BDMA - Massive Graph Management and Analytics

Jose Antonio Lorencio Abril

Fall 2023

Professor: Nacéra Seghouani

Student e-mail: jose-antonio.lorencio-abril@student-cs.fr

This is a summary of the course *Massive Graph Management and Analytics* taught at the Université Paris Saclay - CentraleSupélec by Professor Nacéra Seghouani in the academic year 23/24. Most of the content of this document is adapted from the course notes by Seghouani, [3], so I won't be citing it all the time. Other references will be provided when used.

Contents

1 Introduction

Graph-structured data is at the heart of complex systems and plays a major role in our daily life, science and economy. Examples of this data are the cooperation between billions of individuals, or communication infraestructures with billions of cell phones, computers and satellites, the interactions between thousands of genes and metabolites within our cells, and so on.

Therefore, understanding its mathematical foundations, description, prediction, and eventually being able to control them is one of the major scientific challenges of the 21st century.

2 Preliminaries

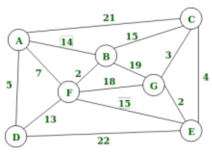
2.1 Graph Theory Preliminaries

A graph is a pair $G = (V, E)$, where V is the set of vertices and $E \subset V \times V$ is the set of edges. Usually, we denote $|V| = n$ and $|E| = m$.

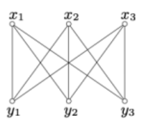
There are different types of graphs:

- **Undirected**: $(u, v) \in E \implies (v, u) \in E$. That is, the edges goes in both directions.
- **Directed**: $(u, v) \in E \not\Longrightarrow (v, u) \in E$. That is, the edges have direction, and it is possible that an edge goes from u to v, but not the other way.
- **Weigthed vertices**: the vertices have a weight. That is, there is a function $w_v : V \to \mathbb{R}$.
- **Weigthed edges**: the edges have a weight. That is, there is a function $w_e : E \to \mathbb{R}$.
- **Labeled vertices**: the vertices have a label, $L_v : V \to \mathcal{L}$, where $\mathcal{L}$ is the set of labels.
- **Labeled edges**: the edges have a label, $L_e : E \to \mathcal{L}$.
- **Bipartite**: a graph $G = (V, E)$ is bipartite if there is a partition of the vertices, $V = V_1 \cup V_2$, such that $V_1 \cap V_2 = \emptyset$ and $E = \{(v_i, v_j) \, | v_i \in V_1, v_j \in V_j\}$. That is, the vertices in V_1 only connect to vertices in V_2, and viceversa.
- **k-Partite**: a graph $G = (V, E)$ is k-partite if there is a k-partition of the vertices, $V = V_1 \cup V_2 \cup ... \cup V_k$, such that $V_i \cap V_j = \emptyset, \forall i \neq j$ and the is no edge $e = (u, v)$ such that $u, v \in V_i$, for the same i.
- **Multigraph** or **multidigraph**: in this case, there can be several edges between two vertices. For this, we define the edges as a separate set E, and a function $r : E \to V \times V$, that assigns the vertices related by that edge.
- **Hypergraph**: in this case, $E \subset 2^V$. That it, the edges can relate 0 or more vertices. In this case, it is more appropriate to interpret E as a set of classes or hierarchies, rather than edges.
- **Complete**: a graph is complete if $E = V \times V$.

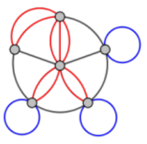
Some examples are:

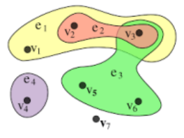


weighted graph



bipartite graph


multigraph



hypergraph

Continuing with definitions, let $G = (V, E)$ be a graph (directed or undirected). Let d_i^+ and d_i^- denote the number of edges coming out and coming to v_i, respectively. The **degree** of v_i is

$$d_i = d_i^+ + d_i^-.$$

Note that it counts double for undirected graphs.

Now, let N_i^+ and N_i^- the set of successors and predecessors of v_i, respectively. Then, the set of **neighbors** of v_i is

$$N_i = N_i^+ + N_i^-.$$

A **path** between two vertices, $u, v \in V$, denoted $u \rightsquigarrow v$, is a sequence of vertices $(u = v_0, v_1, ..., v_{k-1}, v_k = v)$, where $(v_{i-1}, v_i) \in E, \forall i = 1, ..., k$. The length of a path, $L(u \rightsquigarrow v)$, is the number of edges in the cycle, that is, k.

A **cycle** is a path from a vertex to itself, $u \rightsquigarrow u$.

The **distance** between two nodes, $d(u, v)$, is the shortest path length between them:

$$d(u, v) = \min_{u \rightsquigarrow v} L(u \rightsquigarrow v).$$

The **eccentricity** of a node, $ecc(u)$, is the greatest distance between u and any other vertex in the graph:

$$ecc(u) = \max_{v \in V} d(u, v).$$

Note that this could be infinity if we cannot reach some node from u. Usually, we consider only reachable nodes, because this can give us information about the graph, but a value of infinity is not very informative.

The **diameter** of a graph, $diam(G)$, is the greatest distance between two nodes in the graph:

$$diam(G) = \max_{u,v \in V} d(u, v) = \max_{u \in V} ecc(u).$$

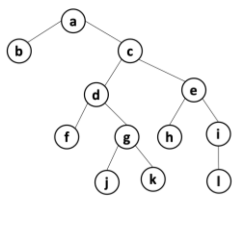
The **radius** of a graph, $rad(G)$, is the minimum eccentricity of any vertex in the graph:

$$rad(G) = \min_{u \in V} ecc(u).$$

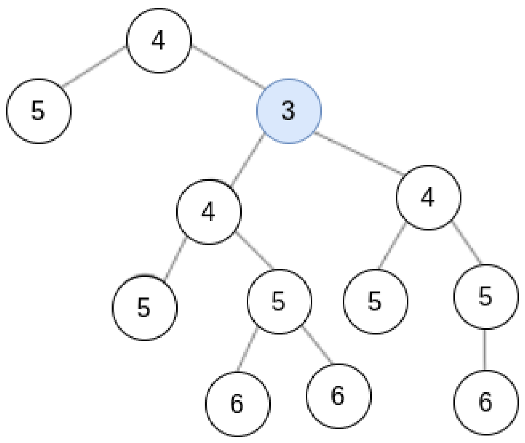
The **center** of a graph, $C(G)$, is the set of all vertices of minimum eccentricity, i.e., the graph radius:

$$C(G) = \{u : ecc(u) = rad(G)\}.$$

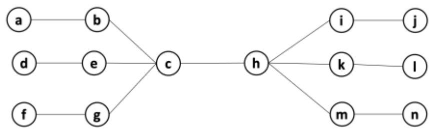
Example 2.1. Compute the diameter, radius and center of the following graphs:



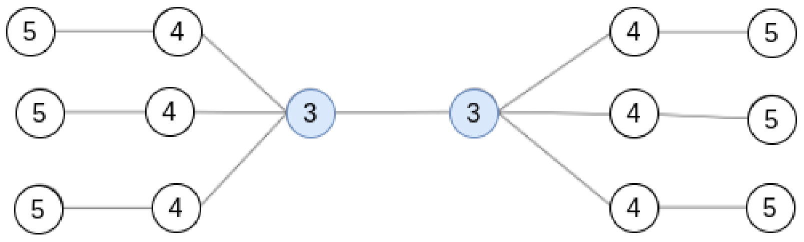
The solution is the following:



In each node, we show its eccentricity. The diameter is 6, the radius is 3 and the center is c (in blue).



Solution:



In this case, the diameter is 5, the radius is 3 and the center is $\{c, h\}$.

A **partial graph** of $G = (V, E)$ is a graph $G' = (V, E')$, where $E' \subset E$.

A **subgraph** of $G = (V, E)$ is a graph $G' = (V', E')$ where $V' \subset V$ and $E' \subset E$. Note that partial graphs are also subgraphs.

A graph $G = (V, E)$ is said to be **connected** if, and only if, $\forall u, v \in V, \exists u \rightsquigarrow v$.

A (strongly) **connected component** of $G = (V, E)$ is a subgraph $G_{cc} = (V_{cc}, E_{cc})$, where $\forall u, v \in V_{cc}, \exists u \rightsquigarrow v \in V_{cc}$. That it, a connected subgraph. It is called strongly when the paths are directed.

A graph $G = (V, E)$ is a **tree** if, and only if, G is a connected graph without cycles. In this case, the graph has $m = n - 1$ edges.

A graph $G = (V, E)$ is a **forest** if, and only if, all connected components of G are trees.

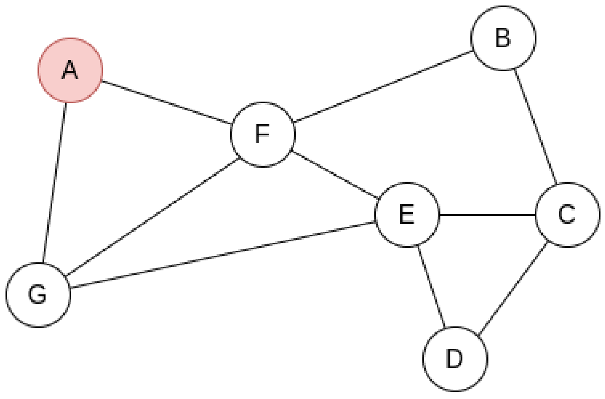
2.1.1 Breadth First Search (BFS)

BFS is a method to traverse the nodes of a graph, by starting at one node and traversing all its neighbours. Then, all neighbours of its neighbours, and so on.

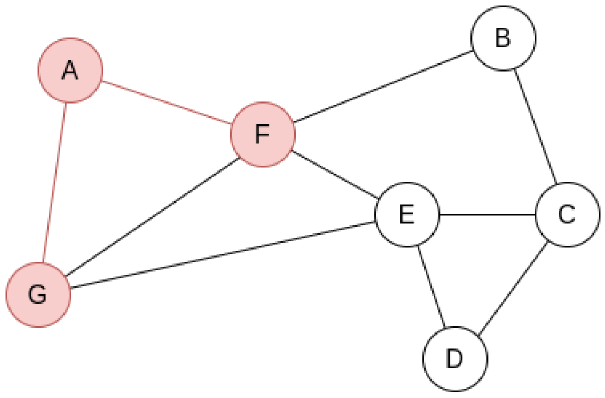
For this, we use a FIFO queue. The algorithm is:

```
procedure BFS(G=(V,E), r)
  Q <- emptyset
  enqueue(Q,r)
  r.label = True

  while Q is not empty do
    v <- dequeue(Q)
    for neig in neighbours(v) do
      if not neig.label then
        enqueue(Q,neig)
        neig.label = True
      end if
    end for
  end while
end procedure
```

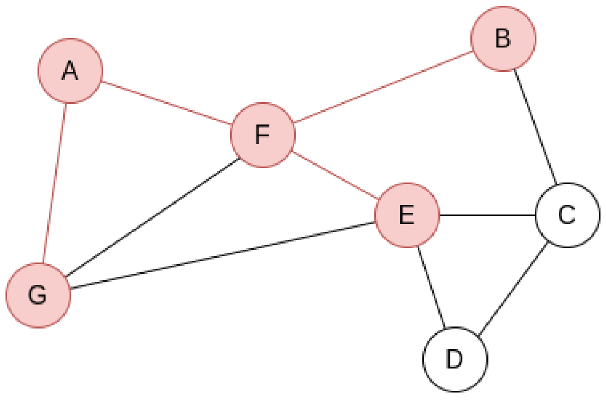
Example 2.2. Apply BFS in the following graph, starting at node A.



Q=[A]. We visit A's neighbours:

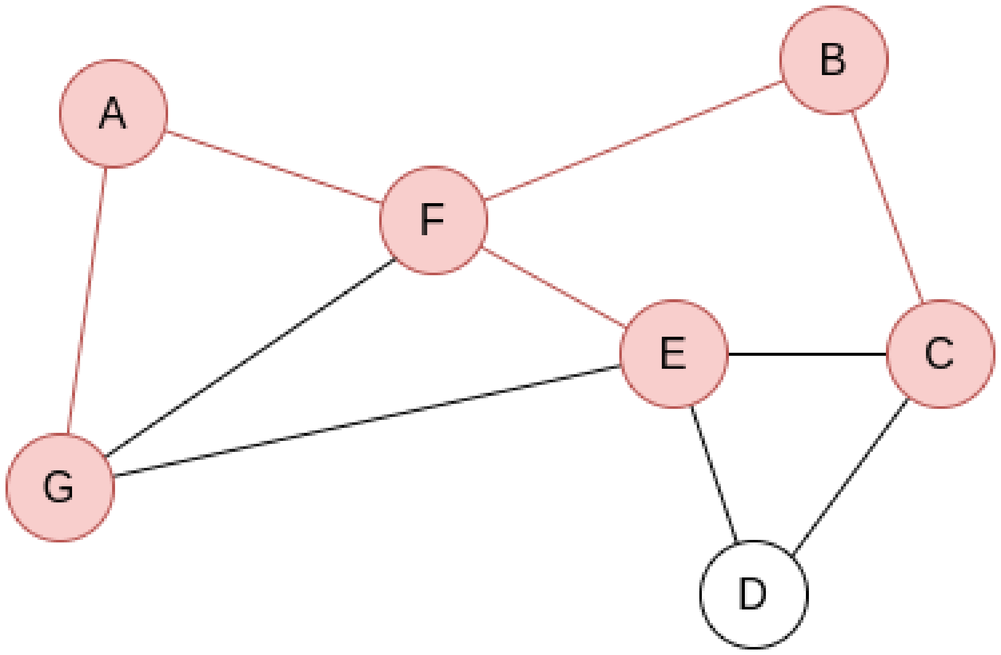


Q=[F,G]. Now, by lexycographical order, we visit F's neighbours:

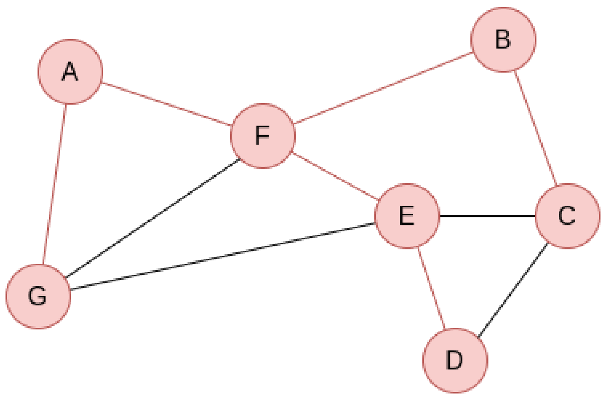


Q=[G,B,E]. Now, we visit G's neighbours. Since it has no new unvisited neighbours, there is no change.

Q=[B,E]. Now, we visit B's neighbours:



Q=[E,C]. Now, we visit E's neighbours:



Q=[C,D]. Everything is visited, so the queue will be slowly emptied!

2.1.2 Depth First Search (DFS)

In the case of DFS, the objective is also to traverse the whole graph. The difference is that in this case we try to go as deep as we can in the graph before visiting more neighbours.

It can be implemented with a stack, let it be a explicit stack, or an implicit one.

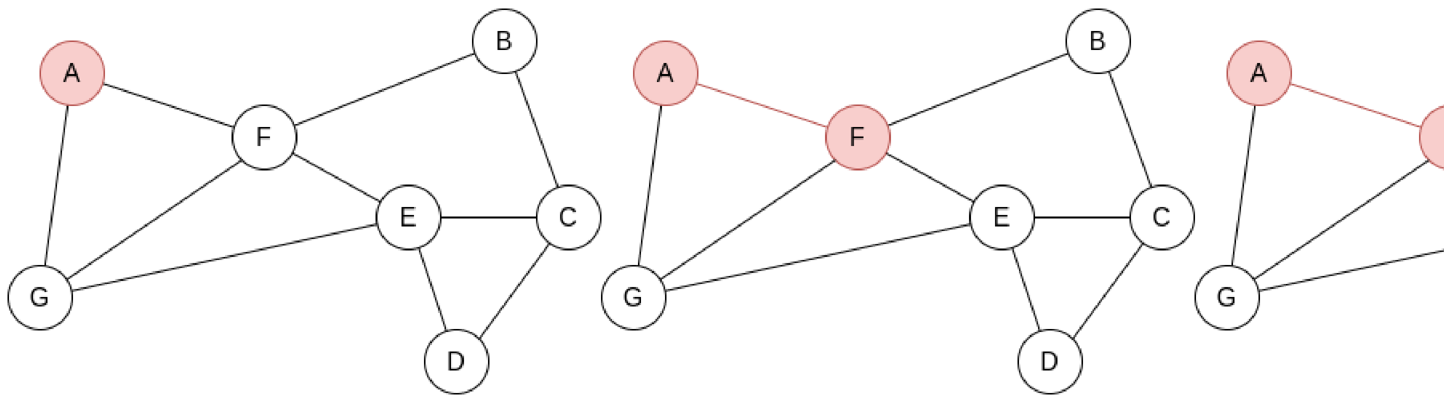
The implementation with an explicit stack is the following:

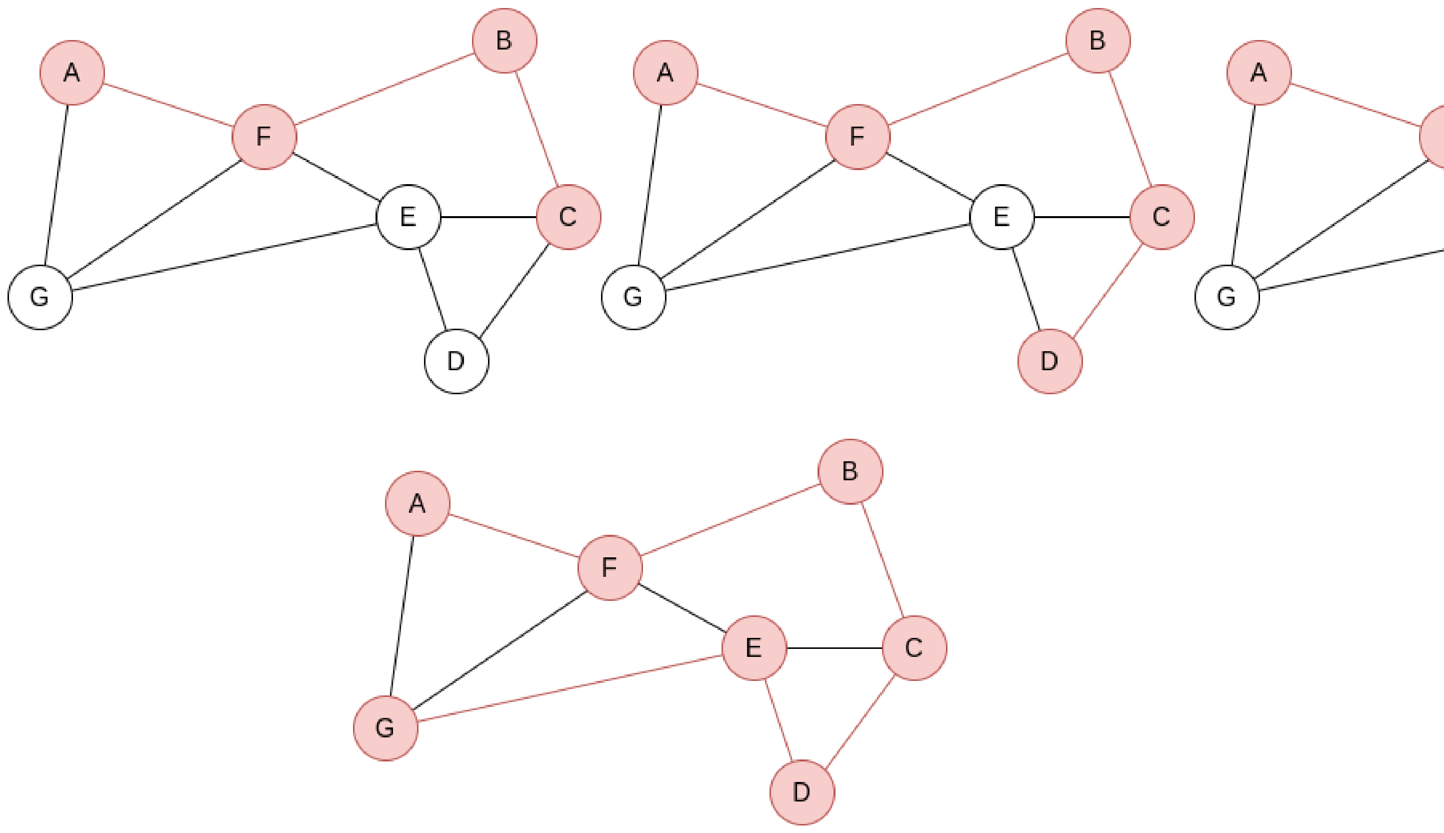
```
procedure DFS(G=(V,E),r)
  S <- emptyset
  push(S,r)
  while S is not empty do
    v <- pop(S)
    if not v.label then
      v.label = true
      for neig in neighbours(v) do
        push(S, neig)
      end for
    end if
  end while
end procedure
```

The implementation with an implicit stack is recursive, and is as follows:

```
procedure BFS(G=(V,E), r)
  Q <- emptyset
  enqueue(Q,r)
  r.label = True

  while Q is not empty do
    v <- dequeue(Q)
    for neig in neighbours(v) do
      if not neig.label then
        enqueue(Q,neig)
        neig.label = True
      end if
    end for
  end whileDFS*(G=(V,E), r)
  r.label = true
  for neig in neighbours(r) do
    if not neig.label then
      DFS*(G, neig)
    end if
  end for
end procedure
```

Example 2.3. Let's repeat the example, now using DFS:





2.1.3 Greaph Representations

A graph, $G = (V, E)$, with n vertices and m edges can be encoded using different structures:

- **Adjacency matrix**: a matrix $A \in \mathcal{M}_{n \times n}$, defined by

$$A_{ij} = \begin{cases} 1 & if,\ (v_i, v_j) \in E \\ 0 & otherwise \end{cases}.$$

  The adjacency matrix is symmetric for undirected graphs.

- **Adjacency list**: a list L of length n in which each vertex holds a list of its neighbours:

$$\forall u \in V, L_u = \{v | \, (u, v) \in E\}\,.$$

  If G is directed, the choice of the direction depends on the analytic needs.

- **Incidence matrix**: a matrix $B \in \mathcal{M}_{n \times m}$, defined by

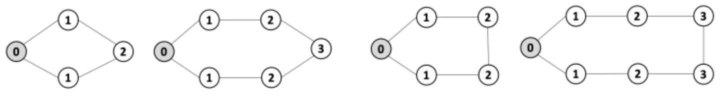
$$B_{ij} = \begin{cases} 1 & if\ e_j = (v_i, v_k) \in E \\ 0 & otherwise \end{cases}.$$

2.1.4 Exercises

1. Using graph traversal algorithms, propose an algorithm that computes the number of edges between a given vertex and all other vertices.

```
procedure n_edges(G=(V,E), r)
  Q <- emptyset
  enqueue(Q,r)
  r.n_edges = 0

  while Q is not empty do
    v <- dequeue(Q)
    for neig in neighbours(v) do
      if not neig.n_edges then
        enqueue(Q,neig)
        neig.n_edges = v.n_edges + 1
      end if
    end for
  end while
end procedure
```

2. Given the following cycles with even and odd lengths (with the distances or depths from the grey vertex), what do you think about the case of graphs with an odd cycle (in number of edges)? Is this a characteristic property? State the general case.



Proposition: a graph contains a cycle C with an odd number of edges if, and only if, $\exists (x, y) \in E | depth\,(x) = depth\,(y)$.

Proof: first, we know that all edges connect vertices of 'neighbouring' depths. That it, $\forall (x, y) \in E$, it holds $|depth\,(x) - depth\,(y)| \leq 1$.

[$\Longrightarrow$] By reduction ad absurdum, seeking a contradiction, suppose that $\forall (x, y) \in C$, with $depth\,(x) \neq depth\,(y)$. This means that $depth\,(x) = depth\,(y) \pm 1$. Therefore, there is, along the cycle, a node of even depth, followed by a node of odd lenght, and so on. When we close the cycle, the final node is the inicial one, so its depth is 0 (even). Therefore, we need an even number of edges, to conserve the parity.

[$\Longleftarrow$] If there is an edge $(x, y) \in E$ with $depth\,(x) = depth\,(y)$, then we can consider the path tree that was used to annotate the depths. In this tree, x and y have a first ancestor z in common, from which we can form an odd cycle of size $2 \cdot (depth\,(x) - depth\,(z)) + 1$ by adding the edge (x, y) to this subtree starting at z.

3. Propose an algorithm that determines if a graph contains an odd cycle.

```
procedure hasOddCycle(G=(V,E))
  v <- a vertex from V
  depths <- n_edges(G,v) #from the first exercise

  for (u,v) in E do
    if depth[u] == depth[v] then
      return True
    end if
  end for
end procedure
```

4. In a bipartite graph, can there be a cycle with an odd number of edges? Is this a characteristic property?

No, it is not possible!

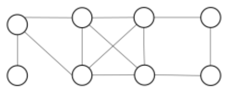
Proposition: A graph is bipartite if, and only if, all cycles are of even size.

[$\Longrightarrow$] If the graph is bipartite, any path alternates between each vertex of each partition to create a cycle ending by the initial vertex. Therefore, all cycles must be of even size.

[$\Longleftarrow$] Consider the partition of vertices with even depth V_1, and the partition of vertices with odd depth V_2.

Since there is no odd cycle, then, from question 2, we know that $\forall (u, v) \in E$ it is $depth\,(u) = depth\,(v) \pm 1$. Therefore, the graph is bipartite.

5. Propose an algorithm that allows to determine if a graph is bipartite. Test your algorithm in the following graph. Is it bipartite? Justify your answer.



The algorithm is the same as in exercise 3, because of exercise 4.

The proposed graph is clearly not bipartite, because there are several odd cycles.

6. Graph coloring is a way of coloring the vertices of a graph in such a way that no two adjacent vertices share the same color. A 2-colorable graph is a graph that can be colored with only 2 colors.

   (a) What is the link with the previous exercise? Justify your answer.

   Proposition: a graph is 2-colorable if, and only if, it is bipartite.

   Proof: [$\Longrightarrow$] If it is 2-colorable, with colors red and blue. Then we take $V_1 = \{u|color\,(u) = blue\}$ and $V_2 = \{u|color\,(u) = red\}$. G is clearly bipartite with this partition.

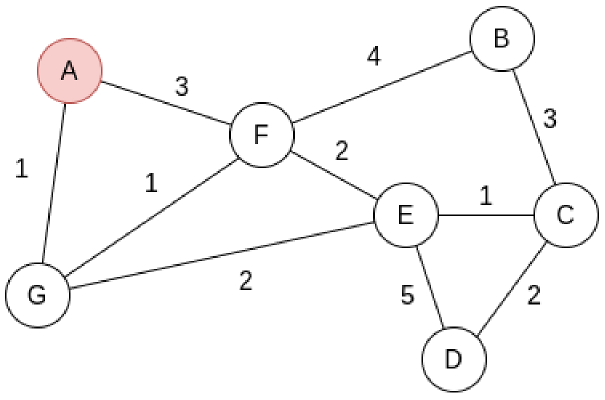
   [$\Longleftarrow$] If it is bipartite, with partition V_1 and V_2, then we can color all nodes in V_1 in blue, and all nodes in V_2 in red. The graph is 2-colorable.

   (b) We want to write an algorithm, inspired by DFS search, which takes as input a graph, $G = (V, E)$, and which returns a pair $(result, color)$ where $result$ is True if the graph is colorable, False otherwise, and $color$ is a dictionary associating a color 0 or 1 to each vertex. This algorithm should stop as soon as possible when the graph is not 2-colorable.

```
procedure coloring(G=(V,E), r)
  color <- {r: 0}
  stack <- emptyset
  push(stack, r)

  while stack is not empty do
    v <- pop(stack)
    for neig in neighbours(v) do
      if neigh is not in color.keys then
        push(stack, neigh)
        color[neig] = 1 - color[v]
      elif color[neig] = color[v] then
        return False, color
      end if
    end for
  end while
  return True, color
end procedure
```

7. Compute the shortest path in the following graph using Dijkstra's algorithm, starting at A:

```
procedure dijkstra(G=(V,E), r)
  dist <- {r:0}
  P <- emptyset

  for v in V-{r} do
    dist[v] = infinity
  end for

  while V-P is not empty do
    w <- select(v in V-P and dist[v]=min_u dist[u])
    P <- P union {w}
    for neig in neighbours(w)-P do
      if dist[w]+weight(neig,w) < dist[neig] then
        dist[neig] <- dist[w]+wight(neig,w)
      end if
    end for
  end while
end procedure
```



We start with: dist =

A	B	C	D	E	F	G
0	∞	∞	∞	∞	∞	∞

, and $P = \emptyset$.

Now, $w = A$ and $P = \{A\}$. We update dist =

A	B	C	D	E	F	G
0	∞	∞	∞	∞	3	1

.

Now, $w = G$ and $P = \{A, G\}$. We update dist =

A	B	C	D	E	F	G
0	∞	∞	∞	3	2	1

.

Now, $w = F$ and $P = \{A, F, G\}$. We update dist =

A	B	C	D	E	F	G
0	6	∞	∞	3	2	1

.

Now, $w = E$ and $P = \{A, E, F, G\}$. We update dist =

A	B	C	D	E	F	G
0	6	4	8	3	2	1

.

Now, $w = C$ and $P = \{A, C, E, F, G\}$. We update dist =

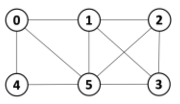
A	B	C	D	E	F	G
0	6	4	6	3	2	1

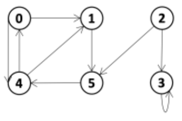
.

Now, $w = B$ and $P = \{A, B, C, E, F, G\}$. dist does not change.

Finally, $w = D$ and $P = \{A, B, C, D, E, F, G\}$. dist does not change.

8. Given the following graphs:





(a) Give the different representations of these graphs.

$$A_1 = \begin{array}{c|cccccc} & 0 & 1 & 2 & 3 & 4 & 5 \\ \hline 0 & 0 & 1 & 0 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 & 1 & 0 & 1 \\ 2 & 0 & 1 & 0 & 1 & 0 & 1 \\ 3 & 0 & 1 & 1 & 0 & 0 & 1 \\ 4 & 1 & 0 & 0 & 0 & 0 & 1 \\ 5 & 1 & 1 & 1 & 1 & 1 & 0 \end{array} , L_1 = \begin{array}{c} 0 : \{1,4,5\} \\ 1 : \{0,2,3,5\} \\ 2 : \{1,3,5\} \\ 3 : \{1,2,5\} \\ 4 : \{0,5\} \\ 5 : \{0,1,2,3,4\} \end{array}$$

$$B_1 = \begin{array}{c|cccccccccc} & 01 & 04 & 05 & 12 & 13 & 15 & 23 & 25 & 35 & 45 \\ \hline 0 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 0 \\ 2 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 1 & 0 & 0 \\ 3 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 1 & 0 \\ 4 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 5 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 1 & 1 & 1 \end{array}$$

$$A_2 = \begin{array}{c} \\ 0 \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{array} \begin{array}{c} \begin{array}{cccccc} 0 & 1 & 2 & 3 & 4 & 5 \end{array} \\ \begin{pmatrix} 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \end{pmatrix} \end{array}, \quad L_2 = \begin{array}{c} 0:\{1,4\} \\ 1:\{5\} \\ 2:\{3,5\} \\ 3:\{3\} \\ 4:\{0,1\} \\ 5:\{4\} \end{array},$$

$$B_1 = \begin{array}{c|ccccccccc} & 01 & 04 & 15 & 23 & 25 & 33 & 40 & 41 & 54 \\ 0 & 1 & 1 & 0 & 0 & 0 & 0 & 2 & 0 & 0 \\ 1 & 2 & 0 & 1 & 0 & 0 & 0 & 0 & 2 & 0 \\ 2 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 \\ 3 & 0 & 0 & 0 & 2 & 0 & 3 & 0 & 0 & 0 \\ 4 & 0 & 2 & 0 & 0 & 0 & 0 & 1 & 1 & 2 \\ 5 & 0 & 0 & 2 & 0 & 2 & 0 & 0 & 0 & 1 \end{array}$$

(b) Compute A^2, A^3. What does A^r_{ij} represents?
A^r_{ij} represents the number of paths of length r from node i to node j.

(c) What is the complexity of A^r? Is it possible to reduce it?
Computing A^r is $O\left(rn^3\right)$, since it requires r products of complexity $O\left(n^3\right)$.
However, we can reuse some results to reduce the complexity:

- If r is even, we can do $A^r = \left(A^{\frac{r}{2}}\right)^2$.
- If r is odd, we can do $A^r = A\left(A^{\frac{r-1}{2}}\right)^2$.

Therefore, we can obtain A^r in $O\left(\log r \cdot n^3\right)$.

2.2 Linear Algebra Preliminaries

A **norm** is a function f that measures the size of a vector. It must satisfy the following properties:

- $f(x) = 0 \iff x = 0$.
- Linear on scale factors:
$$f(\alpha x) = |\alpha| f(x), \forall \alpha \in \mathbb{R}.$$
- Triangle inequality:
$$f(x+y) \leq f(x) + f(y).$$

A widely use family of norms are the p-norms:

$$\|x\|_p = \sqrt[p]{\sum_i |x_i|^p},$$

with the most common one being the Euclidean norm, for $p = 2$:

$$\|x\| = \sqrt{\sum_i x_i^2}.$$

The **determinant** of a square matrix is equal to the hypervolume of the parallelotope defined by the vectors of the matrix. It is 0 if, and only if, the set of vectors is colinear.

The determinant can be used for many things:

- We can represents linear systems with matrices as $Y = AX$, and there are many methods to solve this efficiently.

- With the determinant we can compute the **characteristic polynomial** of A, whose roots are the eigenvalues of A.

Some properties of the determinant are:

- $|I| = 1$, where I is the identity matrix.
- $|A| = 0$ if A is singular (not invertible).
- $|AB| = |A|\,|B|$.
- $|A^T| = |A|$.
- $|cA| = c^n\,|A|$, where n is the dimension of A.

A square matrix, A, is **invertible** (non-singular, non-degenerate), with inverse denoetd A^{-1}, if $\exists B$ such that

$$AB = BA = I,$$

in this case, $A^{-1} = B$.

Proposition 2.1. *For a square matrix, A, the following properties are equivalent:*

- *A is invertible.*
- *All vectors in A are linearly independent.*
- *$|A| \neq 0$.*
- *A^T is invertible.*
- *0 is not an eigenvalue of A.*

Properties of the inverse:

- $\left(A^{-1}\right)^{-1} = A$.
- $\left(A^T\right)^{-1} = \left(A^{-1}\right)^T$.
- $(AB)^{-1} = B^{-1}A^{-1}$.
- $(cA)^{-1} = \frac{1}{c}A^{-1}$ for $c \neq 0$.
- $\left|A^{-1}\right| = \frac{1}{|A|}$.

An **eigenvector** or characteristic vector of a linear transformation, T, is a non-zero vector that changes by a escalar factor, λ, when transformed by T. That is, v is an eigenvector of the linear transformation T if

$$T(v) = \lambda v.$$

There is a direct correspondence between $n \times n$ matrices and linear transformation in the n-dimenstional vector space into itself. That is, every linear transformation T can be represented as a matrix A_T (the matrix depends on the chosen base). Therefore, we can say that A_T has an eigenvector v if

$$A_T v = \lambda v.$$

The scale factors of the eigenvectors are called **eigenvalues**.

We can find the eigenvalues by solving a polynomial function on λ called the **characteristic polynomial** of A_T:

$$(A - \lambda I)\,v = 0.$$

Now, this equation has non-zero solution if, and only if,

$$|A - \lambda I| = 0.$$

Therefore, we can compute $|A - \lambda I|$ and find all values of λ that makes it equal to 0.

Once we have the eigenvalues, we can use them to find the corresponding eigenvectors.

Example 2.4. Compute the eigenvalues and eigenvectors of $A = \begin{pmatrix} 2 & 1 \\ 1 & 2 \end{pmatrix}$.

$$|A - \lambda I| = \begin{vmatrix} 2-\lambda & 1 \\ 1 & 2-\lambda \end{vmatrix} = (2-\lambda)^2 - 1 = \lambda^2 - 4\lambda + 3.$$

This has as solutions

$$\lambda = \frac{4 \pm \sqrt{16-12}}{2} = \frac{4 \pm 2}{2} = 2 \pm 1.$$

Therefore, we have $\lambda_1 = 1, \lambda_2 = 3$.

To find the eigenvectors, we solve

$$Av = \lambda v \iff \begin{cases} 2x + y & = \lambda x \\ x + 2y & = \lambda y \end{cases}.$$

For $\lambda_1 = 1$, it is

$$\begin{cases} 2x + y & = x \\ x + 2y & = y \end{cases} \iff x = -y,$$

so the eigenvector associated to $\lambda_1 = 1$ is

$$v_{\lambda_1} = \begin{pmatrix} t \\ -t \end{pmatrix}.$$

For $\lambda_2 = 3$, it is

$$\begin{cases} 2x + y & = 3x \\ x + 2y & = 3y \end{cases} \iff x = y,$$

so the eigenvector associated to $\lambda_2 = 3$ is

$$v_{\lambda_2} = \begin{pmatrix} t \\ t \end{pmatrix}.$$

We call the **algebraic multiplicity**, t_i, of the eigenvalue λ_i to its multiplicity as root of the characteristic polynomial:

$$P(A) = |A - \lambda I| = (\lambda - \lambda_1)^{t_1} (\lambda - \lambda_2)^{t_2} \cdot \ldots \cdot (\lambda - \lambda_k)^{t_k}.$$

Note that A can have at most n distinct eigenvalues, although some of them may be complex.

Proposition 2.2. *If the eigenvalues of A are all different, then the corresponding eigenvectors are linearly independent.*

The **eigenspace** of an eigenvalue, λ, is the spave generated by the eigenvectors associated to λ.

The dimension of the eigenspace of λ is the **geometric multiplicity** of λ. The geometric multiplicity of an eigenvalue is, at most, its algebraic multiplicity.

Example 2.5. Let's get some eigenspaces:

$A = \begin{pmatrix} -1 & 1 & 0 \\ -4 & 3 & 0 \\ 1 & 0 & 2 \end{pmatrix}$, so

$$\begin{aligned} |A-\lambda I| &= \begin{vmatrix} -1-\lambda & 1 & 0 \\ -4 & 3-\lambda & 0 \\ 1 & 0 & 2-\lambda \end{vmatrix} = (-1-\lambda)(3-\lambda)(2-\lambda) + 4(2-\lambda) \\ &= (2-\lambda)\left[(-1-\lambda)(3-\lambda)+4\right] = (2-\lambda)\left(-3+\lambda-3\lambda+\lambda^2+4\right) \\ &= (2-\lambda)\left(\lambda^2-2\lambda+1\right) \\ &= (2-\lambda)(\lambda-1)^2 . \end{aligned}$$

This has roots $\lambda_1 = 1$, with algebraic multiplicity 2, and $\lambda_2 = 2$, with algebraic multiplicity 1.

Now, we get the eigenvectors associated to them:

$$Av = \lambda v \iff \begin{cases} -x+y & = \lambda x \\ -4x+3y & = \lambda y \\ x+2z & = \lambda z \end{cases} .$$

For λ_1 this is

$$\begin{cases} -x+y & = x \\ -4x+3y & = y \\ x+2z & = z \end{cases} \iff \begin{cases} y & = 2x \\ -4x+3y & = y \\ x & = -z \end{cases} ,$$

so $v_{\lambda_1} = \begin{pmatrix} t \\ 2t \\ -t \end{pmatrix}$, with dimension 1 (it could be 2).

For λ_2 this is

$$\begin{cases} -x+y & = 2x \\ -4x+3y & = 2y \\ x+2z & = 2z \end{cases} \iff \begin{cases} y & = 3x \\ -4x & = -y \\ x+2z & = \lambda z \end{cases} \iff \begin{cases} x = 0 \\ y = 0 \\ 2z & = 2z \end{cases} ,$$

so $v_{\lambda_2} = \begin{pmatrix} 0 \\ 0 \\ t \end{pmatrix}$, with dimension 1 (it could not be differently).

$B = \begin{pmatrix} 4 & 6 & 0 \\ -3 & -5 & 0 \\ -3 & -6 & 1 \end{pmatrix}$, so

$$\begin{aligned} |B-\lambda I| &= \begin{vmatrix} 4-\lambda & 6 & 0 \\ -3 & -5-\lambda & 0 \\ -3 & -6 & 1-\lambda \end{vmatrix} = (4-\lambda)(-5-\lambda)(1-\lambda) + 18(1-\lambda) \\ &= (1-\lambda)\left[(4-\lambda)(-5-\lambda)+18\right] = (1-\lambda)\left(-20-4\lambda+5\lambda+\lambda^2+18\right) \\ &= (1-\lambda)\left(\lambda^2+\lambda-2\right) = (1-\lambda)^2(-2-\lambda) . \end{aligned}$$

This has roots $\lambda_1 = 1$, with algebraic multiplicity 2, and $\lambda_2 = -2$, with algebraic multiplicity 1.

Now, we get the eigenvectors associated to them:

$$Av = \lambda v \iff \begin{cases} 4x+6y & = \lambda x \\ -3x-5y & = \lambda y \\ -3x-6y+z & = \lambda z \end{cases} .$$

For $\lambda_1 = 1$, we have

$$\begin{cases} 4x+6y & = x \\ -3x-5y & = y \\ -3x-6y+z & = z \end{cases} \iff \begin{cases} x & = -2y \\ & \\ z & = z \end{cases}.$$

Therefore, the eigenspace associated to λ_1 is

$$E(\lambda_1) = \left\{ \begin{pmatrix} -2t \\ t \\ 0 \end{pmatrix}, \begin{pmatrix} 0 \\ 0 \\ u \end{pmatrix} \right\}.$$

For $\lambda_2 = 2$, we have

$$\begin{cases} 4x+6y & = -2x \\ -3x-5y & = -2y \\ -3x-6y+z & = -2z \end{cases} \iff \begin{cases} x & = -y \\ & \\ -3y+z & = -2z \end{cases} \iff \begin{cases} x & = -y \\ & \\ y & = z \end{cases}.$$

Thus, the eigenspace associated to $\lambda_2$is

$$E(\lambda_2) = \begin{pmatrix} -t \\ t \\ t \end{pmatrix}.$$

$$C = \begin{pmatrix} 1 & -1 & 0 \\ -1 & 2 & 1 \\ 0 & 1 & 1 \end{pmatrix},$$

$$\begin{aligned} |C-\lambda I| = \begin{vmatrix} 1-\lambda & -1 & 0 \\ -1 & 2-\lambda & 1 \\ 0 & 1 & 1-\lambda \end{vmatrix} &= (1-\lambda)^2(2-\lambda) - 2(1-\lambda) \\ &= (1-\lambda)[(1-\lambda)(2-\lambda)-2] \\ &= (1-\lambda)\left(2-\lambda-2\lambda+\lambda^2-2\right) \\ &= (1-\lambda)\left(\lambda^2-3\lambda\right) \\ &= (1-\lambda)(\lambda-3)\lambda. \end{aligned}$$

$$Cv = \lambda v \iff \begin{cases} x-y & = \lambda x \\ -x+2y+z & = \lambda y \\ y+z & = \lambda z \end{cases}.$$

$\lambda_1 = 0$:

$$\begin{cases} x-y & = 0 \\ -x+2y+z & = 0 \\ y+z & = 0 \end{cases} \iff \begin{cases} x = y \\ y+z & = 0 \end{cases} \iff \begin{cases} x = y \\ y = -z \end{cases},$$

so

$$E(\lambda_1) = \begin{pmatrix} t \\ t \\ -t \end{pmatrix}.$$

$\lambda_2 = 1$:

$$\begin{cases} x-y & = x \\ -x+2y+z & = y \\ y+z & = z \end{cases} \iff \begin{cases} y = 0 \\ -x+z & = \\ z & = z \end{cases} \iff \begin{cases} y = 0 \\ x = z \end{cases},$$

so

$$E(\lambda_2) = \begin{pmatrix} t \\ 0 \\ t \end{pmatrix}.$$

$\lambda_3 = 3$:

$$\begin{cases} x - y &= 3x \\ -x + 2y + z &= 3y \\ y + z &= 3z \end{cases} \iff \begin{cases} y &= -2x \\ y &= 2z \end{cases},$$

so

$$E(\lambda_3) = \begin{pmatrix} -t \\ 2t \\ t \end{pmatrix}.$$

$$D = \begin{pmatrix} 1 & -1 & 4 \\ 3 & 2 & -1 \\ 2 & 1 & -1 \end{pmatrix},$$

$$\begin{aligned} |D - \lambda I| &= \begin{vmatrix} 1-\lambda & -1 & 4 \\ 3 & 2-\lambda & -1 \\ 2 & 1 & -1-\lambda \end{vmatrix} \\ &= (1-\lambda)(2-\lambda)(-1-\lambda) + 12 + 2 - 8(2-\lambda) + 1 - \lambda + 3(-1-\lambda) \\ &= \left(2 - 3\lambda + \lambda^2\right)(-1-\lambda) - 4 + 4\lambda \\ &= -2 - 2\lambda + 3\lambda + 3\lambda^2 - \lambda^2 - \lambda^3 - 4 + 4\lambda \\ &= -\lambda^3 + 2\lambda^2 + 5\lambda - 6 \end{aligned}$$

To obtain the roots, we can use Ruffini:

	-1	2	5	-6
1		-1	1	6
	-1	1	6	0

So $\lambda_1 = 1$ is a root and we have now $-\lambda^2 + \lambda + 6 = 0$, obtaining

$$\lambda = \frac{-1 \pm \sqrt{1+24}}{-2} = \frac{-1 \pm 5}{-2},$$

and we get $\lambda_2 = -2$ and $\lambda_3 = 3$.

$$Dv = \lambda v \iff \begin{cases} x - y + 4z &= \lambda x \\ 3x + 2y - z &= \lambda y \\ 2x + y - z &= \lambda z \end{cases}.$$

$\lambda_1 = 1$:

$$\begin{cases} x - y + 4z &= x \\ 3x + 2y - z &= y \\ 2x + y - z &= z \end{cases} \iff \begin{cases} y &= 4z \\ 3x + 3z &= 0 \\ 2x + 2z &= 0 \end{cases} \iff \begin{cases} y = 4z \\ x = -z \end{cases}.$$

Then, $E(\lambda_1) = \begin{pmatrix} -t \\ 4t \\ t \end{pmatrix}$.

$\lambda_2 = -2$:

$$\begin{cases} x - y + 4z &= -2x \\ 3x + 2y - z &= -2y \\ 2x + y - z &= -2z \end{cases} \iff \begin{cases} 3x - y + 4z &= 0 \\ 3x + 4y - z &= 0 \\ 2x + y + z &= 0 \end{cases} \iff \begin{cases} 5y - 5z &= 0 \\ 2x + y + z &= 0 \end{cases}$$

$$\iff \begin{cases} y = z \\ 2x + 2y &= 0 \end{cases} \iff \begin{cases} y = z \\ x = -y \end{cases}.$$

Then, $E\left(\lambda_2\right) = \begin{pmatrix} -t \\ t \\ t \end{pmatrix}$.

$\lambda_3 = 3$:

$$\begin{cases} x - y + 4z & = 3x \\ 3x + 2y - z & = 3y \\ 2x + y - z & = 3z \end{cases} \iff \begin{cases} -2x - y + 4z & = 0 \\ 3x - y - z & = 0 \\ 2x + y - 4z & = 0 \end{cases} \iff \begin{cases} -2x - y + 4z & = 0 \\ 5x - 5z & = 0 \end{cases}$$

$$\begin{cases} y = 2z \\ x = z \end{cases}.$$

Then, $E\left(\lambda_3\right) = \begin{pmatrix} t \\ 2t \\ t \end{pmatrix}$.

$E = \begin{pmatrix} 6 & -2 & 2 \\ -2 & 3 & -1 \\ 2 & -1 & 3 \end{pmatrix}$,

$$\begin{aligned} |E - \lambda I| = & \begin{vmatrix} 6-\lambda & -2 & 2 \\ -2 & 3-\lambda & -1 \\ 2 & -1 & 3-\lambda \end{vmatrix} \\ & = (6-\lambda)(3-\lambda)^2 + 4 + 4 - 4(3-\lambda) - (6-\lambda) - 4(3-\lambda) \\ & = (6-\lambda)\left(9 - 6\lambda + \lambda^2\right) + 2 - 8(3-\lambda) + \lambda \\ & = 54 - 36\lambda + 6\lambda^2 - 9\lambda + 6\lambda^2 - \lambda^3 - 22 + 8\lambda + \lambda \\ & = -\lambda^3 + 12\lambda^2 - 36\lambda + 32. \end{aligned}$$

Again, we can use the Ruffini rule:

	-1	12	-36	32
2		-2	20	-32
	-1	10	-16	0

So $\lambda_1 = 2$ is a root, and we now have $-\lambda^2 + 10\lambda - 16 = 0$, which gives us

$$\lambda = \frac{-10 \pm \sqrt{100 - 64}}{-2} = \frac{-10 \pm 6}{-2} = 5 \pm 3.$$

Therefore, λ_1 is a double root and the other root is $\lambda_2 = 8$.

$$Ev = \lambda v \iff \begin{cases} 6x - 2y + 2z & = \lambda x \\ -2x + 3y - z & = \lambda y \\ 2x - y + 3z & = \lambda z \end{cases}.$$

$\lambda_1 = 2$:

$$\begin{cases} 6x - 2y + 2z & = 2x \\ -2x + 3y - z & = 2y \\ 2x - y + 3z & = 2z \end{cases} \iff \begin{cases} 4x - 2y + 2z & = 0 \\ -2x + y - z & = 0 \\ 2x - y + z & = 0 \end{cases} \iff \begin{cases} 4x - 2y + 2z & = 0 \\ 2x - y + z & = 0 \end{cases}$$

$$\iff 2x - y + z = 0$$

If $x = 0$: $y = z$.

If $x = t \neq 0$: $-y + z = -2t$, working for $y = t$ and $z = -t$.

So

$$E(\lambda_1) = \left\{ \begin{pmatrix} 0 \\ t \\ t \end{pmatrix}, \begin{pmatrix} t \\ t \\ -t \end{pmatrix} \right\}.$$

$\lambda_2 = 8$:

$$\begin{cases} 6x - 2y + 2z & = 8x \\ -2x + 3y - z & = 8y \\ 2x - y + 3z & = 8z \end{cases} \iff \begin{cases} -2x - 2y + 2z & = 0 \\ -2x - 5y - z & = 0 \\ 2x - y - 5z & = 0 \end{cases} \iff \begin{cases} -3y - 3z & = 0 \\ 2x - y - 5z & = 0 \end{cases}$$

$$\iff \begin{cases} y = -z \\ 2x - 4z & = 0 \end{cases} \iff \begin{cases} y = -z \\ x = 2z \end{cases}.$$

Therefore,

$$E(\lambda_2) = \begin{pmatrix} 2t \\ -t \\ t \end{pmatrix}.$$

$$F = \begin{pmatrix} 0 & -1 & -1 \\ 1 & 2 & 1 \\ 1 & 1 & 2 \end{pmatrix},$$

$$\begin{aligned} |F - \lambda I| &= \begin{vmatrix} -\lambda & -1 & -1 \\ 1 & 2-\lambda & 1 \\ 1 & 1 & 2-\lambda \end{vmatrix} \\ &= -\lambda(2-\lambda)^2 \cancel{-2+2-\lambda+\lambda} + 2 - \lambda \\ &= (2-\lambda)\left[-\lambda(2-\lambda)+1\right] \\ &= (2-\lambda)\left(-2\lambda+\lambda^2+1\right) \\ &= (2-\lambda)(1-\lambda)^2. \end{aligned}$$

One root is $\lambda_1 = 1$ with algebraic dimension 2, and $\lambda_2 = 2$ with algebraic dimension 1.

$$Fv = \lambda v \iff \begin{cases} -y - z & = \lambda x \\ x + 2y + z & = \lambda y \\ x + y + 2z & = \lambda z \end{cases}.$$

$\lambda_1 = 1$:

$$\begin{cases} -y - z & = x \\ x + 2y + z & = y \\ x + y + 2z & = z \end{cases} \iff \begin{cases} x + y + z & = 0 \end{cases}$$

If $x = 0$: $y = -z$.

If $x = t \neq 0$: $y + z = -t$. This works for $y = t, z = -2t$.

Therefore,

$$E(\lambda_1) = \left\{ \begin{pmatrix} 0 \\ t \\ -t \end{pmatrix}, \begin{pmatrix} t \\ t \\ -2t \end{pmatrix} \right\}.$$

$\lambda_2 = 2$:

$$\begin{cases} -y - z & = 2x \\ x + 2y + z & = 2y \\ x + y + 2z & = 2z \end{cases} \iff \begin{cases} -y - z & = 2x \\ x + z & = 0 \\ x + y & = 0 \end{cases} \iff \begin{cases} x = -z \\ x = -y \end{cases}.$$

$$\iff 2x - y + z = 0$$

So

$$E(\lambda_2) = \begin{pmatrix} t \\ -t \\ -t \end{pmatrix}.$$

Another way to represent eigenvalues and eigenvectors is

$$AV = V\Lambda,$$

where $V = [v_1, ..., v_n]$ is the matrix formed by putting each eigenvector as a column, and

$$\Lambda = \begin{pmatrix} \lambda_1 & & & \\ & \lambda_2 & & \\ & & \ddots & \\ & & & \lambda_n \end{pmatrix}$$

is the diagonal matrix formed by all eigenvalues.

A matrix A is **diagonalizable** if there exist n linearly independent eigenvectors. That is, if the matrix V is invertible:

$$\Lambda = V^{-1}AV.$$

This leads naturally to the **eigen-decomposition** of the matrix,

$$A = V\Lambda V^{-1}.$$

A real matrix, U, is **orthogonal** if $U^TU = UU^T = I$.

Proposition 2.3. *The following statements are equivalent:*

- U^T *is orthogonal.*
- $U^T = U^{-1}$.
- $|U| = 1$.
- *U's eigenvectors are orthonormal (the pairwise dot product is 0 and the norm is 1).*

Example 2.6. Some examples of orthogonal matrices:

- Identity: I
- Permutation of coordinates: $\begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix}$
- Rotation: $\begin{pmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{pmatrix}$.
- Reflection: $\begin{pmatrix} \cos\theta & \sin\theta \\ \sin\theta & -\cos\theta \end{pmatrix}$.

A matrix A is said to be **positive semi-definite** when it can be obtained as the product of a matrix by its transpose:

$$\exists X | A = XX^T.$$

Positive semi-definite matrices are always symmetric, because

$$A^T = \left(XX^T\right)^T = XX^T = A.$$

A symmetric matrix A is positive semi-definite if all its eigenvalues are non-negative.

Proposition 2.4. *Let A be a positive semi-definite matrix. Then:*

- $0 \leq \lambda_1 \leq \lambda_2 \leq ... \leq \lambda_n$ *and its eigenvectors are pairwise orthogonal when their eigenvalues are different.*
- *The eigenvalues are composed of real values.*
- *The multiplicity of an eigenvalue is the dimension of its eigenspace.*

In this case, since eigenvectors are orthogonal, it is possible to store all the eigenvectors in an orthogonal matrix.

Therefore, the eigen-decomposition of a positive semi-definite matrix, A, could be

$$A = U \Lambda U^T,$$

with U an orthogonal matrix.

As a consequence, the eigen-decomposition of a positive semi-definite matrix is often referred to as its diagonalization.

An alternative definition for positive semi-definite matrix is:

A is positive semi-definite if $x^T A x \geq 0, \forall x$.

If it is $x^T A x > 0, \forall x$, then it is positive definite.

If it is $x^T A x \leq 0, \forall x$, then it is negative semi-definite.

If it is $x^T A x < 0, \forall x$, then it is negative definite.

The **rank** of a matrix is the dimension of the vector space generated by its columns (or rows). This corresponds to the maximum number of linearly independent columns of A. A matrix whose rank is equal to its size is called a **full rank matrix**. Only full rank matrices have an inverse.

Proposition 2.5. *The sum of the eigenvalues of a matrix is the sum of the elements of its main diagonal. The product of the eigenvalues is equal to the determinant of the matrix.*

We can now define the **Laplacian matrix** for undirected graphs, as

$$L_{ij} = \begin{cases} -1 & ,(v_i, v_j) \in E \\ 0 & ,(v_i, v_j) \notin E \\ d_i & ,i = j \end{cases},$$

or, equivalently,

$$L = D - A,$$

where A is the degree is the matrix of G, and A its adjacency matrix.

2.2.1 Exercises

1. What could you say about these matrices?

   (a) $A = \begin{pmatrix} -1 & \frac{3}{2} \\ 1 & -1 \end{pmatrix}$, $det\,(A) = -\frac{1}{2}$, A is invertible. Its eigenvalues are $\lambda_1 = -1 + \frac{\sqrt{6}}{2}$ and $\lambda_2 = -1 - \frac{\sqrt{6}}{2}$, with $v_{\lambda_1} = \begin{pmatrix} \frac{\sqrt{6}}{2} t \\ t \end{pmatrix}$ and $v_{\lambda_2} = \begin{pmatrix} -\frac{\sqrt{6}}{2} t \\ t \end{pmatrix}$.

   (b) $B = \begin{pmatrix} -1 & \frac{3}{2} \\ \frac{2}{3} & -1 \end{pmatrix}$. The second row is equal to the first row multiplied by $-\frac{2}{3}$. Therefore, it is not invertible.

(c) I: its determinant is 1. It is symmetric, orthogonal, its own inverse. Triple eigenvalue 1, with eigenspace the whole space.

2. Show that $A^n = X\Lambda X^{-1}$.

First, this is only true if A is diagonalizable. If that is the case, then we can proceed by induction on n:

$n = 1$: Obvious.

$n = 2$:
$$A^2 = \left(X\Lambda X^{-1}\right)^2 = X\Lambda X^{-1}X\Lambda X^{-1} = X\Lambda^2 X^{-1}.$$

Suppose it is true for $n-1$:
$$A^{n-1} = X\Lambda^{n-1}X^{-1}.$$

Then, for n, we have:
$$A^n = AA^{n-1} = X\Lambda X^{-1}X\Lambda^{n-1}X^{-1} = X\Lambda^n X^{-1}.$$

3. Find the eigenvalues and unit eigenvectors of A^TA and AA^T with $A = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}$ the Fibonnaci matrix.

First of all, notice that A is symmetric, so $A^TA = AA^T = A^2 = \begin{pmatrix} 2 & 1 \\ 1 & 1 \end{pmatrix}$.

$\begin{vmatrix} 2-\lambda & 1 \\ 1 & 1-\lambda \end{vmatrix} = (2-\lambda)(1-\lambda) - 1 = 2 - 3\lambda + \lambda^2 - 1 = \lambda^2 - 3\lambda + 1$. The roots of this polynomial are
$$\lambda = \frac{3 \pm \sqrt{9-4}}{2} = \frac{3 \pm \sqrt{5}}{2}.$$

Now,
$$A^2 v = \lambda v \iff \begin{cases} 2x + y & = \lambda x \\ x + y & = \lambda y \end{cases} \iff \begin{cases} x = (\lambda - 1)\, y \end{cases}$$

Therefore
$$E(\lambda_1) = \begin{pmatrix} \frac{1+\sqrt{5}}{2}t \\ t \end{pmatrix}$$
with unit eigenvector $v_1 = \frac{1}{\sqrt{4-\sqrt{5}}}\begin{pmatrix} \frac{1+\sqrt{5}}{2} \\ 1 \end{pmatrix}$.

And
$$E(\lambda_2) = \begin{pmatrix} \frac{1-\sqrt{5}}{2}t \\ t \end{pmatrix}$$
with unit eigenvector $v_2 = \frac{1}{\sqrt{4-\sqrt{5}}}\begin{pmatrix} \frac{1-\sqrt{5}}{2} \\ 1 \end{pmatrix}$.

4. Without multiplying
$$S = \begin{pmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{pmatrix}\begin{pmatrix} 2 & 0 \\ 0 & 5 \end{pmatrix}\begin{pmatrix} \cos\theta & \sin\theta \\ -\sin\theta & \cos\theta \end{pmatrix},$$
find the determinant, the eigenvalues and eigenvectors. Why S is positive definite?

We have $S = U\Lambda U^T$ with U orthogonal. Therefore, the eigenvalues of S are 2 and 5. Its determinant is 10. The eigenvectors are the eigenvectors of Λ rotated as well, that is:
$$V = \begin{pmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{pmatrix}\begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}\begin{pmatrix} \cos\theta & \sin\theta \\ -\sin\theta & \cos\theta \end{pmatrix}.$$

S is positive definite because
$$xSx^T = x\begin{pmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{pmatrix}\begin{pmatrix} 2 & 0 \\ 0 & 5 \end{pmatrix}\begin{pmatrix} \cos\theta & \sin\theta \\ -\sin\theta & \cos\theta \end{pmatrix}x^T,$$

now note that

$$\left(x\begin{pmatrix}\cos\theta & -\sin\theta\\ \sin\theta & \cos\theta\end{pmatrix}\right)^T = \begin{pmatrix}\cos\theta & \sin\theta\\ -\sin\theta & \cos\theta\end{pmatrix}x^T,$$

so

$$xSx^T = y\begin{pmatrix}2 & 0\\ 0 & 5\end{pmatrix}y^T \geq 0,$$

because $\begin{pmatrix}2 & 0\\ 0 & 5\end{pmatrix}$ is positive semi-definite (symmetric with positive eigenvalues).

5. For what numbers c and d are the following matrices positive definite?

   (a) $A = \begin{pmatrix}c & 1 & 1\\ 1 & c & 1\\ 1 & 1 & c\end{pmatrix}$: all principal minors must be positive. That is:

   - $c > 0$.
   - $\begin{vmatrix}c & 1\\ 1 & c\end{vmatrix} = c^2 - 1 > 0$. Combined with the previous one, this is $c > 1$.
   - $\begin{vmatrix}c & 1 & 1\\ 1 & c & 1\\ 1 & 1 & c\end{vmatrix} = c^3 + 2 - 3c$. Roots: 1, $\begin{array}{c|cccc} & 1 & 0 & -3 & 2\\ 1 & & 1 & 1 & -2\\ \hline & 1 & 1 & -2 & 0\end{array}$, and we have $c^2 + c - 2$, with roots $c = \frac{-1\pm\sqrt{5}}{2}$. We are only interested in the interval $(1, \infty)$, in which $c^3 - 3c + 2 > 0$.

   Therefore, it is $c > 1$.

   (b) $B = \begin{pmatrix}1 & 2 & 3\\ 2 & d & 4\\ 3 & 4 & 5\end{pmatrix}$:

   - $1 > 0$.
   - $\begin{vmatrix}1 & 2\\ 2 & d\end{vmatrix} = d - 4 > 0 \iff d > 4$.
   - $\begin{vmatrix}1 & 2 & 3\\ 2 & d & 4\\ 3 & 4 & 5\end{vmatrix} = 5d + 24 + 24 - 9d - 16 - 20 = -4d + 12 > 0 \iff -4d > -12 \iff d < 3$.

   Therefore, there is no value for d for which B is positive.

6. Show that if $\lambda_1, \lambda_2, ..., \lambda_n$ are the eigenvalues of a matrix A, then A^m has as eigenvalues $\lambda_1^m, \lambda_2^m, ..., \lambda_n^m$.

   Induction on m.

   $m = 1$: Obvious.

   $m = 2$: Let v_i be the eigenvector associated to λ_i, then

   $$A^2 v_i = A(Av_i) = A(\lambda_i v_i) = \lambda_i A v_i = \lambda_i^2 v_i,$$

   so λ_i^2 is an eigenvalue of A^2, with associated eigenvector v_i.

   Suppose it is true for $m - 1$, then, for m:

   $$A^m v_i = A\left(A^{m-1}v_i\right) = A\left(\lambda_i^{m-1}v_i\right) = \lambda_i^{m-1}Av_i = \lambda_i^m v_i,$$

   and we have the result.

7. What is the determinant of any orthogonal matrix?

   If U is orthogonal, then $UU^T = I$. Then,

   $$1 = |I| = \left|UU^T\right| = |U|\left|U^T\right| = |U|^2.$$

   Therefore, $|U| = \pm 1$.

8. For an undirected graph, both the adjacency matrix and the Laplacian matrix are symmetric. Show that the Laplacian matrix is positive semi-definite.

3 Random Walks on Graphs

3.1 First Perron-Frobenius Theorem

Definition 3.1. A matrix, A, is **positive** if $A_{ij} > 0, \forall i, j$. Similarly, it is **non-negative** if $A_{ij} \geq 0, \forall i, j$. Similar definitions apply for negative and non-positive matrices.

Remark 3.1. Observe that it is not the same for a matrix to be positive as to be positive semi-definite.

The Perron-Frobenius theorem for non-negative matrices leads to the characterization of non-negative primary eigenvectors. This is useful in stationary distributions, such as those of Markov chains and the famous Google's page rank algorithm.

Theorem 3.1. ***Perron-Frobenius Theorem for positive matrices***
If A is a positive matrix, then:

- $\exists \lambda^* > 0, v^* > 0, \|v^*\|_2 = 1$ *such that* $A \cdot v = \lambda^* v^*$ *(v^* is a right column eigenvector).*
- $\exists \lambda^* > 0, w > 0, \|w\|_2 = 1$ *such that* $w \cdot A = \lambda^* w$ *(w is a left row eigenvector).*
- *For any other eigenvalue, λ, it holds, $|\lambda| < \lambda^*$ (λ^* is a dominant eigenvalue, called the **Perron eigenvalue**).*
- *λ^* is unique and v^* is unique (the only vector of unit length associated to λ^*).*

Definition 3.2. A non-negative matrix A is:

- **Irreducible** if, $\forall i, j, \exists k \in \mathbb{N}^*$ such that $A^k_{i,j} > 0$.
- **Primitive** if, $\exists k \in \mathbb{N}^*$ such that $\forall i, j,\ A^k_{i,j} > 0$.

Theorem 3.2. ***Perron-Frobenius Theorem for non-negative matrices***
If A is a non-negative matrix, then:

- $\exists \lambda^* > 0, v^* \geq 0, \|v^*\|_2 = 1$ *such that* $A \cdot v = \lambda^* v^*$ *(v^* is a right column eigenvector).*
- $\exists \lambda^* > 0, w \geq 0, \|w\|_2 = 1$ *such that* $w \cdot A = \lambda^* w$ *(w is a left row eigenvector).*
- *For any other eigenvalue, λ, it holds, $|\lambda| \leq \lambda^*$ (λ^* is a dominant eigenvalue, called the **Perron eigenvalue**).*
- *If A is irreducible, then the vector v^* is unique and it holds $v^* > 0$.*
- *If A is primitive, then the eigenvalue λ^* is unique.*

Note now that a graph, $G = (V, E)$, with adjacency matrix A, then: G is connected $\iff \forall 1 \leq i, j \leq |V|, \exists k \in \mathbb{N}^*$ such that $A^k_{i,j} > 0$. This means that the adjacency matrix of connected graphs is irreducible.

Now, if a graph is k-connected, i.e., there is a k-path between all nodes, then its adjacency matrix is primitive. One sufficient condition for a graph to be k-connected is being connected and having $A_{ii} > 0$ for some i.

3.2 Random Walks on Graphs

A **random walk** on a graph, $G = (V, E)$, is a random process that starts from some vertex v_i, and repeatedly moves to a neighbour v_j chosen at random (for example with uniform distribution). The random walk, ξ_t, is therefore a random variable describing the position of a random walk after t steps. The probability of going from node i to node j is the **transition probability**,

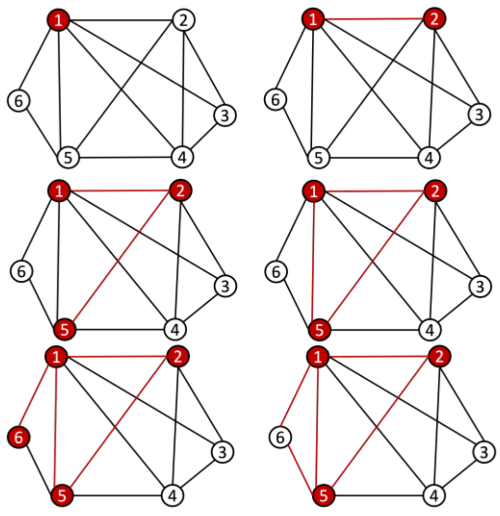
$$P_{ij} = P\left(\xi_{t+1} = j | \xi_t = i\right).$$

The sequence of nodes can be regarded as a Markov chain, i.e. a discrete time stochastic process, where the position ξ_0 is the initial state, according to the **init distribution**, P^0, and from this point the next state only depends on the current state. The t**-step transition probability** is

$$P_{ij}^t = P\left(\xi_t = j | \xi_0 = i\right).$$

Some examples are the path traced by a molecule in a liquid or a gas (Brownian motion), the price of a fluctuating stock, the financial status of a gambler, etc. The term random walk was first introduced by Karl Pearson in 1905.

The following is a basic visual example of a random walk on a graph:



Note that we can express the transition probability P_{ij} in a matrix P. This matrix is the **transition probabilities matrix**, and it is **row-stochastic** or **row-Markov**, meaning,

$$P_{ij} \geq 0, \forall i, j, \text{ and } \sum_j P_{i,j} = 1, \forall i.$$

This implies that

$$P \cdot 1 = P \cdot \begin{pmatrix} 1 \\ \vdots \\ 1 \end{pmatrix} = \begin{pmatrix} 1 \\ \vdots \\ 1 \end{pmatrix}.$$

This means that $\begin{pmatrix} 1 \\ \vdots \\ 1 \end{pmatrix}$ is an eigenvector and 1 is an eigenvalue. 1 is the largest eigenvalue because

$$\|Pv\|_1 \leq \|v\|_1,$$

so, for an eigenvalue λ,

$$|\lambda| \, \|v\|_1 = \|\lambda v\|_1 = \|Pv\|_1 \leq \|v\|_1,$$

so $|\lambda| \leq 1$.

From the Perron-Frobenius theorem for non-negative matrices, we know that:

- $v^* = \begin{pmatrix} 1 \\ \vdots \\ 1 \end{pmatrix}$ is a right Perron eigenvector for P.
- $|\lambda| \leq \lambda^* = 1$ is a Perron eigenvalue.
- There exists a left Perron eigenvector $\pi P = \pi$.
- If P is irreducible, the vector π is unique.
- If P is primitive, the eigenvalue 1 is unique (there are no complex eigenvalues with norm 1).

3.2.1 The Stationary Distribution

Let π^t be the row vector giving the probability distribution of ξ_t, that is, π_i^t is the probability that the random walk is at node i at time t. Therefore, we can write

$$\pi^{t+1} = \pi^t P,$$

which, applied recursively, leads to

$$\pi^{t+1} = \pi^0 P^{t+1}.$$

Or, we can take limits

$$\lim_t \pi^{t+1} = \lim_t \pi^t P.$$

If this limit exists, $\lim_t \pi^t = \pi$, then

$$\pi = \pi P.$$

Convergence is ensured if P is irreducible.

Example 3.1. The following example does not converge:

$$P = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$$

A common way to perform random walks on graphs is with the uniform probability. That is,

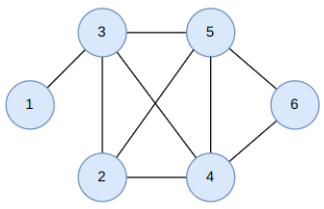
$$P_{ij} = P\left(\xi_{t+1} = j | \xi_t = i\right) = \begin{cases} \frac{1}{d_i} & if \;\; (i,j) \in E, \\ 0 & otherwise, \end{cases}$$

where d_i is the degree of node i. Equivalently,

$$P_{ij} = \frac{A_{ij}}{\sum_{j \in V} A_{ij}} = \frac{A_{ij}}{d_i} = D_{ii}^{-1} A_{ij}.$$

The random sequence of vertices $\xi_0, \xi_1, ..., \xi_t, \xi_{t+1}, ...$ visited on G is a Markov Chain with state space V and matrix transition probabilite $P = D^{-1}A$.

Example 3.2. Given the graph:



The transition matrix for the uniform distribution is:

$$P = \begin{pmatrix} 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & \frac{1}{3} & \frac{1}{3} & \frac{1}{3} & 0 \\ \frac{1}{4} & \frac{1}{4} & 0 & \frac{1}{4} & \frac{1}{4} & 0 \\ 0 & \frac{1}{4} & \frac{1}{4} & 0 & \frac{1}{4} & \frac{1}{4} \\ 0 & \frac{1}{4} & \frac{1}{4} & \frac{1}{4} & 0 & \frac{1}{4} \\ 0 & 0 & 0 & \frac{1}{2} & \frac{1}{2} & 0 \end{pmatrix}$$

3.2.2 Balance Condition

A probability distribution π satisfies the **balance condition** if

$$\pi_i P_{ij} = \pi_j P_{ji}, \forall i, j \in V.$$

If π satisfies the balance condition, then it is the stationary distribution for the undirected graph. To see this, notice that the balance condition can be rewritten as

$$\pi_i \frac{A_{ij}}{d_i} = \pi_j \frac{A_{ji}}{d_j}.$$

Since the graph is considered without direction, $A_{ij} = A_{ji}$, and then

$$\frac{\pi_i}{d_i} = \frac{\pi_j}{d_j} = c,$$

where c is a constant, for all i, j. Now, we know that $\sum_i \pi_i = 1$, so

$$1 = \sum_i \pi_i = \sum_i \frac{\pi_j}{d_i} d_i = \sum_i c d_i = c \sum_i d_i.$$

Therefore

$$\sum_i d_i = \frac{1}{c}.$$

Finally, it must be

$$\pi_i = d_i c = \frac{d_i}{\sum_j d_j} = \frac{d_i}{2\left|E\right|}.$$

In this case:

$$(\pi P)_i = \sum_j \pi_j P_{ji} = \sum_j \pi_j \frac{1}{d_j} A_{ji} = \sum_j c A_{ji} = c \sum_j A_{ji} = c \sum_j A_{ij} = c d_i = \frac{d_i}{\sum_j d_j} = \pi_i.$$

Therefore, we have seen that the stationary probabilities are proportional to the degrees of the vertices.

In particular, if G is d-regular, i.e., all nodes have degree d, then

$$\pi = \frac{d}{2\left|E\right|} = \frac{d}{d \cdot n} = \frac{1}{n}$$

is the uniform distribution. With this setup, a random walk moves along every edge with the same frequence.

The balance condition implies time-reversibility: the reversed walk is also a Markov chain.

3.2.3 Hitting Time

Definition 3.3. The **expected hitting time**, H_{ij}, is the expected number of steps before node j is reached in a random walk starting at node i:

$$H_{ij} = \begin{cases} 1 + \sum_k P_{ik} H_{kj} & \textit{if } i \neq j, \\ 0 & \textit{otherwise.} \end{cases}$$

Remark. In general, $H_{ij} \neq H_{ji}$, so H is not symmetric.

Remark 3.2. H follows the triangle inequality

$$H_{ij} \leq H_{ik} + H_{kj}.$$

Definition 3.4. The **commute time**, C_{ij}, is the expected number of steps in a random walk starting at node i, reaching node j and coming back to i again:

$$C_{ij} = H_{ij} + H_{ji}.$$

3.2.4 Lazy Random Walk

The lazy random walk is a variation of the random walk, in which the walk stays at the current node with probability $\frac{1}{2}$, and continue with the walk with the rest of the probability.

In this case, the transition matrix is

$$P_{ij} = \begin{cases} \frac{1}{2} & if\ i = j, \\ \frac{1}{2d_i} & if\ (i,j) \in E, \\ 0 & otherwise. \end{cases}$$

If Q is the transition matrix for the uniform random walk, then

$$\pi^{t+1} = \pi^t P = \frac{1}{2}\pi^t + \frac{1}{2}\pi^t Q.$$

Proposition 3.1. *If the lazy random walk converges and the uniform random walk is irreducible, then it converges to the same stationary distribution as the uniform random walk.*

Proof. Let Q be the transition matrix for the uniform random walk, then, the stationary distribution is

$$\pi = \pi Q.$$

For lazy random walk, say the stationary distribution is π'. Then:

$$\pi' = \frac{1}{2}\pi' + \frac{1}{2}\pi' Q \iff \frac{1}{2}\pi' = \frac{1}{2}\pi' Q \iff \pi' = \pi' Q.$$

Therefore, since Q is irreducible, the uniqueness of π implies $\pi' = \pi$. □

3.3 PageRank

The web is very heterogeneous bu nature, and certainly huge. We cannot expect the web graph to be connected. Page and Brin proposed a way to overcome this problem, by ensuring the convergence of random walks on the web graph.

The idea is to fix a positive constant, p, between 0 and 1, called the **damping factor**, and which represents the probability that a user leaves the current page and goes to a random web.

Therefore, the **page rank transition matrix** is

$$P_g = (1-p)\,P + pB,$$

where $B = \frac{1}{n}\begin{pmatrix} 1 & 1 & \dots & 1 \\ 1 & 1 & \dots & 1 \\ \vdots & \vdots & \ddots & \vdots \\ 1 & 1 & \dots & 1 \end{pmatrix}$.

p is usually chosen small, like 0.15, modelling a situation in which a surfer will, most of the time, follow the outgoing links and move on to one of the neighbours. A smaller percentage of time, the surfer will dump the current page and choose arbritrarily a different page from the web.

Proposition 3.2. *P_g is stochastic.*

Proof. We need to proof that, for all i, it holds $\sum_j P_{g_{i,j}} = 1$.

$$\begin{aligned}\sum_j P_{g_{i,j}} =& \sum_j (1-p) P_{ij} + pB_{ij} \\ =& (1-p)\sum_j P_{ij} + p\sum_j B_{ij} \\ =& (1-p)\cdot 1 + p\sum_j \frac{1}{n} \\ =& 1 - p + p\cdot n\frac{1}{n} \\ =& 1 - p + p \\ =& 1.\end{aligned}$$

□

4 Centrality Measures

Centrality Measures try to answer the question *'What characterizes an important vertex?'*. They define a real-valued function on the vertices of the graph, $m : V \to \mathbb{R}$, that serves to rank the vertices. However, there are many different ways to define such function, leading to different definitions of centrality, such as cohesiveness, ability to transfer information across the network, to influence other nodes, to control information flow, etc.

There are many centrality measures that count the number of paths through a given vertex. These differ in how relevant walks are defined and counted. For example, if we only consider paths of length one, we would be computing degree centrality, while if we allow paths of arbitrary length, we would be computing eigenvalue centrality.

4.1 Degree Centrality

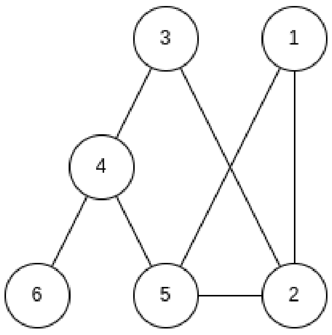
The more neighbours a vertex has, the higher its communication ability is, increasing its importance.

Definition 4.1. Given the graph $G = (V, E)$, with adjacency matrix A, the **degree centrality** is computed as

$$D = Au,$$

where $u = \mathbf{1} \in \mathbb{R}^n$.

Example 4.1. Consider the following graph:



The degree centrality is

$$D = Au = \begin{pmatrix} 0 & 1 & 0 & 0 & 1 & 0 \\ 1 & 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \end{pmatrix} \begin{pmatrix} 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \end{pmatrix} = \begin{pmatrix} 2 \\ 3 \\ 2 \\ 3 \\ 3 \\ 1 \end{pmatrix},$$

so the nodes with highest value are nodes $(2, 4, 5)$.

One drawback of this measure, is that it is very likely that several nodes present the same exact value, difficulting an unique ranking of vertices.

4.2 Neighbourhood centrality

This measure correspond to the average degree of each vertex neighbours. We could understand this measure as measuring how much a vertex is related to influencial vertices.

Definition 4.2. Given the graph $G = (V, E)$, with adjacency matrix A, the **neighbourhood centrality** is computed as

$$N = \mathcal{D}^{-1}AD,$$

where $\mathcal{D}$ is the diagonal matrix where $\mathcal{D}_{ii} = d_i$ is the degree of vertex i and D is the degree centrality. Each vertex' measure is

$$N_v = \frac{\sum_{u \in \mathcal{N}_v} d_u}{d_v}.$$

Example 4.2. The neighbourhood centrality of the previous example graph is

$$N = \mathcal{D}^{-1}Au = \begin{pmatrix} \frac{1}{2} & & & & & \\ & \frac{1}{3} & & & & \\ & & \frac{1}{2} & & & \\ & & & \frac{1}{3} & & \\ & & & & \frac{1}{3} & \\ & & & & & 1 \end{pmatrix} \begin{pmatrix} 0 & 1 & 0 & 0 & 1 & 0 \\ 1 & 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \end{pmatrix} \begin{pmatrix} 2 \\ 3 \\ 2 \\ 3 \\ 3 \\ 1 \end{pmatrix} = \begin{pmatrix} \frac{3+3}{2} \\ \frac{2+2+3}{3} \\ \frac{3+3}{2} \\ \frac{2+3+1}{3} \\ \frac{2+3+3}{3} \\ \frac{3}{1} \end{pmatrix} = \begin{pmatrix} 3 \\ 2.33 \\ 3 \\ 2 \\ 2.66 \\ 3 \end{pmatrix},$$

so the nodes with highest value are nodes $(1, 3, 6)$.

4.3 Eigenvector Centrality

A natural extension of degree centrality is to consider all reachable nodes, not just neighbours. Eigenvector centrality measures a node's importance while considering the importance of its neighbours. A high eigenvector centrality means that a node is connected to many nodes that have high scores themselves.

Definition 4.3. Given the graph $G = (V, E)$, with adjacency matrix A, the **eigenvector centrality** of node v is

$$E_v = \frac{1}{\lambda} \sum_{u \in \mathcal{N}_v} A_{vu} E_u,$$

where λ is a parameter. Note that this can be written as

$$E = \frac{1}{\lambda} AE,$$

or

$$AE = \lambda E.$$

This means that E is an eigenvector of A, for the eigenvalue λ.

Bonacich suggested that the eigenvector of the largest eigenvalue of A could make a good network centrality measure.

The eigenvector E must be non-negative and according to the Perron-Frobenius theorem, the largest λ enforces this property, making it a suitable value.

Example 4.3. Let's compute E for the previous example graph. The matrix A has as largest eigenvalue $\lambda = 2.54$, and the corresponding eigenvector is

$$E = \begin{pmatrix} 2.5 \\ 3.1 \\ 2.2 \\ 2.5 \\ 3.2 \\ 1 \end{pmatrix}.$$

Note that it is usually unfeasible to compute the eigenvalues and eigenvectors. It is more usual to get the vector iteratively as

$$E_k = A \frac{E_{k-1}}{\|E_{k-1}\|}.$$

4.4 PageRank Centrality

Google's PageRank is a variant of the eigenvector centrality, which uses in-degree to award one centrality point for every link a node receives. As we saw, the algorithm is based on a web surfer who is randomly clicking on links, with a certain probability to go to a different place of the web (the damping factor).

Therefore, we define the matrix

$$P_g = (1 - p) P + pB,$$

where $P_{ij} = \begin{cases} \frac{1}{d_i} & if\ j \in \mathcal{N}_i \\ 0 & otherwise \end{cases}$, and $B_{ij} = \frac{1}{n}$.

Now, we apply the eigenvector centrality to this modified matrix, as

$$P_g E_g = \lambda E_g = E_g,$$

with $\lambda = 1$ because P_g is stochastic.

Or, iteratively as

$$E_{g_k} = P_g E_{g_{k-1}}.$$

Note that in this case it is not necessary to normalize the vector at each step, because P_g is stochastic. A good E_{g_0} is $E_{g_0} = \begin{pmatrix} \frac{1}{n} \\ \vdots \\ \frac{1}{n} \end{pmatrix}$.

4.5 Katz (or alpha) centrality

The main problem with eigenvector centrality is that it only works well when the graph is strongly connected (so Perron-Frobenius is applicable in its stronger form). Real networks do not usually have this property, specially if they are directed. The vertices that are not in strongly connected components will have value 0.

A way to work around this problem was proposed by Leo Katz. The idea is to give each node a minimum, positive amount of centrality, that it can transfer to other nodes, so:

$$K_v = \alpha \sum_u A_{vu} K_u + \beta,$$

where K_v is the Katz centrality of node v, β is a vector whose elements are all equal to a given positive constant and $\alpha \in (0,1)$ is a parameter. Equivalently, this is

$$K = \alpha A K + \beta,$$

so

$$\left(I - \alpha A\right) K = \beta,$$

and

$$K = \left(I - \alpha A\right)^{-1} \beta.$$

For this to work, $I - \alpha A$ must be invertible, which happens if and only if $|I - \alpha A| \neq 0 \iff \left|\frac{1}{\alpha} I - A\right| \neq 0$, so $\frac{1}{\alpha}$ must not be an eigenvalue of A. This is ensured if we take $\frac{1}{\alpha} > \lambda_{max}$, or $0 < \alpha < \frac{1}{\lambda_{max}}$.

An iterative way to compute K is

$$K = \left(\sum_{k=1}^{\infty} \alpha^k A^k\right) u.$$

The strength of α decreases at each iteration, acting as attenuation factor.

4.6 Clustering Coefficient Centrality

Triadic closure is the property among three nodes A, B, and C (representing people, for instance), that if the connections A-B and A-C exist, there is a tendency for the new connection B-C to be formed.

The clustering coefficient measures the proportion of neighbours of each node, that connected to each other.

Definition 4.4. Given a graph $G = (V, E)$, with adjacency matrix A, the **clustering coefficient** of node v is

$$CC_v = \frac{\left|\left\{\{u, v, w\} : (u,v), (v,w), (u,w) \in E\right\}\right|}{\binom{d_v}{2}},$$

where the numerator is the number of triangles involving v and its neighbours, and the denominator is the total number of possible links between v's neighbours.

The more densely connected the neighbourhood of v is, the higher is its clustering coefficient.

Example 4.4. The clustering coefficient of the graph example that we've been working with is

$$CC = \begin{pmatrix} \frac{1}{1} \\ \frac{1}{3} \\ \frac{0}{1} \\ \frac{0}{3} \\ \frac{1}{3} \\ 0 \end{pmatrix} = \begin{pmatrix} 1 \\ \frac{1}{3} \\ 0 \\ 0 \\ \frac{1}{3} \\ 0 \end{pmatrix}.$$

4.7 Closeness Centrality

Closeness centrality is a measure of how close a node is, on average, to the rest of the nodes, in terms of shortest paths. It measures the average distance between a node v and all other nodes in the network. Thus, the more central a node is, the closer it is to all other nodes.

Definition 4.5. Given a graph $G = (V, E)$, the **closeness centrality** of node v is

$$CL_v = \frac{1}{\sum_{r \neq v} dist\,(v, r)}.$$

It can be normalized by the factor

$$CL_v = \frac{N-1}{\sum_{r \neq v} dist\,(v, r)}.$$

An alternative is the **harmonic centrality**, obtained as

$$H_v = \sum_{r \neq v} \frac{1}{dist\,(v, r)},$$

with $dist\,(v, r) = 0$ if there is no path from v to r.

4.8 Betweenness Centrality

A family of betweenness measures are defined to capture a node's importance as a conduct of information flow in the network. This has wide applications in network theory, because in a telecommunications network, a node with higher betweenness centrality would have more control over the network, since more information will pass through that node.

The most well-known betweenness metric measures the number of times a node is on a shortest path between two nodes.

Definition 4.6. Given a graph $G = (V, E)$, the **betweenness centrality** of node v is

$$B_v = \sum_{s \neq v \neq t} \frac{\sigma_{s,t}\,(v)}{\sigma_{s,t}},$$

where $\sigma_{s,t}$ is the number of shortest path from source node s to target node t, and $\sigma_{s,t}\,(v)$ is the number of shortest path between these two nodes going through v.
This measure can be normalized by the number of ordered pairs not including v:

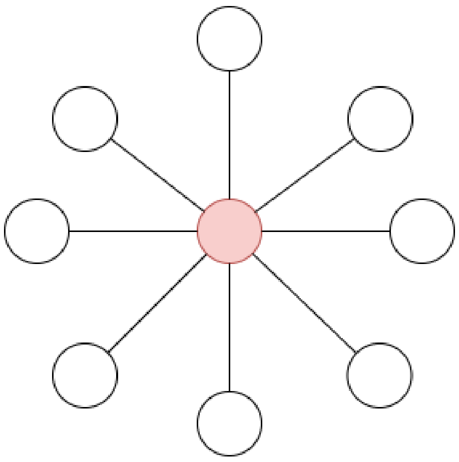
- For directed graphs

$$B_v = \frac{1}{(n-1)\,(n-2)} \sum_{s \neq v \neq t} \frac{\sigma_{s,t}\,(v)}{\sigma_{s,t}}.$$

- For undirected graphs

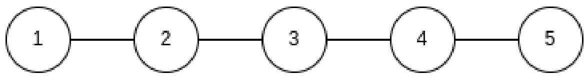
$$B_v = \frac{2}{(n-1)\,(n-2)} \sum_{s \neq v \neq t} \frac{\sigma_{s,t}\,(v)}{\sigma_{s,t}}.$$

Example 4.5. For the undirected star graph:

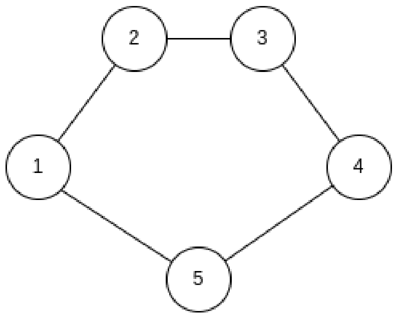


The center vertex has a betweenness of $\frac{(n-1)(n-2)}{2}$ (or 1, if we normalize it), while the leaves have a betweenness of 0.

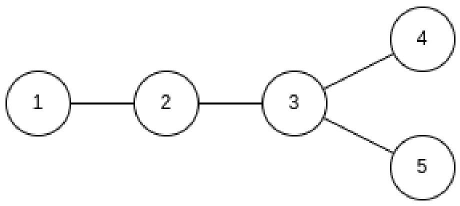
Exercise 4.1. What about the following graphs?

- 

$$B = \begin{pmatrix} \frac{0}{6} \\ \frac{3}{6} \\ \frac{4}{6} \\ \frac{3}{4} \\ \frac{0}{6} \end{pmatrix} = \begin{pmatrix} 0 \\ 0.5 \\ 0.67 \\ 0.5 \\ 0 \end{pmatrix}.$$

- 

$$B = \begin{pmatrix} \frac{1}{6} \\ \frac{1}{6} \\ \frac{1}{6} \\ \frac{1}{6} \\ \frac{1}{6} \end{pmatrix}.$$

- 

$$B = \begin{pmatrix} \frac{0}{6} \\ \frac{3}{6} \\ \frac{5}{6} \\ \frac{0}{6} \\ \frac{0}{6} \end{pmatrix} = \begin{pmatrix} 0 \\ 0.5 \\ 0.83 \\ 0 \\ 0 \end{pmatrix}.$$

5 MapReduce Computation Model

The advent of big data and the increasing analysis needs favorised the design of parallel algorithms, specially in the realm of big data processing pipelines, with a tradeoff between communication costs and degree of parallelism.

MapReduce is a processing paradigm that works on top of distributed environments. More precisely, it was built on top of Google File System (GFS) and Hadoop Distributed File System (HDFS), used to manage large-scale data and to be tolerant to hardware and networks faults. To do this, HDFS splits files into large blocks and distributes thema cross nodes in a cluster, and MapReduce is the programming model used to manage many large-scale parallel computations.

Basically, the idea is that the data is first splitted, then some operation is done to it, and then it's merged to produce the final results. For this, we will just need to define the **Map** and **Reduce** functions, while the system manages the parallel execution on distributed data and the coordination between them, leading with the possibility that one of the tasks may fail.

Example 5.1. Word Counter

Consider a text file splitted into partitions A, B, C, D, across different nodes. We want to count how many times each word appears in the whole document. For this, we can use MapReduce as follows:

1. Map: for each word, w, in each partition, generate the pair $(w, 1)$.
2. Shuffle/sort: collects and groups the pairs by key (word), in order to guarantee that the same key will be processed by the same reduce task. **Shuffling** is the process of redistributing data from Map nodes to Reduce nodes.

   In our example, we would have, for each word w, the pairs $(w, [1, ..., 1])$, with as many 1s as w appearances.
3. Reduce: for each input $(w, [1, ..., 1])$, output (w, N_w), where N_w is the amount of 1s.

The Map task will typically process many words in one or more chunks. If a word, w, appears m times among all chunks assigned to that process, there will be m key-value pairs $(w, 1)$ among its output.

To perform the grouping and distribution to the Reduce task, the master controller merges the pairs by key and produces a sequence of $(w, [1, ..., 1])$. Since it knows how many reduce tasks there will be, r, it will produce r lists, putting a list in one of r local files destined to one of the Reduce tasks. Each key is assigned as input to one, and only one, Reduce task.

The Reduce task executes one or more reducers, one per key. The outputs from all reducers are merges into a single final file.

5.1 The Map Function

In general, a map function can be defined as a function, $m_f : \mathbb{E}_1^n \to \mathbb{E}_2^n$, where $\mathbb{E}_i$ is the domain of the input (1) or output (2) and $f : \mathbb{R} \to \mathbb{R}$, that applies f to each coordinate. That is:

$$m_f([e_1, ..., e_n]) = [f(e_1), ..., f(e_n)].$$

For example:

$$m_{\cdot 2}([2, 3, 6]) = [4, 6, 12].$$

In the MapReduce scheme, map is more restrictive, as the function f must produce a key-value pair. That is, for all $i = 1, ..., n$, it is

$$f(e_i) = (k_i, v_i).$$

For example, in the word counter example:

$$m_f(["a", "b", "a"]) = [("a", 1), ("b", 1), ("a", 1)].$$

5.2 Shuffling/Grouping Function

The shuffle function consists in grouping the outputs of the map function by key, so

$$s\left(\left[\left(k_1, v_1\right), ..., \left(k_n, v_n\right)\right]\right) = \left[\left(k_1, \left(v_j : k_j = k_1, \forall j = 1, ..., n\right)\right), ...\right].$$

Following the previous example:

$$s\left(\left[\left("a", 1\right), \left("b", 1\right), \left("a", 1\right)\right]\right) = \left[\left("a", [1, 1]\right), \left("b", [1]\right)\right].$$

5.3 Reduce Function

Generally, a reduce function applies to a vector/row, and outputs a single value, applying the aggregation function f:

$$r_f\left(\left[v_1, ..., v_n\right]\right) = f\left(v_1, ..., v_n\right).$$

In MapReduce, reduce applies to each output of the shuffle function with the same key:

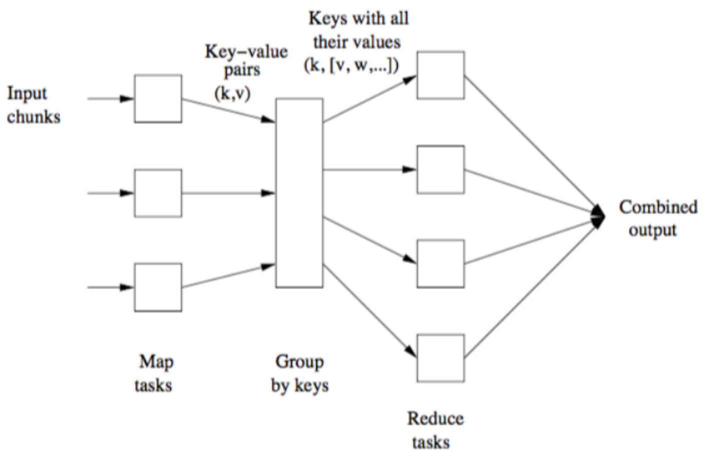
$$r_f\left(\left(k, \left[v_1, ..., v_n\right]\right)\right) = \left[\left(k'_1, f\left(v_1, ..., v_n\right)\right), ..., \left(k'_m, f\left(v_1, ..., v_n\right)\right)\right].$$

Following the previous example:

$$r_{sum}\left(\left[\left("a", [1, 1]\right), \left("b", [1]\right)\right]\right) = \left[\left("a", 2\right), \left("b", 1\right)\right].$$

A **MapReduce pipeline** can be a composition of different $r_{f_r} \circ s \circ m_{f_m}$.

The process is illustrated below:

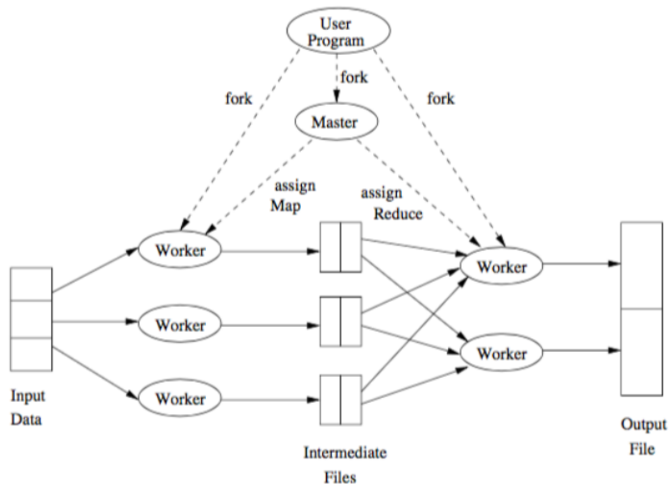


5.4 MapReduce Execution Model

Whenever we launch the execution of a MapReduce pipeline, the following happens:

- The user program forks a master controller process and some number of worker processes at different computer nodes.
- The amster creates some number of map tasks and some number of reduce tasks. It assigns the tasks to worker processes by taking into account the co-location.
- A worker handles either map tasks (a **map worker**) or reduce tasks (a **reduce worker**), but not both.
- A worker process reports to the amster when it finishes a task, and a new task is scheduled by the master for that worker process.

- The master keeps track of the status of each map and reduce task (idle, executing, or completed).



5.4.1 Coping with Node Failures

If the master node fails, the entire MapReduce job must be restarted.

If a worker node fails, it would be detected and managed by the master, since it periodically pings the worker processes. All the map tasks assigned to this worker have to be redone in this case.

5.4.2 Algorithms by MapReduce

This paradigm is not a solution to every problem, and in fact it only makes sense when files are very large, and rarely outdated. Its original purpose was to execute very large matrix-vector multiplications.

5.5 Use-Case: Matrix-Vector Multiplication by MapReduce

Let M be a $n \times n$ squared matrix and V a vector of size n. Their product,

$$W = MV,$$

is defined by

$$w_i = \sum_{j=1}^{n} m_{ij} v_j .$$

We can store M and V in a file in *HDFS* as triples $((i,j), m_{ij})$ for M and pairs (j, v_j) for V[1]. Now we can compute the computation by MapReduce as:

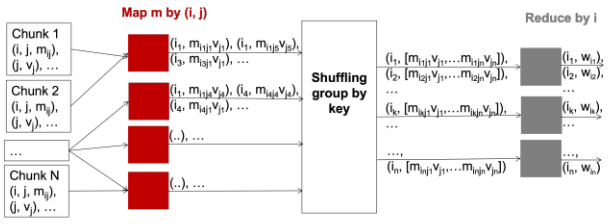
- Map: for each $((i,j), m_{ij})$ and (j, v_j), it returns $(i, m_{ij} v_j)$.
- Reduce: simply sums all the values for each key i, producing the pair (i, w_i).

For this to work, all the pairs from V must be available in all chunks (V cannot be stored distributely).

More concretely, we can define the functions:

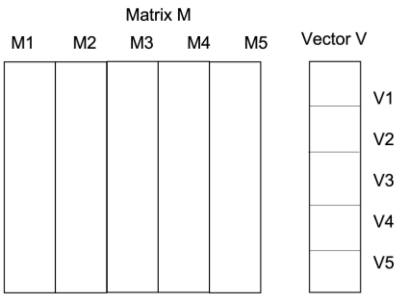
[1]This way is very efficient for sparse matrices.

```
map(key,val):
  i = key(1)
  j = key(2)
  for (j2, v) in V:
    if j == j2:
      emit(i, val*v)

reduce(key, val):
  sum = 0
  for v in val
    sum += v
  emit(key, sum)
```



Now, if n is large, V might not fit in main memory of a worker node, and a large number of disk accesses may be required. We can improve the approach by distributing V and refining the algorithm as follows:

- We devide the matrix into vertical stripes of equal width, and the vector in strips of the same size:



  Here, the size of M_k is $n \times n_k$ and the size of V_k is n_k, so that the product $M_k \cdot V_k$ can be performed, outputing a vector of size n.

- Each map task is assigned a chunk from one of the matrix stripes and gets the entire corresponding stripe of the vector.

- The final result would be

$$W = MV = \sum_{k=1}^{K} M_k \cdot V_k,$$

  where we apply the previously explained algorithm to each sub-multiplication step.

5.5.1 Matrix Multiplication

This approach can be extended to matrix multiplication. Now, let M be a matrix of size $n_1 \times n_2$ and N a matrix of size $n_2 \times n_3$, the product $P = MN$ is a matrix of size $n_1 \times n_3$, where

$$p_{ik} = \sum_{j=1}^{n_2} M_{ij} N_{jk}.$$

The matrices are stored as $(M, (i,j), m_{ij})$ and $(N, (j,k), n_{jk})$.

- Map 1: transform $(M, (i,j), m_{ij})$ into $(j, (M, i, m_{ij}))$ and $(N, (j,k), n_{jk})$ into $(j, (N, k, n_{jk}))$.
- Reduce 1: for each key, j, produces the key-value pair $((i,k), m_{ij}n_{jk})$.
- Map 2: the identity.
- Reduce 2: for each key, (i,k), produce the sum of the list of values associated to this key, $\left((i,k), \sum_j m_{ij}n_{jk}\right)$.

In addition, M could be divided into K vertical stripes of size (n_1, n_k) and N into K horizontal stripes of size (n_k, n_3), where $\sum_k n_k = n_2$. In this setup, we can apply the algorithm to compute each $M_k \cdot N_k$ and then sum them all.

The functions can be defined more precisely as:

```
map_1(T,(i,j),T_ij):
  emit(j , (T,i,T_ij))

reduce_1(key, val):
  for v in val:
    for w in val:
      if v(1) == M and w(1) == N:
        i = v(2)
        M_ij = v(3)
        k = w(2)
        N_jk = w(3)
        emit((i, k), M_ij*N_jk)

map_2(key, val):
  emit(key, val)

reduce_2(key, val):
  sum = 0
  for v in val:
    sum += v
  emit(key, sum)
```

5.6 Relational Algebra by MapReduce

5.6.1 Selection

Let $R(A_1, ..., A_n)$ be a relation stored as a file in HDFS. The elements of this file are the tuples of R. The selection operator, $\sigma_C(R)$ can be defined using MapReduce as:

- Map: for each tuple in R, t, test if t satisfies C. If it does, produce the key-value pair (t,t).
- Reduce: the identity.

5.6.2 Projection

For the projection, $\pi_A(R)$, we can do:

- Map: for each tuple in R, t, construct a tuple t' by removing the attributes that are not in A. Output (t', t').
- Reduce: for each key, t', produced by the map tasks, there will be one or more key-value pairs (t', t'). The reduce function turns $(t', [t', t', ..., t'])$ into (t', t') so it produces exactly one pair.

5.6.3 Join

$R(A) \bowtie_B S(C)$ with A, B, C sets of attributes satisfying $B \subset A, B \subset C$, can be implemented with MapReduce as:

- Map: for each tuple $(a,b) \in R$, produce the key-value pair $(b,(R,a))$. For each tuple $(c,b) \in S$, produce the key-value pair $(b,(S,c))$.
- Reduce: for each key, b, output as many pairs as needed, $(b,[(R,a),(S,c)])$.

5.6.4 Aggregation

The aggregation operator, $\gamma_{A,\theta(B)}(R)$, where $A \cup B$ is the set of attributes of R, and $A \cap B = \emptyset$, can be defined with MapReduce as:

- Map: for each tuple, t, produce (a,b), where a is the A part of t, and b is the B part.
- Reduce: each key represents a group, so we apply θ to the list $[b_1, ..., b_n]$ associated to each value a. We output (a,x), where $x = \theta(b_1, ..., b_n)$.

5.7 Some Issues of MapReduce

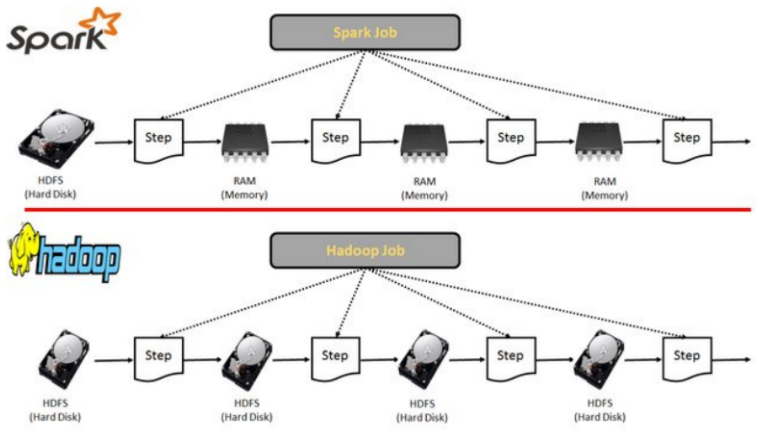
- **Locality**: input data is stored on local disks of machines in the cluster. Each file is divided into blocks of 64MB, each of which is stored several times, as replicas, on different machines. MapReduce master node takes the location information of the input files into account, and attempts to schedule a map task on a machine that contains the needed replica. If this fails, it tries to schedule a map task in a machine that is near to one that has a replica.
- **Granularity**: the amp and reduce steps are divided into M and R pieces. M and R should be much larger than the number of workers. Each worker can perform different tasks, improving dynamic load balancing and speeding up recovery when a worker fails. Some practical bounds on how large these values should be say that the master should take $M + R$ scheduling decisions and keep $M \times R$ states in memory.
- **Refinements**: partitioning input data using different functions according to the problem to be solved.
- **Ordering guarantees**: the intermediate key-value pairs are generally processed in increasing key order, to make it easy to generate a sorted output file per partition. However, this is not guaranteed.

6 Spark Parallel Computing Framework

Nowadays, data is growing faster than processing speeds, and so the only possible solution is to parallelize on large clusters.

Apache Spark is an open source implementation of a framework for large-scale data processing, providing an interface for programming clusters with implicit data parallelism and fault tolerance. It extends a programming language with read-only data structure distributed over a cluster of machines, the **Resilient Distributed Datasets (RDDs)**, maintained in a fault-tolerant way. RDDs were developed in 2012 in response to limitations in Hadoop's MapReduce, which forces a particular linear dataflow as a sequence of HDFS reads and writes.

Spark is up to 100 times faster than traditional Hadoop thanks to its in-memory data processing:



6.1 Spark's Engine Properties and Components

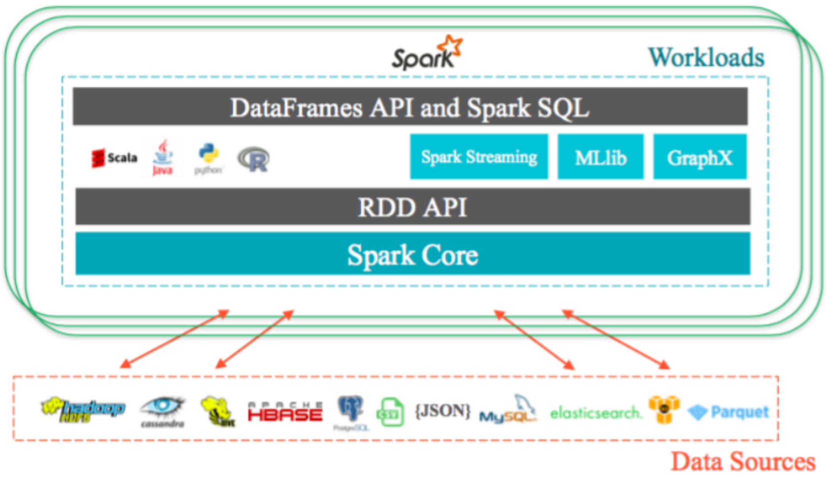
Spark was originally written in Scala, a high level language for JVM. There are APIs for Java, Scala, Python, R,...

The Dataframe API was released as an abstraction on top of the RDD, as well as packages like MLlib or GraphX, that can be used for machine learning and graph analytics. These APIs facilitate the implementation of both iterative algorithms and interactive or exploratory data analysis.

Spark requires a cluster manager and a distributed storage system:

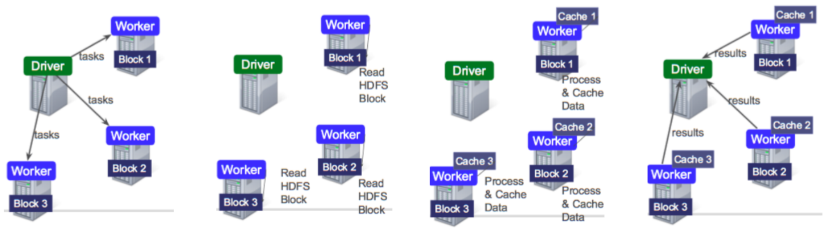
- Cluster management: Spark supports standalone, native Spark clusters, where we can launch a cluster either manually or using scripts, or we can use Hadoop YARN, Apache Mesos, or Kubernetes.
- Distributed storage: Spark cna interface with a wide variety of distributed databases, like Alluxio, HDFS, MapR-FS, Cassandra,...

Spark also supports different sources of data, in different formats and from different databases. All these relationships are shown below:



6.2 Spark's Execution Architecture

Data is splitted into partitions or blocks, and the driver assigns tasks to each worker, which reads a HDFS block, and has a cache. Each worker process and cache data, if necessary, sending the results to the driver when done.



6.3 Resilient Distributed Datasets (RDDs)

RDDs are immutable and distributed collections of objects, spread across a cluster, and stored in RAM or disk (when they are persistent). They are statically typed, i.e., RDD[T] has objects of type T. The types can be of any type of Python, Java, or Scala objects, including user-defined classes.

RDDs are built via parallel **transformations** and computed via parallel **actions** on distributed datasets, executed lazily. For instnace, RDDs are splitted into multiple partitions, which may be computed on different nodes of a cluster.

- **Transformation**: operation on an RDD that returns a new RDD. They are computed lazily, only after an action is called.
- **Action**: operation on an RDD that returns a final result which is not another RDD. When an action is called, all the transformations prior to it are executed in the same order they were defined. Each time a new action is called, the entire RDD must be computed from scratch, but the user can deide to persist intermediate result (caching).

Inside Apache Spark, the workflow is managed as a directed acyclic graph (DAG). Nodes represents RDDs while edges represent the operations executed on the RDDs. Spark keeps track of the set of dependencies between different RDDs. This is called the **lineage graph**.

Example 6.1. Python Example: first line mentioning Python

```python
# spark context creation
sc = pyspark.SparkContext(...)

# creating an RDD of strings with textFile()
lines = sc.textFile("README.txt")

# Transformations:
#   Construct a new RDD from a previous one, one common transformation is filtering data that matches a predicate
pythonLines = lines.filter(lambda line: "Python" in line)

# Actions:
#   Compute a result based on an RDD, and either return it to the driver program or save it to an external sotrage system
pythonLines.first()
```

Some notes:

- Once a SparkContext, *sc*, is set, it is used to build RDDs. The driver program manages a number of workers, and different workers on different machines might count lines in different ranges of the file.
- *filter* () does not mutate the existing input RDD. Instead, it returns a pointer to a new RDD.
- Many transformations work on one element at a time, but this is not true for all transformation, like *union*.

6.3.1 RDDs Operations

- *map* () is used from different purposes, from fetching a large collection to just squaring numbers. It transforms an RDD of length N into another RDD of length N, applying a function to each element in the RDD. For example:

```
nums = sc.parallelize([1,2,3,4])
squared = nums.map(lambda x: x*x).collect()

for num in squared:
  print(num)

# 1 4 9 16
```

- *flatMap* () transforms an RDD of length N into a collection of N collection, then flattens these into a single RDD of results, applying a function to each collection. For example:

```
def tokenize(sentence):
  return [word for word in sentence]

rdd = sc.parallelize(["coffee panda"])

# Using map
map_result = rdd1.map(tokenize).collect()

for token in map_result:
  print(token)

# Output: ["coffee", "panda"]

# Using flatMap
flatmap_result = rdd1.flatMap(tokenize).collect()

for token in flatmap_result:
  print(token)

# Output: "coffee", "panda"
```

- *reduce* () is the most common action on basic RDDs. It operates on two elements of the type of the RDD and returns a new element of the same type. For example:

```
data = sc.parallelize([1,2,3])

data.reduce(lambda x, y: x+y) # Result is 6
```

- *reduceByKey* () operates on RDD of key-value pairs. It runs several parallel reduce operations, for each key, where each operations combines values that have the same key. It returns a new RDD consisting of each key and the reduced value for that key. For example:

```
pets = sc.parallelize([("cat", 3), ("dog", 2), ("cat", 1)])

pets.reduceByKey(lambda x, y: x+y)

# Result: {("cat", 4),("dog", 2)}
```

- *groupByKey* () and *sortByKey* () return the RDD with the values grouped or sorted by the keys. For example:

```
pets = sc.parallelize([("cat", 3), ("dog", 2), ("cat", 1)])

pets.groupsByKey() # Result: {("cat", [3, 1]),("dog", [2])}

pets.sortByKey() # Result: {("cat", 3),("cat", 1),("dog", 2)}
```

- *distinct* () produces a new RDD with only distinct items.

- *union* (*otherRDD*) produces a new RDD consisting of the data from both sources, removing duplicates.
- *RDD.subtract* (*otherRDD*) produces a new RDD consisting in all values present in *RDD* but not in *otherRDD*.
- *RDD.cartesian* (*otherRDD*) returns of possible pairs (a, b) where $a \in RDD$ and $b \in otherRDD$. Note that this operation is very costly.
- *RDD.innerJoin*(*otherRDD*) returns only keys that are present in both pairs to the output RDD.
- *RDD.leftOuterJoin* (*otherRDD*) and *RDD.rightOuterJoin* (*otherRDD*) join the two RDDs together by key, allowing one of them to miss the key (left or right). For example:

```
rdd = sc.parallelize([(1,2),(3,4),(3,6)])
other = sc.parallelzie([3, 9])

rdd.join(other) # result: {(3, (4,9)), (3, (6,9))}

rdd.leftOuterJoin(other) # result: {(1, (2, None)), (3, (4,9)), (3, (6,9))}
```

6.3.2 RDDs Actions

- *collect* () is used to retrieve the entire RDD. It is useful if it filters RDDs down to a very small size to deal with it locally at the driver. The retrieved dataset must fit in memory in a single machine.
- *take* (N) is used to retrieve a small number of elements in the RDD at the driver program and then iterate them over them locally.
- *top* (N) is used to extract the top elements.
- *takeSample* (*withReplacement*, N, *seed*) allows to take a sample of the data either with out without replacement.
- *foreach* () performs computations on each element in the RDD without bringing it back locally.
- *count* () returns a count of the elements.
- *countByValue* () returns a map of each unique value and its count.

6.3.3 Caching

Spark RDDs are lazily evaluated, and sometimes we use the same RDD multiple times. Naively, Spark will recompute the RDD and all of tis dependencies each time we call an action on the RDD. To avoid this, we can ask Spark to persist data using *persist* ().

Notice that calling *persist* () does not force the evaluation of the RDD.

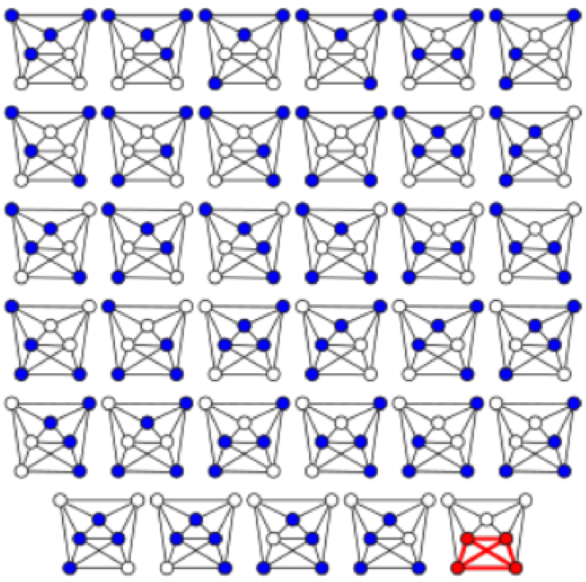
If we cache too much data, Spark will automatically delete old partitions. For the memory-only storage levels, it will recompute these partitions the next time they are accessed. This means that caching unnecessary data can lead to eviction and increased re-computation time.

7 Community Detection Approaches

7.1 k-Clique

Definition 7.1. A **clique** is a complete subgraph.
A k-clique is a complete subgraph with k nodes.

Example 7.1. In the following picture[2], we observe the brute force process to find all 4-cliques in this 7-node graph. For this, we need to check C_7^4 combinations.



Most versions of clique problems are hard, and a common problem is that of finding **maximal cliques**. This is, cliques with the largest number of nodes. For this problem, we have different algorithms, like:

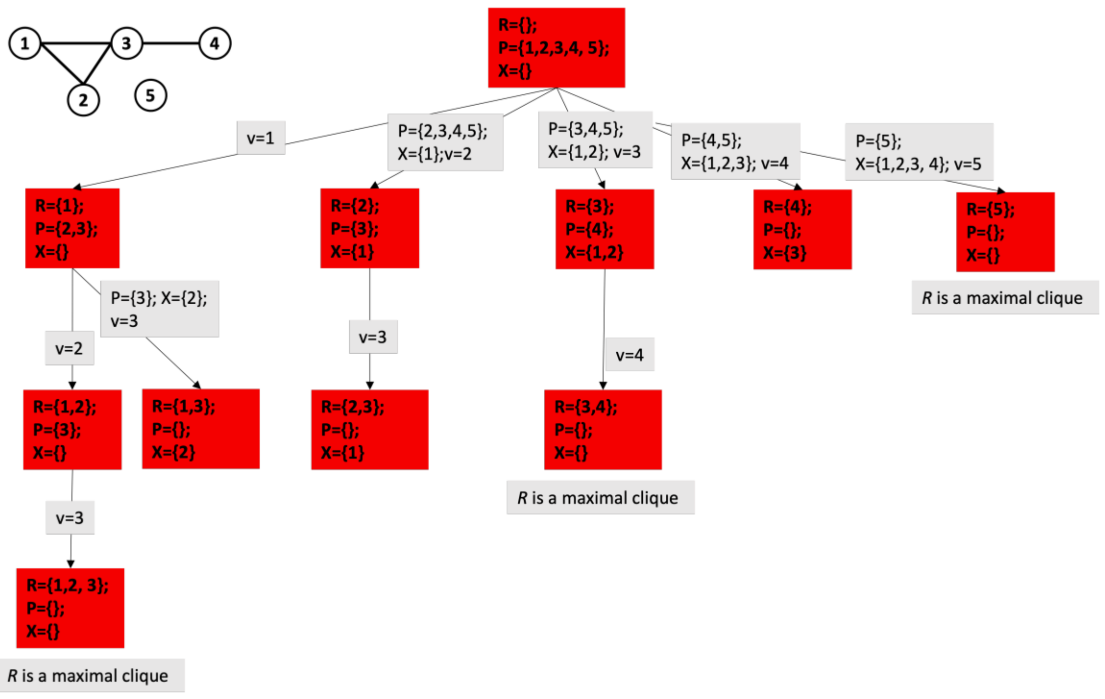
- Bron-Kerbosch algorithm, with complexity $O\left(3^{\frac{n}{3}}\right)$.
- Tarjan and Trojanowski algorithm.
- Janez Konc algorithm.

There are also algorithms with better theoretical complexity, but Bron-Kerbosch and some variants that improve it are more efficient in practice. The basic form of the algorithm is as follows:

```
Bron-Kerbosch(G=(V,E)):
  # Initialization
  P = V
  X = {}
  R = {}

  def BronKerbosch(R,P,X):
    if P = {} and X = {} then
      R is maximal

    for v in P:
      P_prime = P.intersection(neighbours(v))
      X_prime = X.intersection(neighbours(v))
      R_prime = R.union({v})
      BronKerbosch(R_prime, P_prime, X_prime)
      P = P - {v}
      X = X.union({v})
```

For example, the following tree represents the execution of the algorithm for the shown graph:

[2]Example from https://en.wikipedia.org/wiki/Clique_problem.



7.2 k-Core

k-Cliques are interesting as a concept, but they are very restrictive, and so some relaxations have been proposed. A very well known one are k-Cores [1]. The basic idea is that networks may present a core-periphery structure, where in the core there are a lot of connected nodes, while in the periphery we find a more sparse structure, with peripheral nodes called whiskers.

Definition 7.2. The **k-core** of a graph G is a maximal connected subgraph of G where vertices have at least degree k. Also called k-degenerate.

The approach to find k-cores is **k-core decomposition**. The idea is that, given a graph $G = (V, E)$, we delete recursively all vertices, and edges connecting them, of degree less than k, to extract the k-core, such that:

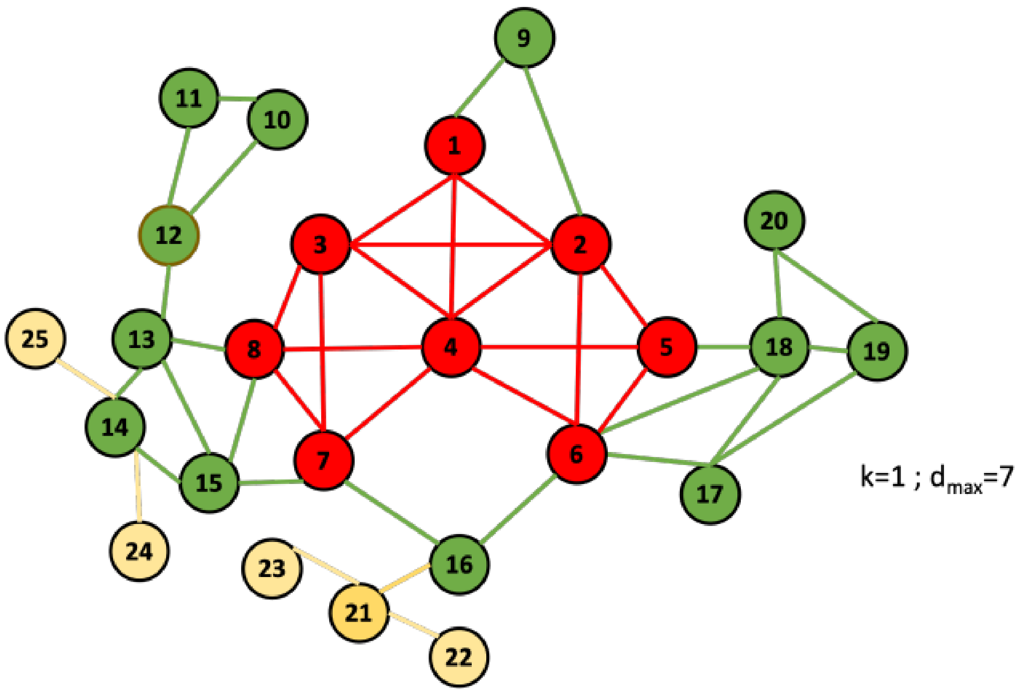
- Each v_i in a k-core graph has $d_i \geq k$.
- A $(k + 1)$-core is a subgraph of a k-core graph.

The algorithms goes as:

```
kCore(G=(V,E)):
  L = {} # Maps each vertex to the highest k-core it belongs
  d = [] # List of degrees for each vertex
  D = {} # Dictionary mapping each degree to all vertices with that degree
  d_max = max([degree(v) for v in V])

  for v_i in V:
    d[v_i] = degree(v_i)
    D[d[v_i]].append(v_i)

  for k in range(0,d_max):
    while not D[k].empty():
      v_i = D[k].pop # Get vertices of k-core
      L[v_i] = k

      for u_j in neighbours(v_i): # Update neighbours removing the edges to the extracted
    vertex
```

```
        if d[u_j] > k:
          D[d[u_j]].pop(u_j)
          D[d[u_j]-1].append(u_j)
          d[u_j] -= 1
```

Example 7.2. Let's go through an example:



First, we initialize d and D:

i	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25
d[v_i]	4	6	5	7	4	6	5	5	2	2	2	3	4	4	4	3	3	5	3	2	3	1	1	1	1

d[v_i]	7	6	5	4	3	2	1
D[d[v_i]]	[4]	[2,6]	[3,7,8,18]	[1,5,13,14,15]	[12,16,17,19,21]	[9,10,11,20]	[22,23,24,25]

Now, we start the procedure. For $k = 1$, the first node to be taken out would be 22, so:

i	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	2
d[v_i]	4	6	5	7	4	6	5	5	2	2	2	3	4	4	4	3	3	5	3	2	~~3~~ 2	1	1	1	1

d[v_i]	7	6	5	4	3	2	1
D[d[v_i]]	[4]	[2,6]	[3,7,8,18]	[1,5,13,14,15]	[12,16,17,19,~~21~~]	[9,10,11,20,21]	[~~22~~,23,24,25]

v_i	22
L[v_i]	1

Now, with 23:

i	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	2
d[v_i]	4	6	5	7	4	6	5	5	2	2	2	3	4	4	4	3	3	5	3	2	~~2~~ 1	1	1	1	1

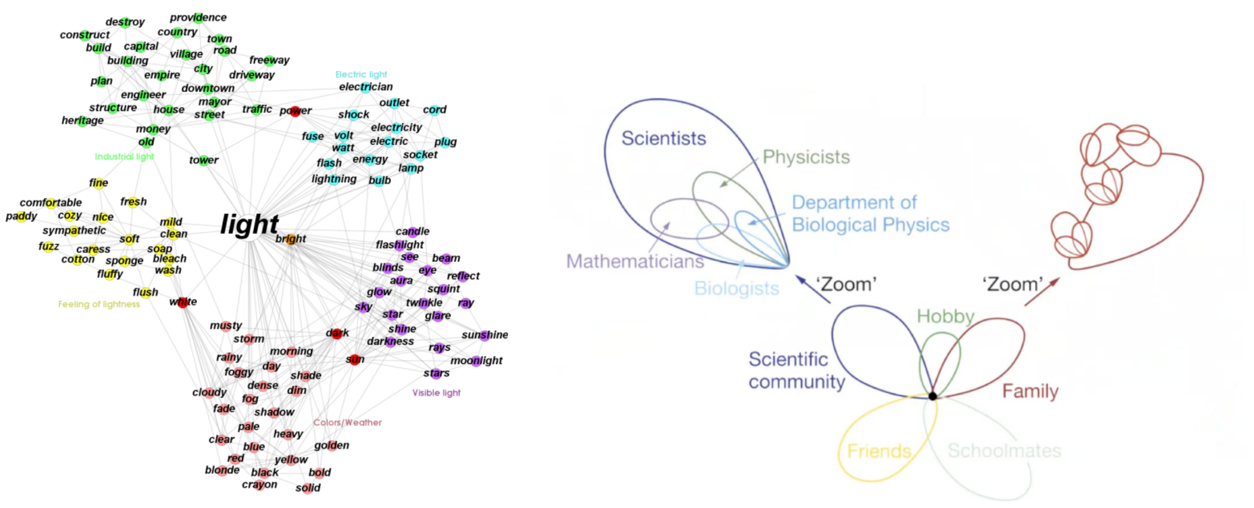


Figure 1: Non-verlapping communities (left) and overlapping communities (right).

d[v _ i]	7	6	5	4	3	2	1
D[d[v _ i]]	[4]	[2,6]	[3,7,8,18]	[1,5,13,14,15]	[12,16,17,19]	[9,10,11,20,~~21~~]	[21,~~23~~,24,25]

v _ i	22	23
L[v _ i]	1	1

Now, with 24:

i	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	2
d[v _ i]	4	6	5	7	4	6	5	5	2	2	2	3	4	~~4~~ 3	4	3	3	5	3	2	1	1	1	1	1

d[v _ i]	7	6	5	4	3	2	1
D[d[v _ i]]	[4]	[2,6]	[3,7,8,18]	[1,5,13,~~14~~,15]	[12,16,17,19,14]	[9,10,11,20]	[21,~~24~~,25]

v _ i	22	23	24
L[v _ i]	1	1	1

And so on...

7.3 Community Detection Problem

A **community** in a network refers to the occurrence of clusters or groups of nodes in a network, that are more densely connected internally than with the nodes outside the community. There are mainly two cases:

- **Non-overlapping communities**: divides into clusters of nodes with dense connections internally, and sparser connections between clusters. Each node is assigned to one community.
- **Overlapping communities**: assumes that pairs of ndoes are more likely to be connected if they are both members of the same communities, and less likely to be connected if they do not share any community.

There are several considerations or properties of communities that can be used to obtain them:

- Mutuality of ties: every node in the group has ties to one another.

- Compactness: closeness or reachability of group nodes.
- Density of edges: high number of edges within a community.
- Separation of higher frequency of ties among group members compared to non members.
- Usual metrics: graph density, internal and external density, graph cut, modularity score.

This problems is a combinatorial optimisation problem, and it is NP-hard if we want a exact solution. However, there are many approaches that leverage greedy techniques or approximate heuristics, that we are going to see now.

7.3.1 k-Clique Community

Definition 7.3. A **k-clique community** is the union of all k-cliques that can be reached from one to the other through a sequence of adjacent k-cliques.
Two k-cliques are **adjacent** if they share $k-1$ vertices.

To find k-clique communities, we can use the Clique Percolation Method [2], which consists in:

1. Find all maximal cliques, sorting them based on degrees.
2. Create clique overlap matrix, where $M[i,j]=1$ if cliques i and j overlap in at least $k-1$ nodes.
3. Communities are the connected components that arise from this.

7.3.2 Louvain Algorithm

The most populat community detection algorithm is the **Louvain algorithm**, which is based in the concept of modularity:

In network science, the **configuration model** is a method for generating random networks from a given degree sequence. It is widely used as a reference model for social networks, allowing the modeler to incorporate arbitrary degree distributions.

The idea is to assign a degree d_v to each node v. Each degree is a half-link, or stub, and the sum of stubs must be even to be able to build a graph, i.e., $\sum_{v\in V} d_v = 2m$. Then, we choose two stubs uniformly at random and connect them by a link, then choose another pair from the remaining $2m-2$ stubs, and connect them. We repeat this process until there are no more stubs. The resulting network keeps the same degrees but randomly pairs up nodes.

A **realization** might include cycles, self-loops or multi-links. The uniform distribution of the matching must be kept, so these are not excluded. However, their expected number goes to zero for large networks, because the probability of v being connected to one of w stubs is $\frac{d_w}{2m-1}$. Since node v has d_v stubs, the probability of v being connected to w is $\frac{d_v d_w}{2m-1}$ which is almost the same as $\frac{d_v d_w}{2m}$ for large m.

Now, **modularity** measures the relative density of links inside communities with respect to links outside communities. There are different methods to compute the modularity, but in the most common version, the randomization of the edges is done to preserve the degree of each vertex.

The basic idea is to compare the number of links within communities with the number expected on the basis of chance. The generated network has less links between nodes of the same community, and more between nodes of different communities.

It can be computed as

$$Q=\frac{1}{2m}\sum_i\sum_j\left[A_{ij}-\frac{d_i d_j}{2m}\right]\delta\left(c_i,c_j\right)$$

or

$$Q = \left[\frac{\sum_i \sum_j A_{ij}}{2m} - \frac{\sum_i d_i \sum_j d_j}{(2m)^2} \right] \delta(c_i, c_j),$$

where d_i is the degree of vertex i, and c_i is the community of vertex i. δ is the Kronecker function, so $\delta(c_i, c_j)$ is 1 if $c_i = c_j$, and 0 otherwise.

The modularity is $-1 \leq Q \leq 1$ and it is positive when links within communities exceed links within communities in a randomly rewired network.

This is the modularity of a graph, but we can also compute the modularity of a community, c, as

$$Q_c = \frac{\sum_{i \in c} \sum_{j \in c} A_{ij}}{2m} - \frac{\sum_{i \in c} d_i \sum_{j \in c} d_j}{(2m)^2}$$

or

$$Q_c = \frac{\sum_{in}}{2m} - \left(\frac{\sum_{in+out}}{2m} \right)^2,$$

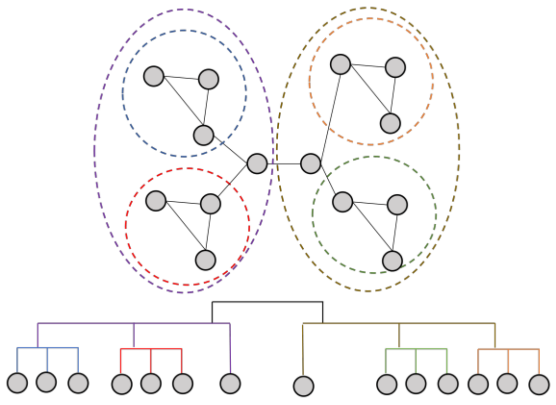
where $\sum_{in}$ is the sum of edge weights between nodes within the community c (each edge is considered twice), and $\sum_{in+out}$ is the sum of all edge weights for nodes within the community, including incident ones from the other communities.

Note that if there is only one community, then $Q = 0$.

Now, finding the optimal value for modularity is impractical, because it needs to go through all possible combinations of the nodes into communities, and so the **Louvain method** is a heuristic method for greedy modularity maximisation in 2 phases:

1. Modularity is optimized by allowing only local changes to node communities memberships.
   (a) Initially, each node is assigned its own community.
2. The identified communities are aggregated into super-nodes, to build a new network.
   (a) This phase is repeated until there is only one super node, the whole graph.

This algorithm is widely used for large networks, because it is efficient and produces high dense communities. It has a complexity of $O(n \log n)$ in time, and can be used on weighted graphs. In addition, it provides hierarchical communities, allowing us to define the level of detail of the communities, and also to obtain subcommunities within communities. A sample visualization is the following:



In more precise terms, the **phase 1** goes as follows: what is ΔQ if we move a node v from C_J to C_I:

$$\Delta Q_{C_J \xrightarrow{v} C_I} = \Delta Q_{C_J \setminus \{v\}} + \Delta Q_{C_I \cup \{v\}}.$$

But how can we derive $\Delta Q_{C_I \cup \{v\}}$? The key here is that the gain in modularity does not depend on the original community of v. It can easily be computed by moving an isolated node v into C_I:

$$\Delta Q_{C_I \cup \{v\}} = Q_{C_I \cup \{v\}} - \left[Q_{C_I} + Q_{\{v\}} \right],$$

and therefore

$$\Delta Q_{C_J \cup \{v\}} = \left[\frac{\sum_{in} + 2 \cdot d_{v,in}}{2m} - \left(\frac{\sum_{in+out} + d_v}{2m} \right)^2 \right] - \left[\frac{\sum_{in}}{2m} - \left(\frac{\sum_{in+out}}{2m} \right)^2 - \left(\frac{d_v}{2m} \right)^2 \right].$$

On the other side,

$$\Delta Q_{C_J \setminus \{v\}} = \left[Q_{C_J \setminus \{v\}} + Q_{\{v\}} \right] - Q_{C_J},$$

so

$$\Delta Q_{C_J \setminus \{v\}} = \left[\frac{\sum_{in} - 2 \cdot d_{v,in}}{2m} - \left(\frac{\sum_{in+out} - d_v}{2m} \right)^2 - \left(\frac{d_v}{2m} \right)^2 \right] - \left[\frac{\sum_{in}}{2m} - \left(\frac{\sum_{in+out}}{2m} \right)^2 \right].$$

Therefore, the gain only depends on $d_{v,in}$ and C.

The complete algorithm is:

1. For each node v:
   (a) Compute modularity gain from removing v from its community and placing it in the community of its neighbours.
   (b) Place v in the community that maximizes ΔQ.
   (c) One iteration is achieved for all the nodes in a sequential manner.
   (d) Repeat the procedure sequentially to all nodes until no more improvement (local maximum of modularity).
   (e) The output of this phase depends on the order in which nodes are considered. Research shows that this does not significantly affect the overall modularity.
2. Nodes from communities are grouped into super nodes.
   (a) Links between nodes of the same community c are represented by self-loops weighted by adding up the links between these nodes: c has a loop edge with weight
   $$w = \sum_{v_i, v_j \in c} A_{i,j}.$$
   (b) Links between communities are weighted by adding up the links between community's nodes. Each (c_i, c_j) has a link with weight
   $$w = \sum_{v_i \in c_i, v_j \in c_j} A_{i,j}.$$

7.3.3 Walktrap Approach

This other approach is based on random walks. The idea is to consider a random walk on a graph, in which at each time step, we move to neighbours uniformly at random:

$$P_{ij} = \frac{A_{ij}}{d_i}, \ P = D^{-1} A,$$

and P_{ij}^t represents the probability to get from i to j in t steps.

We would think that two nodes i and j that lie in the same community should have a high P_{ij}^t, as well as similar $P_{ik}^t \sim P_{jk}^t$, for different k. Therefore, we can define a distance (or similarity) between nodes, as

$$r_{ij}(t) = \sqrt{\sum_{k=1}^{n} \frac{\left(P_{ik}^t - P_{jk}^t \right)^2}{d_k}} = \left\| D^{-\frac{1}{2}} P_i^t - D^{-\frac{1}{2}} P_j^t \right\|.$$

In addition, we approximate the computation by

$$P_{ik}^{t} \sim \frac{N_{ik}}{N_i},$$

where N_{ik} is the number of walks starting at i and passing through k, and N_i is the number of walks starting from i.

The approach goes as follows:

1. Assign each node to its own community.
2. Compute the distance between adjacent nodes, $r_{ij}(t)$.
3. Choose the two closest communities and merge them.
4. Update the distance between communities, as

$$r_{c_1,c_2}(t) = \sqrt{\sum_{k=1}^{n} \frac{\left(P_{c_1k}^{t} - P_{c_2k}^{t}\right)^2}{d_k}} = \left\| D^{-\frac{1}{2}} P_{c_1}^{t} - D^{-\frac{1}{2}} P_{c_2}^{t} \right\|,$$

   where

$$P_{ck}^{t} = \frac{1}{|c|} \sum_{i \in c} P_{ik}^{t}.$$

5. Finish when there is only one community.

7.3.4 Girvan-Newman Approach

Edge betweenness of e is the fraction of shortest paths between all nodes s and t going through edge e:

$$ebt(e) = \sum_{s \neq t} \frac{n_{s,t}^{e}}{n_{s,t}}.$$

In this case, we focus on edges that connect communities, and construct them by progressively removing edges with the highest betweenness value. It goes as:

1. For all $e \in E$
   (a) Compute $ebt(e)$
   (b) Remove the edge e with largets $ebt(e)$
2. Repeat until all edges are gone. Stop when it splits in 2 components. The output is a dendogram.

References

[1] Jure Leskovec, Kevin J. Lang, Anirban Dasgupta, and Michael W. Mahoney. Community structure in large networks: Natural cluster sizes and the absence of large well-defined clusters, 2008.

[2] Gergely Palla, Imre DerÃ©nyi, IllÃ©s Farkas, and TamÃ¡s Vicsek. Uncovering the overlapping community structure of complex networks in nature and society. *Nature*, 435(7043):814–818, 2005.

[3] NacÃ©ra Seghouani. Massive graph management and analytics. Lecture Notes.