Machine and Deep learning for Graphs - an introduction

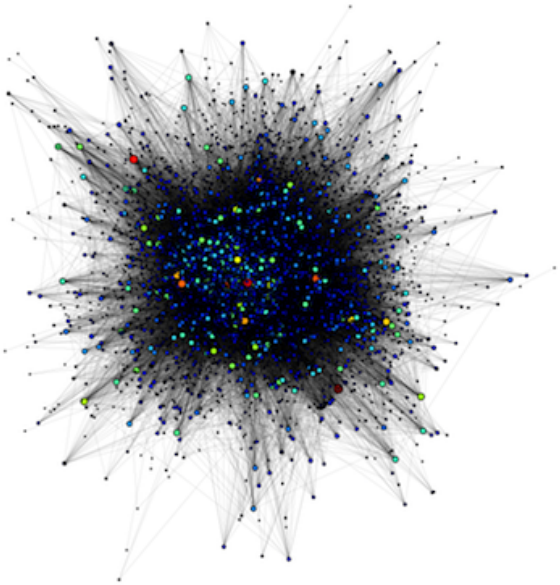
M. Vazirgiannis

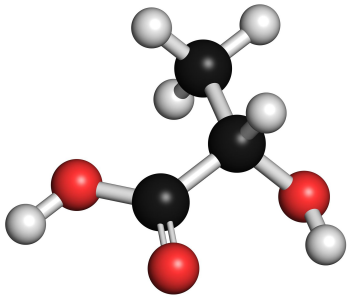
Data Science and Mining Team (DASCIM), LIX
École Polytechnique http://www.lix.polytechnique.fr/dascim
Google Scholar: https://bit.ly/2rwmvQU
Twitter: @mvazirg

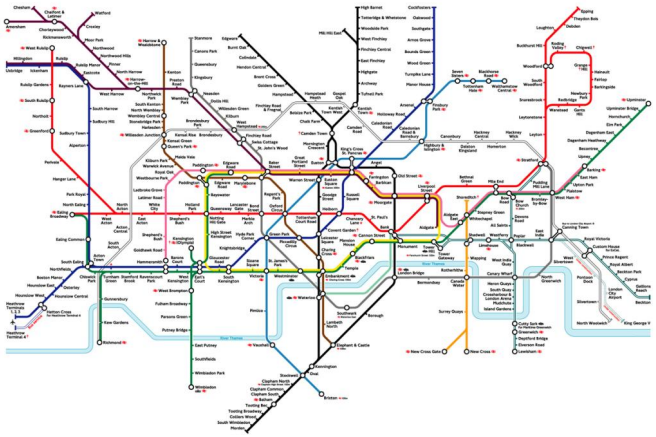
November, 2024

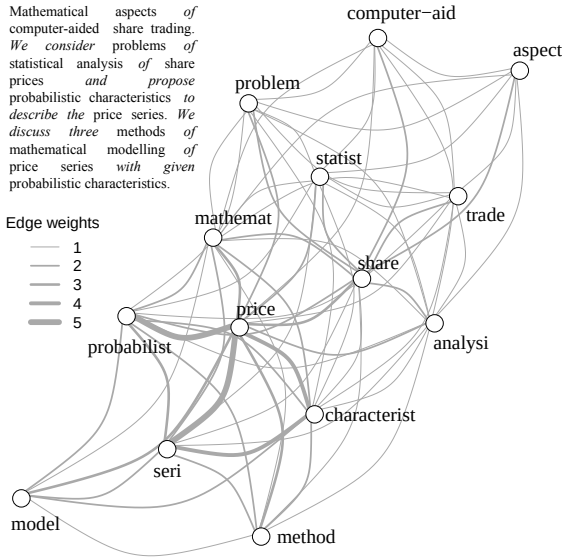
Outline

Graphs Are Everywhere





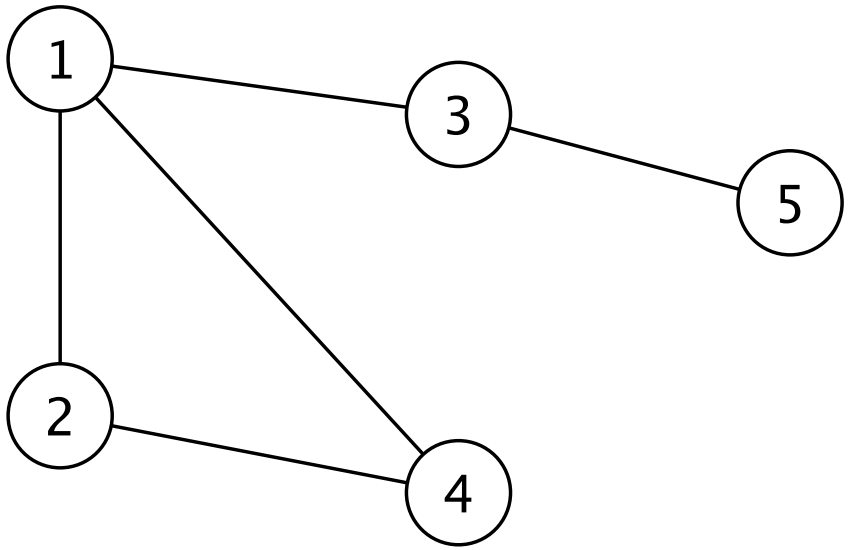




Why graphs?

Graph Preliminaries

Let $G = (V, E)$ be a simple unweighted, undirected graph where V is the set of vertices and E the set of edges



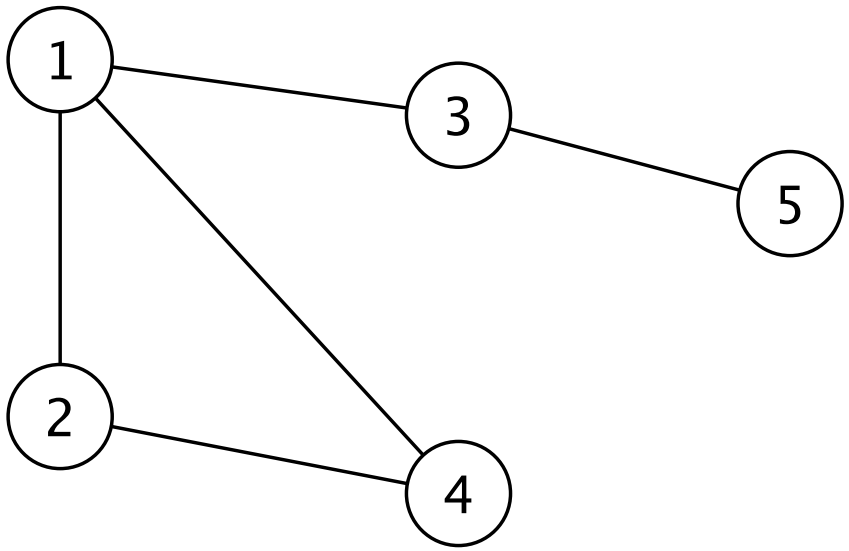
G

$V = \{1, 2, 3, 4, 5\}$

$E = \{(1, 2), (1, 3)(1, 4), (2, 4), (3, 5)\}$

Graph Preliminaries

The neighbourhood $\mathcal{N}(v)$ of vertex v is the set of all vertices adjacent to v, $\mathcal{N}(v) = \{u : (v, u) \in E\}$ where (v, u) is an edge between v and u



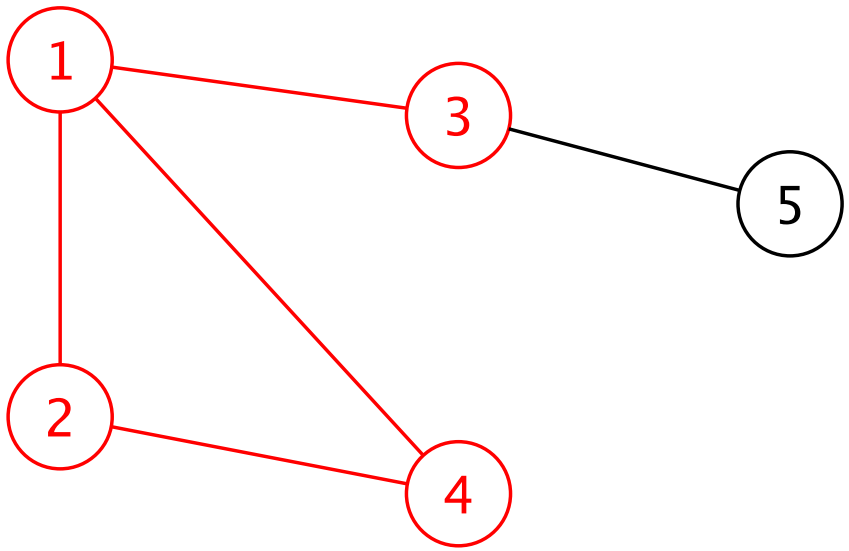
G

$\mathcal{N}(1) = \{2, 3, 4\}$

$\mathcal{N}(5) = \{3\}$

Graph Preliminaries

A walk in a graph G is a sequence of vertices $v_1, v_2, \ldots, v_{k+1}$ where $v_i \in V$ and $(v_i, v_{i+1}) \in E$ for $1 \leq i \leq k$

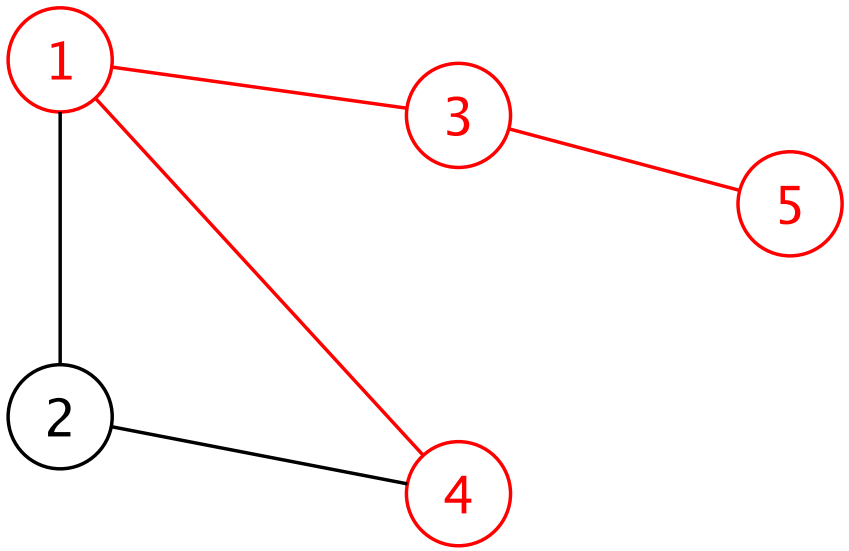


G

Walk: $1 \rightarrow 2 \rightarrow 4 \rightarrow 1 \rightarrow 3$

Graph Preliminaries

A walk in which $v_i \neq v_j \Leftrightarrow i \neq j$ is called a path

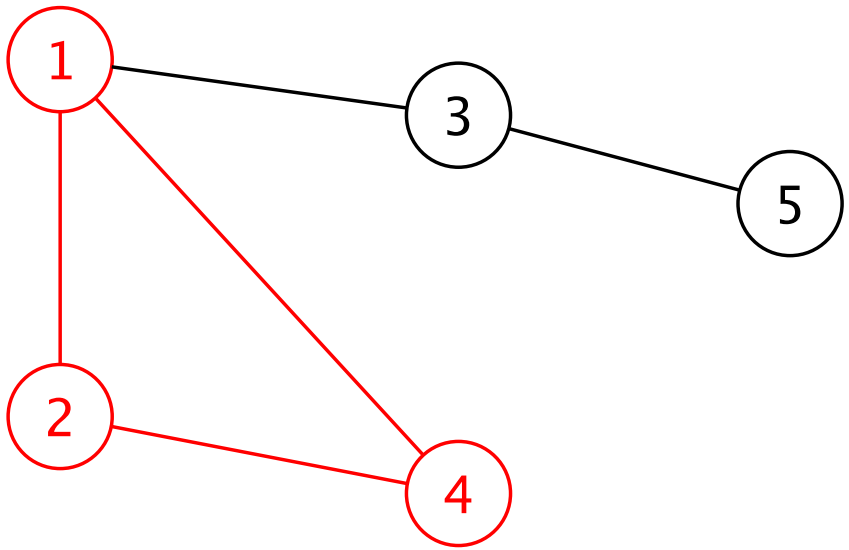


G

Path: $4 \rightarrow 1 \rightarrow 3 \rightarrow 5$

Graph Preliminaries

A cycle is a path with $(v_{k+1}, v_1) \in E$

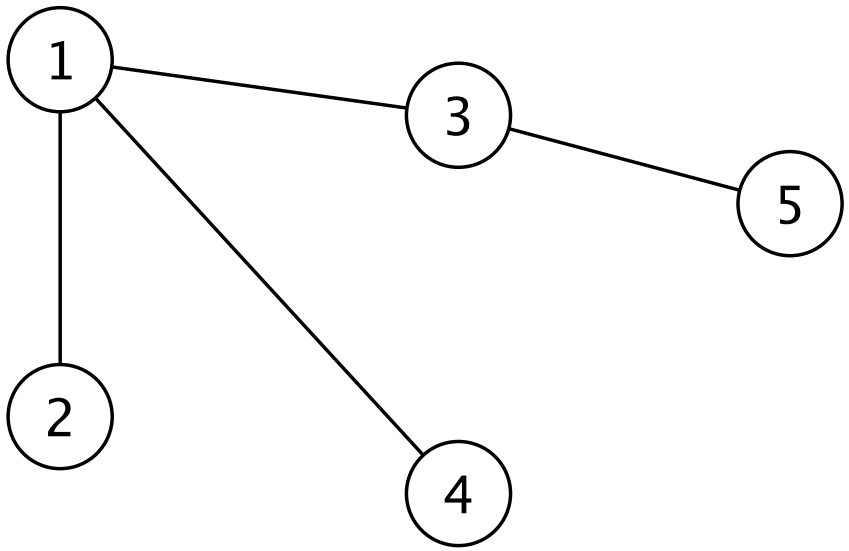


G

Cycle: $1 \rightarrow 2 \rightarrow 4$

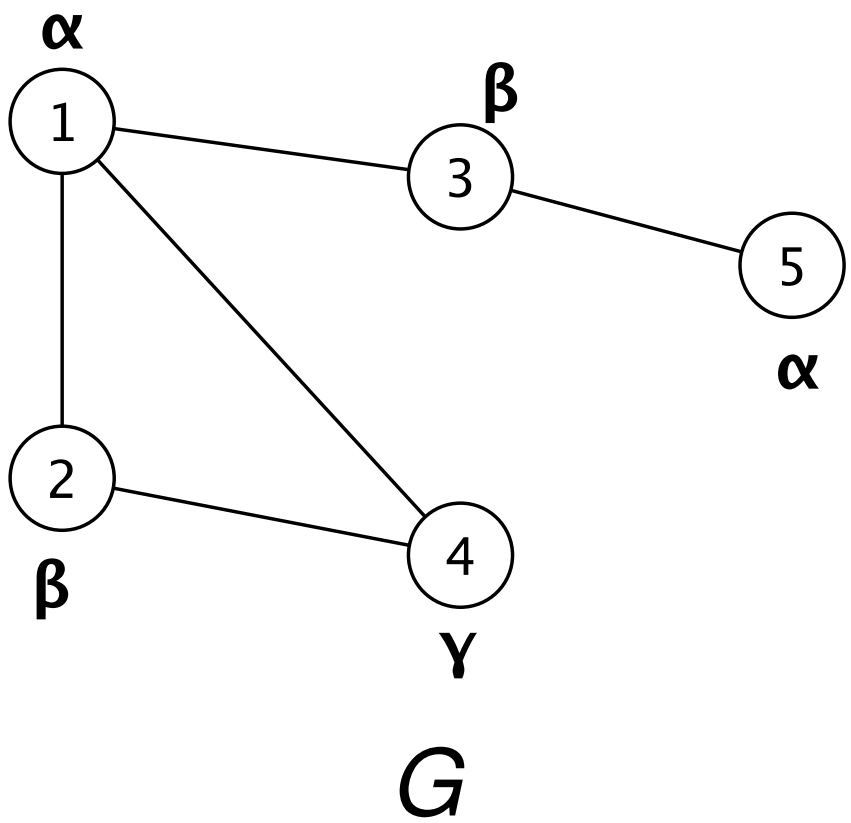
Graph Preliminaries

A subtree is an acyclic subgraph in which there is a path between any two vertices



Graph Preliminaries

A labeled graph is a graph with labels on vertices. Given a set of labels $\mathcal{L}$, $\ell : V \to \mathcal{L}$ is a function that assigns labels to the vertices of the graph

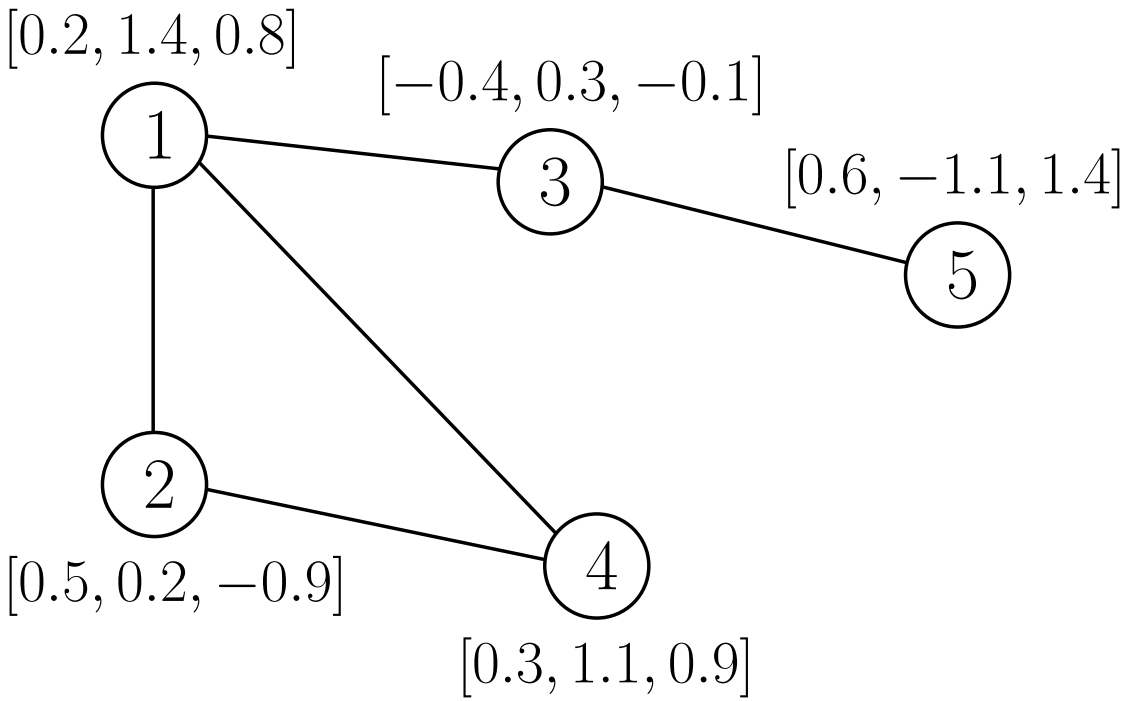


$\mathcal{L} = \{\alpha, \beta, \gamma\}$

$\ell(1) = \alpha \quad \ell(4) = \gamma$

Graph Preliminaries

An attributed graph is a graph with attributes on vertices. Each vertex $v \in V$ is annotated with a feature vector h_v



G

$h_1, \ldots, h_5 \in \mathbb{R}^3$

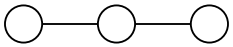
$h_1 = [0.2, 1.4, 0.8]^\top \quad h_3 = [-0.4, 0.3, -0.1]^\top$

Machine Learning on Graphs

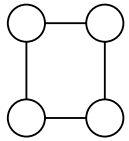
Machine learning tasks on graphs:

- 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

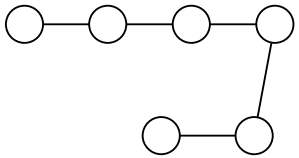
Graph Classification



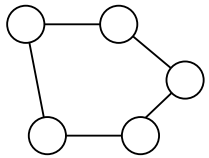
class -1



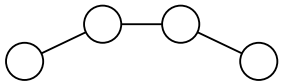
class 1



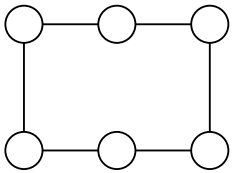
class -1



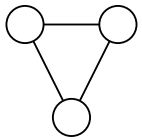
???



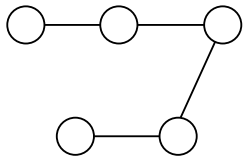
class -1



class 1



class 1



???

- 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 $f(x)$

Graph Comparison

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

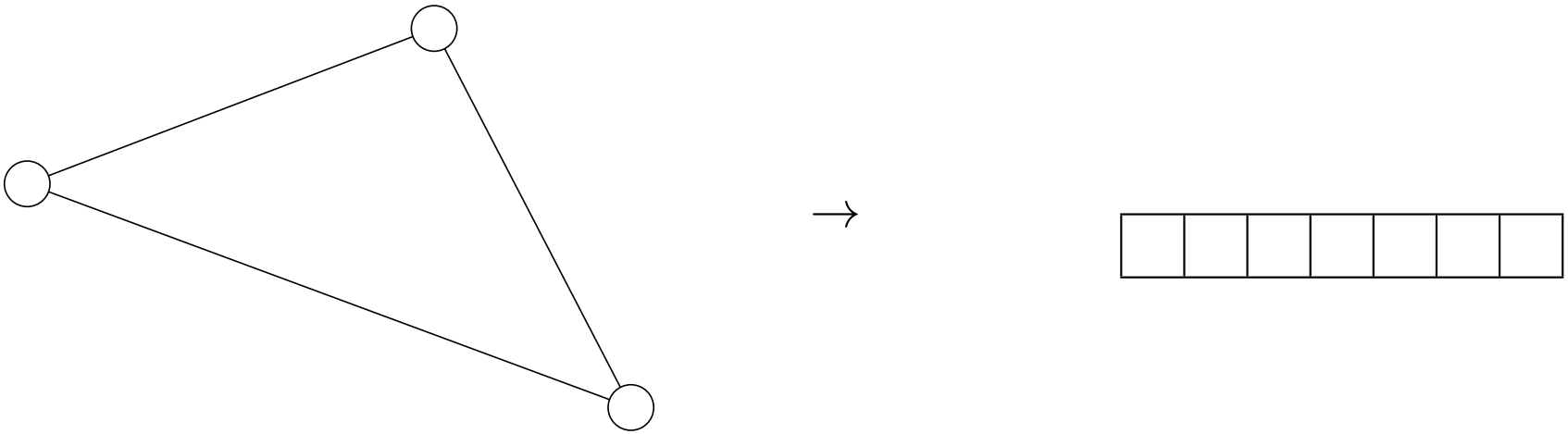
Many problems related to it are **NP-complete**

- 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 $\mathbf{K}$ defined by $\mathbf{K}_{ij} = k(x_i, x_j)$ 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 $\phi : \mathcal{X} \to \mathbb{H}$ into a Hilbert space $\mathbb{H}$, such that:

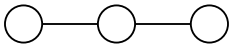
$$k(x, y) = \langle \phi(x), \phi(y) \rangle$$

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

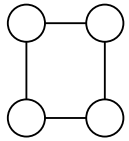
- Informally, $k(x, y)$ is a measure of similarity between x and y

Outline

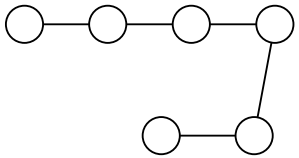
Graph Classification



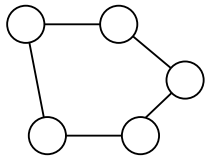
class -1



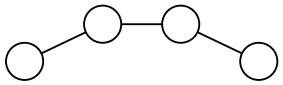
class 1



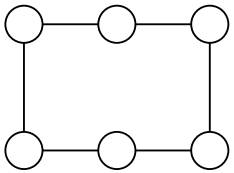
class -1



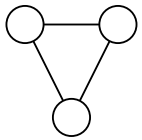
???



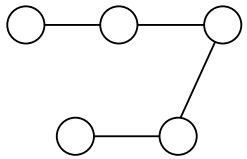
class -1



class 1



class 1



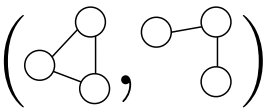
???

- 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 Comparison

Graph classification very related to graph comparison

Example

$f($  $)$

$+$

$k\text{-}nn$

$=$

graph classification

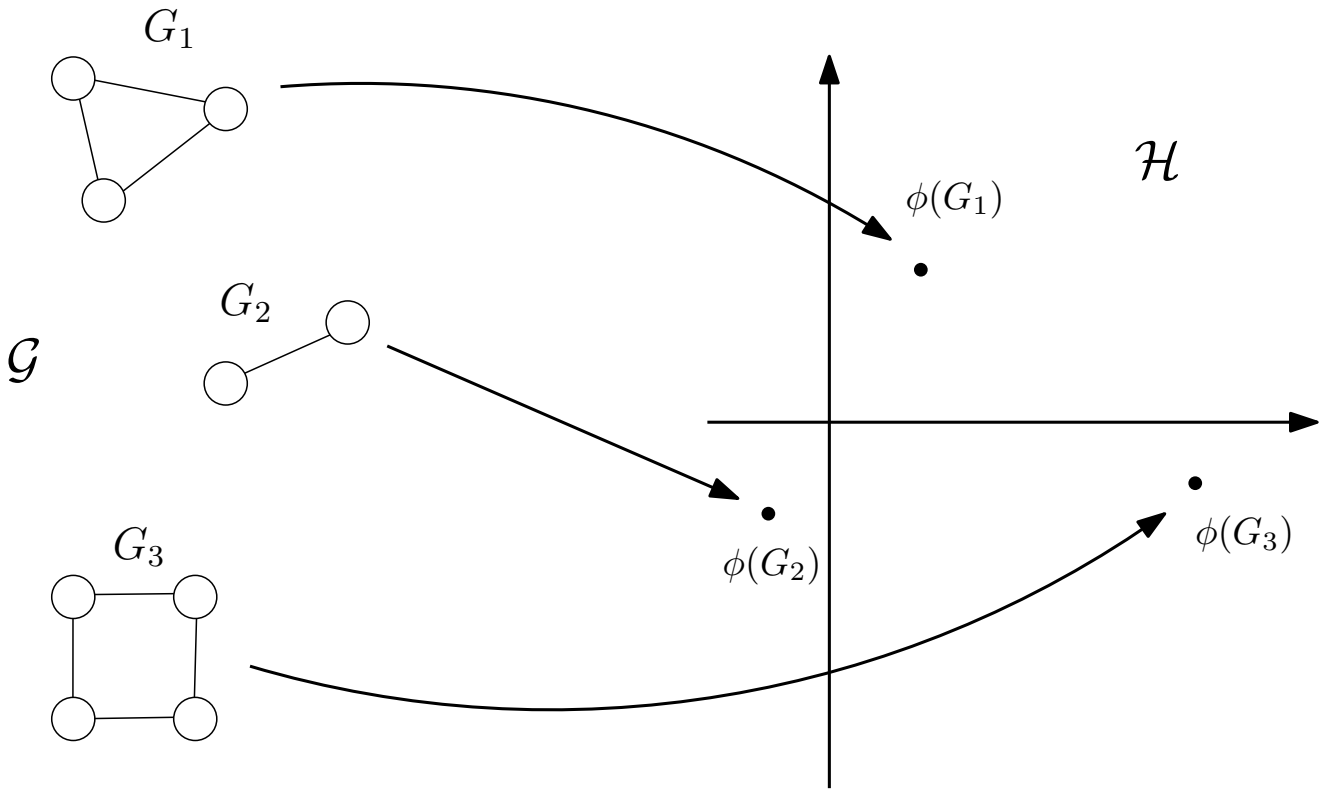
Although graph comparison seems a tractable problem, it is very **complex**

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

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 $\phi : \mathcal{X} \to \mathbb{H}$ of a pair of graphs into a Hilbert space
- Makes the whole family of kernel methods applicable to graphs

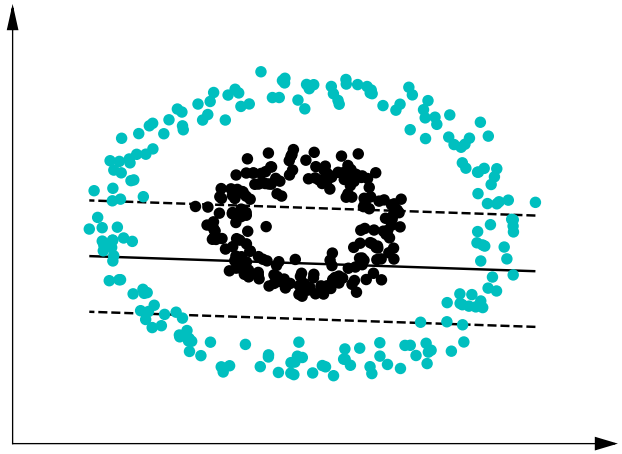


Kernel Trick

- 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
$x = [x_1, x_2]^\top, y = [y_1, y_2]^\top \in \mathcal{X}$

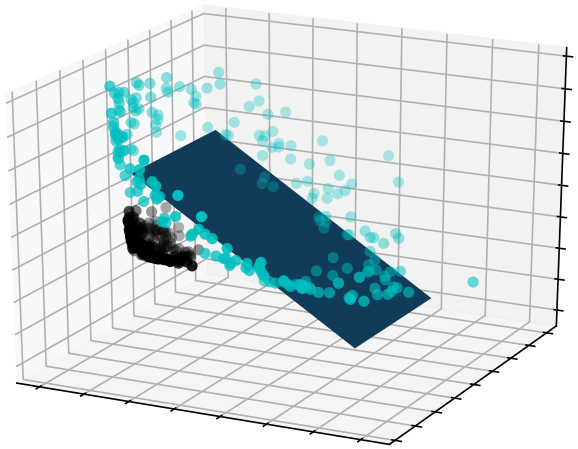


For any $x = [x_1, x_2]^\top$ let ϕ be a map $\phi : \mathbb{R}^2 \to \mathbb{R}^3$ defined as:

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

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

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



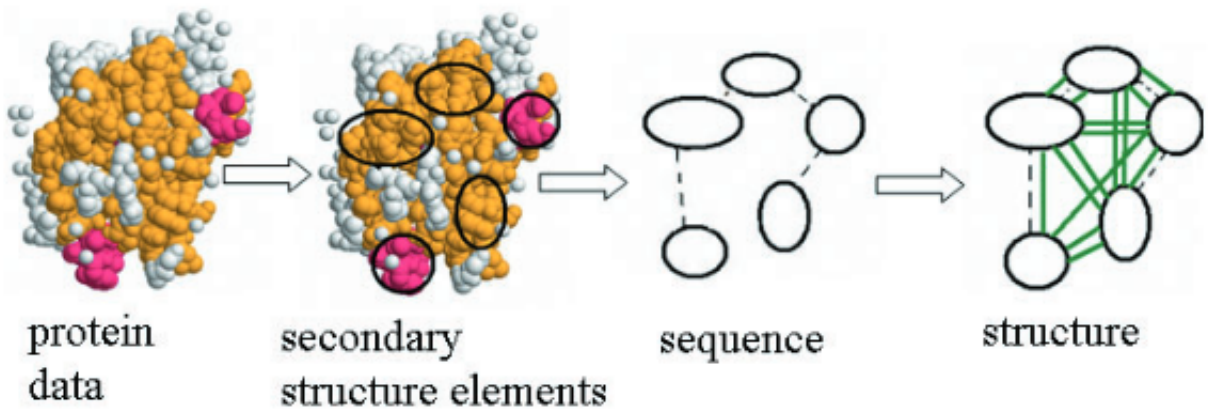
Applications

- 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]

Protein Function Prediction

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

- structure
- sequence
- chemical properties

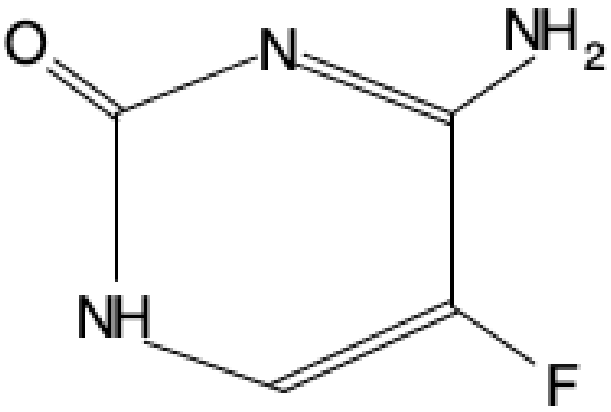


Perform **graph classification** to predict the function of proteins

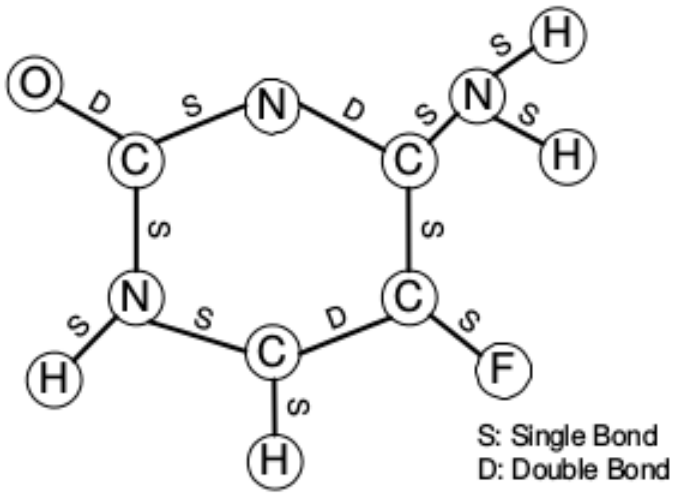
Kernel type	Accuracy
Vector kernel	76.86
Optimized vector kernel	80.17
Graph kernel	77.30
Graph kernel without structure	72.33
Graph kernel with global info	84.04
DALI classifier	75.07

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

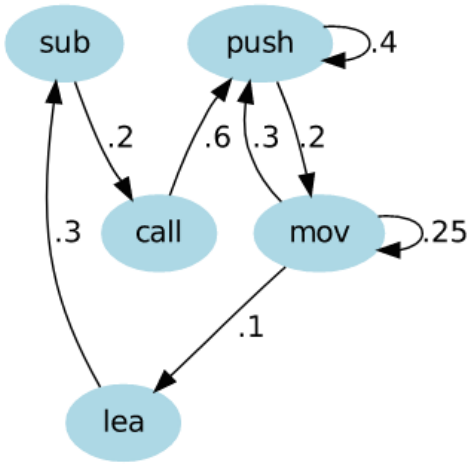
Lin.Reg	*DT*	*NN*	*Progol1*	*Progol2*	*Sebag*	*Kramer*	graph kernels
89.3%	88.3%	89.4%	81.4%	87.8%	93.3%	95.7%	91.2%

[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

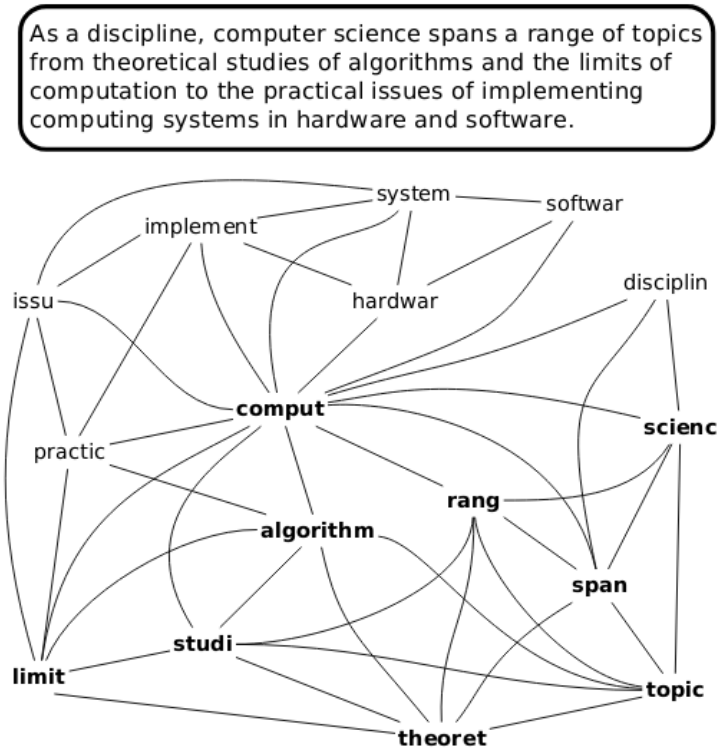
Method	Accuracy (%)
Gaussian kernel	**99.09**
Spectral kernel	96.36
Combined kernel	**100.00**
n-gram ($n = 4$, $L = 1{,}000$, SVM = 2-poly)	94.55
n-gram ($n = 4$, $L = 2{,}500$, SVM = Gauss)	93.64
n-gram ($n = 6$, $L = 2{,}500$, SVM = 2-poly)	92.73
n-gram ($n = 3$, $L = 1{,}000$, SVM = 2-poly)	89.09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

Substructures-based Kernels

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

- walks
- shortest paths
- cyclic patterns
- subtree patterns
- graphlets

These kernels are instance of the R-convolution framework

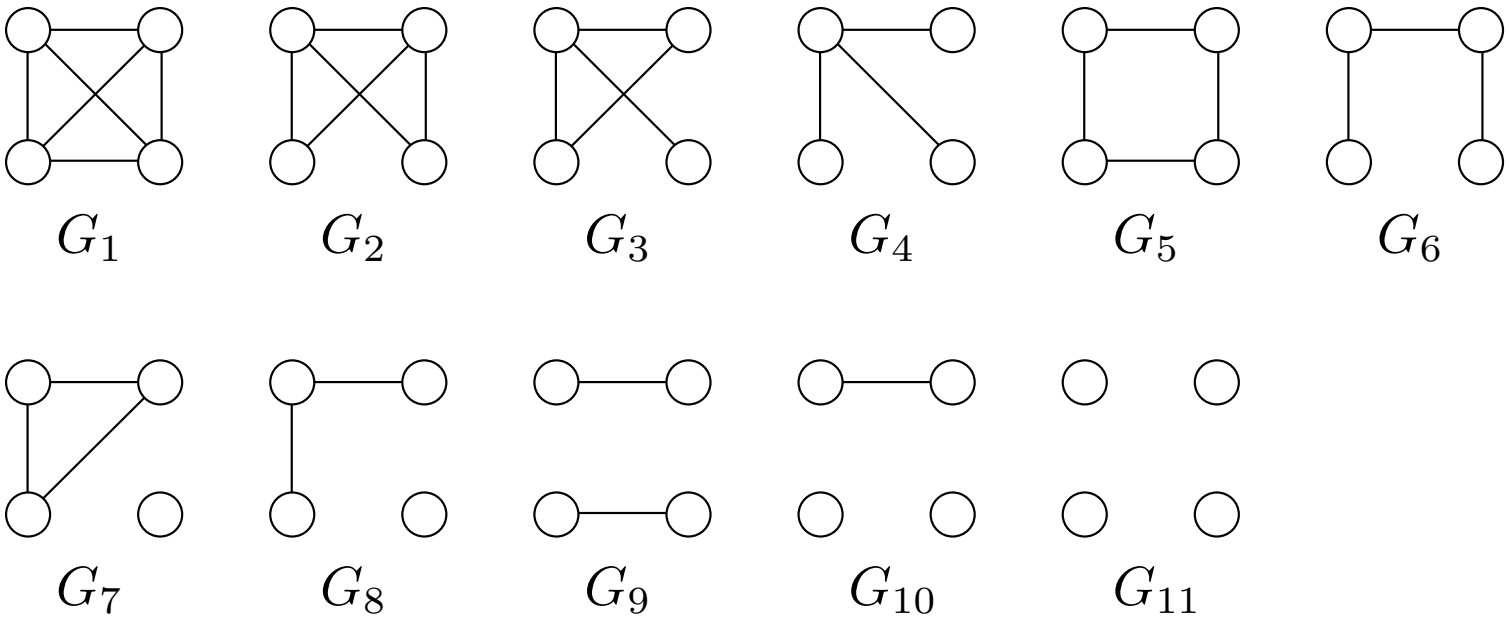
Graphlet Kernel

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

The graphlet kernel is defined as:

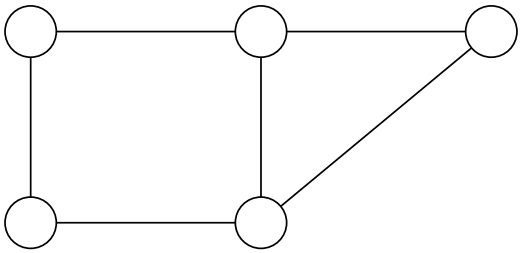
$$k(G_1, G_2) = \langle f_{G_1}, f_{G_2} \rangle$$

Problems:

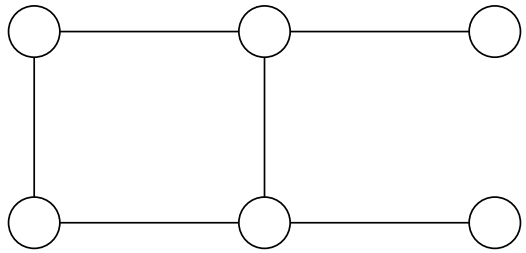
- There are $\binom{n}{k}$ size-k subgraphs in a graph
- Exaustive enumeration of graphlets is very expensive

  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:

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

Shortest Path Kernel

Compares the length of shortest-paths of two graphs

- and their endpoints in labeled graphs

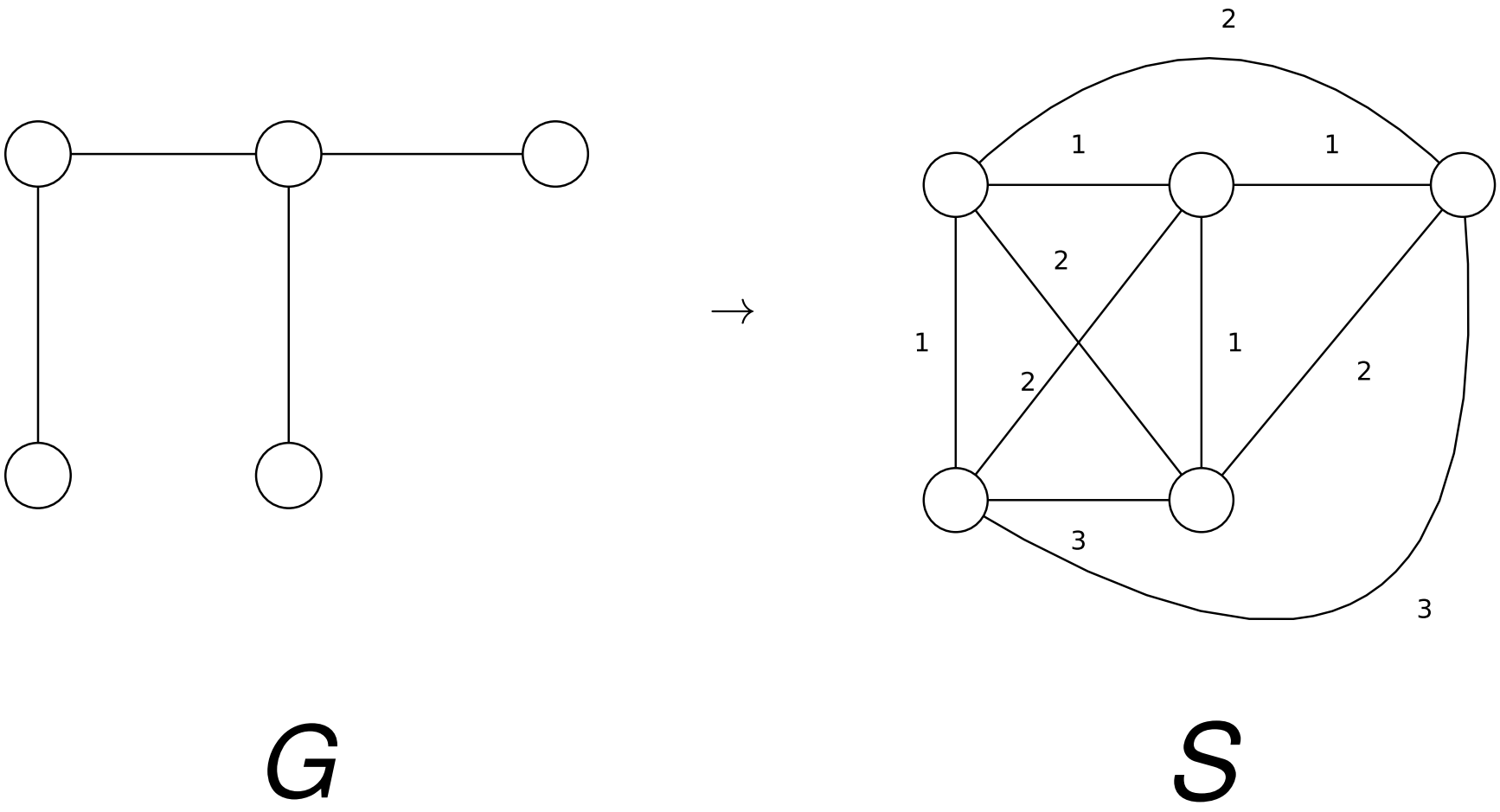
Floyd-transformation

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



G S

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:

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

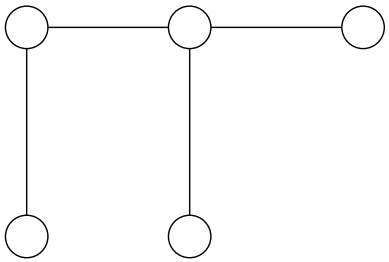
  where $\ell(e)$ gives the label of edge e

- For labeled graphs, it can be:

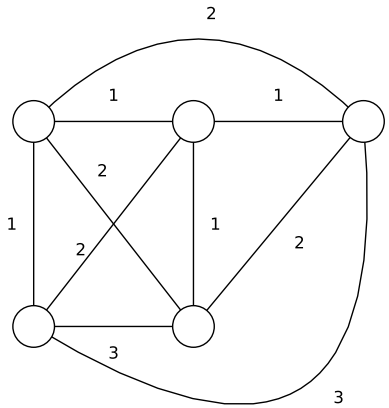
$$k_{edge}(e_1, e_2) = \begin{cases} 1 & \text{if } \ell(e_1) = \ell(e_2) \wedge \ell(e_1^1) = \ell(e_2^1) \wedge \ell(e_1^2) = \ell(e_2^2), \\ 0 & \text{otherwise} \end{cases}$$

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

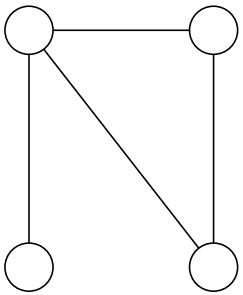
Floyd-transformations



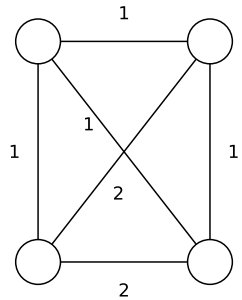
$\Rightarrow$



G_1 S_1



$\Rightarrow$



G_2 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

Graph Invariant Kernels

They decompose graphs into sets of vertices, and compare them based on:

- their attributes
- their structural roles

Let R be a decomposition relation that specifies a decomposition of G into its parts $\rightarrow$ therefore, $R^{-1}(G)$ is the multiset of all patterns in G
$\hookrightarrow$i.e., $R^{-1}(G)$ can be a set of k-hop neighborhood subgraphs for some k
Then, the graph invariant kernel is computed as follows:

$$k(G, G') = \sum_{v \in V} \sum_{v' \in V'} w(v, v')\, k_{attr}(v, v')$$

where k_{attr} is a kernel between vertex attributes, and $w(v, v')$ is a weight function defined as follows:

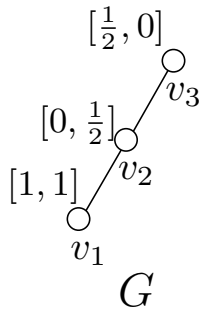
$$w(v, v') = \sum_{g \in R^{-1}(G)} \sum_{g' \in R^{-1}(G')} k_{inv}(v, v')\, \frac{\delta_m(g, g')}{|V_g||V_{g'}|}\, \mathbb{1}\{v \in V_g \wedge v' \in V_{g'}\}$$

where δ_m is a dirac function that determines whether two patterns match, V_g, $V_{g'}$ are the set of vertices of patterns g, g', and $\mathbb{1}$ is an indicator function

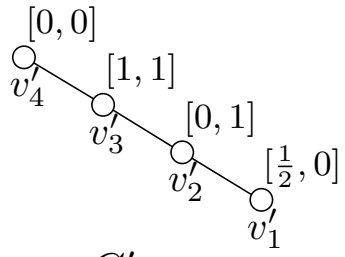
[Orsini et al., IJCAI'15]

Example

Given the following two graphs:



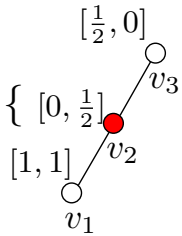
G



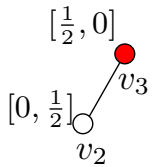
G'

R is a relation that decomposes a graph into its 1-hop neighborhood subgraphs:

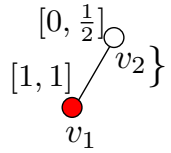
$R^{-1}(G) = \{$



,

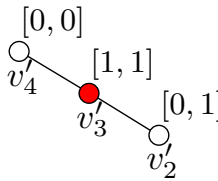


,

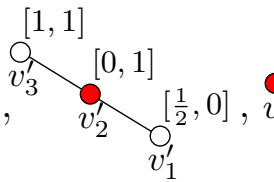


$\}$

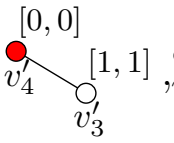
$R^{-1}(G') = \{$



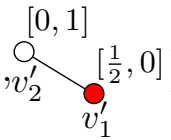
,



,



,



$\}$

Example

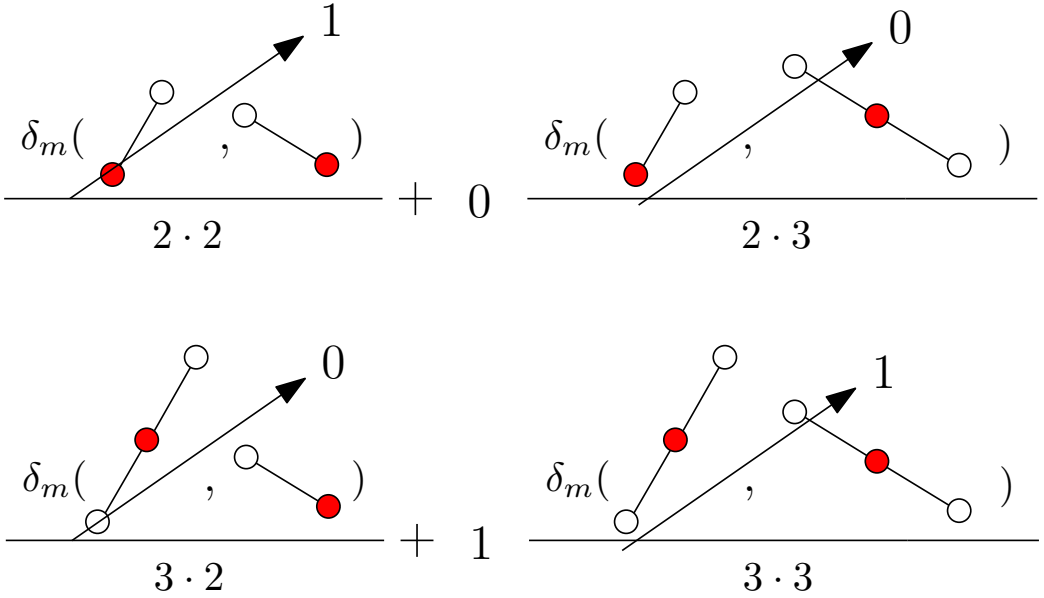
The weight function $w(v_1, v_1')$ is computed as follows:

$$w(v_1, v_1') = 1 \frac{\delta_m(\,,\,)}{2 \cdot 2} + 0 \frac{\delta_m(\,,\,)}{2 \cdot 3} + 0 \frac{\delta_m(\,,\,)}{3 \cdot 2} + 1 \frac{\delta_m(\,,\,)}{3 \cdot 3}$$

Example

The weight function $w(v_1, v_1')$ is computed as follows:

$$w(v_1, v_1') = 1 \cdot \frac{\delta_m(\,,\,)}{2 \cdot 2} + 0 \cdot \frac{\delta_m(\,,\,)}{2 \cdot 3} + 0 \cdot \frac{\delta_m(\,,\,)}{3 \cdot 2} + 1 \cdot \frac{\delta_m(\,,\,)}{3 \cdot 3}$$



$$= \tfrac{13}{36}$$

then the contribution of the two nodes to the sum is:

$$w(v_1, v_1')\, k_{attr}(v_1, v_1') = \frac{13}{36} + \langle [1,1]^\top, [\frac{1}{2}, 0]^\top \rangle = \frac{31}{36}$$

GraKel: A python library for graph kernels

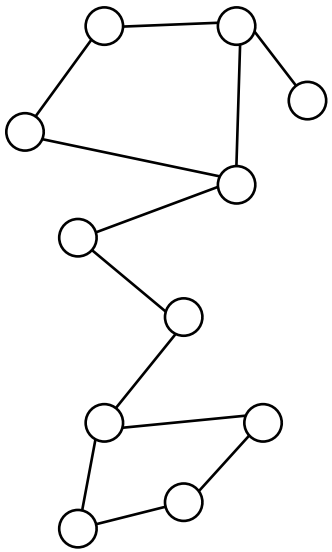
- Python library for graph similarity computations
- Contains practically all known graph kernels
- Compatible with scikit learn
- Open source - can be extended
- Project repository https://ysig.github.io/GraKeL/dev/

Large scale survey on kernels:
”Graph Kernels: a Survey”, *G.Nikolentzos*, *M.Vazirgiannis*, https://arxiv.org/abs/1904.12218

Outline

Deep Learning for Graphs - Node Embeddings

Traditional Node Representation
Representation: row of adjacency matrix

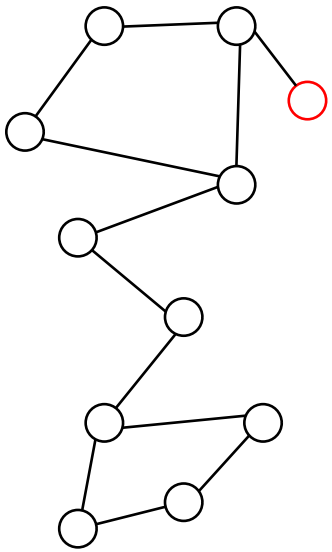


$\rightarrow$

$$\begin{pmatrix} 0 & 1 & \dots & 0 \\ 1 & 0 & \dots & 1 \\ \vdots & \vdots & \vdots & \vdots \\ 0 & 1 & \dots & 0 \end{pmatrix}$$

Deep Learning for Graphs - Node Embeddings

Traditional Node Representation
Representation: row of adjacency matrix

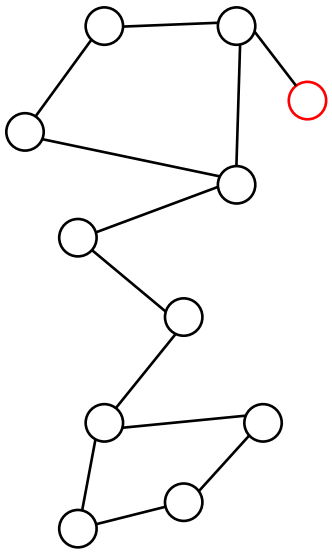


$\rightarrow$

$$\begin{pmatrix} 0 & 1 & \dots & 0 \\ 1 & 0 & \dots & 1 \\ \vdots & \vdots & \vdots & \vdots \\ 0 & 1 & \dots & 0 \end{pmatrix}$$

Deep Learning for Graphs - Node Embeddings

Traditional Node Representation
Representation: row of adjacency matrix



$\rightarrow$

$$\begin{pmatrix} 0 & 1 & \dots & 0 \\ 1 & 0 & \dots & 1 \\ \vdots & \vdots & \vdots & \vdots \\ 0 & 1 & \dots & 0 \end{pmatrix}$$

However, such a representation suffers from:

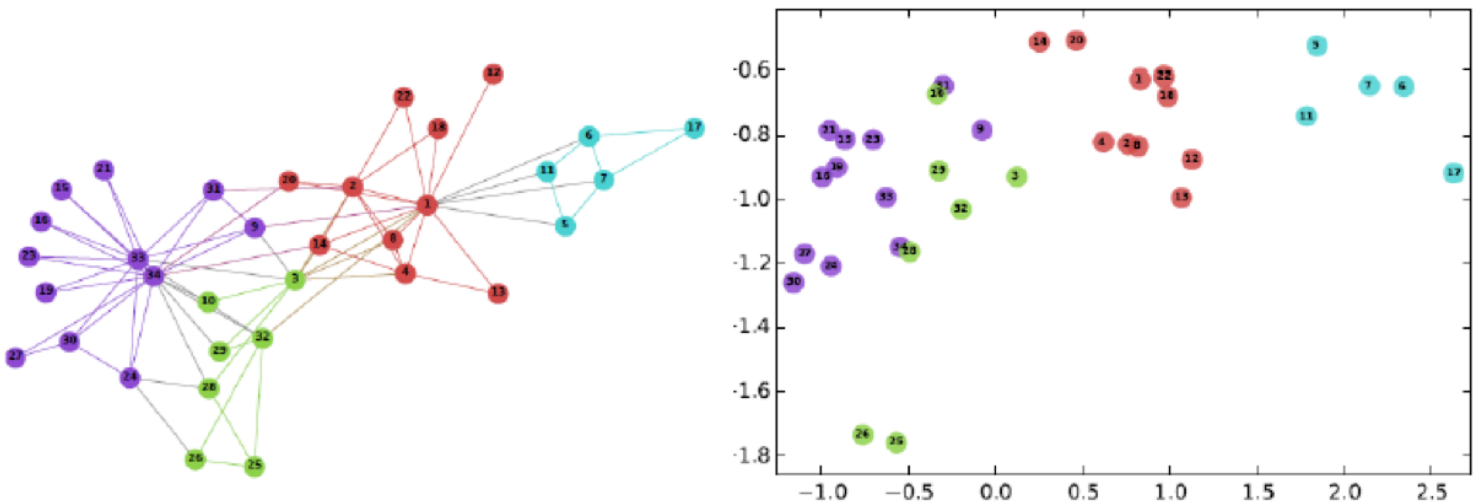
- data sparsity
- high dimensionality

$\vdots$

Node Embedding Methods

Map vertices of a graph into a low-dimensional space:

- dimensionality $d \ll |V|$
- similar vertices are embedded close to each other in the low-dimensional space



Input Output

Early Methods

- Focused mainly on matrix-factorization approaches (e. g., Laplacian eigenmaps)
- Laplacian eigenmaps projects two nodes i and j close to each other when the weight of the edge between the two nodes A_{ij} is high
- Embeddings are obtained by the following objective function:

$$y^* = \arg\min \sum_{i \neq j} (y_i - y_j)^2 A_{ij} = \arg\min y^T L y$$

  where L is the graph Laplacian

- The solution is obtained by taking the eigenvectors corresponding to the d smallest eigenvalues of the normalized Laplacian matrix

[1] Belkin and Niyogi. Laplacian Eigenmaps and Spectral Techniques for Embedding and Clustering. In NIPS'02

Recent Methods

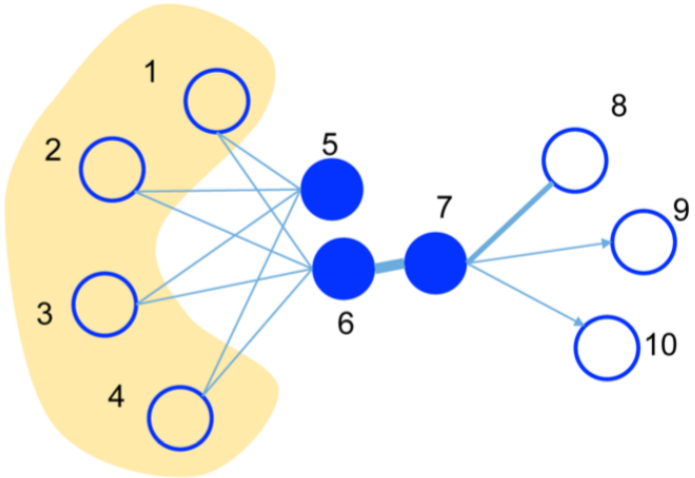
Most methods belong to the following groups:

1. Random walk based methods: employ random walks to capture structural relationships between nodes
2. Edge modeling methods: directly learn node embeddings using structural information from the graph
3. Matrix factorization methods: generate a matrix that represents the relationships between vertices and use matrix factorization to obtain embeddings
4. Deep learning methods: apply deep learning techniques to learn highly non-linear node representations

Proximities

First-order proximity: observed links in the network

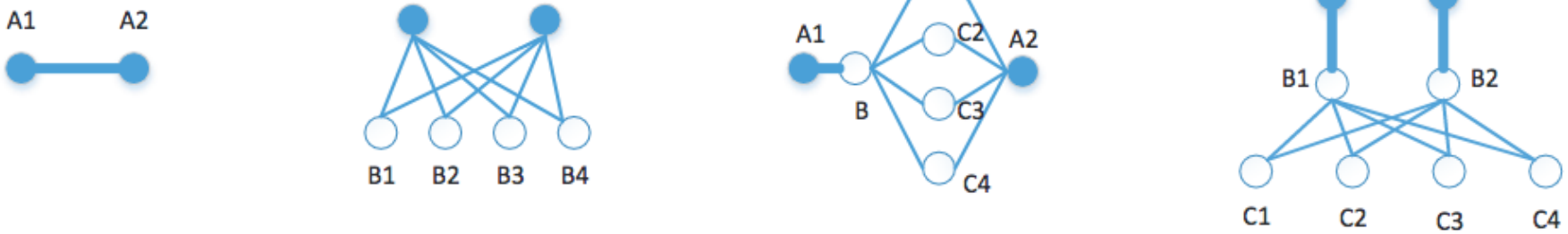
Second-order proximity: shared neighborhood structures



- Vertices 6 and 7 have a high *first-order proximity* since they are connected through a strong tie $\rightarrow$ they should be placed closely in the embedding space
- Vertices 5 and 6 have a high *second-order proximity* since they share similar neighbors $\rightarrow$ they should also be placed closely

Proximities

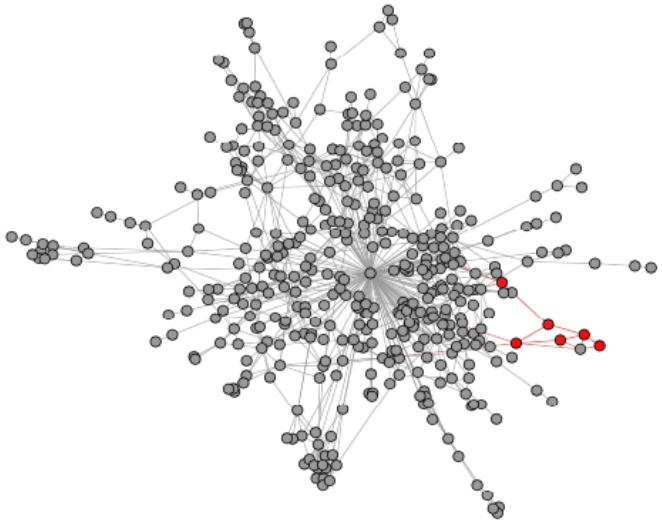
k-order proximities for $k = 1, \ldots, 4$



- Second-order and high-order proximities capture similarity between vertices with similar structural roles
- Higher-order proximities capture more global structure

DeepWalk

Inspired by recent advances in language modeling [1]



$\rightarrow$

$v_5 \rightarrow v_8 \rightarrow v_{32} \rightarrow v_{28} \rightarrow v_6 \rightarrow v_{10} \rightarrow v_9$

$v_3 \rightarrow v_5 \rightarrow v_{28} \rightarrow v_8 \rightarrow v_9 \rightarrow v_{10} \rightarrow v_{25}$

$v_{20} \rightarrow v_{10} \rightarrow v_{12} \rightarrow v_6 \rightarrow v_8 \rightarrow v_4 \rightarrow v_5$

$v_{23} \rightarrow v_5 \rightarrow v_{32} \rightarrow v_{10} \rightarrow v_8 \rightarrow v_3 \rightarrow v_1$

$v_4 \rightarrow v_3 \rightarrow v_1 \rightarrow v_5 \rightarrow v_1 \rightarrow v_{12} \rightarrow v_{10}$

$\vdots$

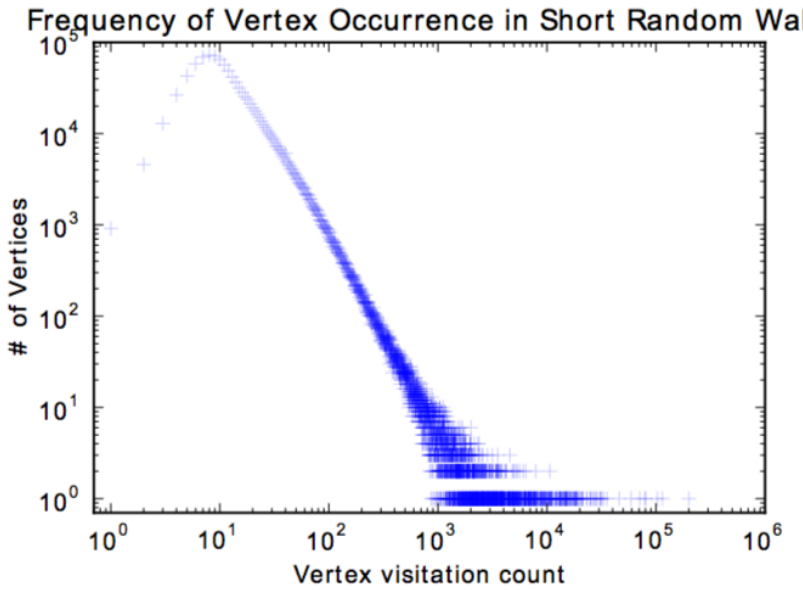
- Simulates a series of short random walks

[1] Mikolov et al. Distributed Representations of Words and Phrases and their Compositionality. In NIPS'13

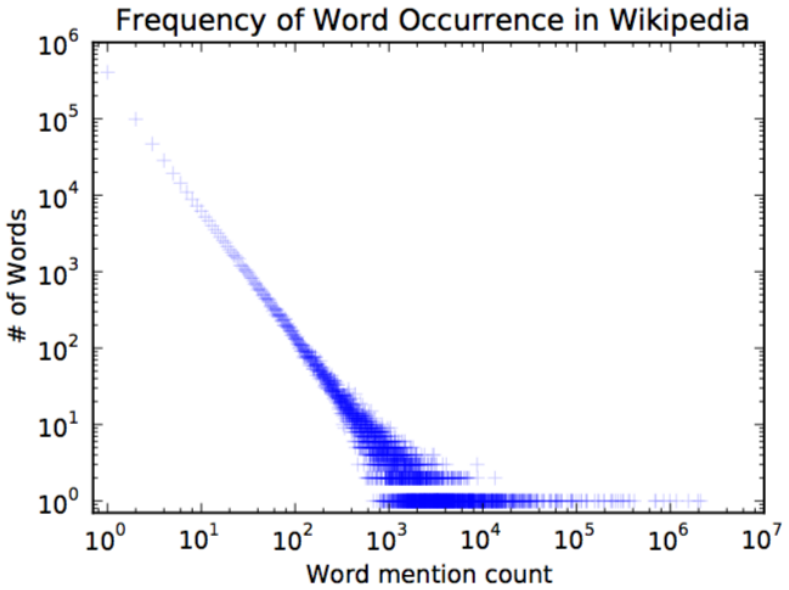
[2] Perozzi et al. DeepWalk: Online Learning of Social Representations. In KDD'14

DeepWalk

Inspired by recent advances in language modeling [1]



(a) YouTube Social Graph



(b) Wikipedia Article Text

- Simulates a series of short random walks
- **Main Idea**: Short random walks = Sentences

[1] Mikolov et al. Distributed Representations of Words and Phrases and their Compositionality. In NIPS'13

[2] Perozzi et al. DeepWalk: Online Learning of Social Representations. In KDD'14

Skipgram

Skipgram is a recently-proposed language model that:

- uses one word to predict the context
- context is composed of words appearing to both the right and left of the given word
- removes the ordering constraint on the problem (i. e. does not take into account the offset of context words from the given word)

In our setting:

$$\mathcal{W}_{v_4} = \begin{matrix} 4 \\ u_k \end{matrix} \begin{bmatrix} 3 \\ 1 \\ 5 \end{bmatrix} v_j \longrightarrow \begin{matrix} \overset{d}{\square\square\square\square\square} \\ \square\square\square\square\square \\ \square\square\square\square\square \end{matrix} j$$

$$\begin{matrix} 1 \\ \vdots \end{matrix}$$

- Slide a window of length $2w + 1$ over the random walk
- Use the representation of central vertex to predict its neighbors

Skipgram

This yields the optimization problem:

$$argmin_f \quad -\frac{1}{T}\sum_{i=1}^{T} \log P(\{v_{i-w}, \ldots, v_{i+w}\} \setminus v_i | f(v_i))$$

v_i: central vertex
$v_{i-w}, \ldots, v_{i+w}$: neighbors of central vertex
$f(v)$: embedding of vertex v
Skipgram approximates the above conditional probability using the following independence assumption:

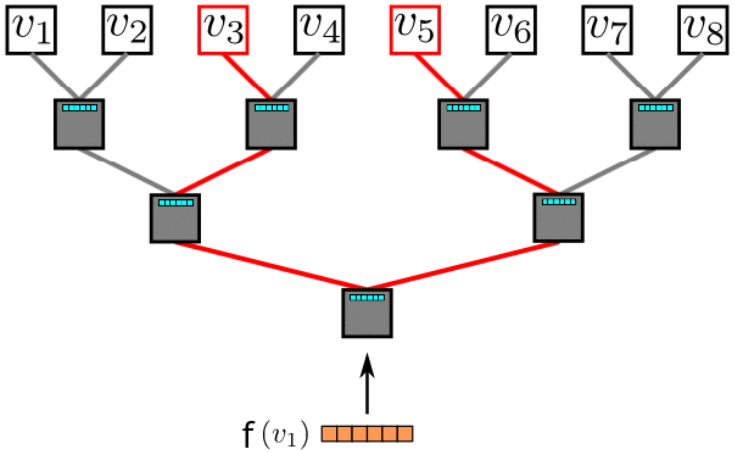
$$minimize_f \quad -\frac{1}{T}\sum_{i=1}^{T}\sum_{\substack{j=i-w \\ j\neq i}}^{i+w} \log P(v_j | f(v_i))$$

- We can learn such a posterior distribution using several choices of classifiers
- **However**, most of them (e. g., logistic regression) would produce a huge number of labels (i. e. $|V|$ labels)
- Instead, we approximate the distribution using the Hierarchical Softmax

Hierarchical Softmax

Reduces complexity from $\mathcal{O}(|V|)$ to $\mathcal{O}(\log|V|)$ using a binary tree

- Assigns the vertices to the leaves of a binary tree
- New problem: Maximizing the probability of a specific path in the hierarchy



If the path to vertex v_j is identified by a sequence of tree nodes $(b_0, b_1, \ldots, b_{\lceil * \rceil \log|V|})$ then

$$P(v_j|f(v_i)) = \prod_{l=1}^{\lceil * \rceil \log|V|} P(b_l|f(v_i))$$

where

$$P(b_l|f(v_i)) = 1/(1+e^{-f(v_i)^\top f'(b_l)}) = \sigma(f(v_i)^\top f'(b_l))$$

and $f'(b_l) \in \mathbb{R}^d$ is the representation assigned to tree node b_l's parent

node2vec

Like DeepWalk, node2vec is also a random walk based method

DeepWalk uses a *rigid* search strategy

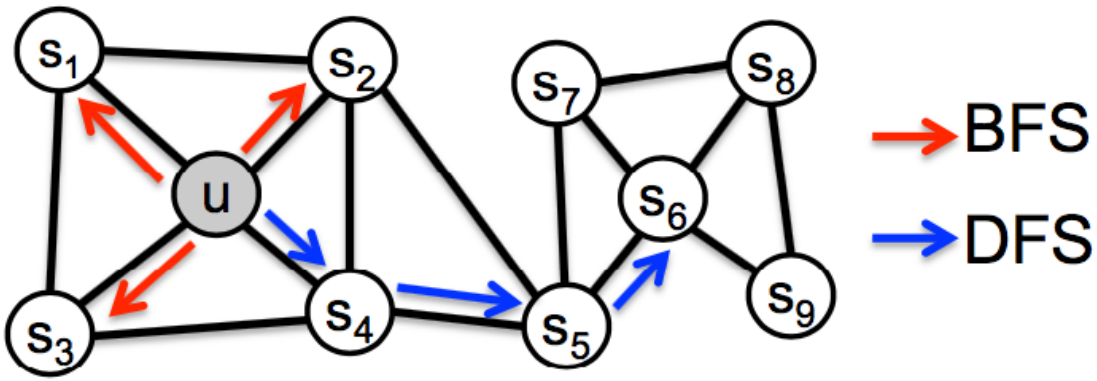
Conversely, node2vec simulates a family of biased random walks which

- explore diverse neighborhoods of a given vertex
- allow it to learn representations that organize vertices based on
  - their network roles
  - the communities they belong to

[1] Grover and Leskovec. node2vec: Scalable Feature Learning for Networks. In KDD'16

Two Extreme Sampling Strategies

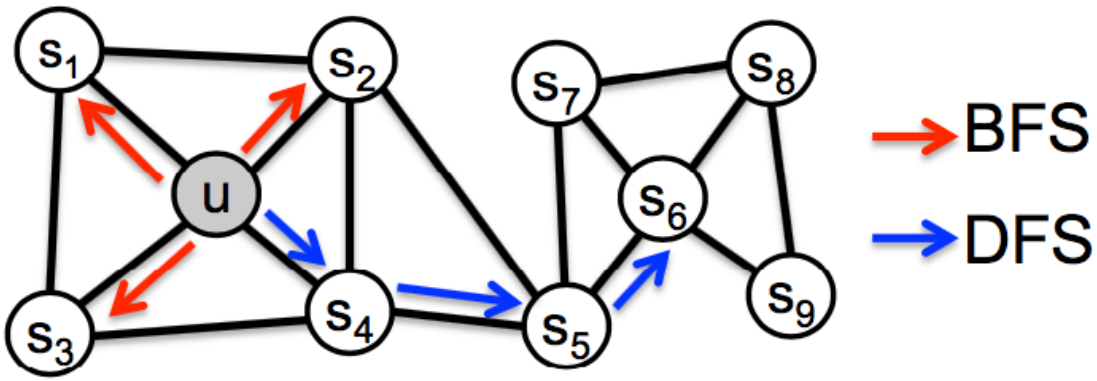
The *breadth-first sampling* (BFS) and *depth-first sampling* (DFS) represent extreme scenarios in terms of the search space



Goal: Given a source node u, sample its neighborhood $\mathcal{N}(u)$ where $|\mathcal{N}(u)| = k$

Two Extreme Sampling Strategies

The *breadth-first sampling* (BFS) and *depth-first sampling* (DFS) represent extreme scenarios in terms of the search space

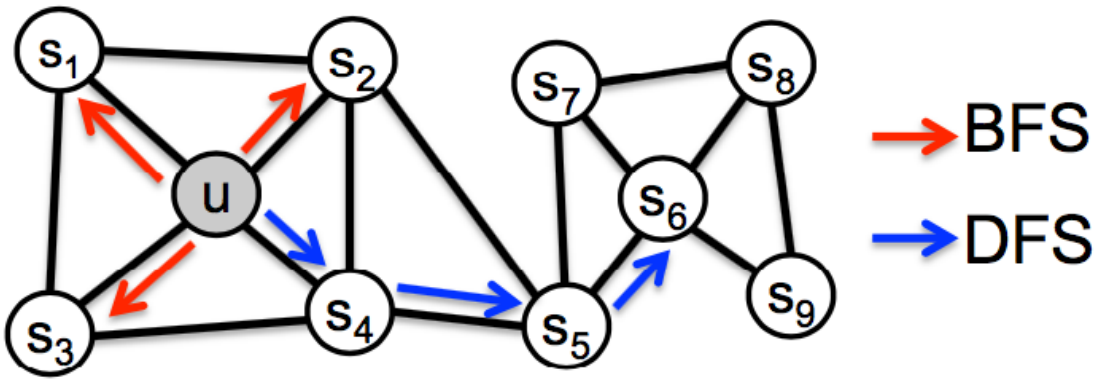


In most applications, we are interested in two kinds of similarities between vertices:

1. homophily: nodes that are highly interconnected and belong to similar communities should be embedded closely together (e. g., s_1 and u)
2. structural equivalence: nodes that have similar structural roles should be embedded closely together (e. g., u and s_6)

Two Extreme Sampling Strategies

The *breadth-first sampling* (BFS) and *depth-first sampling* (DFS) represent extreme scenarios in terms of the search space



BFS and DFS strategies play a key role in producing representations that reflect these two properties:

- The neighborhoods sampled by BFS lead to embeddings that correspond closely to structural equivalence
- The neighborhoods sampled by DFS reflect a macro-view of the neighborhood which is essential in inferring communities based on homophily

Random Walks of node2vec

Given a source node, node2vec simulates a random walk of fixed length l

$$v_1 \rightarrow v_2 \rightarrow v_3 \rightarrow \ldots \rightarrow v_l$$

The i^{th} node in the walk is generated as follows:

$$P(c_i = x | c_{i-1} = v) = \begin{cases} \frac{\pi_{vx}}{Z}, & \text{if } (v, x) \in E \\ 0, & \text{otherwise} \end{cases}$$

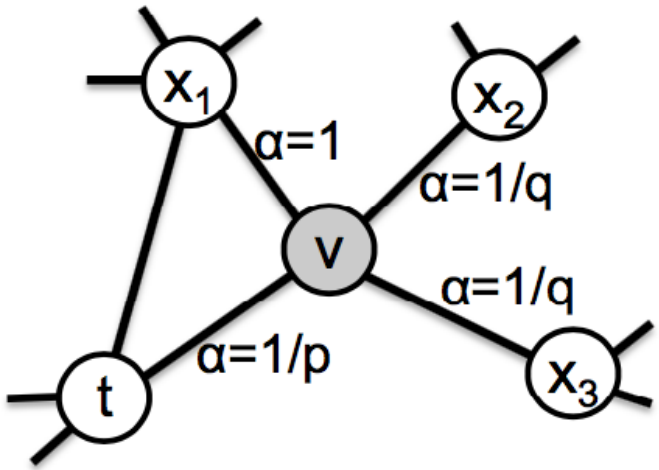
where π_{vx} is the unnormalized transition probability between v and x, and Z is a normalizing factor

To capture both structural equivalence and homophily, node2vec uses a neighborhood sampling strategy which

- is based on a flexible biased random walk procedure
- allows it to smoothly interpolate between BFS and DFS

Random Walks of node2vec

The random walk shown below just traversed edge (t, v) and now resides at node v



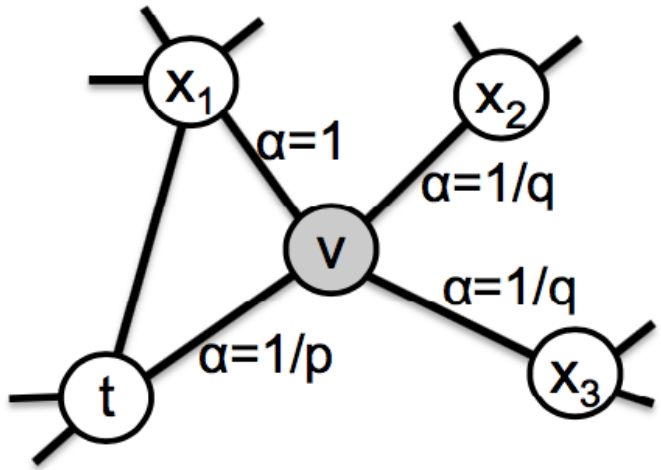
The unnormalized transition probability is $\pi_{vx} = w_{vx}\alpha_{pq}(t, x)$, where:

$$\alpha_{pq}(t, x) = \begin{cases} \frac{1}{p} & \text{if } d_{tx} = 0 \\ 1 & \text{if } d_{tx} = 1 \\ \frac{1}{q} & \text{if } d_{tx} = 2 \end{cases}$$

where d_{tx} denotes the shortest path distance between t and x

Random Walks of node2vec

The random walk shown below just traversed edge (t, v) and now resides at node v

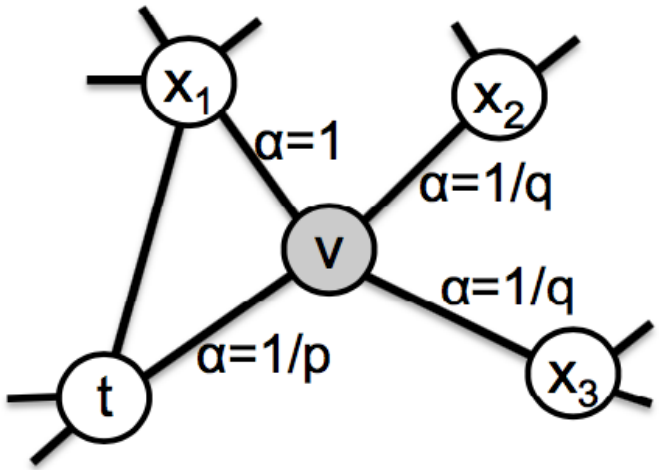


The *return parameter* p controls the likelihood of immediately revisiting a node in the walk

- if p is high, we are less likely to sample an already-visited node in the following two steps
- if p is low, it would keep the walk in the local neighborhood of the starting node

Random Walks of node2vec

The random walk shown below just traversed edge (t, v) and now resides at node v



The *in-out parameter q* allows the search to differentiate between "inward" and "outward" nodes.

- if q is high, the random walk is biased towards nodes close to node t
- if q is low, the walk is more inclined to visit nodes which are further away from the node t

Optimization

After defining the neighborhood $\mathcal{N}(v) \subset V$ of each node v, node2vec uses the Skipgram architecture:

$$minimize_f \quad -\sum_{v \in V} \log \prod_{u \in \mathcal{N}(v)} P(u|f(v))$$

where conditional likelihood is modelled as a softmax unit parametrized by a dot product of their features:

$$P(u|f(v)) = \frac{e^{f'(u)^\top f(v)}}{\sum_{k=1}^{|V|} e^{f'(v_k)^\top f(v)}}$$

and $f'(u) \in \mathbb{R}^d$ is the representation of node u when considered as context

The objective function thus becomes:

$$minimize_{f,f'} \quad -\sum_{v \in V} \left(-\log \sum_{u \in V} e^{f'(u)^\top f(v)} + \sum_{u \in \mathcal{N}(v)} f'(u)^\top f(v) \right)$$

Since learning the above posterior distribution is very expensive, node2vec approximates it using negative sampling

THANK YOU !

Acknowledgements
Dr. I. Nikolentzos, Dr. A. Tixier, Dr. P. Meladianos

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

Relevant Tutorial: Machine Learning on Graphs with Kernels@ CIKM 2019, http://www.cikm2019.net/tutorials.html