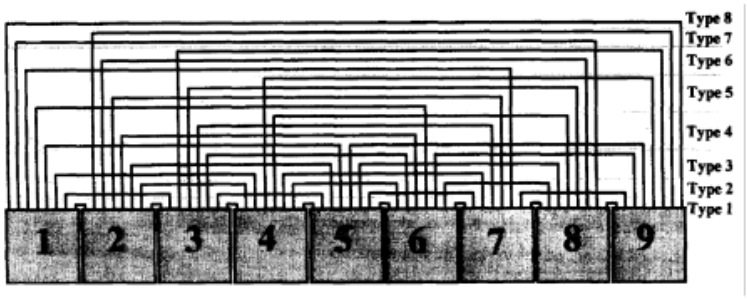
2.4.1 Collinear Layout

It is a complete graph where **all the nodes are placed on the same line, so in order to compute the area, we must find the number of needed tracks**. The edges are drawn by following a simple rule: we add a link of **type-i**, where i is the numerical label that identifies each node, if the nodes' label differ by i. So we will endup into $\frac{n(n-1)}{2}$ nodes, in particular we will have $n-i$ edges of type-i. The number of needed tracks for each type-i is $\min(i, n-1)$, hence the number of total needed tracks is:

substituting $j=n-i$

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

$$= \sum_{i=1}^{n/2} i + \sum_{j=1}^{\frac{n}{2}-1} j = \frac{1}{2}\left[\frac{n}{2}\left(\frac{n}{2}+1\right) + \frac{n}{2}\left(\frac{n}{2}-1\right)\right] = \frac{n^2}{4}$$



The **bisection width of a collinear layout is** $\frac{n^2}{4} + O(n^2)$, and since that the **lower bound of the number of needed track for a collinear layout is the bisection width itself**, the collinear layouts leads to the smallest possible number of tracks. The area of the collinear layout is: $\frac{n^4}{4}$.

2.4.2 Orthogonal Layout

This layout is an area-efficient improvement of the previous seen Collinear Layout. Considering n nodes, the orthogonal layout places all the n nodes in a $m_1 * m_2$ grid where $n = m_1 * m_2$ by

considering m_1 and m_2 in $\theta(\sqrt{n})$. Obviously, not all ns are perfect squares, so can simply take m_1 and m_2 as much closest to $\sqrt{n}$ as possible. The area of these layouts is: $\frac{n^4}{16} + O(n^4)$. Since we said before that a lower bound on the complete graphs layout is given by the square of the bisection width, and since the bisection width of complete graph is $\frac{n^2}{4} + O(n^2)$: **Orthogonal Layouts have the smallest area possible**.

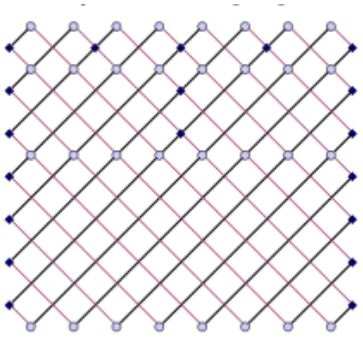
2.5 Butterfly Layouts

2.5.1 Wise Layout

In this type of layout all the wires are equal in the length (that grows exponentially with respect to the size of the layout). This layout is perform on both sides of the board, so it's a 2-sided layout in which:

- First side: lines are drawn from lower-left to upper-right.
- Second side: lines are drawn from lower-right to upper-left.

The path from any input to any output, is equal to the diagonal of the layout itself: $\sqrt{2}(N-1) * \sqrt{2} = 2(N-1)$. In this way there are **no crossing** between edges. The obtained layout seems very good in term of area, however, this is not completely true. The wires are orthogonal between each other but the **layout is rotated by 45° with respect to the grid, hence the area needed for this layout is represented by the square that in which the layout can be circumscribed**: $[2(N-1)]^2$ since the side of the bigger square is equal to the diagonal of the layout (as said $2(N-1)$). Moreover, **the knock-knees are not avoided but arranged using a special device** (that also has an area so the total layout area is enlarged, but still inside the big square).



2.5.2 Even and Even Layout

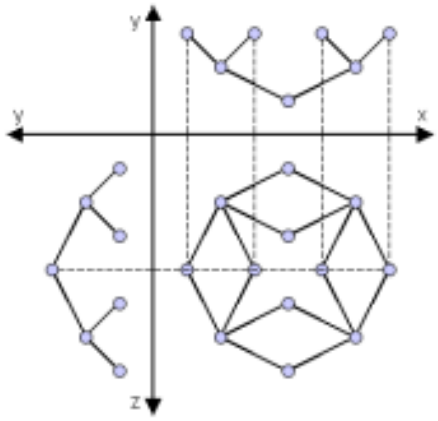
It is based on the **Layered Cross Product** between two layered graphs. Each graph must be composed by $l + 1$ layers that are sets of nodes. Every edge connects nodes only if they lies on adjacent layers. The **layers of the new layout are obtained using the cartesian product of the corresponding layers in the two graphs**. Regarding the edges, the **edge** (v, u) **that connects** (v_1, v_2) **to** (u_1, u_2) **exists in the new graph if** (v_1, u_1) **and** (v_2, u_2) **are edges in the two original graphs**.

Using this layout we can obtain most of the known layouts, such as complete binary trees and butterfly layouts. In particular, by combining two complete binary trees we will obtain a butterfly layout.

Projection Methodology Let G_1 and G_2 be two layered graphs of $l + 1$ layers, G will be the LCP of these two graphs obtained as follows:

1. Let's consider a cube.
2. Project G_1 on the xy face of the cube and G_2 on the yz face.
3. The 3D representation of G is obtained by creating a new node with the coordinates (x_u, i, z_v). Note that the x coordinate is given by the first graph G_1, the z is given by G_2 and the y is the layer in which the two nodes lay on.
4. Project on the xz face the obtained structure to have the 2D representation of G.

We can also **avoid this geometrical construction by using the prolongations on the plane xz of the projections of nodes on the relative axis** (x axis for G_1 and z axis for G_2).

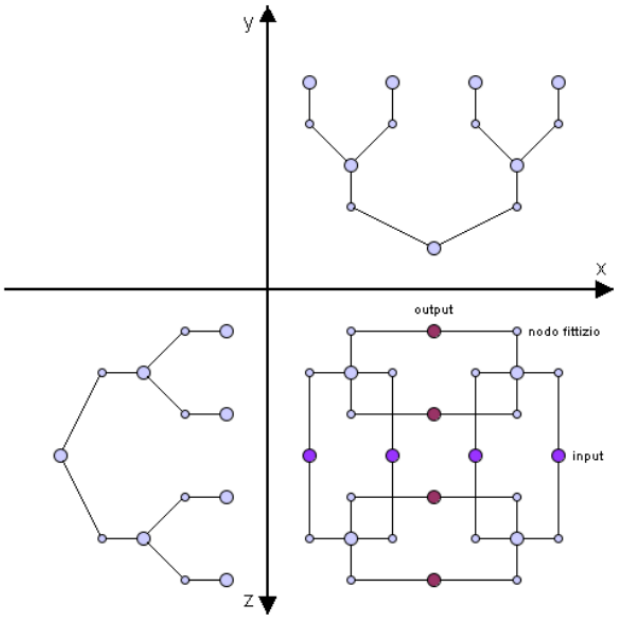


Note that, the PM can produce layouts that are not valid for the Thomposon model constraints:

- **The PM produces edges that are grid lines iff: for every edge e, exactly one of the two factors is diagonal edge**. The **solution** for this problem is **double the nodes in** G_1 by adding fake nodes in a way that all the diagonal edges are in odd layers and the straight edges are in the even, in G_2 (not doubled) will be the opposite situation.
- **The PM generates a layout G in which at most one node is mapped a grid point iff: all the nodes in the two graphs are disjointed**. This means that the projection of all the edges on the relatice axis must be not overlapped (the edges must be **consistent**). The **solution** for this problem is to check that **no nodes shares the same x** (or z in G_2), **except those that are linked through a straight edge**. This is always possible.
- **The PM generates a layout of G in which no grid edge is used twice iff: for every two inconsistent edges of one of the multiplicands, the following conditions are true:**
  - **The two edges are not in the same layer of the multiplicand**
  - **On the two layers in which they appear, there are no straight edges of the other multiplicand that are collinear**.

  This condition is very hard to enforce, hence the PM is used only on two trees, since this condition is always true for them.

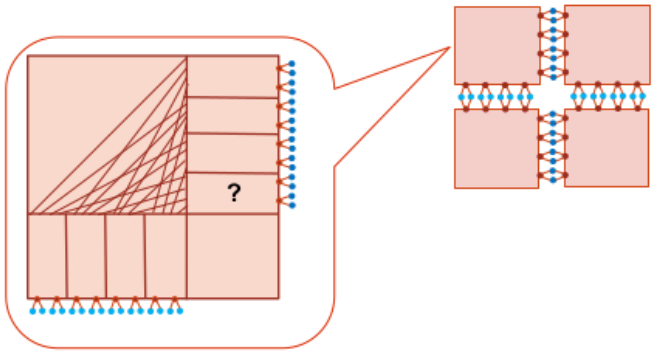
Since LCP is used only for building butterfly networks, we can compare it to the wise layout.



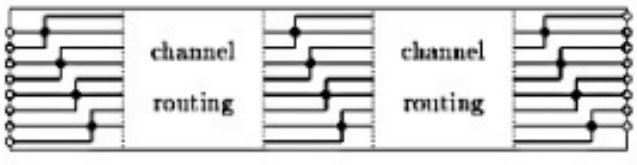
LCP has a bigger area: $\frac{11}{6}N^2 + O(N^2)$ (the shown picture has an area equals to $4N^2 + O(N^2)$, but through some adjustments can be reduced to the one described before) **and the input/output nodes are not on the boundaries** (so they must be accessed from the top). The **main advantages** are that this is a **symmetric construction and all the edges on the same layer have the same length**.

2.5.3 Optimal Area Layout for Butterfly Networks

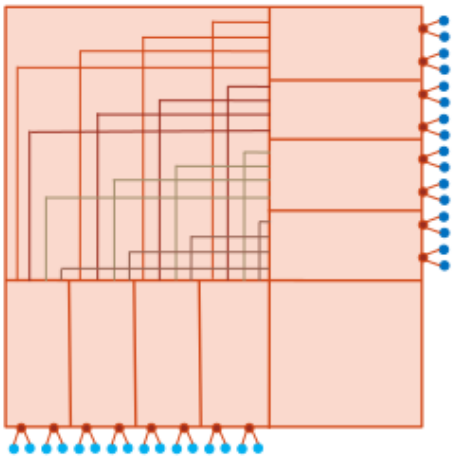
The main idea behind the optimal area is that every n-dimensional butterfly can be built using a pair of $n-1$-dimensional butterflies connected by one node layer and one edge layer. Again those two smaller butterflies can be splitted more and more.



Each reactangle can contains this main layout that is not optimal but it can be used to build the optimal area:



In order to connect the reactangles we can can arrange this connections:



2.6 Hypercube Layout

This layout is mostly used for parallel computation (good fault tolerance, logarithmic diameter, high regularity, etc...). **The n-dimensional hypercube Q_n has $N = 2^n$ nodes and $\frac{n}{2}N$ edges.**

Each node is labeled using a n-bit string, and **two nodes are connected through an edge if their labels differ by exactly one bit**. The hypercube Q_n **can be built by joining two copies** Q_{n-1} **copies**, in particular we add an edge between nodes of the two copies that have the same string and we add 0 in front of the nodes of the first copy and 1 for the other ones.

One of the main property is that Q_n has a **diameter equals to** $\log N$ and the **bisection width is** $\frac{N}{2}$ (so the lower bound of the area is $\frac{N^2}{4}$). Actually the **best area obtained is** $\frac{4}{9}N^2 + O(N^2)$.

To achieve this area, we can recursively build a layout based on the collinear layout starting with $n = 2$ and then we can compute the number of needed tracks as:

- When n is odd: assuming that we have a collinear layout Q_{n-1} that requires $f(n-1)$ tracks, then $f(n) = 2 * f(n-1) + 1$ tracks.
- When n is even: we use 4 copies of Q_{n-2}, the final number of tracks is $f(n) = 4 * f(n-2) + 2$

The total number of needed tracks is $\frac{2}{3}N$. The final achieved area is $\frac{2}{3} * n * N^2 + O(N^2)$, however it can be reduced to $\frac{4}{9}N^2 + O(N^2)$ using the orthogonal layout where we have n_1 copies of smaller collinear layouts and where each layout has n_2 nodes (with $n_1 \approx n_2 \approx n$). Hence, the width (and also the height) of this layout can be calculated using similar calculations as those explained for finding the number of needed traces:

$$\text{width} = \text{height} \approx \frac{2}{3}N + O(N/n_i) \text{ with } n_i = n_1 \vee n_2$$

So the area is width*height $= \approx [\frac{2}{3}N + O(N/n_i)]^2 \approx \frac{4}{9}N^2 + O(N^2)$.

2.7 3D Layouts

The topology is **based on slices**, this allows to **avoid completely wire crossings** (we can simply put those two wires onto two different slices). Moreover, using this layouts we can also **reduce the silicon and the wire lengths used**. A **3D layout of a topology** G **is a ont-to-one function that maps** G **into a 3D grid**. Nodes always are placed according to the grid (and possibly on the external part of the layout to ensure the cooling of them) and wires cannot cross or involve in knock-knees. **We have to minimize the layout volume and the wires length**.

3 Worm Propagation

This problem is related to any networks. A **computer worm is a standalone malware that replicates itself in order to spread to other computers**. Worms differs from viruses since

they do not need to be attached to any external program. Typically, worms tries to replicates itself using:

- Emails: it will insert a copy of itslef into email messages.
- Social engineering techniques: in order to convice people to open attachments containing the worm.
- Bugs of email clients: in order to execute itself even if the message is simply visualized.

They can be classified into **two different categories**:

- **Direct Damages Worms**: resulting from the execution of the worm on the infected machine, such as crypting files or deleting them to extort money or creating backdoors on the infected machine.
- **Indirect Damages Worms**: side effects of the infection, such as bandwidth/computational resources usage.

Let's call T the time needed to transfer an information. **If a worm has infected a set of nodes C such that with a single step all the nodes in the network are infected**, then this is called **first propagation step**. If C is the first propagation step, this means that every other node that hasn't been already infected, has a direct connection to an infected node in C.

3.1 Vertex Cover

Given an undirected graph $G = (V, E)$, the **vertex cover is a subset V' of nodes that contains at least one of the two endpoints of each edge of the graph, obviously we can have more than one vertex cover in a graph**. In the worm problem, **we want to minimize the vertex cover**, since every vertex cover is a perfect start for a worm.

3.2 Minimum Vertex Cover Problem

In general, given $G = (V, E)$ and $V' \subseteq V$, V' is a vertex cover for G iff:

$$\forall a, b \in E \rightarrow a \in V' \vee b \in V'$$

The **minimum vertex cover problem is NP-complete**.

ILP: ILP formulation is an optimization problem that can be translated into mathematical linear formulas. By solving those formulas, we can solve the problem too.

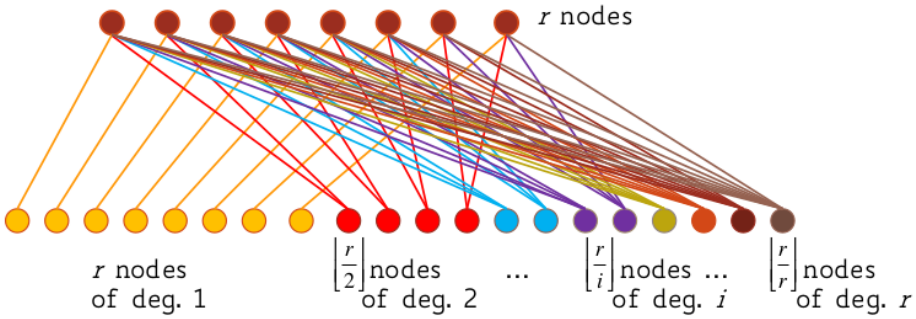
We can define an ILP formulation for VC, for each node $x_i \in G$, then:

$$x_i = \begin{cases} 0, & \text{if } x_i \in V' \\ 1, & \text{otherwise} \end{cases} \tag{1}$$

Obviously, as main constraint of this notation we will have that: $x_i + x_j \geq 1 \forall (i,j) \in E$.

3.2.1 Greedy Approaches

First greedy approach is very simple, **for every edge $(i,j) \in E$, we add i to V' and then delete all the edges in E where i appears**. This solution is far from the optimal one, in fact we can achieve an approximation ratio $\theta(\log r)$ as lower bound (so there are graphs in which this solution is even worse than this). This lower bound is obtained by considering this graph and this situation:



The upper layer is the best solution, the lower layer is the picked one. The algorithm could consider only those node with a lower degree. The approximation ration is given by $\frac{|V'|}{|V^*|}$, where V' is our provided solution and V^* is the optimal one. Since $V' = \theta(r \log r)$, then the approximation ratio is: $\frac{r \log r}{r} = \theta(\log r)$.

The second greedy approach is similar to the first one, but for every edge, we pick the node with the highest degree between the two endpoints. However, even in this case we could end up into the same identical graph obtained with the first approach (simply by starting from the right part of the graph).

3.2.2 First Approximation Approach

This approach is very simple but also counterintuitive, **for every edge $(i, j) \in E$, we add both i and j to V' and then delete all the edges in E where i or j appear**. The approximation ratio for this solution is 2. It is very simple to understand that V' is a vertex cover.

Proof of Approx. Ratio: If we call A the set of edges in E that have been selected during the steps, then trivially $|V'| = 2|A|$. Moreover, since at each step we remove all the edges that were having one of the two nodes in common, then the edges in A are not incident so $|A| \leq |V^*|$. Putting toghether those two statements, we obtain that $|V'| = 2|A| \leq 2|V^*|$. So finally, the approximation ratio for this solution is 2.

3.2.3 Second Approximation Approach

This approach is based on the ILP formulation, in particular by relaxing the ILP (so by removing some constraints). In fact, **we now assume that every node x_i is not a binary integer anymore but a real number in $[0, 1]$, the constraint $\forall (i, j) \in E : x_i + x_j \geq 1$ is kept**. Once said that, the approach is to **loop on the nodes and put every node $x_i \geq \frac{1}{2}$ into V'**. Also in this case the **approximation ratio is 2**. Now is not so clear if V' is or isn't a vertex cover, so we also have to prove it.

Proof that is a Vertex Cover: Due to the constraint $x_i + x_j \geq 1$, at least one node in every edge is $x_i \geq \frac{1}{2}$, so at least one node for every edge is inside V'.

Proof of Approx. Ratio: Let Z^* the sum of all the x_i variables in the optimal solution, then $Z^* \leq |V^*|$. Let's consider $x'_1, ..., x'_n$ the binary variables for the proposed solution V', then $x'_i \leq 2 * x_i$ so $|V'| = x'_1 + ... + x'_n \leq 2(x_1 + ... + x_n) = 2Z^* \leq 2|V^*|$.

3.3 Properties of the Minimum Vertex Cover

3.3.1 Independent Set and Minimum Vertex Cover

An independent set of $G = (V, E)$ is a set of nodes of V, in which there are no two nodes adjacent. **A set of nodes V' is a vertex cover iff its complement $V - V'$ is an independent set**.

Proof: To prove it we have to prove both implication sides:

- V' is VC $\rightarrow V - V'$ is IS: by contradiction, if there exist two adjacent nodes in $V - V'$, then the corresponding edge is not covered by V'.
- $V - V'$ is IS $\rightarrow V'$ is VC: by contradiction, if there exist an edge e that is not covered by any node in V', then the two nodes incident on e are adjacent in $V - V'$.

As consequence of this theorem, the number of nodes of a graph is equal to $\max |IS| + \min |VC|$.

3.3.2 Matching and Minimum Vertex Cover

A matching of $G = (V, E)$ is a subset M of E without common nodes. **If M is the matching of G and V' is a vertex cover for G, then** $|M| \leq |V'|$.

Proof: V' is a vertex cover, so it must cover all the edges in M. On the other side, each edge in M at least one endpoint must be in V', then $|M| \leq |V'|$.

As consequence of this theorem, if $|M| = |V|$ with M matching and V a vertex cover, then M is the maximum matching and V is the minimum vertex cover.

Note that compute the matching takes a polynomial time. Based on the matching we can define a new algorithm to compute the vertex cover, it is very similar to the first approximation algorithm shown before. **Instead of adding both endpoints starting from a random edge, we can take from the edge from the maximum match M.**

The time complexity of this algorithm is $O(m\sqrt{n})$ since the operation is dominated by the time needed to compute the maximum matching M. This algorithm has an approximation ratio 2.

Proof that is a Vertex Cover: V' is a vertex cover since, for any edge e in G then:

- Either e is in M, hence both endpoints are in V'
- Or e is in E/M and at least one of the two endpoints is in V', otherwise it could be added to M (since is a maximum vertex cover).

Proof of Approx. Ratio: By construction $|V'| = 2|M|$ and since each edge $e \in M$ has at least one endpoint in V', then $|M| \leq |V^*|$. By combining these two assumptions, we obtain that $|V'| \leq 2|V^*|$.

3.3.3 Bipartition

If G is bipartite then the VC is solvable in polynomial time. This because for the **König's theorem**, the number of edges in a maximum matching is equals to the number of vertices in a vertex cover.

3.3.4 Related Problem: External Vertex Problem

Considering a graph G in which we place at most one "defender" per node, we have to protect the edges in a way that, if an attacker tries to attack an edge, the defender can cross the edge and defend it. Our aim is to find the set of defenders that can infitely defend every edge, considering that when an edge is under attack, the defender can cross exactly one edge per step.

Let $\alpha(G)$ be the cardinality of a min VC and $\alpha^{\infty}(G)$ the cardinality of a min external CV, trivially $\alpha(G) \leq \alpha^{\infty}(G)$.

Theorem 1: For any cyclic graph with n nodes and $n \geq 3$, then $\alpha(G) = \alpha^{\infty}(G) = \lceil \frac{n}{2} \rceil$.

Theorem 2: For any connected graph then $\alpha(G) \leq \alpha^{\infty}(G) \leq 2\alpha(G)$.

Theorem 3: For any path graph with n nodes and $n \geq 1$, then $\alpha^{\infty}(P_n) = n - 1$.

4 Fixed Wireless Networks

A **fixed wireless network is a collection of wireless devices that form a network using radio or other wireless connections without the need of a central authority**. The main advantages are that we can use this network even in large areas without placing wires to connect devices with themself. **One of the most popular applications is the establishment of fixed cellular telecommunication networks**. The fixed wireless services typically use directional antennas, moreover the devices drain power from public mains.

4.1 Frequency Assignment Problem

The frequency assignment is necessary in many types of wireless networks. Wireless communication between two points is established with the use of a transmitter and a receiver. The transmitter generates electrical oscillations at a radio frequency.

- The receiver must be close enough to the source, in order to be able to detect and correctly receive the message. Moreover, radio signals are subject to variations due to the obstacles that are present in the environment.
- If more messages are sent over same (or very similar) frequencies, they can interfere with each other. We should avoid same frequencies and prefer far enough frequencies. We must pay attention since frequencies are not illimited, so we have to reuse them.
- To transmit in some frequencies we have to pay a certain amount of money, so we also want to reuse them to save money.

In general, the **Frequency Assignment Problems (FAPs), our aim is to set frequencies for the network and we should take them from a set. The constraints are that we should avoid interferences between frequencies** to avoid message losses.

We will have an **interference if both requirements are satisfied**:

- They are **close** to each other in term of **physical space**.
- They are **close** to each other in term of **electromagnetical band**.

4.2 Graph Model

The model that we will use is the **interference graph**, in which we have **one node per station and one edge between two nodes if they can communicate directly with each other** (hence interfere). The **nodes are labeled using integers** that represent different frequency channels. We can distinguish between **two types of collisions**:

- **Direct collision**: when two nodes are at least h distant from each other.
- **Hidden collision**: when two nodes are at least k distant from each other.

In these definitions, h and k are integers that indicate the distance between the nodes.

4.2.1 $L(h,k)$ Labeling Problem

Given a graph $G = (V, E)$, an $L(h,k)$-labeling is a node coloring (coloring means label the nodes) function f such that:

- $\forall u, v \in V | f(u) - f(v)| \geq h$ if $(u, v) \in E$
- $\forall u, v \in V | f(u) - f(v)| \geq k$ if $\exists w \in V$ s.c $(u, w) \in E$ and $(w, v) \in E$

The **first condition** is saying that we need that the **difference between the two channels assigned to two nodes that are directly connected must be at least** h. The **second condition** instead is saying that the **difference between two channels assigned to two nodes that have one common node in the neighborhood must be at least** k.

Our aim with this problem is to **minimize the bandwidth** $\sigma_{h,k}$ and **find the minimum bandwidth** $\lambda_{h,k}$.

Note: Sometimes the second condition is also written in this way:

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

This is not always true, since if $h \leq k$ then the two definitions coincide, otherwise, when $h < k$ only the first formulation allows a triangle graph to be labeled.

The minimum color is 0, so an $L(h, k)$-labeling having span $\sigma_{h,k}$ uses $\sigma_{h,k} + 1$ colors.

The labeling problem is used to model several problems:

- Channel assignment in optical cluster-based networks ($L(0, 1)$ or $L(1, 1)$ if the network has one or more nodes).
- More in general channale assignment problems.
- Integer control code assignment in packet radio networks to avoid hidden collisions $L(0, 1)$.

Note: When we talk about $L(1, 1)$-labeling problem, it is equivalent to a classical vertex coloring in a square graph. Where if G is a graph, the square graph G^2 is the graph having the same G's node set but adding an edge between two nodes if they have a distance=2 in G.

Lemma: $\lambda_{d*h,d*k} = d * \lambda_{h,k}$, to prove this statement, we have to prove both inequalities:

- $\lambda_{d*h,d*k} \geq d * \lambda_{h,k}$: Let f be an $L(d * h, d * k)$ labeling. We can define $f' = \frac{f}{d}$, then f' is a $L(h, k)$ labeling such that $|f'(u) - f'(v)| = |\frac{f(u)}{d} - \frac{f(v)}{d}| \geq h$ (the h value for f is $h * d$ so for f' is $\frac{h * d}{d} = h$), the same is also valid for k. Hence, $\frac{\lambda_{d*h,d*k}}{d} = \sigma_{h,k}(f') \geq \lambda_{h,k}$.
- $\lambda_{d*h,d*k} \leq d * \lambda_{h,k}$: Similarly, let f be an $L(h, k)$-labeling. We can define $f' = f * d$, then f' is a $L(h, k)$ labeling such that $|f'(u) - f'(v)| = |f(u) * d - f(v) * d| \geq h * d$ (the same is also valid for k). Hence, $d * \lambda_{h,k} = \sigma_{h,k}(f') \geq \lambda_{d*h,d*k}$.

This lemma states that we can restrict both h and k to be mutually prime.

What if $f' = \frac{f}{d}$ does not produce integer values?

Lemma: Let $x, y \geq 0$, $d \geq 0$ and $k \in \mathbb{Z}^+$. Then, if $|x - y| \geq k * d$ then $|x' - y'| \geq k * d$, where $x' = \lfloor \frac{x}{d} \rfloor * d$ and $y' = \lfloor \frac{y}{d} \rfloor * d$.

4.3 Special $L(h, k)$-labeling Problem

- $L(h, 0)$ (for any h) is not very studied since coincides with the classica vertex coloring problem.
- $L(h, k)$ with $h = k$ is very studied since is the vertex coloring of the square of a graph.
- $L(2k, k)$ is the most studied one.

The $L(2, 1)$-labeling problem on 2-diameter (the longest distance between two nodes, in this case every node is either at distance 1 or distance 2) **graphs is NP-complete**. Note that, **since the graph is 2-diameter graph, we cannot reuse the same label twice**.

How NP-completeness proof is done: An NP-completeness proof is done by following these two steps:

1. We have to **verify that the problem P is in NP**.
2. **We prove the NP-completeness of a problem P, using a NP problem Q** (that is known to be NP). We have the instance, that is the input for P. By contradiction, suppose that we can solve in polynomial time the problem P. If we can translate in polynomial time the instance of a problem Q into the instance of problem P, and we have an algorithm to transform the solution of the problem P into the solution of the problem Q. Then we have a polynomial algorithm to solve Q. But since Q is NP-complete, this is not possible.

4.3.1 Proof of $L(2, 1)$-labeling NP-completeness:

To prove that the $L(2, 1)$-labeling for 2 diameter graphs is NP-complete, we **must prove the NP-completeness for both DL and IDL problems before**:

- **DL Problem**: **The instance is a graph $G = (V, E)$ that is a 2-diameter graph. The goal is to ensure that $\lambda_{2,1}(G) \leq |V|$.** The general decision $L(2, 1)$-labeling problem is about proving that $\lambda_{h,k} \leq d$ with $d \in \mathbb{N}^+$. In the DL problem we force $d = |V|$. So $\lambda_{h,k} \leq |V|$.
- **IDL Problem**: **The instance is a graph $G = (V, E)$, the goal is to find an injective function f such that $|f(x) - f(y)| \geq 2$ for every edge $e \in E$ and where the function's codomain is** $0, ..., |V| - 1$. So f is a function that associates to every node a unique color i by matching the constraing shown above.

We will firstly prove the IDL NP-completeness, then use the IDL to prove the DL NP-completeness.

IDL NP-completeness proof: Solve IDL means find an Hamiltonian path in G^c (the complement graph is a graph in which we have all the edges that are not present in G). **Since** f **is injective, we can also define** f^{-1}. **Now we can give an order to nodes in this way**:

$$v_i = f^{-1}(i), 0 \leq i \leq |V| - 1, \text{ with } v_i \text{ that is the } i\text{-th node.}$$

Observe that, **since** v_1 **and** v_{i+1} **cannot be adjacent in** G (since f^{-1} maps the color into the unique node, so if x and y are two node such that $f(v_i) = i$ and $f(v_{i+1}) = i + 1$, then $|f(x) - f(y)| = |i - (i+1)| < 2$, hence the constraint is not satisfied), **they must be adjacent in** G^c **so** $v_0, v_1, ..., v_{|V|-1}$ **is a Hamiltonian path**. Trivially, **even the reverse holds**, so starting from an Hamiltonian path we can obtain using the function f^{-1} the color of the nodes in G. **This is why the two problems are equivalent**. Since **finding a Hamiltonian path is well known to be NP-complete, even IDL is NP-complete**.

DL NP-completeness proof: First of all, we need to prove that DL is NP, so prove in polynomial time that the provided solution is correct. ??. Now we can define the transformation **from IDL instance to DL instance**. Given an IDL instance G we can build a new DL instance G':

- $V' = V \cup x$, where x is a dummy node.
- $E' = E \cup (x, a)$ for each $a \in V$.

Hence, G' must be a 2-diameter graph since every node has a direct connection with x, so starting from a node v we can reach every other node w by traversing x.

Now we have to define the transformation **from a DL solution to an IDL solution and viceversa**, this consists in proving that there exists an injective function f such that:

$$|f(x) - f(y)| \geq 2 \text{ for every edge } e \in E \iff \lambda_{2,1}(G) \leq |V'|$$

1. $\implies$: Since f is defined as above, we can simply define a new function g such that:

   $$f(v) = g(v)\ \forall v \in V \text{ and } g(x) = |V| + 1 = |V'| \text{ where } x \text{ is the dummy node.}$$

   Clearly, g is a $L(2,1)$-labeling for G' and $\lambda_{2,1}(G') \leq \max g(v)\ \forall v \in V' \leq |V'|$.

2. $\impliedby$: Suppose that $\lambda_{2,1}(G') \leq |V'|$, so there exists a feasible $L(2,1)$-labeling g such that:

   $$\max g(v)\ \forall v \in V' \leq |V'| = |V| + 1$$

Observe that since G' is a 2-diameter graph, then $g(a) \neq g(b)\ \forall a \neq b$. In fact, suppose that $g(x) \neq |V|+1$ and $g(x) \neq 0$, by the property of $L(2,1)$-labeling there is no $v \in V$ such that $g(v) = g(x)-1$ or $g(v) = g(x)+1$. So we need $|V|+3$ labels for V', hence $\lambda_{2,1}(G') \geq |V'|+1$ and this is a contradiction. $g(x)$ is either 0 or $|V|+1$:

- If $g(x) = |V|+1 \rightarrow f(v) = g(v)$
- If $g(x) = 0 \rightarrow f(v) = g(v) - 2$

In any case, there exists f injective such that its codomain is $0, ..., |V|-1$.

The DL NP-completeness is proved.

4.3.2 Lower Bounds & Upper Bounds

In general, we can define lower bounds in term of number of labels needed. **The most general lower bound for a graph G is**:

$$\lambda_{h,k} \geq (\Delta - 1)k + h \text{ with } h \geq k \text{ and } \Delta = \text{maximum degree of a node in } G.$$

Indeed, the **lower bounds for $L(2,1)$-labeling is** $\lambda_{h,k} \geq (\Delta-1)1+2 = \Delta+1$. There exists a class of graphs called **incidence graph** in which the lower bound can be increased to $\lambda_{h,k} \geq \Delta^2 - \Delta$.

To provide an upper bound, is sufficient to give an algorithm that solves the problem. We have a **greedy algorithm for $L(2,1)$-labeling** that given a graph G, it gives the labels starting from the node v_0 by assigning the lowest color such that there are no conflicts with nodes at distance 1 and 2. The **upper bound provided by this algorithm is** $\lambda_{2,1}(G) \leq \Delta^2 + 2\Delta$. To prove it, Consider a vertex v in G with the maximum degree Δ. By definition, v has Δ neighbors, to ensure that v and its neighbors can be distinguished from each other, we need Δ colors. Extending this argument, for each of the Δ neighbors of v, which have $\Delta - 1$ neighbors (excluding v itself), $\Delta - 1$ additional distinct colors are required to distinguish them from one another and from v. The process is iterated for vertices at increasing distances from v, requiring fewer distinct colors as we move further from v. The total number of distinct colors needed to label the entire graph G is the sum $\Delta + \Delta(\Delta-1) + \Delta(\Delta-2) + ... + 2$ collapsed into $\lambda_{2,1}(G) \leq \Delta^2 + 2\Delta$.

During years, other algorithms have been proposed reaching an upper bound equals to $\lambda_{2,1}(G) \leq \Delta^2 + \Delta - 2$.

4.3.3 Exact Results

Now we will provide some exact results for the $L(2,1)$-labeling problem:

- **Cliques** K_n (Connected Graphs): $\lambda_{2,1}(K_n) = 2(n-1)$. This is simple to understand, since the nodes are pairwise adjacent (we can display them in a circle with all the other links in the middle, we have to label them in this way).

- **Stars** K_1, t: $\lambda_{2,1}(K_{1,t}) = t+1$. To label this graph we have to start form the center and then assign consequently all the other nodes.
- **Trees** T_n: $\lambda_{2,1}(T_n) = \Delta + 1$ or $\lambda_{2,1}(T_n) = \Delta + 2$, proof:
  - $\lambda_{2,1}(T_n) \geq \Delta + 1$: is true since T_n contains $K_{1,\Delta}$.
  - $\lambda_{2,1}(T_n) \leq \Delta + 2$: let's start labeling v_1 with 0. Assume that we have already labeled all the nodes from v_1 to v_i and we are going to label v_{i+1}. Now, if we consider as v_j the v_{i+1}'s parent (with $j < i+1$ since parents have lower indexes), then at most 3 colors are forbidden to v_{i+1} due to v_j and at most $\Delta - 1$ colors are forbidden due to the v_j's adjacent nodes. Therefore, if we have at least $3 + (\Delta - 1) + 1 = \Delta + 2$ colors, we can always determine a color for every node.
- **Paths** P_n:
  - $\lambda_{2,1}(P_2)$ and $\lambda_{2,1}(P_3)$: the same as for the starts (so 2 and 3).
  - $\lambda_{2,1}(P_4) = 3$.
  - $\lambda_{2,1}(P_n) = 4$ for $n \geq 5$.
- **Cycles** C_n: $\lambda_{2,1}(C_n) = 4$, we have 3 main cases
  - $n = 0 \pmod 3$
  - $n = 1 \pmod 3$
  - $n = 2 \pmod 3$

  In all of these cases we can simply define a labeling.
- **Grids**: $\lambda_{2,1}(\Delta) = \Delta + 2$.

4.4 Variation of the $L(2,1)$ Problem

For these variations we will give only the main ideas. The aim of these problem is always to minimize λ (so the total number of colors).

4.4.1 Oriented $L(2,1)$-labeling

In this problem, the graph G is oriented, the definition remains still the same but we must pay attention to the direction of the links. However, the results are very different from the classical problem. In fact, for **direct trees** $\lambda_{2,1} \leq 4$ and the proof is very simple: we can simply assign 0 to the root, 2 to its child with a direct link and 4 to their sons. Using this recursive structure we can label all the nodes.

4.4.2 $L(h_1, ..., h_k)$-labeling Problem

Instead having constraints only for distance 1 and 2 nodes, **we impose more constraints even on nodes a thigher distances**:

$$|f(x) - f(y)| \geq h_i \ \forall 1 \leq i \leq k$$

Also **this problem is NP-hard.**

4.4.3 Backbone Coloring

This problem has been introduced to be more near with real cases. If the topology have a backbone where the transmitting power is higher then: **a backbone coloring of a graph G with respect to the backbone H is a function assigning integer values to nodes such that**:

$$|f(x) - f(y)| \geq 2 \text{ if } (x, y) \text{ is an edge in } H$$
$$|f(x) - f(y)| \geq 1 \text{ if } (x, y) \text{ is an edge in } G - H$$

4.4.4 n-Multiple $L(h, k)$-labeling Problem

Each node can handle more than one channel, so we must assign a set of colors to each station. Given this new definition of distance between two nodes:

$$dist(I, J) = \min |i - j| : i \in I \wedge j \in J$$

So this problem is like the classical $L(h, k)$-labeling problem (even the two properties) but with a new concept of distance.

4.4.5 4 Color Problem

Given a map (e.g. city maps) **that can be considered as a planar graph G, we can build G^* called dual graph**:

- Put a node of G^* in each region of G.
- Put one more node that represents the external region of the graph.
- Connect two nodes in G^* iff their corresponding regions are adjacent (so share an edge in G).

We can always assign a color to a region in a way that the whole map can be colored using 4 colors without that two adjacent regions have the same color.

There are also variations of this problem, 4 is the minimum number to color every graph, but we can use a **polynomial algorithm to decide whether a graph can be colored using only 2 colors. The algorithm is the following**:

1. Assign a color to a region.
2. Assign the other color to the neighboring regions.
3. Loop on every region until all the regions have been colored or a conflict has been detected.

If a graph is bipartite can be 2-colored.

Instead, **understand if a graph is 3-colorable is NP-hard.** We could use a bruteforcing technique to assign all the possible combinations to the regions, however this is not a feasible solution for high number of regions. We could also try to reduce the complexity of the graph by deleting all the regions that have exactly 2 neighbors it can be deleted and then colored with the third color, however the complexity is still the same.

For 5-coloring graphs (nowadays not usefull since we know that the minimum number is 4) is quite easy to do.

5 Minimum Energy Broadcast Problem

This problem still regards fixed wireless network. When we have a large network and we want to update nodes or perform any large-scale operation, we wish to do it automatically by sending only one time the instruction. To **broadcast the information we have 3 requirements**:

- **Reliability**: all the nodes must be covered in the broadcasting.
- **Energy Efficiency**: the process must be done to use the lower amount of energy as much as possible.
- **Scalability**: the protocol must be scalable to accomodate the eventual increasing number of nodes in larger networks.

We have S stations placed in an Euclidian plane (partially realistic) and those stations have **omnidirectional antennas** (can communicate in every direction). Obviously, two stations can either communicate directly (direct link) or using a multi-hop path traversing multiple nodes. We can introduce the concept of **transmission range that is assigned to each station using a range assignment function $r : S \to R$ such that**:

$(x, y) \in E$ iff $dist(x, y) \leq r(x)$, where the distance is the Euclidian distance.

So we **allow direct communications if the node y is in the range of communication $r(x)$ of the node x. Each node can modulate its transmission radius based on the power**:

$$\frac{P_s}{dist(s,t)^\alpha} \geq 1 \text{ where:}$$

- $2 \leq \alpha \leq 4$ and is the distance-power gradient that varies based on the transmission medium (air, water, ...).
- t is the node that we want to reach from the node s.
- P_s is the power assigned to the node s.

Hence, **if we want to reach t we have to use an amount of energy P_s that is proportional with respect to the distance of the two nodes**.

Our goal is to provide specific connectivity properties by adapting the transmission range in order to **reduce the overall consumed energy**. The transmission graph could require to:

- Be strongly connected: this is an NP-hard problem with a 2-approx. algorithm for 2-diameter graphs.
- Have diameter at most h: there are no approximated results yet.
- Include a spanning tree rooted at a given source node s: this is called **Broadcast Problem**.

5.1 Minimum Broadcast Problem

This is a Broadcast Problem in which we want **find the broadcast of minimum overall consumed energy**. This problem **is a NP-hard problem and cannot be approximated within any constant factor**. To prove it, we must define the Minimum Set Cover problem.

Minimum Set Cover Problem: Given a collection C of subsets of a finite universe set U, find a subset C' of C with minimum cardinality such that each element in U belongs to at least one element of C'. Given a set of subsets $C = \{C_1, ..., C_n\}$ of the universe U such that:

$$\cup_{i=1...m} C_i = U$$

We have to find the smallest subset of C, $C' = \{1, ..., n\}$ such that:

$$\cup_{i \in C'} C_i = U$$

It is well known that this problem cannot be approximated neither using a constant factor nor constant factor * log n (with $n = |U|$).

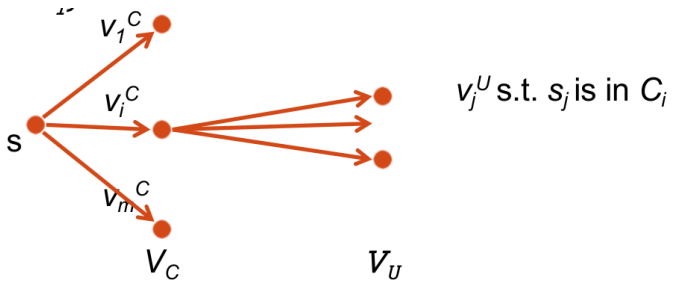
5.1.1 Proof of Inapproximability of MinBroadcast:

The **proof consists in building an instance of MinBroadcast starting from MinSetCover, such that there exists a solution for MinSetCover of cardinality k iff there exists a solution for MinBroadcast of cardinality $k+1$. Therefore, if MinBroadcast is approximable within a constant factor, even MinSetCover can be approximated** (will be proved as contradiction).

1. **Prove INST(MinSetCover) $\rightarrow$ INST(MinBroadcast)**: Let's consider $x = (U, C)$ the instance of MinSetCover with $U = \{s_1, ..., s_n\}$ and $C = \{C_1, ..., C_m\}$. We can build $y = (G, w, s)$ as the instance of MinBroadcast in this way:

   - Nodes of $G : \{s\} \cup \{V_C\} \cup \{V_U\}$ with:
     - s the root of the tree.
     - V_C the set of nodes representing the subsets in C, we will have one node v_i^C for every subset $C_i \in C$.
     - V_U the set of nodes representing the elements in U, we will have one node v_i^U for every element $s_i \in U$.
   - Edges of $G : \{(s, v_i^C)\ 1 \leq i \leq m\} \cup \{(v_i^C, v_j^U)\ 1 \leq i \leq m \text{ s.t. } s_j \in C_j\}$

   Finally, we define the weight of any edge e as $w(e) = 1$.



2. **Prove SOL(MinSetCover) $\implies$ SOL(MinBroadcast)**: Let C' be a solution for x, a solution for y assignes 1 to s and to all the other nodes of V_C in C'. The resulting transmission graph contains a spanning tree rooted at s since each element in U is contained in at least one element of C'. The cost of this solution is $C' + 1$.

3. **Prove SOL(MinBroadcast) $\implies$ SOL(MinSetCover)**: If we assume that r is a solution for y (where $r(v)$ is either 0 or 1) and $r(v) = 0$ if $v \in V_S$, then we can derive the solution C' for x by selecting all subsets C_i s.t. $r(v_i^C) = 1$. It holds that $|C'| = cost(r) - 1$.

5.2 Euclidian MinBroadcast

It is a variation of the classical MinBroadcast in which the range assignment r can be represented by the family $D = \{D_1, ..., D_l\}$ of disks with the overall energy defined as:

$$cost(D) = \sum_{i=1}^{l} r_i^{\alpha}, \text{ where } r_i \text{ is the radius of } D_i.$$

Consider the weighted graph G^{α}, where the weight of each link $(u, v) = dist(u, v)^{\alpha}$. Since the broadcast problem is related to the minimum spanning tree on G^{α}, the connections used to perform a broadcast from s cannot generate a cycle because no node has to be informed twice. Therefore, the tree is the structure the minimize the overall energy (the longer is the arch, the higher is the energy consumption). However, **the MinBroadcast is not equivalent to the MinSpanningTree since the energy used by each node is not the sum of all the outgoing links, but is the maximum required energy in order to reach the farest neighbor**:

$$\text{energy}_u = \max_{(u,v)\in T}\{dist(u, v)^{\alpha}\}$$

Obviously, the leaves do not require any energy consumption. **Nothing has been proved yet about the hardness of this problem**. However, in this case we can provide an **approximation algorithm based on the MST** (Minimum Spanning Tree):

- Compute the MST of the graph.
- Assign a direction to the edges from s to the leaves.
- Assign to each node i a radius equal to the length of the longest arch outgoing from i.

This algorithm has an **approx. ration equal to 6**.

5.3 Recup of MST (Minimum Spanning Tree)

5.3.1 Properties of a MST

- Obs. 1: If the weights are positive, then a MST is a minimum-cost subgraph connecting all nodes.
- Obs. 2: There may be several MST with the same weight. In particular, if the edge weights are all the same, then every spanning tree is a MST.
- Obs. 3: If an edge has a distinct weight, then there is a unique MST. The proof is simple: assume by contradiction that we have two MST with the same total weight T and T'. Let e_1 be an edge in $T - T'$ (in T but not in T'). Since T' is an MST, $\{e_1\} \cup T'$ has a cycle C. Let e_2 be the edge that lies on C and belongs to T', then if $w(e_1) < w(e_2)$ then e_1 should be picked in T' instead of picking e_2, if $w(e_1) > w(e_2)$ then e_2 should be picked in T instead of picking e_1. So the $w(e_1) = w(e_2)$, but this is not possible by hypothesis, hence is a contradiction.

- Obs. 4: For any cycle C in the graph, if the weight of and edge e is higher than all the other weights in C, e cannot be part of an MST. To prove it: Assume the contrary, so that e belongs to T_1 that is a MST. If we delete e from T_1, we obtain two subtrees that can be reconnected using another edge f that is in C and that has a lower weight.
- Obs. 5: If in a graph there exists a unique edge e with the minimum weight, then this edge is included in any MST. To prove it: If e was not included in the MST, removing any of the edges that forms a cycle in T after adding e, would produce a spanning tree of lower weight.
- Obs. 6: For any cut C in the graph, if there exists a unique edge e of C with minimum weight in C, then this edge is included in any MST. To prove it: If e was not included in the MST, adding e to the MST produces a cycle, then removing any of the edges that forms a cycle in T after adding e, would produce a spanning tree of lower weight.

5.3.2 Algorithms

The following algorithms are all greedy algorithms based on the same structure, given a set of arcs A containing a MST arcs, e is a **safe arc with respect to A if $A \cup \{e\}$ contains only MST arcs too**.

This is the basis for all the three algorithms:

```
A=empty set
while A is not a MST
  find a safe arc e w.r.t. A
  A=A ∪ {e}
```

Where A **is acyclic and the graph $G = (V, A)$ is a forest in wich each connected component is either a node or a tree**. Each safe arc connects different connected components of G_A. The while loop is run $n - 1$ times.

Kruskal Algorithm: In this algorithm, **we consider that the set of arcs in G_A is sorted with respect to the weights**. At each step of the while loop, **we choose the safe arc with the lowest weight**. The **time complexity is** $O(m * \log n)$ using the Union-Find as data structure.

Prim Algorithm: In this algorithm, **we assume to start from all single nodes and from a single connected component**. At each step we **take the safe arc with the minimum weight such that it connects the main component to an isolated node**. The isolated nodes are stored using a priority queue where the key(v) = min weight of an arc connecting the node to the main component. Based on the implementation of the queue the **time complexity can be either** $O(m \log n)$ **using a heap or** $O(m + n \log n)$ **using the Fibonacci heap**.

Boruvka Algorithm: For this algorithm **we start from the hypothesis that each arc has a distinct weight** (not unrealistic since typically the weights represent distances). At each step we s**elect for each connected component, the safe arc with the lowest weight that connects that component to another one** (at each step the same arc can be selected twice by two differenct components). The **time complexity is** $O(m \log n)$. Note that thanks to the hypothesis, it is not possible to introduce cycles.

There exists an algorithm designed by Willar, which assumes that all the edges are sorted based on their weights that have a linear time complexity. However, it is not used since the asymptotic notation hides a huge constant.

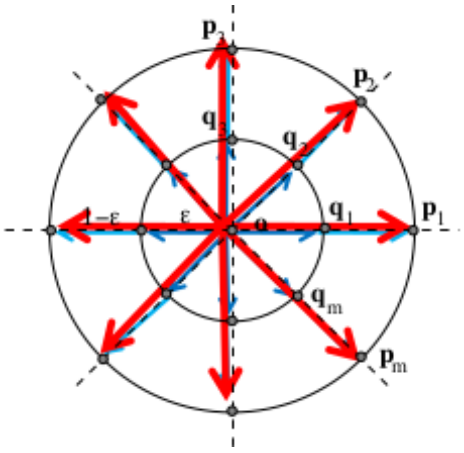
Once that we have built our MST, **we can use some algorithms that can adjust with an average of** $O(\log n)$ **when the graph is sligthly modified**. However, when those changes are too important we have to recompute the MST from scratch.

The **time needed to verify that the provided solution is an MST is** $O(n + m)$.

5.4 Heuristics for Broadcast Problem

We will present three heuristics (therefore we have no bounds) to perform a broadcast operation on a network trying to minimize the overall spent energy:

- **SPT (Spanning Path Tree)**: A SPT is the set of shortest path that connects one node to all the other nodes. SPT and MST can be different, we could have many SPT with different weights but all correct, since we are minimizing the single paths not the overall cost (like for MST). **To build a SPT we run Dijkstra algorithm to get the minimum path three, then it directs the edges from the root to the leaves**. This is an example in which this algorithm does not perform well:

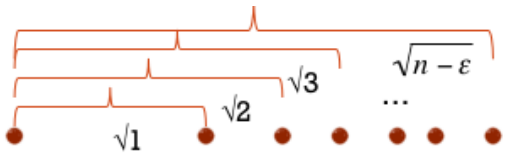


Where half of the nodes are in the inner circle at distance ε from the center and the other nodes are at distance $1-\varepsilon$ from the center. In this case the optimal solution is to set the radius of the central node to 1, hence the energy used is $1^\alpha = 1$. This algorithm chooses to set the radius of the central node to ε and the radius of the middle nodes to $1-\varepsilon$, therefore the final result is $\varepsilon^2 + \frac{n}{2}(1-\varepsilon)^2$ (we set $\alpha = 2$ for the proof). For $\varepsilon \to 0$ the solution picked is far from the optimal one.

- **BAIP (Broadcast Average Incremental Power)**: It is a modification of the Dijkstra algorithm designed to solve the SPT problems, in this case **at each step we check if**:

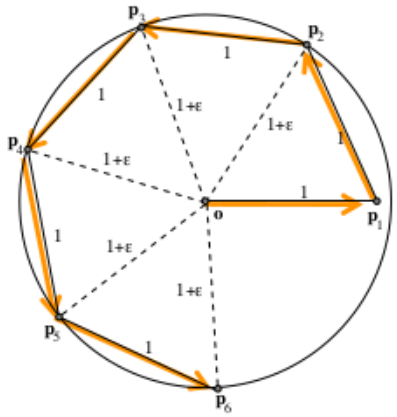
$$\frac{\text{energy increasing}}{\text{number of added nods}} < 1$$

If it is true, we can add the nodes since this operationc can be considered as profitable. Even for this problem we have a case in which the picked solution is far from the optimal one:



The algorithm in this case will consider as solution to set the power of the first node on the left equal to $[\sqrt{n-\varepsilon}]^2 = n - \varepsilon$ (with $\alpha = 2$). However the optimal solution is to set the power of every node to the minimum required to reach the next node on the right.

- **MST (Minimum Spannin Tree)**: **It runs Prim algorithm to get a MST, then it directs the edges of the tree from the root to the leaves**. In this case this **was proved to be an approximation algorithm with an approx. ratio equal to 6**. The worst case scenario is the following:



In this case the selected solution is this one with a total cost of 6, but the best one was to set the radius of the central node to $1 + \varepsilon$. This approx. ratio coincides with the 2-dimensional kissing number, that is the maximum number of 2-dimensional spheres (hence circles) of raidus r, that can touch simultaneously the same 2-dimensional sphere of the same radius r.

These algorithms are not perfect, but they are feasible solution in the real world since the worst case scenarios are very complicated and unprobable networks. However, we have to consider that in the real world scenarios, the ranges are not circles anymore, they are spheres. Moreover, is also unrealistic that those transmissions are propagated in an uniform spherical way.

6 Data Mule Scheduling in Sensor Networks

A **sensor is a device that detects and responds to some type of inputs from the environment**. The input can be light, heat, motion, etc... The output is typically a signal that is converted into a digital output.

Fixed Sensor Networks are wireless networks composed by low-cost devices (sensors).

6.1 The Problem

Even in this case the **energy is a primary issue** since those networks can be deployed over areas where the **power supply is not guaranteed, therefore they rely on batteries**. We should try to optimize the energy efficiency of the protocol to achieve a much longer network life-time. Sensors cannot move (typically) and rely on wireless technology to transfer information. There are some **different approaches that can be used**:

- **Multi-hop communication to the base station**: This is **not always a good solution since the connection in these networks could be instable**. Moreover, if the **deployment is very sparse, the energy needed for a single hop could be too high**. On the other hand, if the **deployment is very dense, nodes that are close to the base station have to send a lot of data**, hence the consumed energy even in this case could be too high.

- **Exploit mobility**: A **data mule is a mobile node that has wireless communication capabilities and a sufficient amount of storage to store the data retrieved by the other nodes**. Data mules have **less problems from an energy point of view, since we could think to recharge them every time that they upload the information to the base station**. In this way the sensors can save a lot of energy due to the fact that the mule can get closer to those, hence the radius needed to transfer the information can be reduced. We can save some additional energy even from the fact that sensors can focus only on sensing the environment without caring of the routing, they can simply send the information to the mule.

6.2 Data Mule Scheduling Problem

We must care about the **scheduling of the mule**, in fact, each mule has to compute the route to get **closer enough to the sensors in order to gather data but trying to minimize the overall time needed to retrieve data from all the sensors** in the network. This problem is much **more complex than a simple scheduling problem, since we can control both speed and path of the mule**. In this way we can modulate the speed according to the path to reduce the velocity as much as needed in order to allow sensors to send the data to the mule without increasing too much the radius.

We can define two different constraints that must be considered:

- **Location Constraints**: The mule must decide how much it has to get closer to a node in order to save time and energy.

- **Time Constraints**: The mule has to modulate the speed in order to allow the full data transmission without stopping itself.

Therefore, the **main issues** for the scheduling problem are:

- **Path Selection**: Determine the trajectory of the mule in order to visit at least once every sensor.

- **Speed Control**: Modulate the speed during the chosen path to accomodate the data transmission (more technical than algorithmic problem).

- **Job Scheduling**: At each step the mule can be closer to more than one node, so it has to decide from which node it has to collect data first. This can be reduced to a simple job scheduling problem.

We will focus on the first sub-problem (path selection).

6.3 Path Selection

We want to schedule the visits of the mule in order that no sensot-buffer is filled at any time. This problem is called **Mobile Element Scheduling (MES)** and is very similar to the **Traveling Salesman Problem (TSP)**.

6.3.1 TSP Definition

Given a set of cities, a salesman has to **visit each one of the cities starting from a certain one and returning to the same city**. The challenge of the problem is that the traveling salesman wants to **minimize the total length of the trip**.

6.3.2 MES vs. TPS

These two problems are different, in fact in **TSP we want to minimize the cost tour that visits each city exactly once**. In **MES we could have to visit the same node more than once**, since that node could produce data more frequently than the others. Therefore, **in MES the deadlines** (time needed for a sensor-buffer to be filled) **are dynamically updated**. Every time that the mule visits a node, the deadline for that sensor is updated based on the buffer size. However, we can **start by solving TSP to find a feasible solution for MES**.

6.3.3 TSP NP-Completeness

We can use the HC problem (Hamiltonian Cycle), so finding a cycle in the graph such that it passes from every node exactly once, to prove that even TSP is NP-complete. Let's **define TSP in a formal way: Let** $K_n = (V, E)$ **be a complete graph with non-negative edges.** K_n **trivially contains always a HC, therefore our aim is to find a HC such that its cost does not exceed** t (non-negative value). The proof for NP-completeness follows the same idea of the $L(2, 1)$ NP-completeness proof:

- **TSP belongs to NP**: check that the tour contains each node exactly once and that the sum of the costs of all the edges in the tour is bounded by t.

- **INST(HC) → INST(TSP)**: Assume to have $G = (V, E)$ that is an instance for HC. We can build $K_n = (V, E')$ in a way that K_n is a complete weighted graph. The vertices-set is the same, we only have to add the needed edges to make K_n completely connected. They weights are added by following these rules:
  - $w(i, j) = 1$ if $(i, j) \in E$
  - $w(i, j) = 2$ if $(i, j) \in E'/E$
- **SOL(HC) → SOL(TSP)**: Assume now that a cycle C is a solution for HC in G. All the edges in C have weight 1 in K_n since they are all in G.
- **SOL(TSP) → SOL(HC)**: Suppose that we found a tour TS for K_n with cost n. Then, all the used edges must belong to G, hence the tour is a Hamiltonian cycle for G.

6.3.4 TSP ILP-Formulation

TSP can be formulated as an ILP. Assume that the **tour is oriented, then we can assign boolean variables $x_{i,j} = 1$ iff the tour traverses the edge (i, j), $x_{i,j} = 0$ otherwise. The goal is to minimize the tour cost**, therefore:

$$\min \sum_{i,j=1}^{n} w(i, j) * x_{i,j}$$

We impose a **first constraint**, **for every node i there can be exactly one edge (i, j) such that $(i, j) = 1$** (only one outgoing edge from i can be picked during the tour, on the other hand only one edge incoming to i can be picked too). This is not enough to ensure that we have exactly one cycle in the tour, therefore we have to impose a **second constraint**:

$$\sum_{i,j \in S} x_{i.j} < |S|, \ \forall S \text{ subset of } \ V$$

However, **due to the constraints that are exponential in the number of checks needed, this formulation cannot be used even for small instances**.

6.3.5 Inapproximability of TSP

TSP cannot be approximated within a constant factor. **By contradiction, if we assume that TSP is approximable in any approximation $r > 1$, then there exists a polynomial algorithm that is exact (best solution) for HC**.

Proof of Inapproximability: Let $G = (V, E)$ be an instance of HC and K_n a complete graph with $|V| = n$. We can define the edge weights on K_n as follows:

- $w(i, j) = 1$ if $(i, j) \in E$
- $w(i, j) = 2 + (r - 1)n$ otherwise

Therefore, a tour in K_n with a cost n exists iff G has a HC. In this case, we can assume that there exists an r-approximation algorithm A for TSP such that if the cost of the optimal solution is n, A finds a solution H with $\text{cost}(H) \leq r * n$. If H contains an edge that is not in E, then:

$$\text{cost}(H) \geq (n - 1) + 2 + (r - 1)n = r * n + 1$$

So, a solution for TSP with $\text{cost}(H) \leq r * n$ exists iff an Hamiltonian cycle is in G and hence, HC can be solved in polynomial time, contradiction!

6.4 Approximation Algorithms for TSP

We can consider some special cases of the TSP in order to solve them using approximation algorithms.

6.4.1 Matric TSP

Given any three nodes a, b, c, if:

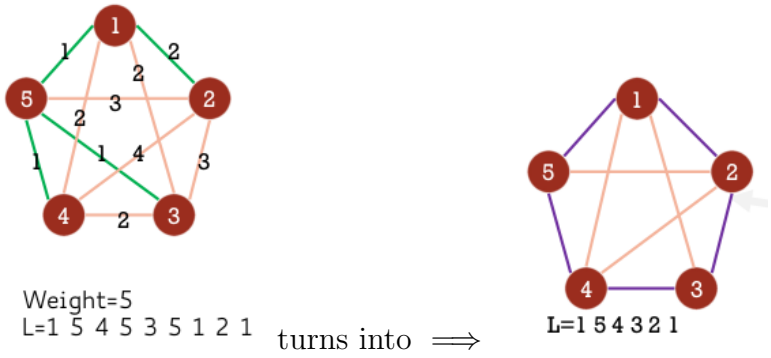
$$w(a, c) \leq w(a, b) + w(b, c)$$

We say that w **satisfies the triangle inequality**. Note that, in this case the weight of a MST T is a lower bound on the cost of the optimal TSP. Indeed, if H^* is the optimal tour, a spanning tree P can be deduced from H^* by deleting an edge. Moreover, P is a path, then:

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

We can build a 2-approx. algorithm called **2-Approx-mTSP**:

1. Given a complete graph $K_n = (V, E)$, let's select a node r.
2. Compute the MST T from r (done in polynomial time).
3. Let L be the sequence of nodes visited in a walk of T.
4. Return the Hamiltonian cycle that visits the nodes by following the same order L without repetitions.



This algorithm is a 2-approx. algorithm if w satisfies the triangle inequality. We know that if H^* is an optimal tour and T is an MST, then $w(T) \leq w(H^*)$. In L every edge is repeated exactly twice so the tour C deduced by L is such that $w(C) = 2w(T)$. However, C is not a Hamiltonian cycle since some nodes are repeated. If in L we have a situation like: $a \to b \to a \to c$, we can skip the middle a by passing directly from $b \to c$ since for the triangle inequality:

$$w(b, c) \leq w(a, b) + w(a, c)$$

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

6.4.2 Euclidian TSP

In this case the **cities of the TSP lies on the Euclidian space**. Like the classical TSP, even the Euclidian TSP is NP-hard, but we have an approximation algorithm. This can be considered as a **special case of the mTSP since the distances that represent weights respects the triangle inequality, hence also for this problem we can use the mTSP approximation algorithm**. Moreover, **we also have a better approximation algorithm with a $\frac{3}{2}$ approximation ratio**. This algorithm is called **Christofides Algorithm**:

1. Given a complete graph K_n, compute the MST T.
2. Let O be the set of nodes with odd degree in T. For a lemma, $|O|$ is even.
3. G^O is the subgraph induced by O (remove from the original graph, all the nodes and edges that are not in O).
4. Find a min-weight perfect matching M in G^O (set of edges that covers every node in G^O with the minimum weight).

5. The graph induced by $M \cup T$ is a connected multigraph where each node has an even degree (every node with even degree in T remains with the same connections, all the nodes with odd degree in T now will be connected with one more edge to other nodes therefore now those nodes have even degree too).

6. Form an Eulerian circuit (closed walk that traverses every edge exactly once).

7. Return the Hamiltonian cycle that visits the nodes without repetitions.

This algorithm is a $\frac{3}{2}$-approx. algorithm if w satisfies the triangle inequality. We know that if H^* is an optimal tour and T is an MST, then $w(T) \leq w(H^*)$. Let's assign a number of nodes in G^O in cyclic order around H^* and then let's partition H^* into two sets of edges:

1. M_o Edges with first node with odd number.

2. M_e Edges with first node with even number.

Each of these sets is a perfect matching: $M_o \cup M_e \subseteq H^*$. Hence, $w(M_o) + w(M_e) \leq w(H^*) \implies w(M_o) \leq \frac{w(H^*)}{2}$. Since M is a min-weight perfect matching of G_O, then $w(M) \leq w(M_o)$. Putting all these inequalities toghether:

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

PTAS: There exists a polynomial time approximation scheme (PTAS) such that for any $c > 0$, if we are in a d-dimensional Euclidian space, we can use a polynomial algorithm that provides us a solution that is at most $(1 + \frac{1}{c})$ times the optimal solution in time $O((n \log n)^{O(c\sqrt{d})^{d-1}})$. So the more c grows, the better will be the approximated solution, but on the other hand the time complexity tends to the exponential complexity.

6.5 Some Generalizations

6.5.1 Relaxing TSP Constraints

We can drop the condition that impose to the salesman to visit each city exactly once, in this way we can stop considering Hamiltonian cycles and start considering simply closed walks. If the problem is metric (respects the triangle inequality), then every optimal tour is an optimal solution.

6.5.2 Asymmetric TSP

Instead of K_n, we consider the complete directed graph K'_n. This problem contains the usual TSP as special case and it is also NP-hard.

6.5.3 General Connected Graph

We can also consider and arbitrary connected graph G with some weight function w instead of K_n. In this case is not clear whether any tour exists, checking that the graph is an Hamiltonian graph (so that there exists an Hamiltonian cycle) is NP-hard.

7 Data Collection in Sensor Network

In this section we are still talking about the collection of the data in sensor networks. When the data mule cannot be used due to several motivations, we have to send the sensed information to the base stations using other approaches:

- **Naive Approach**: Every sensor increases its transmission range to reache the sink directly. This is not a feasible solution due to the fact that the distance from the base station could be too high, therefore in order to reach it we should provide to much energy.
- **Multi-hop Data Routing**: Use a multi-hop protocol to send data (even this has been discussed in the previous section).
- **Clusters**: Sensors are grouped in clusters, every sensor sends to the **cluster head** (node elected as leader of the cluster) and then it will forward the data to other clusters/sink. In this solution, the cluster head will probably will drain its battery more quickly than the others. We can balance this problem by selecting a new cluster head periodically.
- **Duty Cycle Based Mode**: The sensor transmitters are active only in certain periods in order to save energy. The duty cycle can be implemented through a fixed cycle-time or can be triggered whenever the sensor reaches a certain amount of sensed data. An active sensor can communicate only with another active sensor, so it may wait that some near node wakes up thus increasing the overall delay.
- **Mixed Approach**: This solution mixes the duty cycle with a connected sub-network (backbone) constructed through some nodes. Nodes inside the backbone use a fixed time interval to switch between active/sleeping mode, instead the nodes outside of the backbone turn off their radio transmitters if there are no data to be sent. Those nodes send the data to the nodes that compose the backbone that will forward the packets to the sink. Since the

backbone-node batteries are drained quickly with respect to the nodes that are outside of the backbone, a new backbone is created in order to balance the overall battery lifes in the network, indeed the nodes with the highest residual energy will be picked in the new backbone. Backbone requirements:

- Number of nodes inside the backbone must be smallest as possible.
- The backbone must be connected with respect to the sink (at least one path for every node to the sink).
- The other nodes in the network must communicate directly with at least one node in the backbone (dominating set).

7.1 Min Connected Dominating Set

A **min connected dominating set is the key to find the nodes that can be elected formers of a backbone**.

Given a graph $G = (V, E)$, a **dominating set (DS)** for G, is a subset D of V such that every node in $V - D$ is adjacent to at least one member of D. A **min DS** is a dominating set with the smallest cardinality possible.

A **min connected DS is a dominating set for G such that it's a min DS and it induces a connected subgraph of G.**

7.2 Maximum Leaf Spanning Tree and CDS Theorem

In any n-node graph G with $n > 2$, $n = d + l$ where:

- d: is the **cardinality of the min CDS** of G.
- l: the **number of leaves of the maximum leaf spanning tree** of G, so the spanning tree of G with the maximum number of leaves.

Proof: We have to prove the two inequalities:

- $n \leq d+l$: If D is a CDS, then there exists a spanning tree for G whose leaves include all the nodes that are not in D. In fact, if we start from the connected subgraph induced by D, then we can add as leaf of nodes that are in D, every node $v \notin D$. Hence the number of leaves $l' = n-d$, therefore if l is the maximum number of leaves: $l \geq l' \implies l \geq n-d \implies n \leq l+d$.
- $n \geq d+l$: Let T be the maximum leaves spannin tree of G. The nodes of T that are not leaves forms a CDS D' of G. Hence, $|D'| = d' \geq d$ and since $d' = n-l$ then $n-l \geq d \implies n \geq d+l$.

7.2.1 Computational Complexity

Thanks to this theorem, we can derive that calculating the max leaves number we also obtain the connected domintation number. However, t**esting whether there exists a CDS with size less than a certain threshold and testing if there exists a spanning tree with at least a given number of leaves** are two problems **both NP-complete**.

In terms of approximation algorithms, these two problems are completely different. In fact, there exists an **approx. algorithm that calculates the min CDS using a** $2 + \ln \Delta + O(1)$ **ratio** (with Δ the maximum degree in G). Instead, calculate the **max leaf spanning tree can be approximated within a factor 2**.

If G **is a graph where the maximum degree is 3, then both problems are solvable in polynomial time**.

Even if we drop the connected condition, so we are not anymore trying to find the min CDS but a simple **min DS, the problem remains NP-complete**. Indeed, there exists a **bijection between solving the min DS and the MinSetCover problems**.

7.3 Two-Step Greedy Algorithm

This algorithm is an approx. algorithm for solving the min CDS problem:

1. Consider G and a subset C of its nodes.
2. All nodes in G can be divided into three classes with respect to C:
   - B (Black nodes): nodes that are in C.
   - Gr (Gray nodes): nodes not in C but adjacent to at least a B-node.
   - W (White nodes): nodes neither in C nor adjacent to it.

   Clearly $B \cup Gr \cup W = V$ and C is a CDS iff there is no W-node and the subgraph induced by B-nodes is connected.
3. Call CC the number of connected components in this black subgraph, then $|W| + CC = 1$ (hence, $W = \varnothing$ and $CC = 1$).

The greedy algorithm is based upon the potential function $Pf = |W| + CC$.

First-Step:

First-Step Greedy Algorithm (G)

Repeat

if there exists a white or gray node s. t. coloring it in black and its adjacent white nodes in gray would reduce the value of Pf

then choose such a node and reduce the value of Pf

else return

In this first step we are producing a simple DS that may be not connected, therefore we have to apply the second step.

Second-Step:

Second-Step Greedy Algorithm (G)

Repeat

color either one or two gray nodes in black to reduce CC

Until CC=1

This step guarantees to obtain a connected dominating set.

The final **approximation ratio of this algorithm is** $3 + \ln \Delta$. However, it is **possible to design a single-step greedy algorithm with a more complex potential function that achieves a ratio** $2 + \ln \Delta$.

7.4 Disk Graphs

Consider a set of n equally sized circles in a plane called **intersection model**, then **intersection graph (a.k.a Disk Graphs)** of these circles, is a **n-node graph in which every node corresponds t a circle and there is an edge between two nodes if the corresponding circles intersect**.

Disk graphs are suitable to **model the wireless networks**, where every node of the network is a node in the graph and the transmission range of every node it's the radius of the circle. If the **network is homogeneous every circle will has radius 1**.

Even if we **restrict the min CDS to the UDGs** (Unit Disk Graph) or **to grids** (subclass of the UDGs), **the problem remains NP-hard**. There **exists a PTAS for min CDS**, for any small $\varepsilon > 0$, there exists a polynomial approximation $(1 + \varepsilon)$, however it is **not used in practice**.

7.4.1 Distributed Algorithm for Reducing CDSs

This algorithm is a heuristic, therefore we don't have any guarantees about neither lower nor upper bounds. It is based on the **convex hull** concept. A convex hull for a set of points X in a 2D space, is the minimum convex set containing X:

Repeat
  Select a minimum degree node u from the given CDS
  Compute $CH(N[u])$ and, $\forall i \in N(u), CH(N[i])$
  if $CH(N[u]) \subseteq \bigcup_{i \in N(u)} CH(N[i])$
    then remove node u from the given CDS
Until there are unconsidered nodes in the given CDS

Where $N[v]$ **is the closest neighborhood of a node** v**, therefore** v **and its adjacent nodes**. This algorithm is very simple and since can be run in a distributed fashion it does't require any global knowledge.

8 Mobile Sensor Networks

Wireless sensor networks are large-scale wireless multi-hop networks where nodes have limited resources such as energy, bandwidth, storage, and processing power. They are deployed into some Area of Interest AoI for many applications. Their aim is to sense the AoI without leaving unsensed points inside it. We can deploy these network in two ways:

- **Carrier-Base Method**: Using robots that carry the sensors and drop them into specific points of the AoI.
- **Self-Deployment Method**: Use nodes that automatically change their position until reaching the desidered distribution.

Mobile sensors are tiny and low-cost nodes that can produce measurable responses based on the physical environment. They are **equipped with**:

- Detecting/Monitoring Unit (for sensing the environment).
- Communication Unit.
- Small Battery.

- Computing Unit.
- Motion System.

Each sensor can be seen as a device with **two ranges**:

- **Communication Range** r_c: Range in which it can receive or send packets.
- **Sensing Range** r_s: Range in which it can sense the environment.

These two ranges are not related, however it can be proved that protcols in which we assume to have $r_c = 2 * r_s$ satisfy the constraints.

The **coverage is one of the most important problem that we have to face**, it is defined as **measurement of the quality of survaillance of sensing function**. The quality depends from how well the nodes have been deployed in the AoI, we have to sense each point in the area with a good overall quality and also we must be able to send those information to the sink. The coverage problem can be divided into two different problems:

- **Point Coverage**: Set of discrete points that must be sensed continuously (e.g. building access).
- **Area Coverage**: All points inside a region must be monitored continuously (e.g. forest monitoring, every location of the forest must be covered by at least one sensor node in order to immediately detect any unusual activities like forest fire).
- **Barrier Coverage**: A specified path or the boundaries of a region that must be continuously monitored (e.g. montioring a restricted-access area).
- **Sweep Coverage**: A Poit of Interest (PoI) is said to be **t-sweep covered** iff it has the need too be visited every t time periods, where t is called **sweep period**. In this case we want to find the minimum number of nodes that can guarantee sweep-coverage for a set of PoI. It has been proven that using both mobile and static sensors is much more effective rather than using only mobile ones.

The **location knowledge of a sensor** is essential since the deployment protocols depend often on this information. In **outdoor applications** we can locate sensors using **GPS**. Concerning the indoor applications, we have to distinguish between:

- **Indoor Centralized Applications**: Typically a grid-based approach is used when the global position is needed for the deployment. The grids are used as landmarks where the sensors have to be placed, sensors can act also as landmarks themself.
- **Indoor Distributed Applications**: In this case some techniques such as the received signal strenght, time difference of arrival of two signals and so on, are exploited.

8.1 Deployment Problem (Area Coverage)

The deployment problem for area coverage is can be expressed in two different ways:

- **Given an AoI to cover**, the aim is to **cover the whole area by minimizing the number of needed sensors**.
- **Given a set of sensors**, we want to **maximize the sensed area**.

The **coverage can be either single or multiple coverage**, respectively, each point is sensed by one sensor or more sensors can sense the same point. The second coverage guarantees that in case of faults, the area that was sensed by the dead node, remains still sensed.

8.1.1 Coordination Algorithm

We start from an initial configuration, that can be random or from a safe location, and we want to achieve a final configuration. The **final deployment** can be either a **regular tasselation** (organized) or **any configuration** (not organized) such that we are covering the AoI.

While building the deployment we have to keep account of several parameters that must be optimized:

- **Traversed distance**: It's the distance that a node has to travel to reach the final position and it is the dominant cost that must be minimized.
- **Number of start/stop moves**: They're are important since stop and then re-start moving is much more expansive rather than a continuous move.
- **Communication Cost**: It depends on the number of exchanged packets.
- **Computation Cost**: Usually it's not considered unless the process are extremely complicated.

The r**andom deployment is the easiest deployment** in which we **drop** over the AoI **sensors in a random way**. When we have **no knowledge about the area or it changes very fast** (like in warfields), this method reaches a **good overall coverage**. On the other hand, this method **doesn't guarantee a uniform distribution**, that is needed for achieving long-life networks. **Another well-known deployment** in the literature is constructed by **placing all the nodes in a triangular grid such that the three disks intersect in a single point**. This method is used in cellular networks and requires a carrier-based method in order to place the nodes in the right spots, therefore we have to minimize the number of robot's movements.

The self-deployment method can be either:

- **Centralized Approach (Global)**: It relies on global information that is not usually scalable.
- **Distributed Approach (Local)**: These algorithms are based on the iterative nearest neighbor exchange.

8.1.2 Centralized Deployment Problem

We have to distinguish between two main cases:

- **Incremental Approach**: We have no information about the AoI, therefore we can use a one-at-a-time approach in which every node calculates its position based on the information gathered from the previous node. At each step the sensor gets its ideal position thanks to the base station, once that it places into that position, it will forward the information about its positioning to the base station in order to used them in the next iteration. We don't need any localization technique since the sensors act as landmarks. This method guarantees that the best position is given to any node and since they are fixed once placed, the energy consumption is low. However, it can require a lot of time and computational power to be performed.
- **Min Weight Perfect Matching Approach**: We have information about the AoI and we can model the problem as a min weight perfect matching problem in the bipartite graphs.

8.2 Min Weight Perfect Matching in Bipartite Graphs

8.2.1 Graph Model

We have a **set of n mobile sensors $S = \{S_1, ..., S_n\}$, a set of p locations in the AoI $L = \{L_1, ..., L_p\}$ such that $n \geq p$. For each sensor S_i we have to determine its location L_j such that the total energy is minimized**.

We can define a **weighted completed bipartite graph** $G = (S \cup L, E, w)$ such that we have one node for every sensor, one node for every location, an edge between every node S_i and L_j with the weight that is proportional to the energy needed by S_i to reach L_j.

8.2.2 Matching

In general, given a graph $G = (V, E)$, **a matching is a subset of edges M such that every node is adjacent to at most one edge in M**:

- **Maximal Matching**: M is a maximal matching if there is no edge e such that $M \cup \{e\}$ is a matching.

- **Maximum Matching**: M is a maximum matching if M is the matching with the highest cardinality possible. It's **not unique**.
- **Perfect Matching**: M is a perfect matching if $|M| = \frac{n}{2}$, so if each node is adjacent to exactly one edge in M. If G is bipartite then $|M| = \min\{|V_1|, |V_2|\}$.

A node that is not inside a matching is called **free node**. Given a graph G, **finding a maximal matching can be done in a polynomial time** using a greedy algorithm. The **same can be done with maximum matching**, even if it's **more complicated but still polynomial**. If the **graph is bipartite is much more easier**. The **perfect matching is a special case of maximum matching**.

8.2.3 Wedding Problem

This problem was about finding the maximum matching such that it maximizes the number of couples in a bipartite graphs where the nodes are divided in male and female.

8.2.4 Hall's Marriage Theorem

We have a theorem called **Hall's Marriage Theorem** that states the following: **Given a bipartite graph G with $|V_1| \leq |V_2|$, then G has a perfect matching iff $\forall$ set $S \subseteq V_1$ of k nodes, there are at least k nodes in V_2 that are adjacent to some node in S**. Formally, $\forall S \subseteq V_1 : |S| \leq |adj(S)|$.

Proof: We have to prove both side of the implication:

- If G is a perfect matching M and $S \subseteq V_1 \implies$ each node in S is matched in M with a different node $adj(S)$, therefore $|S| \leq |adj(S)|$.
- We have to prove that if the Hall condition is true, then there exists a perfect matching. By contradiction, assume that M is a maximum matching but $|M| < |V_1|$. By hypothesis $|M| < |V_1| \implies \exists u_0 \in V_1 : u_0 \notin M$. If we consider $S = \{u_0\}$, the Hall condition holds $1 = |S| \leq |adj(S)|$, therefore there exists $v_1 \in V_2$ adjacent to u_0. Consideration on v_1:
  - $v_1 \notin M$: We can add it the edge (u_0, v_1) to M, otherwise M is not a maximum matching.
  - $v_1 \in M$: We can consider as u_1 the node matched with v_1. If we consider $S = \{u_0, u_1\}$ then $2 = |S| \leq |adj(S)|$, there exists v_2 that is adjacent to u_0 or u_1 that can be either in M or not.

  We can repeat this until we reach a contraddiction since G is a finite graph.

This **theorem does not provide an algorithm for solving perfect matching**.

8.2.5 Perfect Matching and Flow Network

Finding a **perfect matching in a bipartite graph is equivalent in finding the maximum flow problem in a network**. In this problem we have a network composed by a source and a tail, from the source we have to send the maximum number of flows to the tail.

We can build an instance for the maximum flow problem by **starting from the bipartite graph $G = (V_1 \cup V_2, E)$, then the graph representing the flow network G' is built as follows**:

- $V' = V \cup \{s\} \cup \{t\}$
- E': We add an edge between the source and every node in V_1 and one edge between every node in V_2 and the tail. The capacity of every edge is set to 1.

Let M be a matching in bipartite graph G, then there exists a flow f in the network G' such that $|M| = |f|$. Even the reverse holds, if there is a flow f in G', then there exists a matching M in G such that $|M| = |f|$.

The max value of the flow f in G' is equal to the cardinality of the max matching M.

Alternating Path: Given a matching M in a graph G, an alternating path with respect to M is the path alternating edges of M and $E \setminus M$.

Augmenting Path: Given a matching M in a graph G, an augmenting path with respect to M is an alternating path that starts and ends into two free nodes. By **swapping the roles of an augmenting path, we can increase the cardinality of the matching** M. We have a theorem which states: M is a max matching iff there are no augmenting paths:

- $\implies$: Trivially, if M is max and there is an augmenting path we could swap the roles of that path increasing the cardinality of M, therefore this would end into a contradiction.
- $\impliedby$: Skipped.

Ford-Fulkerson Algorithm: This algorithm is used to get the max flow in a network in $O(m * |f|)$, since in our case the flow is upperbounded by $f \leq \min\{|V_1|, |V_2|\}$, the time complexity becomes $O(m * n)$. We **exploit the augmenting path theorem to design an iterative algorithm**:

1. **Start from a matching** (even the empty one).
2. **While there is an augmenting path P: swap the roles of the edges**.

The time complexity depends on how we find the augmenting path. **We can direct the edges in G with respect to M such that the edges in M goes from V_1 to V_2 and those that are not in M goest from V_2 to V_1**. If we call this graph D then we can state that **there exists an augmenting path in G iff there exists a path in D that starts in a free node in V_1 and ends in a free node in V_2**. The idea is to **run for every free node in V_1 a DFS** (Depth First Search) such that as soon as a **free node in V_2 is met, a new augmenting path is found**. Since we are running this algorithm for $\frac{n}{2}$ nodes in V_1, the **time complexity** becomes $O(n+m) * \frac{n}{2} = O(n * m)$.

Hopcroft-Karp Algorithm: This algorithm improves the Ford-Fulkerson algorithm by finding the maximum matching in $O(m\sqrt{n})$. The idea behind the algorithm is the same, instead of performing a DFS we **use a modified BFS** (Breadth First Search) **such that we are searching not a single augmenting path but a maximal set of those**. During the k-th step we **run the modified BFS starting from all the free nodes in V_1 until some nodes V_2 is reached in layer k** (each of these augmenting path have length at least $2k-1$). All these free nodes V_2 are added to a set F. Then **using a DFS we climb up the free nodes until reaching the starting node in V_1 by finding the maximal set of augmenting path that are node disjoint**. Each of these paths are used to enlarge the matching M. Each step consists into a BFS and DFS, hence the time complexity is $O(n+m) = O(m)$ (we are considering a connected graph, therefore $m >> n$).

If we consider our partial solution M after the first $\sqrt{n}$ steps, we can compute the symmetric difference between the maximum perfect matching and M (symmetric difference = remove only common elements and merge the two sets). This is a vertex-disjoint of alternating cycles, alternating paths and augmenting paths that can be used to increase the partial solution M. **Considering that at this point the length of the augmenting paths is at least $2\sqrt{n}-1$, there are at most $\sqrt{n}$ paths. The whole algorithm executes at most $2\sqrt{n} = O(\sqrt{n})$ steps, therefore the time complexity is $O(m * \sqrt{n})$.**

This is true for the worst case, however it has been proved that in **random sparse bipartite graphs the augmenting paths have a logarithm length, therefore the time complexity is $O(m * \log n)$.**

8.3 Weighted Matching in Bipartite Graphs

When we introduce the weights in our problem, we've to take account that some definition are not longer the same. In particular, in weighted graphs the **augmenting path is any alternating path such that the weights of the edges out of the matches is greater than the weight of the edges inside the match**. In this case there is **no need to end at a free node**.

8.3.1 Find the Min Weight Perfect Matching

If we want to find the min weight perfect matching, it's much easier to found the max weight perfect matching by negating the weights.

Algorithm: The time complexity is $O(n * m)$:

1. Start with an empty matching.
2. Until we can find an augmenting path P, select the path with the highest weight and swap the edges to increase M.

8.3.2 Min Weight Perfect Matching ILP Formulation

It is also possible to model the problem using the ILP formulation. This is called **Hungarian method**, **the bipartite graph is modeled as a matrix where the rows represent the nodes in V_1 and the columns represent the nodes in V_2. Every $x_{i,j}$ in the matrix is 1 if the edge $(i, j) \in M$ and 0 otherwise**. We want to **minimize** the following value:

$$\sum_{i,j} c_{i,j} \; x_{i,j}$$

The **time complexity is** $O(n^3)$.

8.4 Maximum Matching in General Graphs

If the graph is not bipartite the main problem are the presence of **blossoms**. A **blossom is an odd cycle with a maximal of number of edges**. These cycles are problematic since **every free node in a blossom is connected with two edges that are not in M, so if we use the classical algorithms seen previously, the node may be not detected by the DFS**. Hence we have to remove these cycles to be able to solve the problem.

8.4.1 Cycle Contraction Lemma

Let M be a matching of G and let B be a blossom that is disjointed with respect to M (hence there are no edge in M that has an endpoint in B). Let's call G' the graph obtained by G contracting B in a single node. Then M' **of** G' **induced by** M **is maximum in** G' **iff** M **is maximum in** G. The proof is the following:

- $M \max \text{in } G \implies M' \max \text{in } G'$: By contradiction, if M' is not maximum, then there exists an augmenting path P in G' with respect to M'. If we call b the node that represents the contracted blossom B:

- P does not cross b: therefore P is an augmenting path for M too. Contradiction.
- P crosses b: then b is and endpoint of some edge in P. So, we can create a new augmenting path $P' = P \cup P^*$ where P^* is the path in B. P' is an augmenting path for G, contradiction.

- $M' \max \text{in } G' \implies M \max \text{in } G$: Suppose that M is not maximum, so there exists an augmenting path P in G with respect to M:
  - P does not cross b: P is an augmenting path for G', contradiction.
  - P crosses b: Since B contains only one free node, at least one node of P must be outside B. Let's call this node w, then P' is the sub-path of P joining w with b. P' is an augmenting path for G', contradiction.

8.4.2 Edmonds Algorithm

To solve the max matching for general graph is sufficient to contract blossoms and apply the standard algorithm for bipartite graphs, indeed, a **general graph without odd cycles is a bipartite graph**. **For each found blossom B we can use the lemma to contract the graph G, then we can use the found augmenting paths to enlarge the matching**:

1. Starting from a matching M (even the empty set).
2. L is a **subset of free nodes**.
3. F is the **forest of trees such that every node in L is a root** of a tree.
4. At **each step we increase the trees by adding one edge that is not inside M and one that is part of M**. In this way the nodes that are at odd distances are 2-degree nodes and they're called **internal nodes** (there are in the between an edge inside M and an edge not inside M). The other nodes are leaves and are called **external nodes**.
5. Considering the neighbors of an external node x:
   - There exists a node y that is not in F and that is adjacent to x. Hence, we can add to F the edges (x, y) and (y, z). The (y, z) edge is added to M.
   - There exists an external node y that is an external nodes of another tree in F. We can connect them in order to form an augmenting path that starts from the root of the first component and goes to the root of the second component.
   - There exists an external node y that is adjacent to x and is in the same component, therefore we found a blossom. We can apply the lemma.

- All the external nodes are adjacent to internal nodes, M is maximum.

At each step of the algorithm either the dimension of F is increased, or the dimension of G is decreased (blossom contraction), or an augmenting path is found or M is maximum. For what concerns the time complexity, we know that the number of needed iterations $\leq$ that num. of times F is increased (at most n)+ num. of times a blossom is shrinked (at most n)+ num. of found aug. paths (at most n/2). Therefore, the **final complexity depends on how the blossom are handled, it can be $O(n^3)$ or $O(m * n^2)$. The best known time complexity is $O(m * \sqrt{n})$.**

8.5 Switch Buffer

Recalling what we've seen during the Interconnection Topologies section, when two packets want to use the same link at the same time, a buffer is needed to prevent the collision. These multistage topologies are usefull since are modular and scalar, however the global throughput is decreased and possible delays are added. A **possible solution could be to use input buffers that are external to the topology**.

8.5.1 Head Of Line Buffer (HOL)

The HOL buffers are built as s**everal FIFOs connected through a mesh to the inputs of the topology. The mesh's switches are set by an external scheduler**. This **solution still introduces throughput lowerings** since if two item on different buffers must be sent to the same input, one of the two is kept idle in that step.

8.5.2 Backlog Matrix

If we now introduce a new buffer that can send even items that are not in the head, we could improve the throughput. This is done using the **backlog matrix**, their **rows are represented by the input buffers and the colums by the outputs. Every cell (x, y) in the matrix represents the number of packets in the buffer x that must be delivered to the output** y. At each step the scheduler can send at most one packet to each output, the scheduling can be seen as a problem in bipartite graphs where: the graph G is composed by V_1 that are the nodes representing the buffers and V_2 that are the nodes representing the outputs. The edges E are built as follows, we draw an edge (x, y) if the corresponding matrix cell value is greater than 0 (at least one packet that must be delivered from buffer x to output y). **To achieve a full usage of the network, we have also to assign weights to the edegs, in detail, at each edge is assigned a weight equal to its corresponding matrix cell value**. To **find the largest set**

of packets that can be sent simultaneously, it is sufficient to solve the max matching for that graph.

9 Distributed Deployment of Mobile Sensors Network (i.e. Voronoi Diagram Construction Problem)

The centralized approach is not always a good solution since we need a connection to the server, there can be some delays due to the computation and the solution is not fault-tolerant. We want to exploit a possbile solution for self-deployment in which nodes can calculate by themself their position that can be adjust on-demand in case of failures. The self-deployment is also necessary in warfields, disaster areas... In general, the best approach is the **Look-Compute-Move cycle**, in which every sensor checks the neighborhood and choose its new position continuously.

A possible approach is the **virtual forces approach**, the sensors are similar to charged particles subject to magnetic force. **Two sensors repel eachother if they are too close and attract eachother if they are far but can still communicate**. We need a lot of **manual parameter tuning and some friction forces/stopping conditions to manage the possible continuous oscillations**. Moreover, in some versions of the protcol, sensors are attracted by the obstacles and borders, therefore they have **difficulties to get through doors or narrows**.

9.1 Voronoi Diagram

The idea is to **assign to each sensor an AoI portion**. The **sensor is satisfied when it is covering the whole portion or it is fully using the sensing radius**, otherwise it will **move until being satisfied**. Sensors move until they're all satisfied. A **Voronoi Diagram** is formally defined as follows, c**onsidering as set P of n sites on the plane, the $VD(P)$ (Voronoi Diagram of P) is a partition of the plane into n cells V_i such that**:

- **Each V_i contains exactly one site**.
- **If a point Q in the plane is in V_i, then $dist(Q, P_i) < dist(Q, P_j)$ for any $P_j \in P$ $s.t.$ $j \neq i$.**

The diagram is composed by:

- Voronoi Axis: boundaries of the regions.
- Voronoi Vertex: vertices of the regions.
- Voronoi Cells: regions.

9.1.1 General Position Assumption

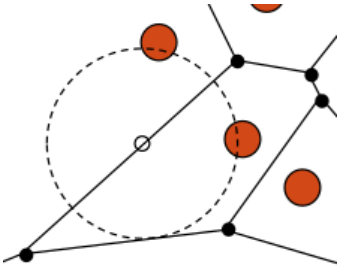
A Voronoi Diagram of a **single node** is, trivially, the **entire plane**. When we consider **two nodes**, the **plane is splitted in half**. In the case of all **collinear sites, the plane is splitted in parallel sub-planes**. If **3 nodes are not collinear**, we can always draw the circumference that passes by all the nodes, if we **tack the segment that links the nodes and then we track the orthogonal axes that split the segments into two halfs, we can see that the intersection between the axis is exactly the center of the circle**. The Voronoi **vertex has a degree 3**. When we consider **4 non-collinear sites, we will obtain a closed cell and 3 opened cells**. However, if these sites lies on a cicle, the resulting VD is composed by 4 opened cells (built as for the 3 sites case).

We will assume that in out plane the sites respect the **General Position Assumption**:

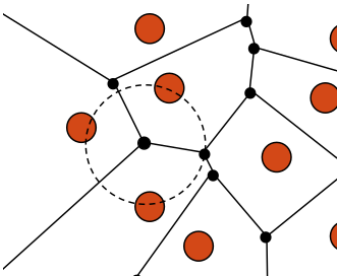
- **Each 3 sites are not collinear**.
- **Each 4 sites are not cocircular** (do not lie on a circle).

9.1.2 Voronoi Diagram Properties

A point q in the plane lies on a Voronoi segment between p_i and p_j (a Voronoi segment is a subset of the Voronoi axis) **iff the largest empty circle touch exactly only p_i and p_j**.



A point q in the plane is a Vornoi vertex iff the largest empty circle centered in q touches at least 3 sites of P. Hence, a Voronoi vertex is the intersection of at least 3 axes, each generated by a pair of sites.



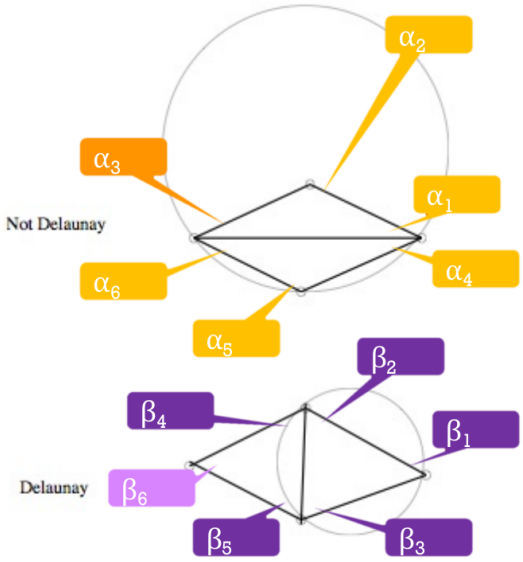
In a Voronoi diagram, the **number of voronoi vertex** v **is such that** $|v| \leq 2n - 5$ and the **number of voronoi segments** e **is such that** $|e| \leq 3n - 6$.

Proof: If the sites are collinear then $|v| = 0$ and $|e| = 1$ so it's proved. If they're not collinear we have to insert a dummy node to transform the Voronoi diagram into a graph, because some of the cells are not limited. With the dummy node, the graph is a connected graph, hence we can apply the Euler formula: $|v|-|e|+f = 2$ where $f = n+1$. Moreover, $\sum_{v \in D} deg(v) = 2|e|$ and since we are assuming that all the sites respect the general position assumption, $deg(v) \geq 3 \implies 2|e| \geq 3|v|$. Therefore, joining the Euler formula and this inequality we get the two values presented before.

9.2 Algorithm for Computing Voronoi Diagram

9.2.1 Deluney Triangolation

This problem is the dual problem of the Voronoi Diagram composition problem. **Given a set of discrete points** P **in general position, a DT is a set of triangulations such that there is no point** p_i **inside the cirumcirle of any triangle in the DT**. Considering two triangulations α and β, the **DT maximizes the minum angle in the triangulations**, therefore if $\min \alpha_i < \min \beta_j$ then the **associated segment is called illegal**. There cannot be illegal segment, so **we can switch the triangulation** α **with the** β **obtaining a valid triangulation**.



9.2.2 Planes Intersection

A V**oronoi diagram can be obtained by intersecting half-planes, this operation must be iterated for each site**. Therefore, **each of the** n **voronoi cells is the result of** $k = \theta(n)$ **intersections**. To calculate the time complexity of these operations we can use the divide-et-impera technique from the geometrical algorithm:

- **Divide**: The set of k halfplanes is recursively split until k single halfplanes are obtained.
- **Impera**: The halfplane on each leaf is intersected with a rectangle R (the search space). In this way, each leaf contains now a polygon.
- **Combine**: Recursively, bottom-up, compute the intersection of two sibling polygons and put the result on the father node.

Two polygons with p and p' vertices can be intersected in $O(p + p')$, a single step of the algorithm takes $O(k \log k)$. This is an optimal time since the step can be reduced in the sorting problem, whose lower bound is exactly $O(n \log n)$. In order to obtain n cells, we need $O(n)$ intersections of $O(n \log n)$ so the **final time complexity is** $O(n^2 \log n)$.

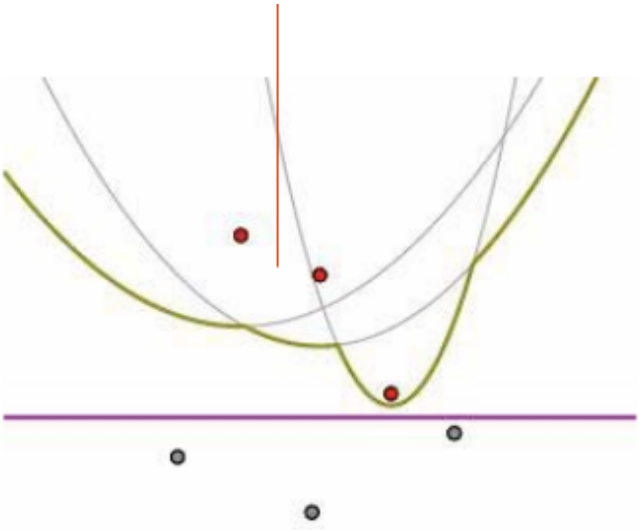
9.2.3 Fortune Algorithm

A **sweep line** is used to solve bidimensional problems through a sequence of almost onedimensional problems. **It's an infinite line that moves from top to bottom checking for those onedimensional problems** (such as segment, intersections, point of interest, etc...). In its classical form, this algorithm cannot be used on Vornoi diagrams since the site position should be predicted before the sweep line moves.

Fortune developed the his algorithm based on the **beach line** concept. Given any point p and any exteranl line l, the set of point equally distant from p and l forms a parabola $P_{p,l}$. Hence, considering any point $q = (q_x, q_y)$ and the sweep horizontal line l such that its y-coordinate is l_y, then: $dist(q, l) = l_y - q_y$. So, by considering another point (site) p, if:

- $dist(p, q) < l_y - q_y$, then q lies above the parabola.
- $dist(p, q) = l_y - q_y$, then q lies on the parabola.
- $dist(p, q) > l_y - q_y$, then q lies under the parabola.

At each step, the sweep line goes down and all the sites above it are considered. The **beach line is the union of all the lower parabola arches**:



vertical line? If a **point is above the beach line, it is closer to one of the sites above the sweep line rather than the others**. Therefore, the **point must belong to the Voronoi cell of a site already met by the sweep line. The Voronoi diagram above the beach line is completely determined by the sites above the sweep line**. If a **point** q **is touched by the beach line portion generated by the site** p**, then it will belong to the Voronoi cell associated to** p. Trivially, when a **point lies on two parabolas it will belong to the Voronoi axis between the two associated sites**, the intersection point between two parabolas are called **breakpoints**. In order to draw the whole Voronoi diagram is sufficient to keep track of the breakpoints. When a new site appears above the sweep line then a new parabola is drawn (**site event**), hence a new intersection appears generating the two breakpoints. **When a parabola arch disappears, a Voronoi vertex is created**. We can **detect the arch appearing simply by checking when the sweep line encounters a new site**. A **vertex is detect when 3 parabolas intersect**, one of the 3 will disappear after that point (**circle event**).

In order to determine the vertex and segment, we have to keep track of the parabola arches from left to right (based on the coordinates):

- **Site event is detected**: New parabola must be insert in the list.
- **Circle event is detected**: ??

Each **event can be determined and stored in** $O(1)$. Each d**ata structure that stores the information can store** $O(n)$ **information** and every **access to that structure takes** $O(\log n)$. The whole **time complexity is** $O(n \log n)$**, that is also optimal** since the sorting problem can be reduced to this problem.

9.3 Heterogeneous Sensors

Sensors are not necessarly all equal in term of communication and sensing capabilities. It can happen that sensors are different or that the sensing/ communication depend on their position (obstacles, terrain, ...). The virtual force and voronoi approaches do not keep account of the coverage capabilities. In fact, if we have a situation in which a subset of nodes can completely sense their cells (but could cover even more) and the other subset of nodes that are not able to fully cover their cells, a stale will occur.

We can define a new notion of distance based on the Euclidian distance and the heterogeity of the devices. We want to keep the diagram axis as straight lines and the set of points equally distant from two sensors contains the intersection of their sensing circles.

Laguerre Distance: Defined in $R3$, given two point $P = (x, y, z)$ and $Q = (x', y', z')$ their Laguerre distance is: $d_L^2(P, Q) = (x - x')^2 + (y - y')^2 + (z - z')^2$.

The two points can be seen also as circles, in which (x, y) are the center coordinates and $|z|$ is the raidus. Indeed, we can define the following lemma.

Lemma: Given two circles $\mathcal{C}_1$ and $\mathcal{C}_2$ centered in C_1 and C_2 (with $C_1 \neq C_2$) and having radius r_1 and r_2, the set of points equally distant from $\mathcal{C}_1$ and $\mathcal{C}_2$ (called **radical axis**) is a straight line orthogonal to the segment joining C_1 and C_2.

Lemma: C_1 and C_2 lies on the same side with respect to the associated radical axis iff $d_E^2(C_1, C_2) < |r_1^2 - r_2^2|$.

9.3.1 Vornoi-Laguerre Diagrams

??

10 UAVs

UAVs are flying vehicles able to autonomously decide their route (different from drones, that are remotely piloted). Historically, used in the military, mainly deployed in hostile territory to reduce pilot losses. Nowadays, they are also used in civil applications (weather monitoring, forest fires detection, ...).

10.1 UAV Probelm

Given an AoI whose map is known, we have a fleet of m UAVs that start from a safe location (**depot**) v_0, each of them with a battery endurance B. There are a set of sites $S = v_1, ..., v_n$ that must be examinated, each of them needs t_i time to be inspected. Each UAV must periodically go back to the depot to recharge the battery and this takes R time, $\approx [2.5, 5]$ B. We want to **overfly the sites in the shortest time possible**.

10.2 Graph Model

The nodes in the graph represent the **sites + the depot, therefore there are** $n + 1$ **nodes** in the graph. From every UAV we can reach every site and viceversa, hence we will build a **complete graph**. **Every edge has a weight equal to the time needed to traverse that distance**. Each UAV has a **flying+inspection time upperbounded by** B. Each **UAV will be assigned to a cycle** in which it will visit the sites assigned to it and then go back to the depot in the shortest time possible. The term "shortest time" can assume **different meanings**:

- **Minimize the total completation time**.
- **Minimize the average waiting time**.
- **Minimize the number of cycles**.
- ...

The energy consumption minimization is not a possible interpretation of the problem.

10.2.1 Problem Similarities

This problem is similar to many problems:

- **mTSP**: Multiple salesmen have to visit n sites, but in this case the optimization is about the length of the path and there are not battery constraints.
- **kTRPR** (k-Traveling Repairperson Problem with Repairtimes): The aim is to create k cycles to cover n points starting from a common depot. The latency is the time elapsed before visiting a certain point. We want to minimize all the latencies, so it is very similar to the UAV problem, but even in this case no battery constraints are considered.
- **mTRPD** (multiple Traveling Repairperson Problem with Distance constraints): In this case we have k repairperson that have to visit n customers and they are not allowed to traverse distance longer than a fixed one. The aim is to minimize the customers' waiting time. However, no repairtimes are considered and the number of cycles is fixed to k.

- **VRP** (Vehicle Routing Problem): Similar to mTRPD but with a fixed number of customers per vehicle.
- **TOP** (Team Orienteering Problem): equivalent to the first round of our problem, however the goal is not the right one since we want to maximize the number of covered sites.

None of these problem is equivalent to the UAV problem. From all these similarities, we can deduce that the **UAV problem is NP-hard**.

10.2.2 Connection with RMCCP (Minimum Bounded Rooted Cycle Cover Problem)

A **cycle cover** $\mathcal{C} = C_1, ..., C_k$ for V is a set of cycles such that each location of V belongs to at least one cycle. Given a fixed value $x \geq 0$, an **x-bounded cycle cover** is a cycle cover in which each cycle has a cost $cost(C) \leq x$. A **rooted cycle** C is a cycle where $v_0 \in C$. A **rooted cycle cover** is a cycle cover where all the cycles are rooted. The **completion time of a rooted cycle cover** $\mathcal{C}$ is $ct(\mathcal{C}) = \max cost(C)$ for all $C \in \mathcal{C}$.

The RMCCP is about finding an x-bounded rooted cycle cover of minimum cardinality (if it exists), given in input $\langle G, v_0, d, x \rangle$ with G the graph, v_0 the depot, d the weight function based on distances. This problem can be approximated within $O(\log x)$ factor (a better solution was proved to be $O(\frac{\log x}{\log \log x})$).

Theorem: If RMCCP can be approximated within α factor, the UAV problem can be approximated in $5\alpha + 1$.

Theorem: If the UAV probem can be approximated within γ factor, RMCCP can be approximated in $2\gamma + 1$.

Even if the two problems are tighly connected, RMCCP is about minimizing the number of cycles, the UAV probles aims to minimization the completion time.

10.3 MDMT-VRP-TCT

Multi Depot Multi Trip Vehicle Routing Problem with Total Completion Times minimization. This problem perfectly fits our problem since every UAV can perform many trips starting from any depot and the goal is to minimize the total completion time. Keep in mind that, splitting the AoI based on the depot position is not a correct solution.

10.3.1 MILP Formulation

This problem can be formulated as a Mixed ILP in which:

- **Sequence**: Ordered set k of target nodes. The **duration of a sequence** d_k is the sum of time needed to travel all around the nodes and to inspect them.
- **Trip**: Sequence k assigned to an UAV u with the addition of its depot o_u. The **duration of a trip** $d_{k,u}$ is the duration d_k + travelling time from the depot to the first node and the travelling time from the last node and the depot.

A sequence k is compatible with an UAV u if the associated trip duration is upperbounded by B. The main idea was consisting into produce all the possible sequences and choose the best solution, however it could be a huge number. Therefore, a relaxation is applied to the problem by creating a subset of the possible sequences and choose among them.

10.4 Problem Variations

Some variations are the following:

- **Priority Constraints**: Consider some points that must be visited with an higher priority with respect to the other sites.
- **Double Badget**: Consier also the memory as constraint and not only the battery.
- **Cooperation**: Allow the communication among UAVs.
- **Relaxed Multi Depot**: Allow UAVs to start from a depot and end into a different one.
- **Emergency Criteria**: Some criteria that could change the UAV behaviour dynamically (like when a persone is detected on a disaster area).