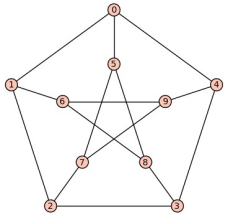
Data Mining for Networks

Frédéric Giroire
Université Côte d'Azur/CNRS/Inria COATI*
frederic.giroire@inria.fr
Master 2

***Combinatorics, Optimisation et Algorithms For Telecommunications**



$$\min \sum_{e \in \mathcal{E}} y_e$$

$$\text{s.t.} \sum_{a \in A_i^+(u)} f_a^i - \sum_{a \in A_i^-(u)} f_a^i = \begin{cases} |V_i| - 1 & \text{if } u = s_i \\ -1 & \text{if } u \neq s_i \end{cases} \quad \forall u \in V_i,\ V_i \in C$$

$$f_a^i \leq |V_i| \cdot x_a, \quad \forall V_i \in C, a \in A$$

$$x_{(u,v)} \leq y_{uv}, \quad \forall uv \in \mathcal{E}$$

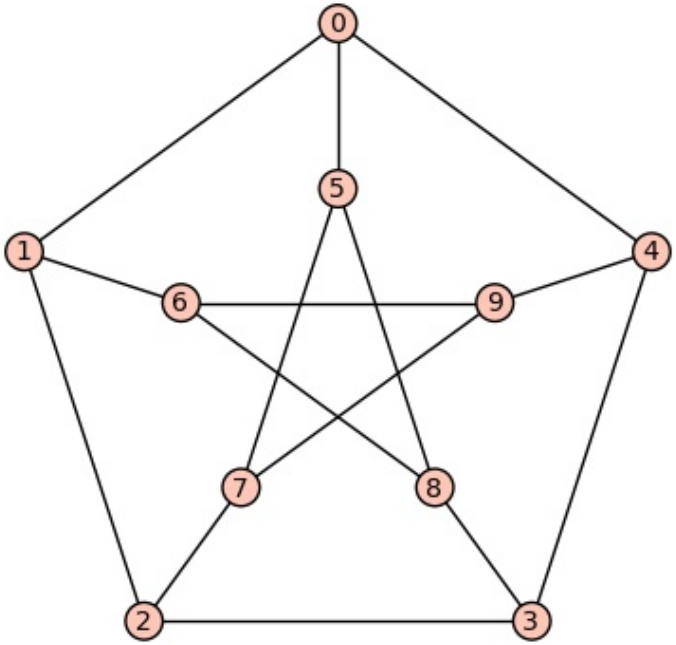
$$x_{(v,u)} \leq y_{uv}, \quad \forall uv \in \mathcal{E}$$

Table of Content

Table of Content

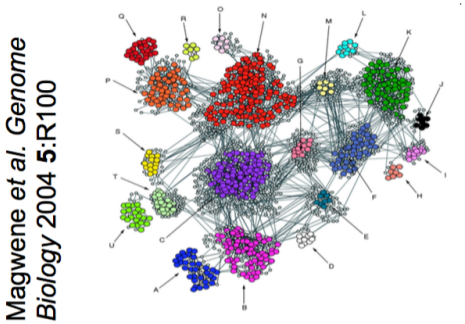
Machine Learning on Graphs

- Graphs in reality:
  - Graphs model objects and their relationships.
  - Also referred to as networks.
  - All common data structures can be modeled as graphs.

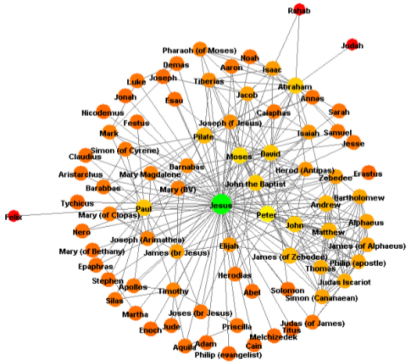


Machine Learning on Graphs

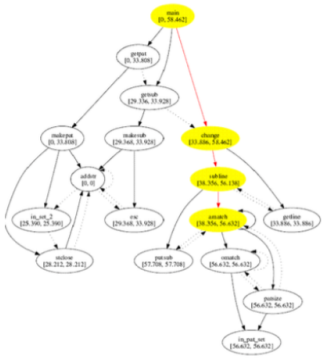
- Graphs are everywhere



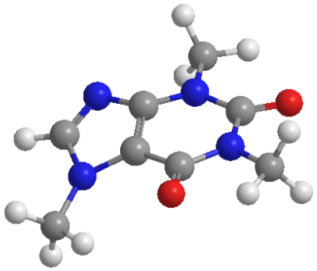
Co-expression Network



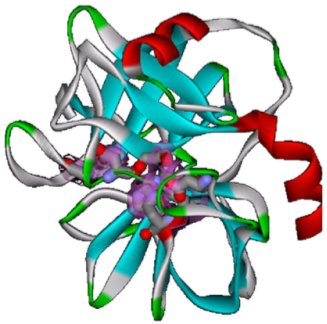
Social Network



Program Flow



Chemical Compound



Protein Structure

Machine Learning on Graphs

Important **Machine learning tasks** on graphs:

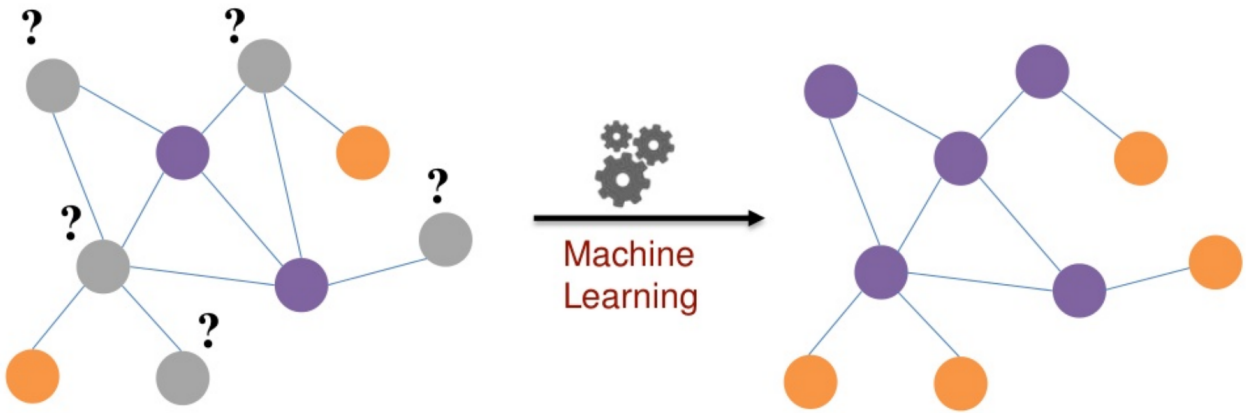
- Node classification
- Graph clustering
- Link Prediction
- Graph classification

Machine Learning on Graphs

Machine learning tasks on graphs:

- Node classification: given a graph with labels on some nodes, provide a high quality labeling for the rest of the nodes

  Labels: demographic (age, gender, location), interests (hobbies), etc.

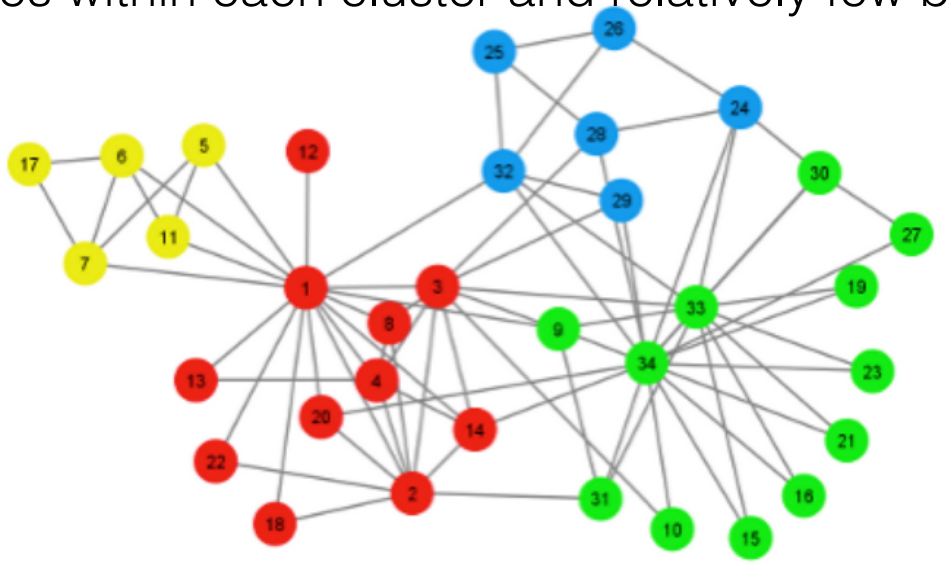


Classification can be done on many criteria (Node degree, pagerank, neighbors, …)

Machine Learning on Graphs

Machine learning tasks on graphs:

- Graph clustering: given a graph, group its vertices into clusters taking into account its edge structure in such a way that there are many edges within each cluster and relatively few between the clusters

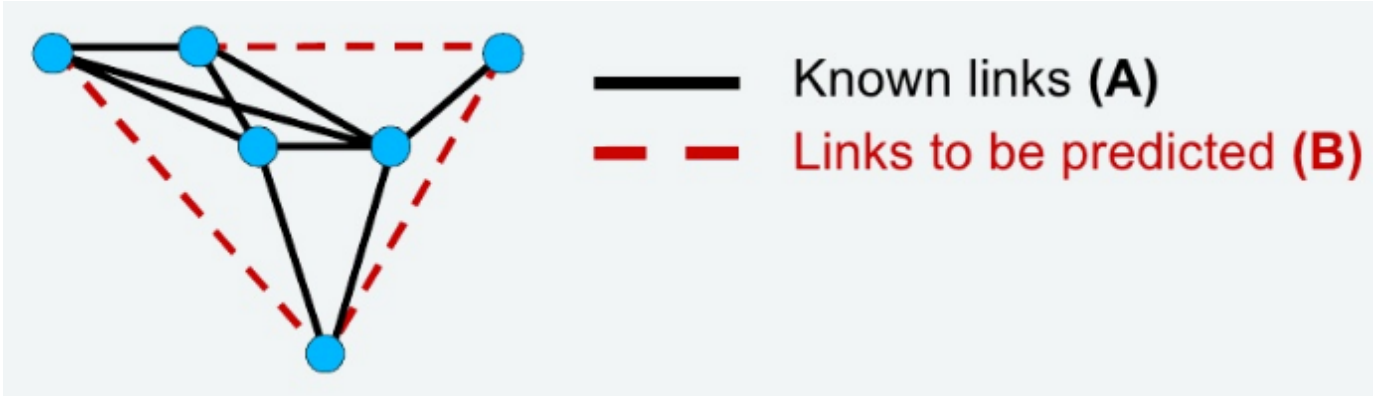


Clusters: people with the same label. (Age, citizenship, interest, …)

Machine Learning on Graphs

Machine learning tasks on graphs:

- Link Prediction: given a pair of vertices, predict if they should be linked with an edge

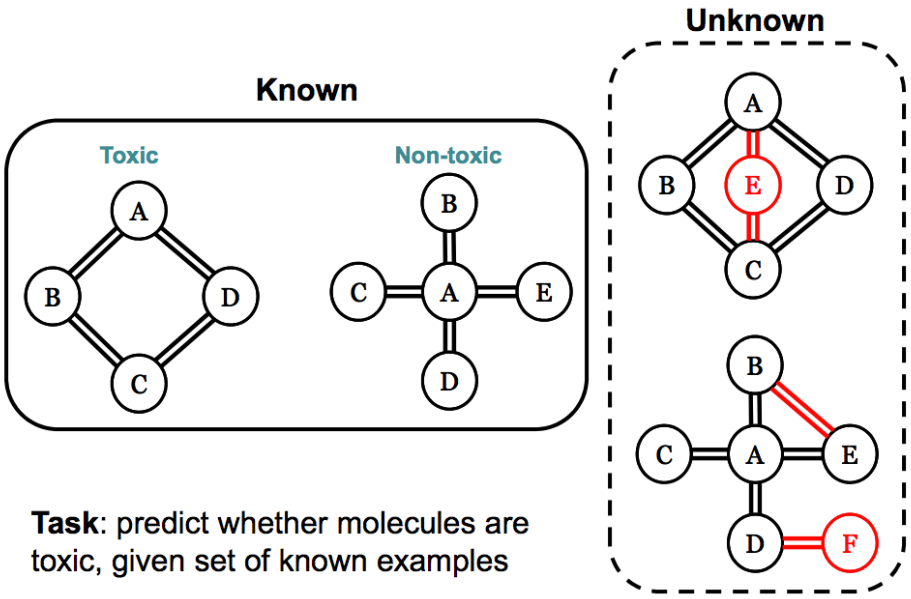


Used to predict future link in a social network (friends or interest) for recommendation, or reconstruct missing links due to incomplete data

Machine Learning on Graphs

Machine learning tasks on graphs:

- Graph classification: given a set of graphs with known class labels for some of them, decide to which class the rest of the graphs belong

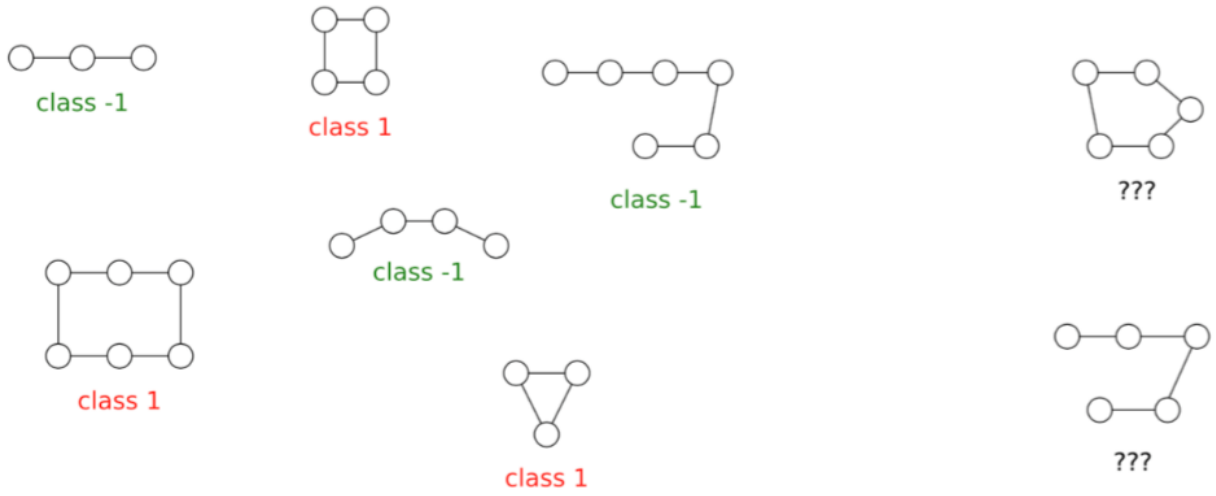


Machine Learning on Graphs

Machine learning tasks on graphs:

- Node classification: given a graph with labels on some nodes, provide a high quality labeling for the rest of the nodes
- Graph clustering: given a graph, group its vertices into clusters taking into account its edge structure in such a way that there are many edges within each cluster and relatively few between the clusters
- Link Prediction: given a pair of vertices, predict if they should be linked with an edge
- **Graph classification: given a set of graphs with known class labels for some of them, decide to which class the rest of the graphs belong**

Graph Classification



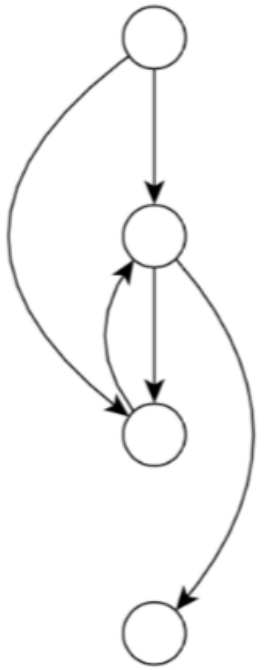
- Input data $x \in \mathcal{X}$
- Output $y \in \{-1, 1\}$
- Training set $\mathcal{S} = \{(x_1, y_1), \ldots, (x_n, y_n)\}$
- Goal: estimate a function $f : \mathcal{X} \to \mathbb{R}$ to predict y from f(x)

Graph Classification Motivation - Anomaly Detection

Given a computer program, create its control flow graph

```
            processed_pages.append(processed_page)
            visited += 1
            links = extract_links(html_code)
            for link in links:
                if link not in visited_links:
                    links_to_visit.append(link)

    return create_vocabulary(processed_pages)


def parse_page(html_code):
    punct = re.compile(r'([^A-Za-z0-9])')
    soup = BeautifulSoup(html_code, 'html.parser')
    text = soup.get_text()
    processed_text = punct.sub(" ", text)
    tokens = processed_text.split()
    tokens = [token.lower() for token in tokens]
    return tokens


def create_vocabulary(processed_pages):
    vocabulary = {}
    for processed_page in processed_pages:
        for token in processed_page:
            if token in vocabulary:
                vocabulary[token] += 1
            else:
                vocabulary[token] = 1

    return vocabulary
```

→



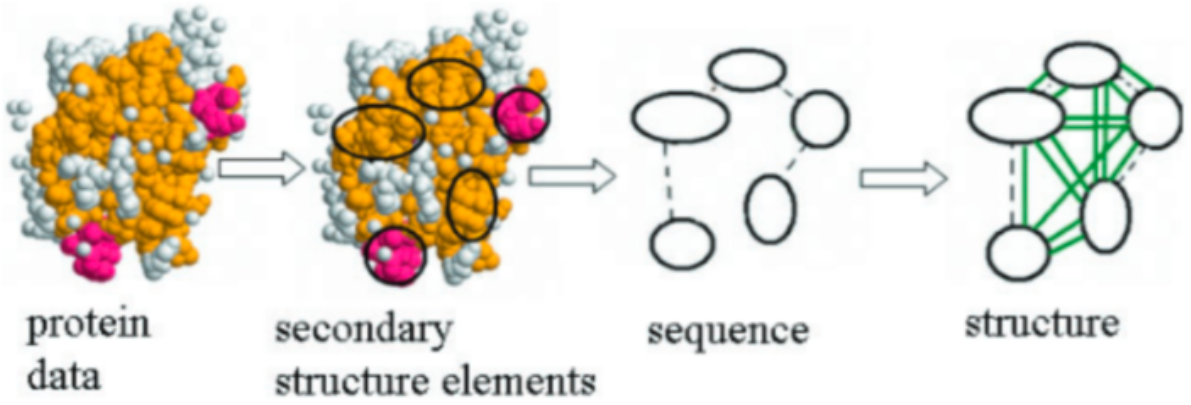
Perform **graph classification** to predict if there is malicious code inside the program or not

Gascon et al. "Structural detection of android malware using embedded call graphs". In AISec'13

Graph Classification Motivation - Protein Function Prediction

For each protein, create a graph that contains information about its

- structure
- sequence
- chemical properties



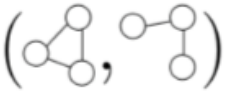
Perform **graph classification** to predict the function of proteins

Borgwardt et al. "Protein function prediction via graph kernels". Bioinformatics 21

Graph Classification

Graph Classification very related to Graph Comparison

Example

$f($  $)$

$+$

$k\text{-}nn$

$=$

graph classification

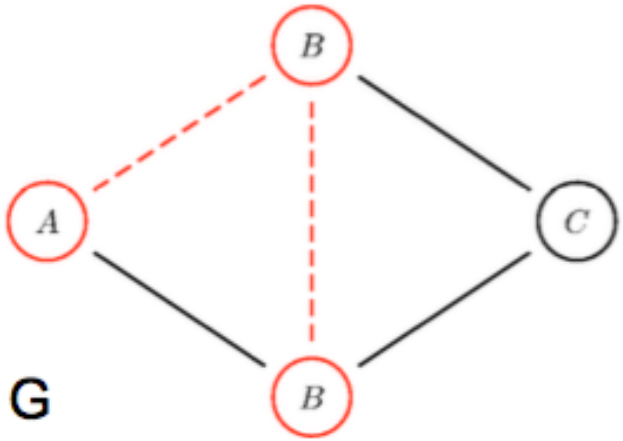
- Although graph comparison seems a very tractable problem, it is very complex.
- We are interested in algorithms capable of measuring the similarity between two graphs in polynomial time.

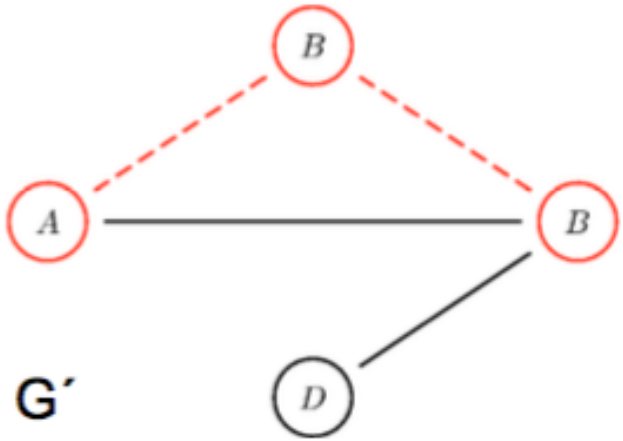
Graph Comparison

Definition 1 (Graph Comparison Problem) *Given two graphs G and G' from the space of graphs $\mathcal{G}$. The problem of graph comparison is to find a mapping*

$$s : \mathcal{G} \times \mathcal{G} \rightarrow \mathbb{R}$$

such that $s(G, G')$ quantifies the similarity (or dissimilarity) of G und G'.





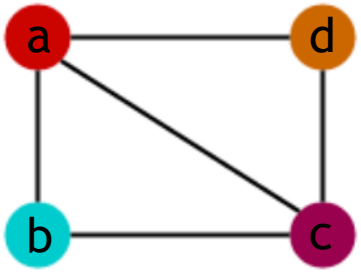
Approaches for Comparing Graphs

A first approach. Graph isomorphism.

1. Graph isomorphism: find a mapping of the vertices of G_1 to the vertices of G_2 s.t. G_1 and G_2 are identical

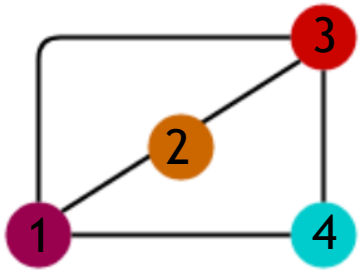
   i.e. (x,y) is an edge of G_1 iff (f(x),f(y)) is an edge of G_2. Then f is an isomorphism, and G_1 and G_2 are called isomorphic

G_1



Neighbors of d are a,c

G_2



Neighbors of 2=f(d)
are 1=f(c) and 3 = f(a)

mapping

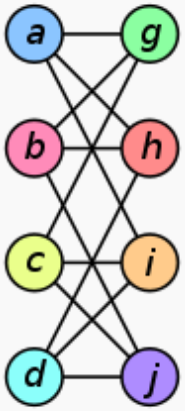
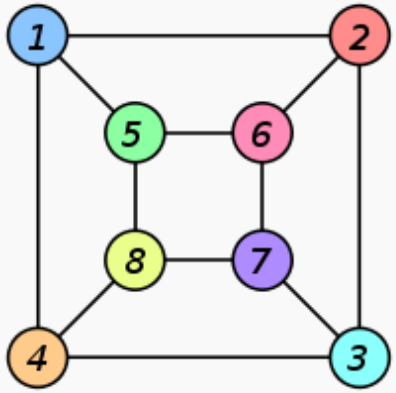
f(a) = 3
f(b) = 4
f(c) = 1
f(d) = 2

G_1 and G_2 are isomorphic.

Approaches for Comparing Graphs

A first approach. Graph isomorphism.

1. Graph isomorphism: find a mapping of the vertices of G_1 to the vertices of G_2 s.t. G_1 and G_2 are identical
   i.e. (x,y) is an edge of G_1 iff (f(x),f(y)) is an edge of G_2. Then f is an isomorphism, and G_1 and G_2 are called isomorphic

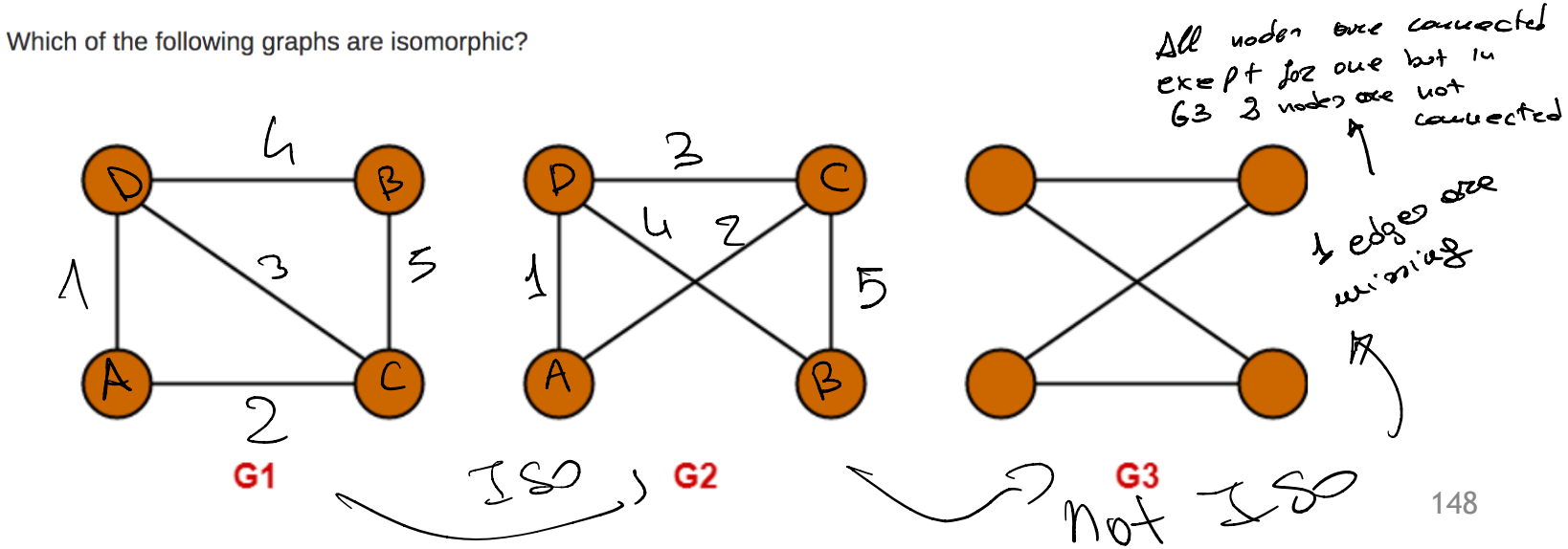
Graph G	Graph H	An isomorphism between G and H
		*f*(*a*) = 1 *f*(*b*) = 6 *f*(*c*) = 8 *f*(*d*) = 3 *f*(*g*) = 5 *f*(*h*) = 2 *f*(*i*) = 4 *f*(*j*) = 7

Approaches for Comparing Graphs

A first approach. Graph isomorphism.

1. Graph isomorphism: find a mapping of the vertices of G_1 to the vertices of G_2 s.t. G_1 and G_2 are identical
   i.e. (x,y) is an edge of G_1 iff (f(x),f(y)) is an edge of G_2. Then f is an isomorphism, and G_1 and G_2 are called isomorphic

Exercise:



Approaches for Comparing Graphs

A first approach. Graph isomorphism.

1. Graph isomorphism: find a mapping of the vertices of G_1 to the vertices of G_2 s.t. G_1 and G_2 are identical
   i.e. (x,y) is an edge of G_1 iff (f(x),f(y)) is an edge of G_2. Then f is an isomorphism, and G_1 and G_2 are called isomorphic
   1. No polynomial-time algorithm is known
   2. Neither is it known to be NP-complete

2. Subgraph isomorphism: find if any subgraph of G_1 is isomorphic to a smaller graph G_2
   1. NP-complete

Approaches for Comparing Graphs

A first approach. Graph isomorphism.

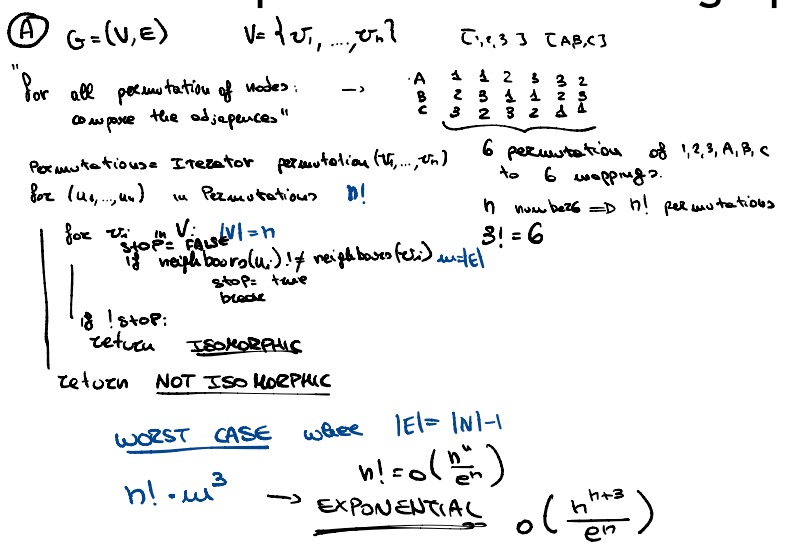
- NP-completeness
  - A decision problem C is NP-complete iff
    - C is in NP
    - C is NP-hard, i.e. every other problem in NP is reducible to it.
- Problems for the practitioner
  - Excessive runtime in worst case
  - Runtime may grow exponentially with the number of nodes
  - For larger graphs with many nodes and for large datasets of graphs, this is an enormous problem

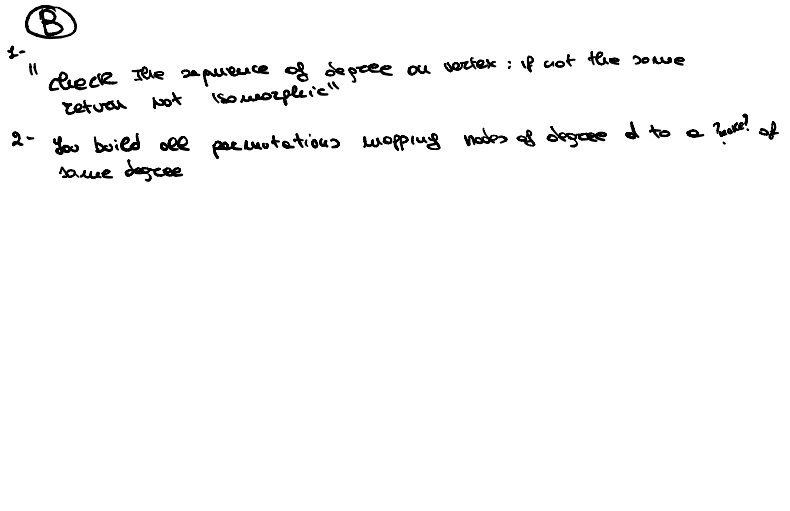
Approaches for Comparing Graphs

A first approach. Graph isomorphism.

Exercise:

1. Propose algorithms to check if 2 graphs (without labels) are isomorphic.
   - A. A brute force one.
   - B. A faster one based on using node degrees.
2. Adapt them to labeled graphs.

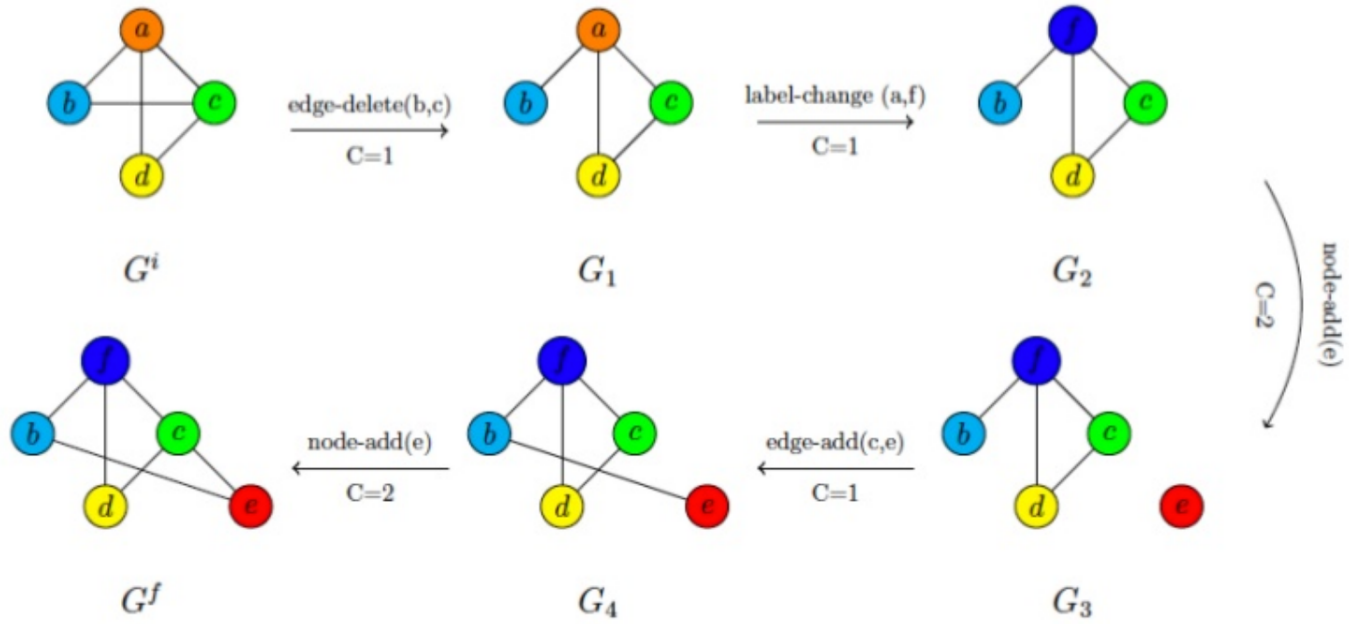




Approaches for Comparing Graphs

A second approach. Graph edit distance:

- Principle:
  - Count necessary operations to transform G1 into G2
  - Assign costs to different types of operations (edge/node insertion/deletion, modification of labels)



Approaches for Comparing Graphs

A second approach. Graph edit distance:

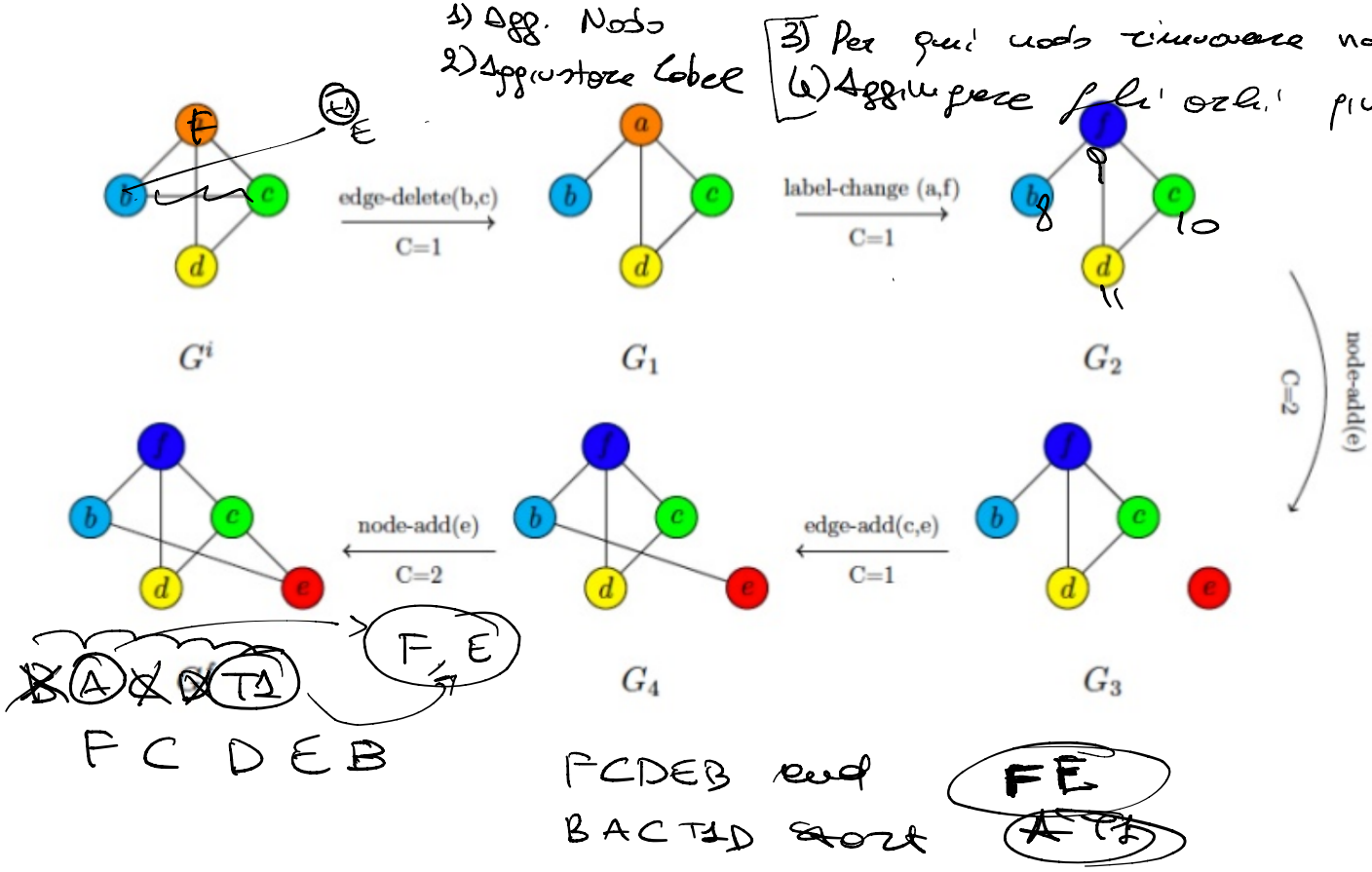
- Advantages
  - Captures partial similarities between graphs
  - Allows for noise in the nodes, edges and their labels
  - Flexible way of assigning costs to different operations

- Disadvantages
  - Contains subgraph isomorphism check as one intermediate step
  - Choosing cost function for different operations is difficult

Approaches for Comparing Graphs

A second approach. Graph edit distance:

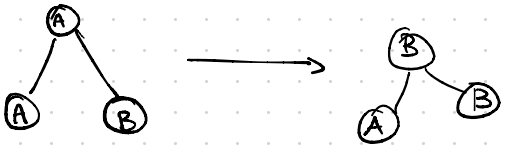
- Exercise: Propose an algorithm to estimate graph edit distance.



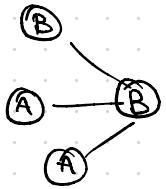
1) Consider a graph of graphs $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ in which each node is a graph and an edge [arc = directed edge] between two graphs G and G' IF we can go from G to G' with a basic operation (Add/Remove Node/Edge modify label)

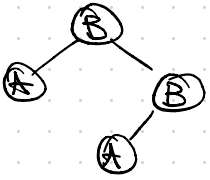
2) Finding the graph edit distance between two graphs G_i, G_j in $\mathcal{G}$ "boils" down to computing the minimum cost

G_i



G_j





PATH between G_i and G_j in $\mathcal{G}$

There are very efficient algorithms to compute it: Dijkstra, Bellman-Ford, A*]

The hard part is building $\mathcal{G}$ so that $\mathcal{G}$ is not too big.

Approaches for Comparing Graphs

A third approach. Topological Descriptors.

- Principle
  - Map each graph to a feature vector
  - Use distances and metrics on vectors for learning on graphs
- Advantages
  - Reuses known and efficient tools for feature vectors
- Disadvantages
  - Efficiency comes at a price: feature vector transformation leads to loss of topological information (or includes subgraph isomorphism as one step)

Approaches for Comparing Graphs

A third approach. Topological Descriptors.

- Wanted: Polynomial-time similarity measure for graphs

-> Graph kernels: compare substructures of graphs that are computable in polynomial time.

Table of Content

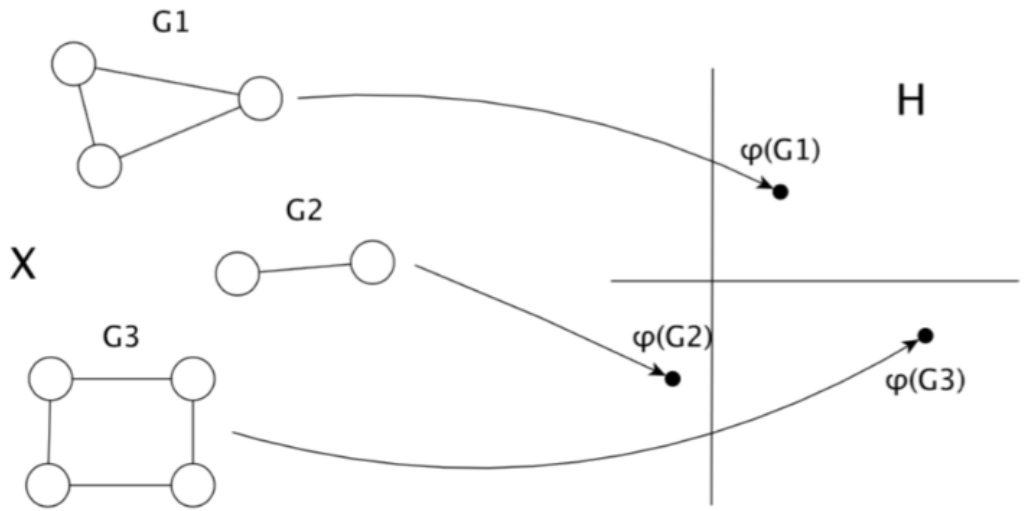
Graph Kernels

A third approach. Topological Descriptors.

Definition (Graph Kernel)

A graph kernel $k : \mathcal{G} \times \mathcal{G} \to \mathcal{R}$ is a kernel function over a set of graphs $\mathcal{G}$

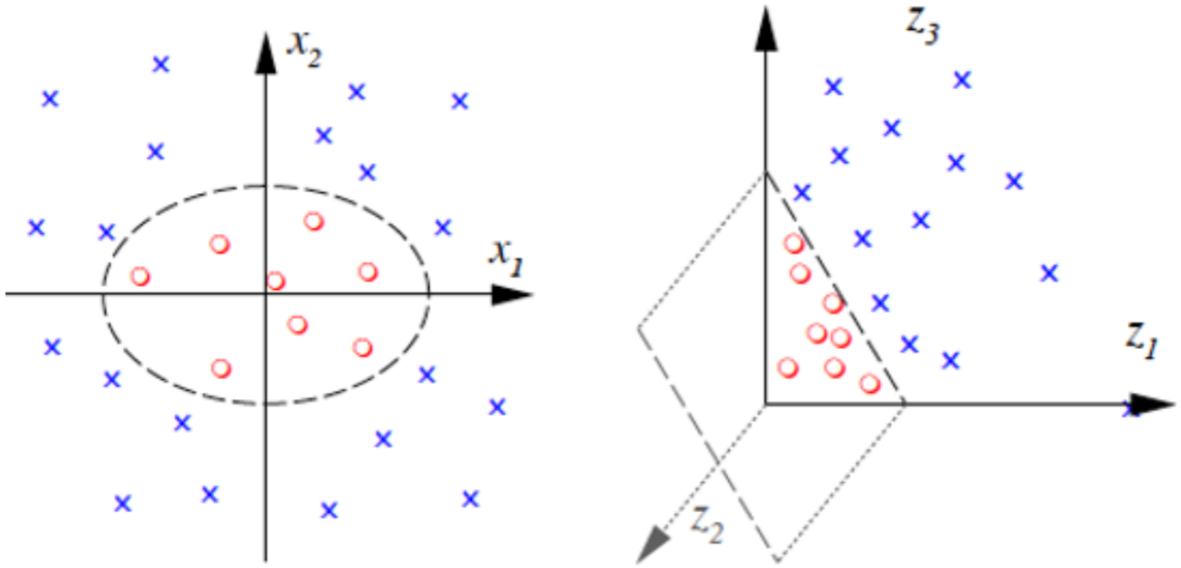
- It is equivalent to an inner product of the embeddings $\phi : \mathcal{X} \to \mathbb{H}$ of a pair of graphs into a Hilbert space: $k(G_1, G_2) = \langle \phi(G_1), \phi(G_2) \rangle$
- Makes the whole family of kernel methods (e.g. SVMs) applicable to graphs



The Kernel Trick (Reminder)

- ML method to use a linear classifier to solve a non linear problem.

$$\Phi : R^2 \to R^3$$
$$(x_1, x_2) \mapsto (z_1, z_2, z_3) := (x_1^2, \sqrt{(2)} x_1 x_2, x_2^2)$$



The Kernel Trick (Reminder)

- In higher dimensions:

The problem in higher dimension can be written as

$$\min_{\alpha_i} \quad L_D(\alpha) = \sum_{i=1}^{n} \alpha_i - \frac{1}{2} \sum_{i=1}^{n} \sum_{i'=1}^{n} \alpha_i \alpha'_i \quad \phi(x_i) \cdot \phi(x'_i)$$
$$\alpha_i \geq 0 \qquad \text{for all } i, 1 \leq i \leq n$$
$$\sum_{i=1}^{n} \alpha_i y_i = 0$$

where $\phi(x)$ is the transformation in the higher dimensional space and

$$K(x_i, x'_i) = \phi(x_i) \cdot \phi(x'_i)$$

is the Kernel function.

- Implicit feature space: coordinates of the data in the high-dimensional space are not computed.

  Principle: use Kernel functions.

Approaches for Comparing Graphs

A third approach. Topological Descriptors.

- Wanted: Polynomial-time similarity measure for graphs

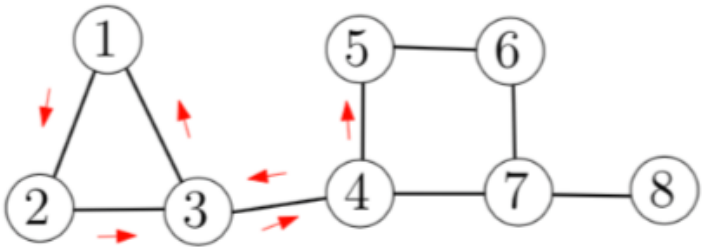
-> Graph kernels: compare substructures of graphs that are computable in polynomial time.

- Criteria for a good graph kernel
  - Expressive
  - Efficient to compute
  - Positive definite
  - Applicable to wide range of graphs

Substructure Based Graph Kernels

A large number of graph kernels compare substructures of graphs that are computable in polynomial time:

- walks



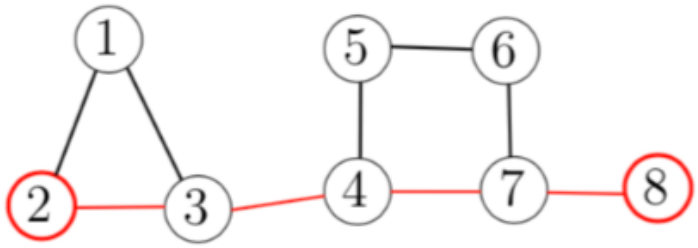
Walk: $4 \to 3 \to 1 \to 2 \to 3 \to 4 \to 5$

Vishwanathan et al. "Graph Kernels". JMLR 11, 2010

Substructure Based Graph Kernels

A large number of graph kernels compare substructures of graphs that are computable in polynomial time:

- walks
- shortest path lengths



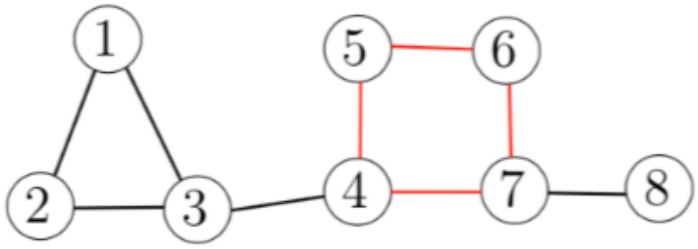
SP length between vertices 2 and 8 : 4

Borgwardt and Kriegel. "Shortest-path kernels on graphs". In ICDM'05

Substructure Based Graph Kernels

A large number of graph kernels compare substructures of graphs that are computable in polynomial time:

- walks
- shortest path lengths
- cyclic patterns



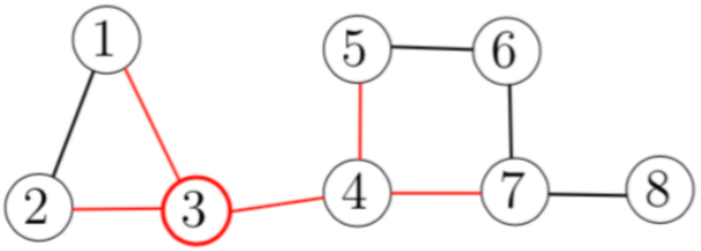
Cycle: $4 \rightarrow 7 \rightarrow 6 \rightarrow 5 \rightarrow 4$

Horváth et al. "Cyclic pattern kernels for predictive graph mining". In KDD'04

Substructure Based Graph Kernels

A large number of graph kernels compare substructures of graphs that are computable in polynomial time:

- walks
- shortest path lengths
- cyclic patterns
- rooted subtrees



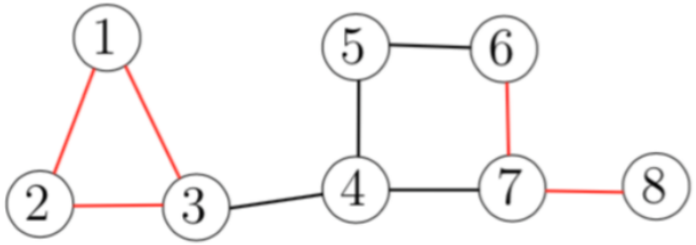
Subtree rooted at vertex 3

Shervashidze et al. "Weisfeiler-Lehman Graph Kernels". JMLR 12, 2011

Substructure Based Graph Kernels

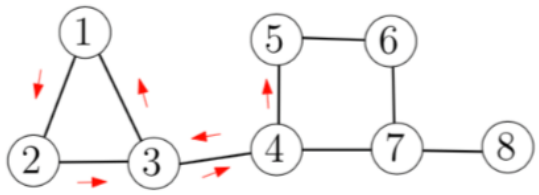
A large number of graph kernels compare substructures of graphs that are computable in polynomial time:

- walks
- shortest path lengths
- cyclic patterns
- rooted subtrees
- graphlets

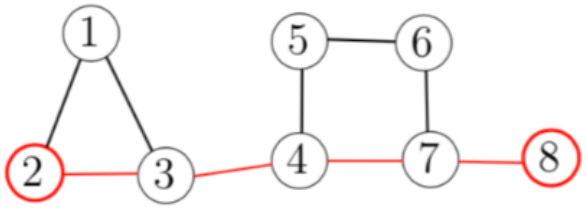


Shervashidze et al. "Efficient graphlet kernels for large graph comparison.". In AISTATS'09

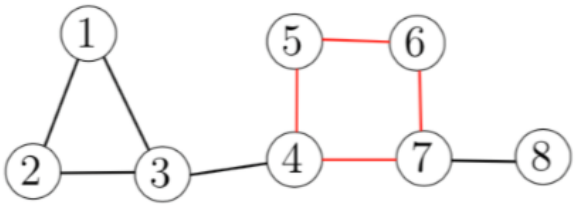
Substructure Based Graph Kernels



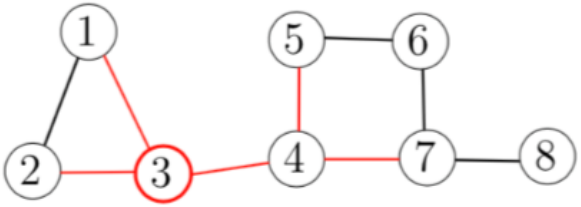
Walk: $4 \to 3 \to 1 \to 2 \to 3 \to 4 \to 5$



SP length between vertices 2 and 8 : 4



Cycle: $4 \to 7 \to 6 \to 5 \to 4$

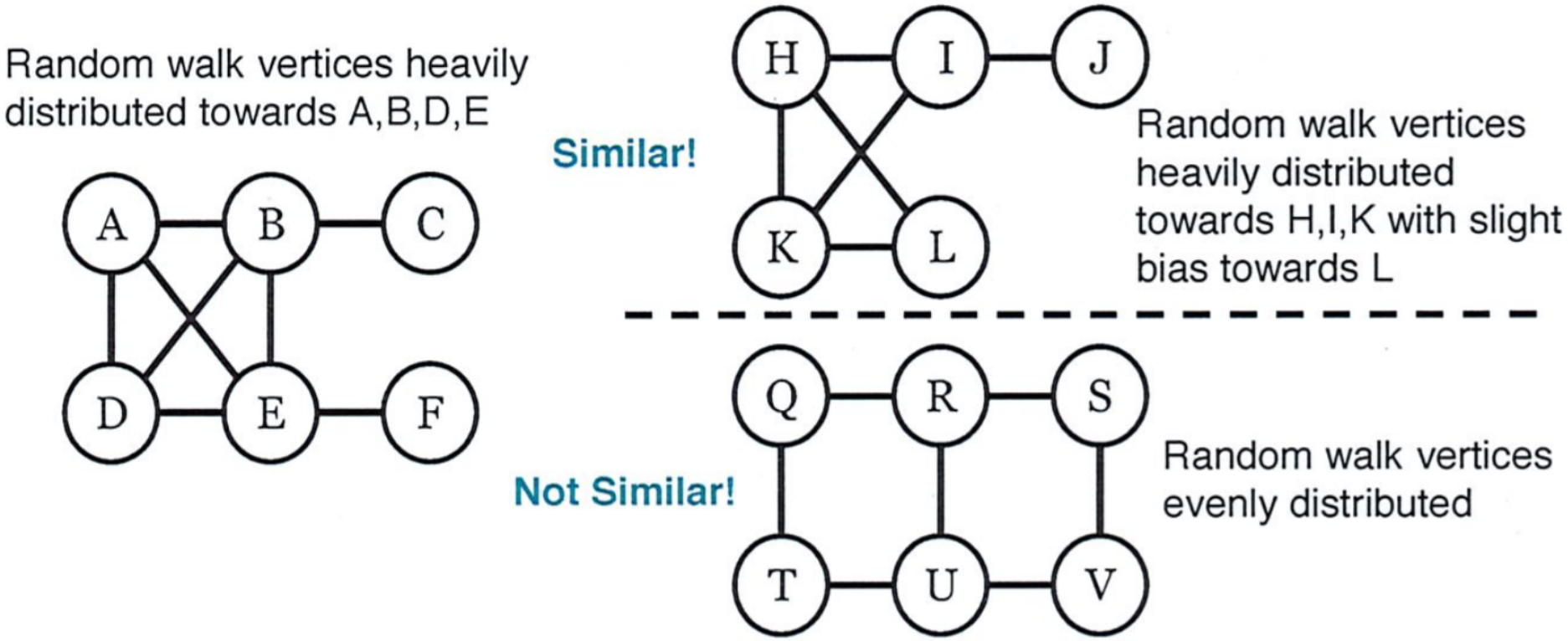


Subtree rooted at vertex 3

Table of Content

Walk-based Similarity

- Intuition — two graphs are similar if they exhibit similar patterns when performing random walks



References

Some slides are courtesy of

- Karsten Borgwardt (University of Cambridge), Xifeng Yan (IBM T. J. Watson Research Center), and Oliver Stegle (Max Planck Institute)

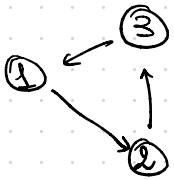
- Bibliography:
  - *Shortest-path kernels on graphs*, K. M. Borgwardt and H.-P. Kriegel, ICDM, 2005.
  - *Marginalized Kernels Between Labeled Graphs*, H. Kashima, K. Tsuda, and A. Inokuchi, ICML, 2003.
  - *On Graph Kernels: Hardness Results and Efficient Alternatives*, Thomas Gartner, P. Flach, and S. Wrobel, COLT, 2003
  - *Deep Graph Kernels*, P. Yanardag, S.V.N. Vishwanathan, KDD, 2015.
  - *Profiling the End Host*, T. Karagiannis, K. Papagiannaki, N. Taft, and M. Faloutsos, PAM 2007.

GRAPHS KERNEL

Power ?? of graphs

A graph G is a set of (n) nodes (or vertices) V and m edges (or labels) E.
An attributed graph is a graph with labels on nodes and/or edges,
we refer to labels as attributes.
The adjacent matrix A of G is identified as

$$[A]_{ij} = \begin{cases} 1 & \text{if } (v_i, v_j) \in E \\ 0 & \text{otherwise} \end{cases}$$



$G = (V, E)$

$V = \{1, 2, 3\}$

$E = \{(1,2), (2,3), (3,1)\}$

$1 \to 2 \to 3$ PATH

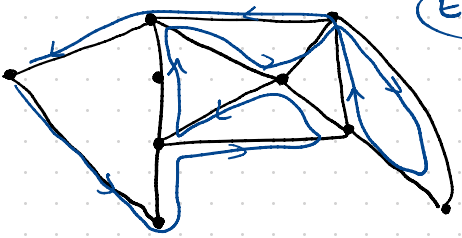
$1 \to 2 \to 3 \to 1 \to 2$ WALK

$$A = \begin{array}{c} \\ 1 \\ 2 \\ 3 \end{array} \begin{array}{c} \begin{array}{ccc} 1 & 2 & 3 \end{array} \\ \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \end{bmatrix} \end{array}$$

- When v_i and v_j are nodes of G.
  A **WALK** w of length $k-1$ in a graph is a sequence of nodes $v_1, v_2, \dots, v_k$ where
  $(v_{i-1}, v_i) \in E$ for $1 < i \leq k$

- A walk W is a **PATH** if $v_i \neq v_j$ iff $i \neq j$ $\forall i \in [1, \dots, k]$
  ↳ WALK with no repetition of vertices

- A **cycle** is a WALK with $v_1 = v_k$
- A **SIMPLE CYCLE** is a cycle with no repeated nodes [except v_1]
- A **graph** is **fully connected** if there is an edge for every pair of node
- A **Hamiltonian Path** is a path that visit every node exactly once.
- A **EULER WALK** is a walk that visit every edge exactly once.



EULERIAN GRAPH

- Determine if an Eulerian CYCLE exists is TRIVIAL:
  - In an undirected graph, such a cycle exist IF:
    - all vertices have even degree

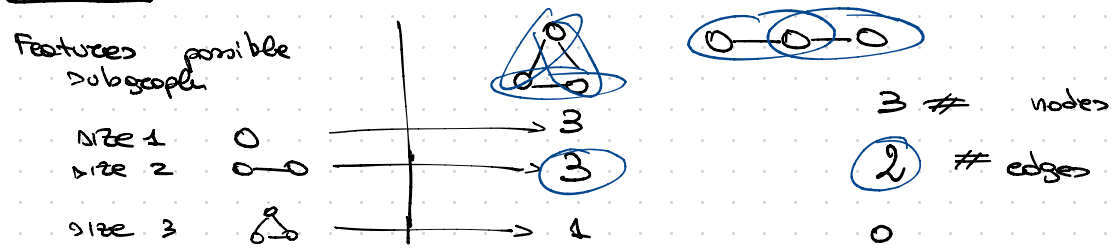
- Determine whether a Hamiltonian path or cycle is NP-COMPLETE

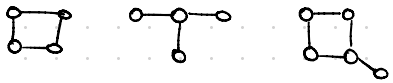
AN IDEAL GRAPH KERNEL

Consider an Ideal graph kernel that has $h-1$ features ϕ_H for each possible graph H.
Each feature measures how many subgraphs of G have the same structure as graph H

(EXAMPLE)

Features possible subgraphs



B) 

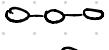
Size 1

0 4 4 5

Size 2

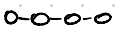
 4 3 5

Size 3

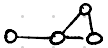
 4 3 5

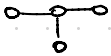
 0 0 0

SIZE 4

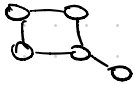
 4 0 1

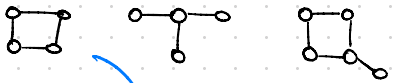
 1 1 1



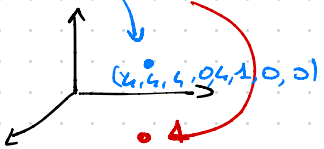
 0 0 1

SIZE 5

 0 0 1



-> For ideal graphs
korvee
1 map the graph
on the axes



Too expressive → NP-COMPLETE

However, the ideal kernel allows graphs with specific substructures, such as a hamiltonian cycle, to be identified.

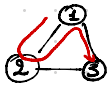
↳ A sequence of adjacent vertices and edges containing all vertices exactly one. which is NP-HARD → we know an exponential algorithm

RANDOM WALK KERNEL

Refresh: walks on sequence of sequences of nodes that allow repetition of nodes.

PRINCIPLE OF RWK: Count COMMON walks in two input graphs G and G'

Feature space: The numbers of walks of length k, $k = 0, 1, 2, 3, \dots$ from node u to node v in a graph G.

EXAMPLE 

A walk of length 3 from nod 1 to node 3

How many walks of length 0?

1 → 1	# 1	i.e.	I have 1 walk	from 1 to 1
1 → 2	# 0	i.e.	I have 0 walk	from 1 to 2
1 → 3	# 0	//	//	// 1 to 3

Put in a table

	1	2	3
1	1	0	0
2	0	1	0
3	0	0	1

How many walks of length 1

	1	2	3
1	0	1	1
2	1	0	1
3	1	1	0

Because if I use 1 edge I go from 1 to 2 or to 3 but I cannot go back

How many walks of length 2

	1	2	3
1	2	1	1
2	1	2	1
3	1	1	2

Length 3?

	1	2	3
1	2	3	3
2	3	2	3
3	3	3	2

Important Trick

- How to compute an efficient method to compute the number of walks of length k

- **Using matrix multiplication**

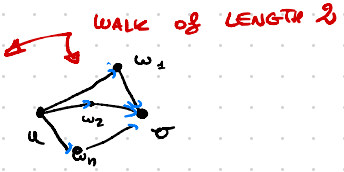
$A = [a_{ij}]$ Adjacency matrix for which $a_{ij} = 1$ if vertices i and j are linked by an edge

	1	2	3
1	0	1	1
2	1	0	1
3	1	1	0

$\emptyset$ otherwise

A^+ represent the walks of length 1

The number of walks of length 2 are the sum of the number of walks from u to w_n times the number of walk from w_n to v



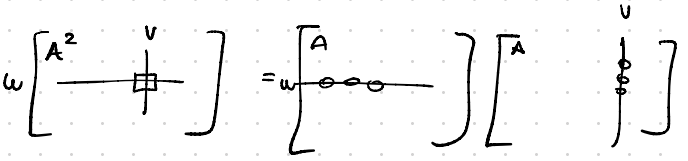
A walk of size 2 from u to v going through v can be noted $u\ w\ v$

$$\#\text{walks}_{\substack{u\to v\\ \text{of size } 2}} = \sum_{w\in V(G)} \#\text{walks}_{u\,w\,v \text{ of size } 2}$$

$$\overset{!}{=} \sum_{w\in V(G)} \#\text{walks}_{u\,w \text{ of size } 1} \bullet \#\text{walks}_{w,v \text{ of size } 1}$$

↑ times

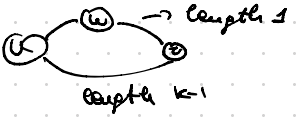
$$\overset{!}{=} \sum_{w\in V(G)} [A^1]_{uw}\ [A^1]_{wv}$$



↳ mimic the matrix multiplication

The number of walks $u\to v$ of length 2 $= A^2_{uv}$

More generally, the number of walks of length k is given by A^k

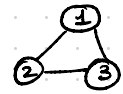


Proof: by induction

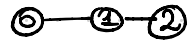
$$\#\text{walks}_{\substack{u\to v\\ \text{length } k}} = \sum_{w\in V(G)} \#\text{walks}_{u\to w \text{ of length } k-1} \bullet \underbrace{\#\text{walks}_{w\to v \text{ of length } 1}}$$

$$A^k_{uv} = \sum_{w\in V(G)} A^{k-1}_{uw} \cdot A_{wv}$$

EXERCISE 1

Compute # of walks of length 4 for 

EXERCISE 2

Compute the walks of length 0, 1, 2, 3, 4 for the graph 

EXERCISE 1

length 0

	1	2	3
1	1	0	0
2	0	1	0
3	0	0	1

1

	1	2	3
1	0	1	1
2	1	0	1
3	1	1	0

2

	1	2	3
1	2	1	1
2	1	2	1
3	1	1	2

3

	1	2	3
1	2	3	3
2	3	2	3
3	3	3	2

$$A^4 = A^3 \cdot A^1 = \begin{bmatrix} 2 & 3 & 3 \\ 3 & 2 & 3 \\ 3 & 3 & 2 \end{bmatrix} \begin{bmatrix} 0 & 1 & 1 \\ 1 & 0 & 1 \\ 1 & 1 & 0 \end{bmatrix} = \begin{bmatrix} 6 & 5 & 5 \\ 5 & 6 & 5 \\ 5 & 5 & 6 \end{bmatrix}$$

EXERCISE 2

$$A^0 = I = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

$$A^1 = \begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 1 \\ 0 & 1 & 0 \end{bmatrix}$$

$$A^2 = A^1 \cdot A^1 = \begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 1 \\ 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 1 \\ 0 & 1 & 0 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 1 \\ 0 & 2 & 0 \\ 1 & 0 & 1 \end{bmatrix}$$

$$A^3 = A^2 \cdot A^1 = \begin{bmatrix} 0 & 2 & 0 \\ 2 & 0 & 2 \\ 0 & 2 & 0 \end{bmatrix}$$

$$A^4 = A^3 \cdot A^1 \overset{!}{=} A^2 \cdot A^2 = \begin{bmatrix} 2 & 0 & 2 \\ 0 & 4 & 0 \\ 2 & 0 & 2 \end{bmatrix}$$

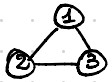
We take a set of features the number of walks of any length $k=0,1,\dots$, between each pair of nodes.

For example for graph

walks of length $0 = [\ ]$

" $1 = [\ ]$

" $2 = [\ ]$



# of nodes

$n = |V(G)|$

If we limit ourselves to walks of length k_{MAX}, we have $(k_{MAX}+1)\cdot n^2$ features

Note that the features can be represented as a vector X, the features vector

$$X = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix} \text{ line 1 of } A^0 \quad \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix} \text{ " 2 "} \quad \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} \text{ 3 "} \quad \begin{bmatrix} 0 \\ 1 \\ 1 \end{bmatrix} \text{ line 1 of } A^1 \quad \begin{bmatrix} \bullet \\ \bullet \\ \bullet \end{bmatrix}$$

Remark for the future:

In general, we will not stop at k_{MAX}, we will have an infinite number of features

---

Comparing Graphs / Comparing Vectors

To compare 2 graphs, we can compute the feature values (feature vector) and, then, compare them using a similarity value.

Classic similarity value: **The cosmic similarity**

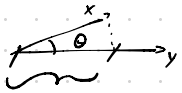
The cosmic similarity is a measure of similarity between 2 and $\emptyset$ vectors measuring the cosine of the angle θ between the similarity $\Rightarrow \cos\theta = \frac{X\cdot Y}{\|X\|\|Y\|}$

Remark: The "dot" product of two vectors

$$X \cdot Y = \begin{bmatrix} x_1 \\ \vdots \\ x_n \end{bmatrix} \begin{bmatrix} y_1 \\ \vdots \\ y_n \end{bmatrix} = x_1y_1 + x_2y_2 + \dots + x_ny_n$$

The metric ranges from 1 to -1

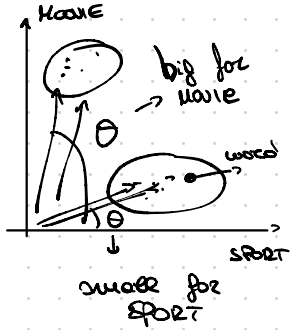
- 1 means similarity
- -1 " dissimilarity
- 0 " decorrelation



It is a judgement of orientation and not of magnitude [vector words ???]

The cosine similarity is particular used in the positive state, where the outcome is between 0 and 1 and for high dimensional space
In particular in information retrieval value and text meaning

Each term is associated to different dimension
A document is ?characterized? by a vector where the value in each dimension corresponds to the number of times the term appears in the document



Cosine Similarity then gives a measures how similar two documents are likely to be in terms of their subject matter

θ near ø -> similarity close to 1
θ near 90° -> " " " ø

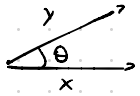
The fact that the vector norm is not taken into account allows not to discriminate short or long text.

For graphs, we will not use the cosine similarity value, but a kernel value

$$k(x, y) = x \cdot y$$

Reminder:

$$\cos\theta = \frac{x \cdot y}{\|x\| \cdot \|y\|}$$



Similarity and Graph Kernel

For RW for graph kernel, the similarity is given by the kernel function:

$k(x, y) = x \cdot y$ with : x = The number of walks between all pairs of nodes in graph G
y = // // // // in graph G'

Another trick How to compute efficiently the number of common walks in two graph

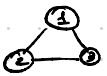
→ Use the **Directed Product Graph** of G and G'

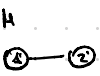
Def: Directed Product Graph: The direct product $G \times H$ of graphs G and H is a graph such that:
- $V(G \times H)$ is the cartesian product $V(G) \times V(H)$ and
- distinct vertices (u, u') and (v, v') are adjacent in $G \times H$ if and only if u is adjacent to v (in G) and u' is adjacent to v' (in H)

Def: The cartesian product of two sets A and B denoted as $A \times B$ is the set of all ordered pairs (a, b) when $a \in A$ and $b \in B$. That is

$$A \times B = \{ (a, b) \mid a \in A \text{ and } b \in A \}$$

EXAMPLE

$G =$ 

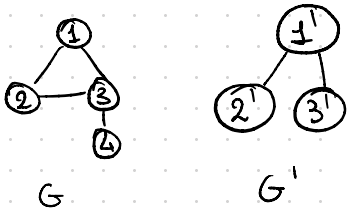
H 

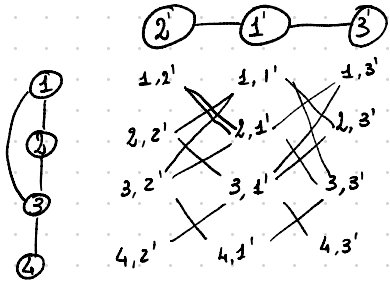
$G \times H$

	1'	2'
1	1,1'	1,2'
2	2,1'	2,2'
3	3,1'	3,2'

→ chose the edges; if the nodes are adjacent!

EXERCISE write the directed product of





Each walk in the direct product graph corresponds to a walk in G x H

EXAMPLE

G x H

	1'	2'
1	1,1'	1,2'
2	2,1'	2,2'
3	3,1'	3,2'

walk 

correspond to $1 \to 3 \to 2 \to 1$ in G

$1' \to 2' \to 1' \to 2'$ in H

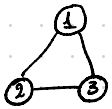
And Vice-versa thus:

The number of walks in $G \times H$ of length k = $\sum_{i,j}$ (# walks in G of length k)$_{ij}$ · (# walks in H of length k)$_{ij}$

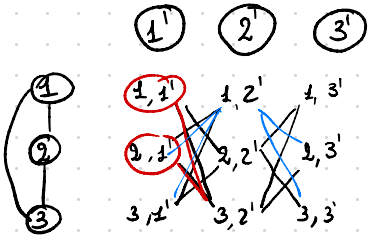
This is what we need to compute for similarity measure for? Kernel function.

Related to the kernel trick, No need to explicitly compute the feature vectors of G and H'

How many walks between $(1,1')$ and $(2,3')$ of length 2 in the direct product graph of

$G =$ 

$G' =$ 



$\nexists$ walks of length 2 from $(1,2')$ to $(1,2')$ is $2 \times 2 = 4$

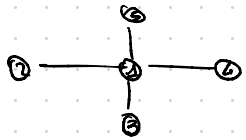
Kernel function $K(x,y) = x \cdot y$

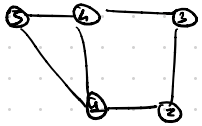
When using the direct product graph, the kernel function of two graphs G and G' is given by

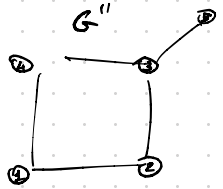
$$K_\times(G,G') = \sum_{i,j=1}^{|V_\times|} \left[\sum_{k=0}^{\infty} A_\times^k \right]_{ij}$$ where $A_\times$ is the adjacent matrix of $G \times G'$

EXERCISE

HW3

G 

G' 

G'' 

Compute the kernel value of G and G' = $K(G,G')$ } discuss it
$K(G',G'')$

"

Instead of ∞ in the formula I go to 4 $\sum_{k=0}^{4}$

Play with graphs and exhibit triplets of graphs for which

2 similar and 2 different

KERNEL FUNCTION

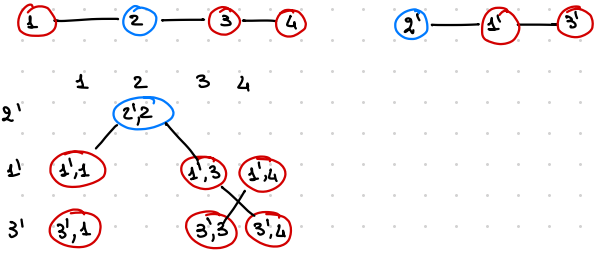
Kernel function $K(x,y) = x \cdot y$.
With kernel trick I avoid to make all the computation because I use the direct product graph.
The kernel function of two graphs G and G' is given by $k_\times(G,G') = \sum_{i,j=1}^{|V_\times|} \left[\sum_{k=0}^{\infty} A_\times^k \right]_{i,j}$

where $A_\times$ is the adjacency matrix of $G \times G'$

- Product graph in the case of labeled graphs

Def: $G_\times = (V_\times, E_\times)$ s.t.
$V_\times = \{(v_1, w_1) \in V_1 \times V_2 \mid \text{label}(v_1) = \text{label}(v_2)\}$
$E_\times = \{(v_1, w_1), (v_2, w_2) \in V(G_1 \times G_2), (v_1, v_2) \in E_1 \text{ and } (w_1, w_2) \in E_1 \text{ and } \text{label}(v_1, v_2) = \text{label}(w_1, w_2)\}$

Meaning: the direct product graph consists of pairs of identically labeled nodes and edges from G_1 and G_2
ex:



- To summarize let G and G' be two labeled graphs.
The RWK of G and G' denoted as $K_\times(G,G')$ is defined as the number of pairs of matching walks (when matching means "same length and labels") weighted by their length.
It can be computed with the following formula:

$$K_\times(G,G') = \sum_{i,j=1}^{|V_\times|} \left[\sum_{n=0}^{\infty} \lambda^n A_\times^n \right]_{i,j} = e^T (I - \lambda A_\times)^{-1} \cdot e \quad \text{where } e = \begin{bmatrix} 1 \\ \vdots \\ 1 \end{bmatrix}$$

where $\lambda < 1$ is a decaying factor for the sum to converge with $A_\times$ the adjacency matrix of $G_\times$ the labeled direct graph of G and G'

- Computing $\sum_{k=0}^{\infty} A^k$

I) A solution is of course to limit the length of the walk to k_{max}.
For example, if we limit the length of the walks to $k_{max} = 3$.
We get $\sum_{k=0}^{3} A^k = A^0 + A^1 + A^2 + A^3$

II) But, there exists efficient ways to compute the infinite sum.

Intuition: If we consider the geometric series $\sum_0^{\infty} \gamma^i = 1+\gamma+\gamma^2$.
This is known to converge if and only if $|\gamma|<1$.
In this case, the limit is given by: $\lim_{n\to+\infty} \sum_{i=0}^{n} \gamma^i = \frac{1}{1-\gamma}$

we have an equivalent with matrices under some condition, we have:

$$\lim_{n\to+\infty} \sum_{i=0}^{n} \gamma^i \epsilon^i = \frac{I}{I-\gamma\epsilon}$$

more formally:

Def: Let ϵ be a square matrix, then the sequence $\{S_n\}_{n\geq 0}$ defined by $S_n = I+\epsilon+\epsilon^2+\dots+\epsilon^n \quad S_0 = I$ is called the geometric series generated by ϵ.

The series converges if the sequence $\{S_n\}_{n\geq 0}$ converges, we then write $\sum_{n=0}^{\infty} \epsilon^n = \lim_{n\to+\infty} S_n$

THEOREM: The geometric series generated by ϵ converges if and only if:

1) $|\lambda_i|<1$ for each eigenvalue λ_i of ϵ.
If this condition holds, then $I-\epsilon$ is invertible and we have

2) $S_n = \sum_{k=0}^{n-1} \epsilon^k = (I-\epsilon)^{-1}(I-\epsilon)^n$

and hence the series converges to

3) $\sum_{k=0}^{\infty} \epsilon^k = (I-\epsilon)^{-1} = \frac{I}{(I-\epsilon)}$

REMARK: condition (1) is equivalent to (4)
$\lim_{n\to+\infty} \epsilon^n = 0$

$$K_x(G,G') = \sum_{i,j=1}^{|V_x|} \left[\sum_{h=0}^{\infty} \lambda^h A_x^h \right]_{i,j}$$

$$= e^T (I-\lambda A_x)^{-1} e \quad \text{with} \quad e = \begin{bmatrix} 1 \\ \vdots \\ 1 \end{bmatrix}$$

RUNTIME / COMPLEXITY of RANDOM WALKS

NOTATION: Given two graphs G and G', n is the number of nodes in G and G'

1st STEP: Compute the direct graph

- **paired nodes**: create all pairs of nodes in G and G' $\Rightarrow$ $O(n^2)$

- **paired edges**: an edge is a pair of nodes in V_X.
  Compare all pairs of nodes (n^2) in G and G' $\Rightarrow$ $O(n^4)$

  i.e. [number of edges in a graph with $O(n^2)$ vertices]

2° STEP Invert the adjacency matrix A_X

- Matrix multiplication of inversion for $n^2 \times n^2$ matrix $\Rightarrow$ $O(n^6)$ runtime

TOTAL RUNTIME is $O(n^6)$ ← "what we use"

↳ can speed up to $O(n^3)$

Classic algorithms:

$O(n^3)$ for $n \times m$ matrix

Best Algorithms:

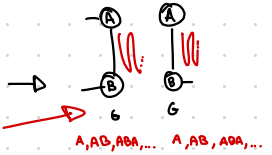
$O(n^a) = O(n^{2.371})$

We achieved our goal to have a polynomial method to compare graphs

PROBLEM OF RANDOM WALK KERNEL: TOTTERING

- Walks allow repetitions of nodes
- A walk can visit the same cycle of nodes one or more times

↳ A small substructure can cause a huge kernel value (TOTTERING)



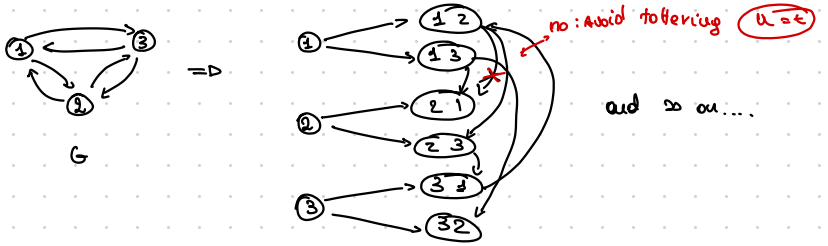
PREVENT TOTTERING

- Explicitely forbid tottering between 2 nodes that is any walk $(v_1, v_2, \dots, v_\ell)$ such that $v_i = v_{i+2}$ for any $i \in \{1, \dots, \ell-2\}$
- Special transformation of each of the input graph $G = (V, E)$:
  - Create new graph G_T with $V_T = V \cup E$ and

$$E_T = \{(v, (v,\epsilon)) \mid v \in V, (v,\epsilon) \in E\} \cup \{((u,v),(v,\epsilon)) \mid (u,v) \in E, (v,\epsilon) \in E, u \neq \epsilon\}$$

  - The set of nodes of G_T is the set of vertices and edges of G
  - In G_T, there are directed edges between each node from G and each adjacent edge, and between edges from G that share exactly a node

[i.e. the target node is one edge and the sequence node in the other]



PROPERTY 1: Walks in G_T correspond to walks in G, but it is not possible to totter between two nodes.

EXERCISE: Build H_T from H