Machine Learning for Graphs based on Kernels @ DaSciM

M. Vazirgiannis & G. Nikolentzos

Data Science and Mining Team (DASCIM), LIX
École Polytechnique
Google Scholar: https://bit.ly/2rwmvQU
Twitter: @mvazirg

January, 2022

Graphs Are Everywhere

Why graphs?

Motivation - Text Categorization

Given a text, create a graph where

- vertices correpond to terms
- two terms are linked to each other if they co-occur within a fixed-size sliding window

Rousseau et al. "Text categorization as a graph classification problem.". ACL'15

Motivation - Text Categorization

Intuition: documents sharing same subgraphs belong to the same class

Given a set of documents and their graph representations:

Extract frequent subgraphs

- from the set of graphs

  or

- from the set of the main cores of the graphs

Then, use frequent subgraphs as features for classification

Motivation - Protein Function Prediction

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

- structure
- sequence
- chemical properties

Use graph kernels to

- measure structural similarity between proteins
- predict the function of proteins

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

Motivation - Chemical Compound Classification

Represent each chemical compound as a graph

Use a frequent subgraph discovery algorithm to discover the substructures that occur above a certain support constraint

Perform feature selection

Use the remaining substructures as features for classification

Deshpande et al. "Frequent substructure-based approaches for classifying chemical compounds". TKDE 17(8)

Motivation - Malware 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
```

$\rightarrow$

Compare the control flow graph of the problem against the set of control flow graphs of known malware

If it contains a subgraph isomorphic to these graphs $\rightarrow$ malicious code inside the program

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

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 $G \in \mathcal{X}$
- Output $y \in \{-1, 1\}$
- Training set $\mathcal{D} = \{(G_1, y_1), \ldots, (G_n, y_n)\}$
- Goal: estimate a function $f : \mathcal{X} \to \mathbb{R}$ to predict y from $f(x)$

Definition (Graph Comparison Problem)

Given two graphs G_1 and G_2 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 $s(G_1, G_2)$ quantifies the similarity of G_1 and G_2.

Graph comparison is a topic of high significance

- It is the central problem for all learning tasks on graphs such as clustering and classification
- Most machine learning algorithms make decisions based on the similarities or distances between pairs of instances (e.g. k-nn)

Not an Easy Problem

Although graph comparison seems a tractable problem, it is very complex

Many problems related to it are **NP-complete**

- subgraph isomorphism
- finding largest common subgraph

We are interested in algorithms capable of measuring the similarity between two graphs in **polynomial** time

Graphs to Vectors

- To analyze and extract knowledge from graphs, one needs to perform machine learning tasks
- Most machine learning algorithms require the input to be represented as a fixed-length feature vector
- There is no straightforward way to transform graphs to such a representation

What is a Kernel?

Definition (Kernel Function)

The function $k : \mathcal{X} \times \mathcal{X} \to \mathbb{R}$ is a kernel if it is:

1. symetric: $k(x, y) = k(y, x)$
2. positive semi-definite: $\forall x_1, x_2, \ldots, x_n \in \mathcal{X}$, the Gram Matrix K defined by $\mathrm{K}_{ij} = k(x_i, x_j)$ is positive semi-definite

- If a function satisfies the above two conditions on a set $\mathcal{X}$, it is known that there exists a map $\phi : \mathcal{X} \to \mathcal{H}$ into a Hilbert space $\mathcal{H}$, such that:

$$k(x, y) = \langle \phi(x), \phi(y) \rangle$$

  for all $(x, y) \in \mathcal{X}^2$ where $\langle \cdot, \cdot \rangle$ is the inner product in $\mathcal{H}$

- Informally, $k(x, y)$ is a measure of similarity between x and y

Definition (Graph Kernel)

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

- It is equivalent to an inner product of the embeddings $\phi : \mathcal{X} \to \mathcal{H}$ of a pair of graphs into a Hilbert space
- Makes the whole family of kernel methods applicable to graphs

Kernel Trick

- Many machine learning algorithms can be expressed only in terms of inner products between vectors
- Let $\phi(G_1), \phi(G_2)$ be vector representations of graphs G_1, G_2 in a very high (possibly infinite) dimensional feature space
- Computing the explicit mappings $\phi(G_1), \phi(G_2)$ and their inner product $\langle \phi(x), \phi(y) \rangle$ for the pair of graphs can be computationally demanding
- The kernel trick avoids the explicit mapping by directly computing the inner product $\langle \phi(x), \phi(y) \rangle$ via the kernel function

Example

Let $\mathcal{X} = \mathbb{R}^2$ and
$x = [x_1, x_2]^\top, y = [y_1, y_2]^\top \in \mathcal{X}$

For any $x = [x_1, x_2]^\top$ let ϕ be a map
$\phi : \mathbb{R}^2 \to \mathbb{R}^3$ defined as:

$$\phi(x) = [x_1^2, \sqrt{2}x_1x_2, x_2^2]^\top$$

Let also $k : \mathcal{X} \times \mathcal{X} \to \mathbb{R}$ a kernel defined as $k(x, y) = \langle x, y \rangle^2$. Then

$$\begin{aligned} k(x, y) &= \langle x, y \rangle^2 \\ &= (x_1y_1 + x_2y_2)^2 \\ &= x_1^2y_1^2 + 2x_1y_1x_2y_2 + x_2^2y_2^2 \\ &= \langle \phi(x), \phi(y) \rangle \end{aligned}$$

Classification using SVM

- The standard SVM classifier addresses the following problem:
  Given a set of N training objects along with their class labels $\mathcal{D} = \{(x_i, y_i)\}_{i=1}^N$, $x_i \in \mathcal{X}$, $y_i \in \mathcal{Y} = \{-1, +1\}$, learn a classifier $f : \mathcal{X} \to \mathcal{Y}$ that predicts the class labels of new objects

- SVM belongs to the family of large margin classifiers
  $\hookrightarrow$ it seeks a hyperplane that separates the instances belonging to class -1 from those belonging to class 1

- This1leads to the following dual optimization problem:

$$
\begin{aligned}
\textit{maximize}_\alpha \quad & \sum_{i=1}^N \alpha_i - \frac{1}{4} \sum_{i=1}^N \sum_{j=1}^N \alpha_i \alpha_j y_i y_j \langle \phi(x_i), \phi(x_j) \rangle \\
\text{subject to} \quad & \sum_{i=1}^N \alpha_i y_i = 0 \\
& C \geq \alpha_i \geq 0 \quad \forall i \in \{1, \ldots, N\}
\end{aligned}
$$

Graph Classification using SVM

- The standard SVM classifier addresses the following problem:
  Given a set of N training objects along with their class labels $\mathcal{D} = \{(G_i, y_i)\}_{i=1}^{N}$, $G_i \in \mathcal{G}$, $y_i \in \mathcal{Y} = \{-1, +1\}$, learn a classifier $f : \mathcal{X} \rightarrow \mathcal{Y}$ that predicts the class labels of new objects

- SVM belongs to the family of large margin classifiers
  $\hookrightarrow$ it seeks a hyperplane that separates the instances belonging to class -1 from those belonging to class 1

- This leads to the following dual optimization problem:

$$\begin{aligned} \textit{maximize}_{\alpha} \quad & \sum_{i=1}^{N} \alpha_i - \frac{1}{4} \sum_{i=1}^{N} \sum_{j=1}^{N} \alpha_i \alpha_j y_i y_j \langle \phi(G_i), \phi(G_j) \rangle \\ \text{subject to} \quad & \sum_{i=1}^{N} \alpha_i y_i = 0 \\ & C \geq \alpha_i \geq 0 \quad \forall i \in \{1, \ldots, N\} \end{aligned}$$

Graph Classification using SVM

- The standard SVM classifier addresses the following problem:
  Given a set of N training objects along with their class labels $\mathcal{D} = \{(G_i, y_i)\}_{i=1}^{N}$, $G_i \in \mathcal{G}$, $y_i \in \mathcal{Y} = \{-1, +1\}$, learn a classifier $f : \mathcal{X} \rightarrow \mathcal{Y}$ that predicts the class labels of new objects

- SVM belongs to the family of large margin classifiers
  $\hookrightarrow$ it seeks a hyperplane that separates the instances belonging to class -1 from those belonging to class 1

- This leads to the following dual optimization problem:

$$\begin{aligned} \textit{maximize}_{\alpha} \quad & \sum_{i=1}^{N} \alpha_i - \frac{1}{4} \sum_{i=1}^{N} \sum_{j=1}^{N} \alpha_i \alpha_j y_i y_j k(G_i, G_j) \\ \text{subject to} \quad & \sum_{i=1}^{N} \alpha_i y_i = 0 \\ & C \geq \alpha_i \geq 0 \quad \forall i \in \{1, \ldots, N\} \end{aligned}$$

Two Simple Kernels

The two kernels assume node/edge-labeled graphs

Vertex histogram kernel:

f_i : frequency of label i

- The vertex label histogram of a graph G is a vector $f = [f_1, f_2, \ldots, f_d]^\top$, such that $f_i = |\{v \in V : \ell(v) = i\}|$ for each $i \in \mathcal{L}$
- The vertex histogram kernel is then defined as:

$$k(G, G') = \langle f, f' \rangle$$

Edge histogram kernel:

- The edge label histogram of a graph G is a vector $f = [f_1, f_2, \ldots, f_d]^\top$, such that $f_i = |\{(v, u) \in E : \ell(v, u) = i\}|$ for each $i \in \mathcal{L}$.
- The edge histogram kernel is then defined as:

$$k(G, G') = \langle f, f' \rangle$$

Example

The vector representations of the two graphs are:

$$f_G = [2, 2, 1, 0]^\top$$
$$f_{G'} = [1, 1, 3, 1]^\top$$

Hence, the value of the kernel is:

$$k(G, G') = \langle f_G, f_{G'} \rangle = 7$$

Expressiveness vs Efficiency

Complete Graph Kernels

Definition (Complete Graph Kernel)

A graph kernel $k(G_1, G_2) = \langle \phi(G_1), \phi(G_2) \rangle$ is complete if ϕ is injective

Hence, for complete graph kernels, $\phi(G_1) = \phi(G_2)$ iff G_1 and G_2 are isomorphic

How **hard** is to compute a complete graph kernel?

Proposition

Computing any complete graph kernel is at least as hard as the graph isomorphism problem

[Gartner et al., COLT/Kernel'03]

Complete Graph Kernels

Clearly, the vertex and edge histogram kernels are **not** complete

G

G'

The two graphs are not isomorphic. However

$$f_G = f_{G'} = [2, 2, 1, 0]^\top$$

Expressiveness vs Efficiency

If the kernel is complete:

- Computation is at least as hard as the graph isomorphism problem
  $\hookrightarrow$ No polynomial algorithm for the graph isomorphism problem is known

If the kernel is not complete:

- It can be computed efficiently
- We can have $\phi(G_1) = \phi(G_2)$ even if $G_1 \not\cong G_2$
  $\hookrightarrow$ The kernel is not expressive enough

We are interested in kernels that can be computed in polynomial time (with small degree)

Expressive Power of Graph Kernels

Capitalize on concepts from property testing to measure the expressive power of graph kernels

Definition

A graph kernel *identifies* a property if no two graphs are mapped to the same feature vector unless they both have or both do not have the property (e. g., connected vs disconnected)

Property \ Kernel	Weisfeiler-Lehman subtree kernel	Random Walk kernel	Shortest Path kernel	Graphlet kernel
Connectivity	X	X	V	X
Planarity	X	X	X	X
Bipartiteness	X	X	X	X
Triangle-freeness	X	X	X	V

Well-established kernels fail to identify fundamental properties
$\hookrightarrow$ However, still they achieve state-of-the-art results on many datasets

Early Days of Graph Kernels

Convolution Kernels in a Nutshell

- Decompose structured objects into comparable parts
- Aggregate the values of similarity measures for individual parts

[Haussler. Tech Report'99]

R-Convolution

- Let X be a set of composite objects (e.g., cars), and $\bar{X}_1, \ldots, \bar{X}_D$ be sets of parts (e.g., wheels, brakes, etc.). All sets are assumed countable.
- Let R denote the relation "being part of":

$$R(\bar{x}_1, \ldots, \bar{x}_D, x) = 1, \text{ iff } \bar{x}_1, \ldots, \bar{x}_D \text{ are parts of } x$$

- The inverse relation R^{-1} is defined as:

$$R^{-1}(x) = \{\bar{\mathrm{x}} : R(\bar{\mathrm{x}}, x) = 1\}$$

  In other words, for each object x, $R^{-1}(x)$ is a set of component subsets, that are part of x
- We say that R is finite, if R^{-1} is finite for all $x \in X$

Example

- x is a string

- Subpart relation $R(\bar{x}_1, \bar{x}_2, x) = 1$ iif $\bar{x}_1, \bar{x}_2$ are (non-empty) strings such that $x = \text{concat}(\bar{x}_1, \bar{x}_2)$

$x = \text{table}$

$\bar{x}_1 = \text{t}$, $\bar{x}_2 = \text{able}$
$\bar{x}_1 = \text{ta}$, $\bar{x}_2 = \text{ble}$
$\bar{x}_1 = \text{tab}$, $\bar{x}_2 = \text{le}$
$\bar{x}_1 = \text{tabl}$, $\bar{x}_2 = \text{e}$

R-Convolution Kernel

Definition

Let $x, y \in X$ and $\bar{\mathrm{x}}$ and $\bar{\mathrm{y}}$ be the corresponding sets of parts. Let $K_d(\bar{x}_d, \bar{y}_d)$ be a kernel between the d-th parts of x and y $(1 \leq d \leq D)$. Then the convolution kernel between x and y is defined as:

$$K(x, y) = \sum_{\bar{\mathrm{x}} \in R^{-1}(x)} \sum_{\bar{\mathrm{y}} \in R^{-1}(y)} \prod_{d=1}^{D} K_d(x_d, y_d)$$

Substructures-based Kernels

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

- walks
- shortest paths
- subtree patterns
- graphlets

$\vdots$

These kernels are instance of the R-convolution framework

Graphlet Kernel

The graphlet kernel compares graphs by counting *graphlets*

A graphlet corresponds to a small subgraph

- typically of 3,4 or 5 vertices

Below is the set of graphlets of size 4:

[Shervashidze et al., AISTATS'09]

Graphlet Kernel

Let $\mathcal{G} = \{graphlet_1, graphlet_2, \ldots, graphlet_r\}$ be the set of size-k graphlets

Let also $f_G \in \mathbb{N}^r$ be a vector such that its i-th entry is $f_{G,i} = \#(graphlet_i \sqsubseteq G)$

The graphlet kernel is defined as:

$$k(G_1, G_2) = \langle f_{G_1}, f_{G_2} \rangle$$

Problems:

- There are $\binom{n}{k}$ size-k subgraphs in a graph
- Exaustive enumeration of graphlets is very expensive

  Requires $O(n^k)$ time
- For labeled graphs, the number of graphlets increases further

Example

G_1

G_2

The vector representations of the graphs above according to the set of graphlets of size 4 is:

$$f_{G_1} = [0, 0, 2, 0, 1, 2, 0, 0, 0, 0, 0]^\top$$
$$f_{G_2} = [0, 0, 0, 2, 1, 5, 0, 4, 0, 3, 0]^\top$$

Hence, the value of the kernel is:

$$k(G_1, G_2) = \langle f_{G_1}, f_{G_2} \rangle = 11$$

Subtree Kernel

Compares subtree patterns in two graphs

A subtree pattern is a subgraph of a graph which has

- a root vertex
- no cycles

Subtree of height 2 rooted at vertex 1

The height of a subtree is the maximum distance between the root and any other node in the subtree

If there are cycles in the graph, a vertex can appear more than once in a subtree pattern

- it is treated as a distinct vertex such that the pattern is still a cycle-free tree

For all pairs of nodes v from G_1 and u from G_2:

- Create the subtree patterns of height h rooted at v, u
- Compare v and u via a kernel function
- Recursively compare all vertices of the subtree patterns of v and u via a kernel function

[Ramon and Gartner. MGTS'03]

Subtree Kernel

Given a pair of graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$, the subtree kernel of height h is defined as:

$$k(G_1, G_2) = \sum_{v_1 \in V_1} \sum_{v_2 \in V_2} k_h(v_1, v_2)$$

where

$$k_h(v_1, v_2) = \begin{cases} \delta\big(\ell(v_1) = \ell(v_2)\big) & \text{if } h = 0 \\ \lambda_{v_1} \lambda_{v_2} \delta\big(\ell(v_1) = \ell(v_2)\big) \sum\limits_{R \in \mathcal{M}(v_1, v_2)} \prod\limits_{(w_1, w_2) \in R} k_{h-1}(w_1, w_2) & \text{if } h > 0 \end{cases}$$

where $\delta(\cdot, \cdot)$ is the Kronecker delta function that equals 1 if its arguments are equal, 0 otherwise, λ_{v_1} and λ_{v_2} are weights associated with nodes v_1 and v_2, and

$$\mathcal{M}(v_1, v_2) = \Big\{ R \subseteq \mathcal{N}(v_1) \times \mathcal{N}(v_2) \big| \big(\forall (u_1, u_2), (w_1, w_2) \in R : u_1 = w_1 \Leftrightarrow u_2 = w_2\big) \\ \wedge \big(\forall (u_1, u_2) \in R : \ell(u_1) = \ell(u_2)\big) \Big\}$$

Example

We are given the following graphs

G_1

G_2

Example

Below are given the subtrees of G_1 and G_2 with height 2 rooted at 1 and a respectively

We will compute $k_2(1, a)$

Example

We set $\lambda_v = 1$ for all $v \in V_1 \cup V_2$ and we have:

$$k_2(1, a) = \delta\big(\ell(1) = \ell(a)\big) \sum_{R \in \mathcal{M}(1,a)} \prod_{(v_1, v_2) \in R} k_1(v_1, v_2)$$

- $\delta\big(\ell(1) = \ell(a)\big) = 1$ since $\ell(1) = \ell(a) = l1$
- $\mathcal{M}(1, a) = \big\{\{(2, b)\}\big\}$ since $\ell(2) = \ell(b) = l2$

Hence, we will next compute $k_1(2, b)$

$$k_1(2, b) = \delta\big(\ell(2) = \ell(b)\big) \sum_{R \in \mathcal{M}(2,b)} \prod_{(v_1, v_2) \in R} k_0(v_1, v_2)$$

- $\delta\big(\ell(2) = \ell(b)\big) = 1$ since $\ell(2) = \ell(b) = l2$
- $\mathcal{M}(2, b) = \big\{\{(1, a), (3, d)\}\big\}$ since $\ell(1) = \ell(a) = l1$ and $\ell(3) = \ell(d) = l2$

Example

At height 0, we have:

$$k_0(1, a) = k_0(3, d) = 1$$

Therefore,

$$k_1(2, b) = k_0(1, a)k_0(3, d) = 1$$

And finally,

$$k_2(1, a) = k_1(2, b) = 1$$

<u>Subtree kernel</u>

Pros: Richer representation of graph structure

Cons: Very high complexity

- $\mathcal{O}(n^2 h 4^d)$ where d is the maximum degree of the pair of graphs

Shortest Path Kernel

Compares the length of shortest-paths of two graphs

- and their endpoints in labeled graphs

Floyd-transformation

Transforms the original graphs into shortest-paths graphs

- Compute the shortest-paths between all pairs of vertices of the input graph G using some algorithm (i. e. Floyd-Warshall)
- Create a shortest-path graph S which contains the same set of nodes as the input graph G
- All nodes which are connected by a walk in G are linked with an edge in S
- Each edge in S is labeled by the shortest distance between its endpoints in G

[Borgwardt and Kriegel. ICDM'05]

Floyd-transformation

Shortest Path Kernel

Given the Floyd-transformed graphs $S_1 = (V_1, E_1)$ and $S_2 = (V_2, E_2)$ of G_1 and G_2, the shortest path kernel is defined as:

$$k(G_1, G_2) = \sum_{e_1 \in E_1} \sum_{e_2 \in E_2} k_{edge}(e_1, e_2)$$

where k_{edge} is a kernel on edges

- For unlabeled graphs, it can be:

$$k_{edge}(e_1, e_2) = \delta(\ell(e_1), \ell(e_2)) = \begin{cases} 1 & \text{if } \ell(e_1) = \ell(e_2), \\ 0 & \text{otherwise} \end{cases}$$

  where $\ell(e)$ gives the label of edge e

- For labeled graphs, it can be:

$$k_{edge}(e_1, e_2) = \begin{cases} 1 & \text{if } \ell(e_1) = \ell(e_2) \wedge \ell(e_1^1) = \ell(e_2^1) \wedge \ell(e_1^2) = \ell(e_2^2), \\ 0 & \text{otherwise} \end{cases}$$

  where e^1, e^2 are the two endpoints of e

Floyd-transformations

G_1

$\Rightarrow$

S_1

G_2

$\Rightarrow$

S_2

Example

In S_1 we have:

- 4 edges with label 1
- 4 edges with label 2
- 2 edges with label 3

In S_2 we have:

- 4 edges with label 1
- 2 edges with label 2

Hence, the value of the kernel is:

$$k(G_1, G_2) = \sum_{e_1 \in E_1} \sum_{e_2 \in E_2} k_{edge}(e_1, e_2) = 4 \cdot 4 + 4 \cdot 2 = 24$$

Shortest Path Kernel

Computing the shortest path kernel includes:

- Computing shortest paths for all pairs of vertices in the two graphs: $\mathcal{O}(n^3)$
- Comparing all pairs of shortest paths from the two graphs: $\mathcal{O}(n^4)$

Hence, runtime is $\mathcal{O}(n^4)$

Problems:

- Very high complexity for large graphs
- Shortest-path graphs may lead to memory problems on large graphs

Cyclic Pattern Kernel

The cyclic pattern kernel

- decomposes a graph into cyclic and tree patterns
- counts the number of common patterns which occur in two graphs

Cycles:

- Let $\mathcal{S}(G)$ denote the set of cycles of a graph G
- Let also $\pi(C)$ denote the canonical representation of a cycle C
- The set of cyclic patterns of G is defined by $\mathcal{C}(G) = \{\pi(C) : C \in \mathcal{S}(G)\}$

Trees:

- By removing all the edges of all cycles, the kernel obtains a set of trees
- The kernel computes the canonical representation $\pi(T)$ of each tree T
- The set of tree patterns of G is then defined by $\mathcal{T}(G) = \{\pi(T) : T \text{ is a tree}\}$

The cyclic pattern kernel is then defined as

$$k(G, G') = |\mathcal{C}(G) \cap \mathcal{C}(G')| + |\mathcal{T}(G) \cap \mathcal{T}(G')|$$

Problems:

- Number of cyclic and tree patterns can be exponential in the number of vertices n
- Computing the cyclic pattern kernel on general graphs is NP-hard
- Can only be applied to graphs where the number of cycles is polynomially bounded

[Horvath et al., KDD'04]

Example

G

G'

Extract cyclic and tree patterns from G, G'

Example

C_1

C_1'

$$\mathcal{C}(G) = \{\pi(C_1)\} = \{(1,2,3)\}$$
$$\mathcal{C}(G') = \{\pi(C_1')\} = \{(1,2,3)\}$$

Example

$$\mathcal{T}(G) = \{\pi(T_1)\} = \{(1,3,4)\}$$
$$\mathcal{T}(G') = \{\pi(T_1'), \pi(T_2')\} = \{(1,3),(3,4)\}$$

Example

G

G'

Hence, kernel equal to

$$k(G, G') = |\mathcal{C}(G) \cap \mathcal{C}(G')| + |\mathcal{T}(G) \cap \mathcal{T}(G')| = 1$$

Random Walk Kernel

- Probably the most well-studied family of graph kernels
- Counts matching walks in two graphs

Product graph

Given two graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$, their direct product $G_\times$ is a graph with vertex set: each vertex has two nodes, from G_1 and G_2

$$V_\times = \{(v_1, v_2) : v_1 \in V_1, v_2 \in V_2\} \text{ for unlabeled graphs}$$

or

$$V_\times = \{(v_1, v_2) : v_1 \in V_1, v_2 \in V_2, \ell(v1) = \ell(v2)\} \text{ for labeled graphs}$$

and edge set:

$$E_\times = \{((v_1, v_2), (u_1, u_2)) : (v_1, u_1) \in E_1, (v_2, u_2) \in E_2\}$$

- vertices: pairs of vertices from G_1 and G_2
- draw edge if corresponding vertices of G_1 and G_2 are adjacent in G_1 and G_2

[Gartner et al., COLT/Kernel'03]

Example

G_1 G_2

there is edge between (1,3) in G_1

and (4,7) in G_2

$G_\times$

Random Walk Kernel

The k-th power of the adjacency matrix A of G computes walks of length k
$\hookrightarrow A^k_{ij}$ = number of walks of length k from vertex i to vertex j

Performing a random walk on $G_\times$ is equivalent to performing a simultaneous random walk on G_1 and G_2

- Common walks of length k can be computed using $A^k_\times$

For $k \in \mathbb{N}$, the k-step random walk kernel is defined as:

$$K^k_\times(G_1, G_2) = \sum_{i,j=1}^{|V_\times|} \Big[\sum_{r=0}^{k} \lambda_r A^r_\times \Big]_{ij}$$

where $\lambda_0, \lambda_1, \ldots, \lambda_k$ positive weights and $A^0_\times = I$

random walk length	#
2	20
3	10
⋮	⋮
7	5
8	3
⋮	⋮

Aggregate

↓

$k(G_1, G_2)$

G_1

G_2

In product graph:

$(1,a) - (2,b) - (3,c) - (4,d)$

$\vdots$

$(8,h)$

$\boxed{\text{Longer the random walk, more similar the } G_1 \text{ and } G_2}$

Random Walk Kernel

For $k \to \infty$, we obtain the geometric random walk kernel $K_\times^\infty(G_1, G_2)$

If $\lambda_r = \lambda^r$, $K_\times^\infty(G_1, G_2)$ can be directly computed as follows:

$$K_\times^\infty(G_1, G_2) = \sum_{i,j=1}^{|V_\times|} \Big[\sum_{r=0}^{\infty} \lambda^r A_\times^r \Big]_{ij} = e^\top (I - \lambda A_\times)^{-1} e$$

where e the all-ones vector

Problem: computational complexity is $\mathcal{O}(n^6)$

Solution: Efficient computation (almost $\mathcal{O}(n^3)$) using:

- Sylvester equations
- Conjugate gradient solver
- Fixed-point iterations
- Spectral decompositions
- λ should be non greater than the largest eigenvalue of $A_\times$

[Vishwanathan et al., JMLR 11.Apr (2010)]

Neighborhood Aggregation Approaches

Weisfeiler-Lehman Test of Isomorphism

May answer if two graphs are not isomorphic

Run the Weisfeiler-Lehman algorithm for the following pair of graphs

G_1

G_2

Iteration 1

First step: Augment the labels of the vertices by the sorted set of labels of neighbouring vertices

Copy the label of its neighbors.

G_1

G_2

Iteration 1

Second step: Compress the augmented labels into new, short labels:

- $1, 11 \rightarrow 2$
- $1, 111 \rightarrow 3$
- $1, 1111 \rightarrow 4$

G_1

G_2

Iteration 1

Are the label sets of G_1 and G_2 identical?

$\{2, 2, 2, 3, 3, 4\}$ for G_1, G_2

G_1

G_2

Yes!!!

Continue to the next iteration

Iteration 2

First step: Augment the labels of the vertices by the sorted set of labels of neighbouring vertices

G_1

G_2

Iteration 2

Second step: Compress the augmented labels into new, short labels:

- $2, 24 \rightarrow 5$
- $2, 33 \rightarrow 6$
- $2, 34 \rightarrow 7$
- $3, 234 \rightarrow 9$
- $4, 2233 \rightarrow 10$

G_1

G_2

Iteration 2

Are the label sets of G_1 and G_2 identical?

G_1

G_2

No!!!

Graphs are not isomorphic

Weisfeiler-Lehman Framework

Let $G^1, G^2, \ldots, G^h$ be the graphs emerging from graph G at the iteration $1, 2, \ldots, h$ of the Weisfeiler-Lehman algorithm

Iteration 1: labels of neighbors; Iteration 2: labels of neighbors of neighbors.

...

Then, the Weisfeiler-Lehman kernel is defined as:

$$\boxed{k^h_{WL}(G_1, G_2) = k(G_1, G_2) + k(G^1_1, G^1_2) + k(G^2_1, G^2_2) + \ldots + k(G^h_1, G^h_2)}$$

where $k(\cdot, \cdot)$ is a base kernel (e.g. subtree kernel, shortest path kernel, ...)

stop iteration if G^1_h, G^2_h set of labels are not anymore identical.

At each iteration of the Weisfeiler-Lehman algorithm:

- run a graph kernel for labeled graphs
- the new kernel values are added to the ones of the previous iteration

[Shervashidze et al., JMLR 12.Sep (2011)]

Weisfeiler-Lehman Subtree Kernel

Counts matching pairs of labels in two graphs after each iteration

G_1

G_2

Initialization

Feature vector for a graph G:

$$\phi(G) = \{\#\text{nodes with label } 1, \#\text{nodes with label } 2, \ldots, \#\text{nodes with label } l\}$$

G_1

G_2

$$\phi(G_1) = [1, 2, 1, 1, 1]^\top \qquad \phi(G_2) = [1, 1, 2, 1, 1]^\top$$

$$k(G_1, G_2) = \langle \phi(G_1), \phi(G_2) \rangle = 7$$

Iteration 1

First step: Augment the labels of the vertices by the sorted set of labels of neighbouring vertices

G_1

G_2

Iteration 1

Second step: Compress the augmented labels into new, short labels:

- $1, 24 \rightarrow 6$
- $2, 14 \rightarrow 7$
- $2, 1334 \rightarrow 8$
- $2, 3 \rightarrow 9$
- $3, 24 \rightarrow 10$
- $3, 245 \rightarrow 11$
- $3, 25 \rightarrow 12$
- $4, 1235 \rightarrow 13$
- $5, 34 \rightarrow 14$

G_1

G_2

Iteration 1

Third step: Compute kernel value for iteration $h = 1$ and add it to previous kernel value

G_1

G_2

$$\phi(G_1^1) = [1, 1, 0, 1, 0, 1, 0, 1, 1]^\top \quad \phi(G_2^1) = [1, 0, 1, 0, 1, 0, 1, 1, 1]^\top$$

$$k(G_1^1, G_2^1) = \langle \phi(G_1^1), \phi(G_2^1) \rangle = 3$$

$$k_{WL}^1(G_1, G_2) = k(G_1, G_2) + k(G_1^1, G_2^1) = 10$$

Weisfeiler-Lehman Subtree Kernel

Computing the Weisfeiler-Lehman Subtree Kernel takes $\mathcal{O}(hm)$ time

- very **efficient**

Comparison to other well-known kernels

More Recent Approaches

Lovász ϑ kernel

Compares graphs based on the orthonormal representation associated with the Lovász number

- the orthonormal representation captures **global** graph properties

Orthonormal representation of a graph $G = (V, E)$:

- each vertex $i \in V$ is assigned a unit vector u_i, $||u_i|| = 1$
- let $U_G = \{u_1, u_2, \ldots, u_n\}$ be the set of all vectors
- for $i, j \in V$, if $(i, j) \notin E$, then $u_i^\top u_j = 0$

An interesting orthonormal representation is associated with the Lovász number $\vartheta(G)$

Definition (Lovász number)

For a graph $G = (V, E)$,

$$\vartheta(G) = \min_{c, U_G} \max_{i \in V} \frac{1}{(c^\top u_i)^2}$$

where the minimization is taken over all orthonormal representations U_G and all unit vectors c

[Johansson et al., ICML'14]

Lovász ϑ kernel

Given a subset of vertices $S \subseteq V$, the Lovász *value* of the subgraph induced by S is:

$$\vartheta_S(G) = \min_c \max_{u_i \in U_{G|S}} \frac{1}{(c^\top u_i)^2}$$

where $U_{G|S} = \{u_i \in U_G : i \in S\}$

The Lovász kernel is then defined as:

$$k_\vartheta(G_1, G_2) = \sum_{\substack{S_1 \subseteq V_1 \\ |S_1|=|S_2|}} \sum_{S_2 \subseteq V_2} \frac{1}{Z} k(\vartheta_{S_1}(G_1), \vartheta_{S_2}(G_2))$$

where $Z = \binom{n_1}{d}\binom{n_2}{d}$, $d = |S_1| = |S_2|$ and $k(\cdot,\cdot)$ is a base kernel (e.g. linear, gaussian)

Problem: Computing the Lovász ϑ kernel is very expensive since $\rightarrow$ requires computing the Lovász value for all subgraphs of the two graphs

Solution: Sampling
$\hookrightarrow$ Evaluate the Lovász value for a smaller number of subgraphs of size d

Ordered Decomposition DAGs Kernel

General idea:

- Decomposes graphs into multisets of directed acyclic graphs (DAGs)
- Uses existing tree kernels to compare these DAGs

Generates one unordered rooted DAG for each vertex (keeps only edges belonging to the shortest paths)

Then, the kernel is defined as:

$$k(G, G') = \sum_{D \in DD(G)} \sum_{D' \in DD(G')} k_{DAG}(D, D')$$

where $DD(G)$ and $DD(G')$ are multisets that contain the DAGs extracted from G and G', respectively, and k_{DAG} is a kernel between DAGs

[Da San Martino et al., SDM'12]

Ordered Decomposition DAGs Kernel

DAGs are unordered (i. e. the set of neighbours of each node is unordered)

There is a vast literature on kernels for ordered trees. Hence, the kernel:

- transforms the unordered DAGs to ordered DAGs (based on node labels, outdegrees of nodes, etc.)
- projects subdags to a tree space (see Figure below)
- applies a kernel for ordered trees

The kernel between two DAGs is computed as follows:

$$k_{DAG}(D, D') = \sum_{v \in V_D} \sum_{v' \in V_{D'}} k_{tree}\big(root(v), root(v')\big)$$

V_D, $V_{D'}$: sets of vertices of D and D'
k_{tree}: kernel between ordered trees

Assignment Kernels

Assignment Kernels

- Another design paradigm for developing kernels
- Only a few instances in the literature
- They compute a matching between substructures of one object and substructures of a second object such that the overall similarity of the two objects is maximized
- Such a matching can reveal structural correspondences between the two objects

Pyramid Match Kernel

Embed all vertices in the d-dimensional vector space $\mathbb{R}^d$ as follows

- compute the eigendecomposition of the adjacency matrix
- use the eigenvectors of the d largest in magnitude eigenvalues

Such embeddings capture **global** properties of graphs

Example: eigenvector corresponding to greatest eigenvalue contains eigenvector centrality scores of vertices $\rightarrow$ global property

After embedding: each vertex is a point in the d-dimensional unit hypercube

Then, use pyramid match kernel, a kernel function over unordered feature sets:

- Each feature set is mapped to a multiresolution histogram
- The histogram pyramids are then compared using a weighted histogram intersection computation

[Nikolentzos et al., AAAI'17]

Node Embeddings

Node embeddings: represent nodes as points in a vector space

- Generate embeddings using eigenvectors of adjacency matrix $A = U \Lambda U^\top$
  - i^{th} row u_i of U corresponds to embedding of vertex v_i

- Such embeddings capture global properties of graphs

Bag-of-vectors Representation

Graphs represented as **bags-of-vectors**:

- A graph is represented as a set of vectors: $\{u_1, \ldots, u_n\}$
- Each vector $u_i \in \mathbb{R}^d$ represents the embedding of the ith node in the d-dimensional space
- This is a natural representation

  $\hookrightarrow$ There is no canonical ordering for the nodes of a graph

Pyramid Match Graph Kernel

The Pyramid Match Graph Kernel

- partitions feature space into cells
- at level $l \rightarrow 2^l$ cells along each dimension

Number of nodes (i.e. embeddings) that match at l:

$$I(H^l_{G_1}, H^l_{G_2}) = \sum_{i=1}^{2^l d} \min\left(H^l_{G_1}(i), H^l_{G_2}(i)\right)$$

where $H^l_G(i)$ is the number of nodes of G that lie in the ith cell

Example

Example

Example

$$I(H^0_{G_1}, H^0_{G_2}) = 9 + \dots$$

Example

$$I(H^0_{G_1}, H^0_{G_2}) = 9 + \dots$$

Example

$$I(H^0_{G_1}, H^0_{G_2}) = 9 + 9 = 18$$

Example

Example

$$I(H^1_{G_1}, H^1_{G_2}) = (5 + 4) + \dots$$

Example

$$I(H^1_{G_1}, H^1_{G_2}) = (5 + 4) + \dots$$

Example

$$I(H^1_{G_1}, H^1_{G_2}) = (5+4) + (5+4) = 18$$

Example

Example

$$I(H^2_{G_1}, H^2_{G_2}) = (2 + 2 + 1 + 3) + \dots$$

Example

$$I(H^2_{G_1}, H^2_{G_2}) = (2 + 2 + 1 + 3) + \dots$$

Example

$$I(H^2_{G_1}, H^2_{G_2}) = (2+2+1+3) + (2+2+1+2) = 15$$

Pyramid Match Graph Kernel

PM takes a weighted sum of the matches that occur at each level (levels 0 to L):

$$k_\Delta(G_1, G_2) = I(H^L_{G_1}, H^L_{G_2}) + \sum_{l=0}^{L-1} \frac{1}{2^{L-l}} \left(I(H^l_{G_1}, H^l_{G_2}) - I(H^{l+1}_{G_1}, H^{l+1}_{G_2}) \right)$$
$$= 15 + \frac{1}{2}(18 - 15) + \frac{1}{4}(18 - 18) = 16.5$$

- Matches within lower levels weighted less
- Only new matches are taken into account

Complexity: $\mathcal{O}(dnL)$

Optimal Assignment Kernel

- $\{x_1, \ldots, x_n\}$ are substructures of G, e. g., nodes
- $\{x'_1, \ldots, x'_{n'}\}$ are substructures of G', e. g., nodes
- κ is a non-negative kernel comparing substructures
- π is a permutation of the integers $\{1, \ldots, \min(n, n')\}$
- Then, the optimal assignment kernel is defined as follows:

$$k(G, G') = \begin{cases} \max\limits_{\pi} \sum\limits_{i=1}^{n} \kappa(x_i, x'_{\pi(i)}), & \text{if } n' > n \\ \max\limits_{\pi} \sum\limits_{j=1}^{n'} \kappa(x_{\pi(j)}, x'_j), & \text{otherwise} \end{cases}$$

[Frohlich et al., ICML'05]

Optimal Assignment Kernel

- $\{x_1, \ldots, x_n\}$ are substructures of G, e. g., nodes
- $\{x'_1, \ldots, x'_{n'}\}$ are substructures of G', e. g., nodes
- κ is a non-negative kernel comparing substructures
- π is a permutation of the integers $\{1, \ldots, \min(n, n')\}$
- Then, the optimal assignment kernel is defined as follows:

$$k(G, G') = \begin{cases} \max\limits_{\pi} \sum\limits_{i=1}^{n} \kappa(x_i, x'_{\pi(i)}), & \text{if } n' > n \\ \max\limits_{\pi} \sum\limits_{j=1}^{n'} \kappa(x_{\pi(j)}, x'_j), & \text{otherwise} \end{cases}$$

- However, **not** positive semidefinite in general

[Vert. arXiv:0801.4061]

Valid Optimal Assignment Kernels

- Let $\mathcal{X}$ be a set, and $[\mathcal{X}]^n$ denote the set of all n-element subsets of $\mathcal{X}$
- Let also $X, X' \in [\mathcal{X}]^n$ for $n \in \mathbb{N}$, and $\mathfrak{B}(X, X')$ denote the set of all bijections between X and X'
- The optimal assignment kernel on $[\mathcal{X}]^n$ is defined as

$$K^k_{\mathfrak{B}}(X, X') = \max_{B \in \mathfrak{B}(X,X')} \sum_{(x,x') \in B} k(x, x')$$

  where k is a kernel between the elements of X and X'
- The above function $K_{\mathfrak{B}}(\mathcal{X}, \mathcal{X}')$ is a valid kernel only if the base kernel k is strong

Definition (Strong Kernel)

A function $k : \mathcal{X} \times \mathcal{X} \to \mathbb{R}_{\geq 0}$ is called strong kernel if $k(x, y) \geq \min\{k(x, z), k(z, y)\}$ for all $x, y, z \in \mathcal{X}$.

Strong kernels are equivalent to kernels obtained from a *hierarchy* defined on set $\mathcal{X}$.

Frameworks

Diagonal Dominance Problem

Diagonal dominance problem of kernels that compare specific substructures of graphs:

- Very large feature space, hence, unlikely that two graphs will contain similar substructures
- However, substructures (i. e. features) often **related** to each other
- Kernel value between pairs of graphs $\ll$ kernel value between a graph and itself

For example, when the features correspond to large graphlets (e. g., $k \geq 5$), two graphs may be composed of many similar graphlets, but not any identical

g_1, g_2, g_3 extracted from G

g_1', g_2', g_3' extracted from G'

g_1 nearly isomorphic to g_1'

g_2 nearly isomorphic to g_2'

g_3 nearly isomorphic to g_3'

Diagonal Dominance Problem

Diagonal dominance problem of kernels that compare specific substructures of graphs:

- Very large feature space, hence, unlikely that two graphs will contain similar substructures
- However, substructures (i. e. features) often **related** to each other
- Kernel value between pairs of graphs $\ll$ kernel value between a graph and itself

This leads to the diagonal dominance problem

The resulting kernel matrix is close to the identity matrix

A Structural Smoothing Framework

To deal with diagonal dominance, it applies *smoothing*

First construct a Directed Acyclic Graph (DAG):

- each vertex corresponds to a substructure
- for each substructure s of size k determine all possible substructures of size $k-1$ that s can be reduced into
- these correspond to the parents of s
- draw a weighted directed edge from each parent to its children vertices

DAG provides a topological ordering of the vertices

- all descendants of a given substructure at depth $k-1$ are at depth k

DAG for graphlets of size $k \leq 3$

[Yanardag and Vishwanathan, NIPS'15]

A Structural Smoothing Framework

The structural smoothing for a substructure s at level k is defined as:

$$P_{SS}^{k}(s) = \frac{\max(c_s - d, 0)}{m} + \frac{dm_d}{m} \sum_{p \in \mathcal{P}_s} P_{SS}^{k-1}(p) \frac{w_{ps}}{\sum_{c \in \mathcal{C}_p} w_{pc}}$$

where

- c_s denotes the number of times substructure s appears in the graph
- $m = \sum_i c_i$ denotes the total number of substructures present in the graph
- $d > 0$ is a discount factor
- $m_d := |\{i : c_i > d\}|$ is the number of substructures whose counts are larger than d
- w_{ij} denotes the weight of the edge connecting vertex i to vertex j
- $\mathcal{P}_s$ denotes the parents of vertex s
- $\mathcal{C}_p$ the children of vertex p

Even if the graph does not contain a substructure s ($c_s = 0$), its value in the feature vector can be **greater** than 0 ($P_{SS}(s) > 0$)

Example

Kernel matrix before smoothing

$\Rightarrow$

Kernel matrix after smoothing

Deep Graph Kernels

To deal with diagonal dominance, the deep graph kernels framework computes the kernel as follows:

$$k(G, G') = \phi(G)^\top \; M \; \phi(G')$$

M: a positive semidefinite matrix that encodes the relationships between substructures
Each component of $\phi(G), \phi(G')$ corresponds to a substructure (e. g., the complete graphlet of size 5)

Matrix M is learned using techniques inspired from the field of natural language processing:

- An embedding for each substructure is generated using the CBOW or Skip-gram model
- Then M corresponds to the inner products of these embeddings

However, unlike words in documents, substructures of graphs do not have a *linear co-occurrence relationship*

Such co-occurrence relationships are manually defined for 3 kernels:
(1) the Weisfeiler-Lehman subtree kernel
(2) the graphlet kernel
(3) the shortest path kernel

[Yanardag and Vishwanathan, KDD'15]

A Degeneracy Framework for Graph Comparison

Definition (k-core)

The k-core of a graph is defined as a maximal subgraph in which every vertex is connected to at least k other vertices within that subgraph

A *k-core decomposition* of a graph consists of finding the set of all k-cores

The set of all k-cores forms a nested sequence of subgraphs

The degeneracy $\delta^*(G)$ is defined as the maximum k for which graph G contains a non-empty k-core subgraph

[Nikolentzos et al., IJCAI'18]

Degeneracy Framework for Graph Comparison

Idea: use the nested sequence of subgraphs generated by k-core decomposition to capture structure at multiple different scales

The core variant of the base kernel k is defined as:

$$k_c(G, G') = k(C_0, C_0') + k(C_1, C_1') + \ldots + k(C_{\delta^*_{min}}, C'_{\delta^*_{min}})$$

where δ^*_{min} is the minimum of the degeneracies of the two graphs, and $C_0, C_1, \ldots, C_{\delta^*_{min}}$ and $C_0', C_1', \ldots, C'_{\delta^*_{min}}$ are the 0-core, 1-core,..., δ^*_{min}-core subgraphs of G and G', respectively

The degeneracy framework can:

- increase the expressive power of existing algorithms
- be applied to any algorithm that compares graphs

Example

G

G'

Example

Core 0 = G

C_0

C_0'

$$k_c(G, G') = k(C_0, C_0')$$

Example

core = 1

C_1

core - 1

C_1'

$$k_c(G, G') = k(C_0, C_0') + k(C_1, C_1')$$

Example

core-2

C_2

core-2

C_2'

$$k_c(G, G') = k(C_0, C_0') + k(C_1, C_1') + k(C_2, C_2')$$

Example

C_3

C_3'

$$k_c(G, G') = k(C_0, C_0') + k(C_1, C_1') + k(C_2, C_2') + k(C_3, C_3')$$

Successive Embeddings

Graph kernels compute implicitly the inner product between the representations of input graphs in $\mathcal{H}$

- Equivalent to computing the linear kernel on feature space $\mathcal{H}$
- Linear kernel limits expressiveness of derived representations

Idea: Obtain complex kernels by stacking simpler kernels on top of one another

[Nikolentzos et al., CIKM'18]

Successive Embeddings

Embedding 1: Embed graphs in a Hilbert space $\mathcal{H}_1$ using a graph kernel k

Embedding 2: Embed emerging representations $\phi(G)$, $\phi(G')$ into another Hilbert space $\mathcal{H}_2$ using kernels for vector data:

1. *Polynomial kernel*: $k_P(\phi(G), \phi(G')) = \big(\langle \phi(G), \phi(G') \rangle\big)^d, \quad d \in \mathbb{N}$
2. *Gaussian kernel*: $k_G(\phi(G), \phi(G')) = exp\Big(-\frac{||\phi(G)-\phi(G')||^2}{2\sigma^2}\Big), \quad \sigma > 0$

Problem: Usually $\phi(G)$ and $\phi(G')$ not computed explicitly. How to apply **Embedding 2**?
$\hookrightarrow$ Use an implicit computation scheme

The two kernels for vector data can be computed as:

1. *Polynomial kernel*:

$$k_P(\phi(G), \phi(G')) = \big(\langle \phi(G), \phi(G') \rangle\big)^d = \big(k(G, G')\big)^d, \quad d \in \mathbb{N}$$

2. *Gaussian kernel*:

$$k_G(\phi(G), \phi(G')) = exp\Big(-\frac{k(G,G)-2k(G,G')+k(G',G')}{\ldots}\Big), \quad \sigma > 0$$

Successive Embeddings Example

- Figure below illustrates a sequence of two embeddings
- Separation of the data points associated with the two classes progressively increased

Applications of Graph Kernels

Applications

- Bioinformatics [Borgwardt et al., Bioinformatics 21(suppl_1); Borgwardt et al., PSB'07; Sato et al., BMC bioinformatics 9(1)]
- Chemoinformatics [Swamidass et al., Bioinformatics 21(suppl_1); Ralaivola et al., Neural Networks 18(8); Mahé et al., JCIM 45(4); Ceroni et al., Bioinformatics 23(16); Mahé and Vert, Machine Learning 75(1)]
- Computer Vision [Harchaoui and Bach, CVPR'07; Bach, ICML'08; Wang and Sahbi. CVPR'13; Stumm et al., CVPR'16]
- Cybersecurity [Anderson et al., JCV 7(4); Gascon et al., AISec'13; Narayanan et al., IJCNN'16]
- Natural Language Processing [Glavas and Snajder, ACL'13; Bleik et al., TCBB 10(5); Nikolentzos et al., EMNLP'17]
- Social Networks [Yanardag and Vishwanathan, KDD'15]

$\vdots$

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

Kernel type	Accuracy
Vector kernel	76.86
Optimized vector kernel	80.17
Graph kernel	77.30
Graph kernel without structure	72.33
Graph kernel with global info	84.04
DALI classifier	75.07

Chemical Compound Classification

Represent each chemical compound as a graph

$\Rightarrow$

Perform **graph classification** to predict if a chemical compound displays the desired behavior against the specific biomolecular target or not

Lin.Reg	*DT*	*NN*	*Progol1*	*Progol2*	*Sebag*	*Kramer*	graph kernels
89.3%	88.3%	89.4%	81.4%	87.8%	93.3%	95.7%	91.2%

[Mahé et al., JCIM 45(4)]

Malware Detection

Given a computer program, create its control flow graph

call	[ebp+0x8]
push	0x70
push	0x010012F8
call	0x01006170
push	0x010061C0
mov	eax, fs:[0x00000000]
push	eax
mov	fs:[], esp
mov	eax, [esp+0x10]
mov	[esp+0x10], ebp
lea	ebp, [esp+0x10]
sub	esp, eax
...	...

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

Method	Accuracy (%)
Gaussian kernel	**99.09**
Spectral kernel	96.36
Combined kernel	**100.00**
n-gram ($n = 4$, $L = 1{,}000$, SVM = 2-poly)	94.55
n-gram ($n = 4$, $L = 2{,}500$, SVM = Gauss)	93.64
n-gram ($n = 6$, $L = 2{,}500$, SVM = 2-poly)	92.73
n-gram ($n = 3$, $L = 1{,}000$, SVM = 2-poly)	89.09
n-gram ($n = 2$, $L = 500$, 3-NN)	88.18

[Anderson et al., JCV 7(4)]

Graph-Of-Words

Each document is represented as a graph $G = (V, E)$ consisting of a set V of vertices and a set E of edges between them

- vertices $\rightarrow$ unique terms
- edges $\rightarrow$ co-occurrences within a fixed-size sliding window
- no edge weight
- no edge direction

Graph representation more flexible than n-grams. Takes into account

- word inversion
- subset matching
- e. g., "*article about news*" vs. "*news article*"

[Rousseau and Vazirgiannis, CIKM'13]

Custom Shortest Path Kernel

Transforms the original graphs into shortest-paths graphs
$\hookrightarrow$ Edges correspond to shortest paths of length at most d in original graph

Given the SP-transformed graphs $C_1 = (V_1, E_1)$ and $C_2 = (V_2, E_2)$ of G_1 and G_2, the shortest path kernel is defined as:

$$k(G_1, G_2) = \frac{\sum_{v_1 \in V_1, v_2 \in V_2} k_{node}(v_1, v_2) + \sum_{e_1 \in E_1, e_2 \in E_2} k^{(1)}_{walk}(e_1, e_2)}{norm}$$

where k_{node} is a kernel for comparing two vertices, $k^{(1)}_{walk}$ a kernel on edge walks of length 1 and *norm* a normalization factor. Specifically:

$$k_{node}(v_1, v_2) = \begin{cases} 1 & \text{if } \ell(v_1) = \ell(v_2), \\ 0 & \text{otherwise} \end{cases}$$

$$k^{(1)}_{walk}(e_1, e_2) = k_{node}(u_1, u_2)\, k_{edge}(e_1, e_2)\, k_{node}(v_1, v_2)$$

$$k_{edge}(e_1, e_2) = \begin{cases} \ell(e_1)\,\ell(e_2) & \text{if } e_1 \in E_1 \wedge e_2 \in E_2, \\ 0 & \text{otherwise} \end{cases}$$

[Nikolentzos et al., EMNLP'17]

Example

d_1: "*barclays bank cut its base lending rate*"

d_2: "*base rate of barclays bank dropped*"

G_1

G_2

Example

SP-transformation ($d = 2$)

C_1

C_2

Compute the similarity between C_1 & C_2

Example

$\sum_{v_1 \in V_1, v_2 \in V_2} k_{node}(v_1, v_2) = 4$

C_1

C_2

Example

$$\sum_{e_1 \in E_1, e_2 \in E_2} k^{(1)}_{walk}(e_1, e_2) = 1 + \frac{1}{2} = \frac{3}{2}$$

C_1

C_2

Example

$norm = 13.07$

$k(G_1, G_2) = \frac{4+\frac{3}{2}}{13.07} = 0.42$

C_1

C_2

Text Categorization

Method \ Dataset		WebKB		News		Subjectivity		Amazon		Polarity	
		Acc	F1	Acc	F1	Acc	F1	Acc	F1	Acc	F1
Dot product	$n = 1$	90.26	89.23	81.10	77.64	89.92	89.92	91.88	91.88	76.27	76.26
	$n = 2$	90.47	89.50	80.91	77.32	91.01	91.01	92.00	92.02	77.46	77.45
	$n = 3$	90.26	89.17	80.72	77.10	90.90	90.90	91.81	91.85	77.41	77.40
	$n = 4$	89.40	88.13	80.31	76.51	90.39	90.39	91.31	91.33	77.19	77.18
Cosine	$n = 1$	92.48	91.88	81.17	77.66	90.03	90.02	94.00	94.00	76.70	76.69
	$n = 2$	93.05	92.75	81.49	77.97	90.94	90.94	94.13	94.13	77.56	77.56
	$n = 3$	92.98	92.59	80.97	77.38	90.99	90.99	94.19	94.18	77.65	77.65
	$n = 4$	92.48	92.08	80.76	77.09	90.76	90.75	94.13	94.13	77.53	77.53
Tanimoto	$n = 1$	90.62	89.83	81.55	78.15	90.94	90.93	92.25	92.26	77.49	77.48
	$n = 2$	90.40	89.45	80.75	77.00	90.61	90.60	91.81	91.85	77.35	77.35
	$n = 3$	92.41	91.80	79.80	75.75	90.21	90.20	93.44	93.47	76.48	76.48
	$n = 4$	91.76	90.84	78.99	74.83	89.53	89.52	93.00	93.00	75.86	75.86
DCNN		89.18	87.99	79.91	76.15	90.26	90.26	91.81	91.81	73.26	73.26
CNN	static,rand	> 1 day		77.57	73.37	87.16	87.15	88.81	88.82	71.50	71.50
	non-static,rand	> 1 day		81.13	77.49	89.61	89.60	93.56	93.56	76.54	76.53
SPGK	$d = 1$	93.27	92.78	81.04	77.49	91.48	91.48	94.00	94.01	77.76	77.75
	$d = 2$	93.70	93.36	80.89	77.29	91.46	91.46	94.13	94.13	77.89	77.88
	$d = 3$	92.91	92.33	80.78	77.03	91.37	91.37	94.44	94.44	77.61	77.60
	$d = 4$	92.91	92.23	80.97	77.30	91.18	91.18	94.63	94.63	77.80	77.80

Image Classification

Represent each image as a graph based on its segmentation mosaic

Perform **graph classification** to categorize images

	H	W	TW	wTW	M
Coil100	1.2%	0.8%	0.0%	0.0%	0.0%
Corel14	10.36%	8.52%	7.24%	6.12%	5.38%

[Harchaoui and Bach, CVPR'07]

Experimental Evaluation

GraKeL

- Python library for graph kernels
- Contains implementations of a large number of graph kernels
- Compatible with scikit-learn
- Project repository: `https://github.com/ysig/GraKeL`

[Siglidis et al., arXiv:1806.02193]

Evaluation

Standard datasets from graph classification containing:

- unlabeled graphs
- node-labeled graphs
- node-attributed graphs

Classification using:

- SVM $\rightarrow$ precompute kernel matrix
- Hyperparameters of both SVM (i. e. C) and graph kernels optimized on training set using cross-validation

Perform 10 **times** 10-fold cross validation and report:

- *Average accuracy* over the 10 repetitions
- Standard deviation over the 10 repetitions

Graph Classification (Node-Labeled Graphs)

Kernels	DATASETS			
	MUTAG	ENZYMES	NCI1	PTC-MR
Vertex Histogram	71.87 (± 1.83)	16.87 (± 1.56)	56.09 (± 0.35)	58.09 (± 0.62)
Random Walk	82.24 (± 2.87)	12.90 (± 1.42)	TIMEOUT	51.26 (± 2.30)
Shortest Path	82.54 (± 1.00)	40.13 (± 1.34)	72.25 (± 0.28)	59.26 (± 2.34)
WL Subtree	84.00 (± 1.25)	53.15 (± 1.22)	85.03 (± 0.20)	63.28 (± 1.34)
WL Shortest Path	82.29 (± 1.93)	28.23 (± 1.00)	61.43 (± 0.32)	55.51 (± 1.68)
WL Pyramid Match	88.60 (± 0.95)	57.72 (± 0.84)	85.31 (± 0.42)	64.52 (± 1.36)
Neighborhood Hash	87.74 (± 1.17)	43.43 (± 1.45)	74.81 (± 0.37)	60.50 (± 2.10)
Neighborhood Subgraph Pairwise Distance	82.46 (± 1.55)	41.97 (± 1.66)	74.36 (± 0.31)	60.04 (± 1.15)
Ordered DAGs Decomposition	79.01 (± 2.04)	31.87 (± 1.35)	75.03 (± 0.45)	59.08 (± 1.85)
Pyramid Match	84.72 (± 1.67)	42.67 (± 1.78)	73.11 (± 0.49)	57.99 (± 2.45)
GraphHopper	82.11 (± 2.13)	36.47 (± 2.13)	71.36 (± 0.13)	55.64 (± 2.03)
Subgraph Matching	84.04 (± 1.55)	35.68 (± 0.80)	TIMEOUT	57.91 (± 1.73)
Propagation	77.23 (± 1.22)	44.48 (± 1.63)	82.12 (± 0.22)	59.30 (± 1.24)
Multiscale Laplacian	86.11 (± 1.60)	53.08 (± 1.53)	79.40 (± 0.47)	59.95 (± 1.71)
CORE WL	85.90 (± 1.44)	52.37 (± 1.29)	85.12 (± 0.21)	63.03 (± 1.67)
CORE Shortest Path	85.13 (± 2.46)	41.55 (± 1.66)	73.87 (± 0.19)	58.21 (± 1.87)

Kernels	DATASETS			Avg. Rank
	D&D	PROTEINS	AIDS	
Vertex Histogram	74.83 (± 0.40)	70.93 (± 0.28)	79.78 (± 0.13)	13.7
Random Walk	OUT-OF-MEM	69.31 (± 0.29)	79.52 (± 0.58)	15.0
Shortest Path	78.93 (± 0.53)	75.92 (± 0.35)	99.41 (± 0.12)	6.7
WL Subtree	78.88 (± 0.46)	75.45 (± 0.33)	98.51 (± 0.05)	4.8
WL Shortest Path	75.66 (± 0.42)	71.88 (± 0.22)	99.36 (± 0.02)	11.8
WL Pyramid Match	OUT-OF-MEM	75.63 (± 0.49)	99.37 (± 0.04)	2.1
Neighborhood Hash	76.02 (± 0.94)	75.55 (± 1.00)	99.54 (± 0.02)	5.0
Neighborhood Subgraph Pairwise Distance	78.76 (± 0.56)	73.17 (± 0.76)	98.04 (± 0.20)	8.0
Ordered DAGs Decomposition	75.82 (± 0.54)	70.49 (± 0.64)	90.75 (± 0.30)	11.4
Pyramid Match	76.98 (± 0.84)	71.90 (± 0.79)	99.56 (± 0.08)	8.2
GraphHopper	TIMEOUT	74.19 (± 0.42)	99.57 (± 0.02)	9.6
Subgraph Matching	OUT-OF-MEM	OUT-OF-MEM	91.96 (± 0.18)	11.2
Propagation	78.43 (± 0.55)	72.71 (± 0.62)	96.51 (± 0.38)	8.4
Multiscale Laplacian	78.28 (± 0.99)	73.89 (± 0.93)	98.48 (± 0.12)	6.0
CORE WL	78.91 (± 0.50)	75.46 (± 0.38)	98.70 (± 0.09)	4.1
CORE Shortest Path	79.33 (± 0.65)	76.31 (± 0.40)	99.47 (± 0.05)	5.5

[Nikolentzos et al., arXiv:1904.12218]

Running Time (Node-Labeled Graphs)

Kernels	DATASETS			
	MUTAG	ENZYMES	NCI1	PTC-MR
Vertex Histogram	0.01s	0.04s	0.84s	0.02s
Random Walk	1m 46.86s	4h 24m 16.26s	TIMEOUT	6m 41.20s
Shortest Path	0.92s	11.03s	1m 9.69s	1.52s
WL Subtree	0.21s	3.81s	7m 5.33s	0.55s
WL Shortest Path	7.02s	1m 27.07s	15m 29.50s	12.55s
WL Pyramid Match	3m 42.07s	1h 5m 37.26s	13h 31m 34.36s	11m 8.16s
Neighborhood Hash	0.40s	11.17s	7m 4.54s	1.31s
Neighborhood Subgraph Pairwise Distance	4.05s	27.02s	6m 9.81s	7.66s
Ordered DAGs Decomposition	1.54s	50.05s	46m 2.13s	4.03s
Pyramid Match	2.59s	31.38s	37m 37.50s	11.35s
GraphHopper	24.70s	15m 38.33s	3h 45m 8.31s	1m 33.90s
Subgraph Matching	1m 57.25s	3h 25m 43.59s	TIMEOUT	4m 19.80s
Propagation	0.48s	12.05s	10m 27.83s	1.81s
Multiscale Laplacian	10m 3.15s	56m 43.76s	5h 30m 56.29s	19m 22.43s
CORE WL	0.55s	12.52s	14m 30.56s	17m 2.27s
CORE Shortest Path	2.69s	48.02s	3m 16.54s	3.97s

Kernels	DATASETS			Avg. Rank
	D&D	PROTEINS	AIDS	
Vertex Histogram	0.24s	0.10s	0.25s	1.0
Random Walk	OUT-OF-MEM	51m 10.11s	1h 51m 56.47s	13.6
Shortest Path	55m 58.79s	1m 18.91s	13.93s	4.4
WL Subtree	5m 52.96s	32.48s	40.49s	2.8
WL Shortest Path	7h 27m 21.90s	8m 3.68s	1m 33.46s	10.1
WL Pyramid Match	OUT-OF-MEM	5h 37m 10.33s	5h 55m 20.37s	14.6
Neighborhood Hash	6m 17.21s	41.81s	33.30s	3.5
Neighborhood Subgraph Pairwise Distance	4h 36m 28.97s	9m 9.80s	1m 12.31s	8.1
Ordered DAGs Decomposition	27m 59.18s	4m 7.81s	2m 5.32s	8.7
Pyramid Match	5m 48.51s	1m 26.82s	2m 48.04s	8.0
GraphHopper	TIMEOUT	3h 43m 1.54s	38m 51.78s	12.1
Subgraph Matching	OUT-OF-MEM	OUT-OF-MEM	4h 26m 46.71s	14.0
Propagation	9m 34.30s	51.20s	1m 43.62s	5.5
Multiscale Laplacian	3h 40m 30.72s	2h 20m 39.57s	1h 11m 58.23s	13.2
CORE WL	17m 2.27s	1m 16.74s	54.79s	7.2
CORE Shortest Path	5h 2m 39.71s	3m 31.97s	40.11s	7.2

[Nikolentzos et al., arXiv:1904.12218]

Graph Classification (Unlabeled Graphs)

Kernels	DATASETS						Avg. Rank
	IMDB BINARY	IMDB MULTI	REDDIT BINARY	REDDIT MULTI-5K	REDDIT MULTI-12K	COLLAB	
Vertex Histogram	46.54 (± 0.80)	29.59 (± 0.40)	47.32 (± 0.66)	17.92 (± 0.42)	21.73 (± 0.00)	52.00 (± 0.00)	12.4
Random Walk	63.87 (± 1.06)	45.75 (± 1.03)	TIMEOUT	TIMEOUT	OUT-OF-MEM	68.00 (± 0.07)	7.6
Shortest Path	55.18 (± 1.23)	39.37 (± 0.84)	81.67 (± 0.23)	47.90 (± 0.13)	TIMEOUT	58.80 (± 0.08)	8.3
Graphlet	65.19 (± 0.97)	39.82 (± 0.89)	76.80 (± 0.27)	34.06 (± 0.38)	23.08 (± 0.11)	70.63 (± 0.25)	7.0
WL Subtree	72.47 (± 0.50)	50.76 (± 0.30)	67.96 (± 1.01)	OUT-OF-MEM	OUT-OF-MEM	78.12 (± 0.17)	4.2
WL Shortest Path	55.87 (± 1.19)	39.63 (± 0.68)	TIMEOUT	TIMEOUT	TIMEOUT	58.80 (± 0.06)	10.8
Neighborhood Hash	73.34 (± 0.98)	50.68 (± 0.50)	81.65 (± 0.28)	49.36 (± 0.18)	39.62 (± 0.19)	79.99 (± 0.39)	2.3
Neighborhood Subgraph Pairwise Distance	68.81 (± 0.71)	45.10 (± 0.63)	TIMEOUT	TIMEOUT	TIMEOUT	TIMEOUT	7.5
Lovász-ϑ	49.21 (± 1.33)	39.33 (± 0.95)	TIMEOUT	TIMEOUT	TIMEOUT	TIMEOUT	15.0
SVM-ϑ	51.35 (± 1.54)	38.40 (± 0.60)	74.54 (± 0.27)	29.65 (± 0.53)	23.04 (± 0.18)	55.72 (± 0.31)	10.1
Ordered DAGs Decomposition	64.70 (± 0.73)	46.80 (± 0.51)	50.61 (± 1.06)	42.99 (± 0.09)	29.83 (± 0.08)	52.00 (± 0.00)	7.5
Pyramid Match	66.67 (± 1.45)	45.25 (± 0.79)	86.77 (± 0.42)	48.22 (± 0.29)	41.15 (± 0.17)	74.57 (± 0.34)	4.1
GraphHopper	57.69 (± 1.31)	40.04 (± 0.91)	TIMEOUT	TIMEOUT	TIMEOUT	60.21 (± 0.10)	9.3
Subgraph Matching	TIMEOUT	TIMEOUT	OUT-OF-MEM	OUT-OF-MEM	OUT-OF-MEM	TIMEOUT	–
Propagation	51.15 (± 1.67)	33.15 (± 1.08)	63.41 (± 0.77)	34.32 (± 0.61)	24.07 (± 0.11)	58.67 (± 0.15)	10.1
Multiscale Laplacian	70.94 (± 0.93)	47.92 (± 0.87)	89.44 (± 0.30)	35.01 (± 0.65)	OUT-OF-MEM	75.29 (± 0.49)	3.8
CORE WL	73.31 (± 1.06)	50.79 (± 0.54)	72.82 (± 1.05)	OUT-OF-MEM	OUT-OF-MEM	OUT-OF-MEM	3.8
CORE Shortest Path	69.37 (± 0.68)	50.79 (± 0.57)	90.76 (± 0.14)	TIMEOUT	OUT-OF-MEM	TIMEOUT	2.5

[Nikolentzos et al., arXiv:1904.12218]

Running Time (Unlabeled Graphs)

Kernels	Datasets						Avg. Rank
	IMDB BINARY	IMDB MULTI	REDDIT BINARY	REDDIT MULTI-5K	REDDIT MULTI-12K	COLLAB	
Vertex Histogram	0.07s	0.15s	0.67s	2.20s	6.37s	1.12s	1.0
Random Walk	7m 20.94s	13m 40.75s	TIMEOUT	TIMEOUT	TIMEOUT	13h 38m 11.49s	13.6
Shortest Path	11.51s	7.92s	4h 48m 11.19s	12h 40m 19.50s	TIMEOUT	1h 9m 5.50s	7.0
Graphlet	22m 45.89s	21m 44.30s	44m 45.42s	44m 6.52s	53m 14.22s	2h 58m 1.14s	9.5
WL Subtree	4.49s	6.16s	16m 2.65s	OUT-OF-MEM	OUT-OF-MEM	38m 42.24s	4.2
WL Shortest Path	1m 32.66s	1m 40.46s	TIMEOUT	TIMEOUT	TIMEOUT	10h 27m 41.97s	10.3
Neighborhood Hash	21.83s	26.07s	23m 3.42s	2h 44m 44.66s	9h 11m 23.67s	35m 49.96s	6.3
Neighborhood Subgraph Pairwise Distance	4m 18.12s	2m 49.45s	TIMEOUT	TIMEOUT	TIMEOUT	TIMEOUT	12.5
Lovász-ϑ	5h 19m 27.17s	6h 33m 6.55s	TIMEOUT	TIMEOUT	TIMEOUT	TIMEOUT	17.0
SVM-ϑ	39.40s	1m 0.57s	19m 24.73s	23m 14.31s	52m 10.36s	5m 57.31s	5.3
Ordered DAGs Decomposition	4.47s	4.85s	1m 53.50s	4m 48.92s	8m 20.66s	2h 1m 9.55s	3.1
Pyramid Match	1m 28.02s	2m 13.01s	10m 9.24s	51m 45.10s	3h 50m 38.60s	36m 26.14s	7.0
GraphHopper	2m 11.15s	2m 3.71s	TIMEOUT	TIMEOUT	TIMEOUT	5h 51m 32.27s	10.3
Subgraph Matching	TIMEOUT	TIMEOUT	OUT-OF-MEM	OUT-OF-MEM	OUT-OF-MEM	TIMEOUT	–
Propagation	7.41s	14.26s	1m 23.42s	5m 49.01s	20m 41.73s	4m 34.26s	3.1
Multiscale Laplacian	1h 22m 6.04s	1h 41m 13.74s	8h 21m 18.76s	47m 51.91s	OUT-OF-MEM	9h 24m 15.22s	10.0
CORE WL	36.74s	1m 1.82s	45m 1.09s	OUT-OF-MEM	OUT-OF-MEM	OUT-OF-MEM	8.0
CORE Shortest Path	3m 58.29s	4m 29.55s	10h 37m 3.94s	TIMEOUT	OUT-OF-MEM	TIMEOUT	12.3

[Nikolentzos et al., arXiv:1904.12218]

Graph Classification (Node-Attributed Graphs)

Kernels	DATASETS					Avg. Rank
	ENZYMES	PROTEINS_full	SYNTHETICnew	Synthie	BZR	
Shortest Path	TIMEOUT	TIMEOUT	TIMEOUT	TIMEOUT	TIMEOUT	–
Subgraph Matching	TIMEOUT	OUT-OF-MEM	TIMEOUT	TIMEOUT	80.52 (± 0.43)	3.0
GraphHopper	66.25 (± 1.24)	72.49 (± 0.34)	76.43 (± 1.97)	71.75 (± 1.65)	82.58 (± 1.05)	1.0
Propagation	15.42 (± 1.00)	59.56 (± 0.01)	47.90 (± 3.26)	48.90 (± 2.05)	78.76 (± 0.02)	3.0
Multiscale Laplacian	65.55 (± 0.93)	70.55 (± 0.99)	47.90 (± 2.13)	69.42 (± 1.98)	82.33 (± 1.29)	2.0

[Nikolentzos et al., arXiv:1904.12218]

Running Time (Node-Attributed Graphs)

Kernels	Datasets					Avg. Rank
	ENZYMES	PROTEINS_full	SYNTHETICnew	Synthie	BZR	
Shortest Path	TIMEOUT	TIMEOUT	TIMEOUT	TIMEOUT	TIMEOUT	–
Subgraph Matching	TIMEOUT	OUT-OF-MEM	TIMEOUT	TIMEOUT	8h 2m 3.79s	4.0
GraphHopper	16m 36.12s	5h 16m 46.48s	13m 54.36s	24m 20.00s	4m 24.79s	2.6
Propagation	15.85s	1m 43.58s	13.44s	34.68s	10.40s	1.0
Multiscale Laplacian	26.05s	4h 29m 35.69s	2h 54m 31.22s	15m 11.29s	49m 33.60s	2.4

[Nikolentzos et al., arXiv:1904.12218]

THANK YOU !

http://www.lix.polytechnique.fr/dascim/

Software and data sets:
http://www.lix.polytechnique.fr/dascim/software_datasets/

Preprint available at: https://arxiv.org/pdf/1904.12218.pdf