Machine and Deep learning for Graphs - an introduction

M. Vazirgiannis

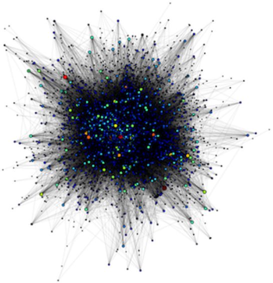
Data Science and Mining Team (DASCIM), LIX
École Polytechnique
and AUEB
http://www.lix.polytechnique.fr/dascim
Google Scholar: https://bit.ly/2rwmvQU
Twitter: @mvazirg

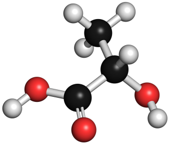
November, 2020

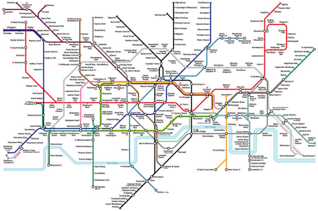
Outline

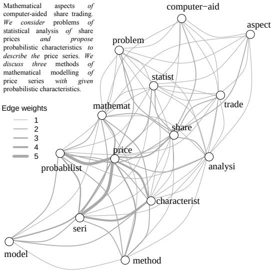
1. Intro to graphs - ML for graphs tasks
2. Graph Kernels
3. Deep Learning for Graphs - Node Embeddings

Graphs Are Everywhere







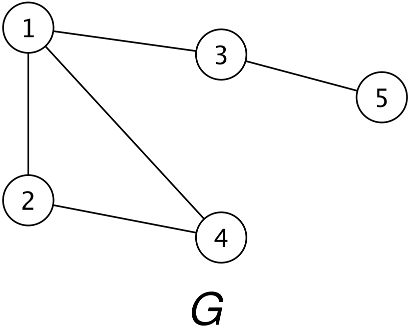


Mathematical aspects of computer-aided share trading. *We consider problems of* statistical analysis *of* share prices *and propose* probabilistic characteristics *to describe the* price series. *We discuss three* methods *of* mathematical modelling *of* price series *with given* probabilistic characteristics.

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

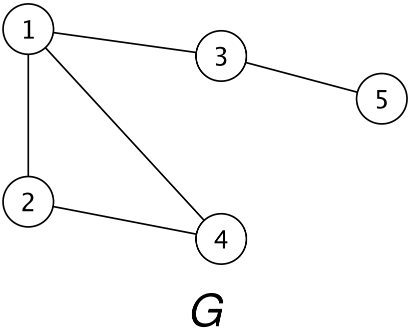


$V = \{1, 2, 3, 4, 5\}$

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

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

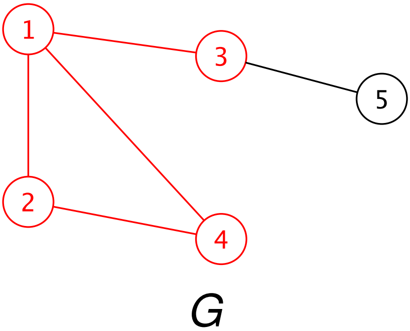


$\mathcal{N}(1) = \{2, 3, 4\}$

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

Graph Preliminaries

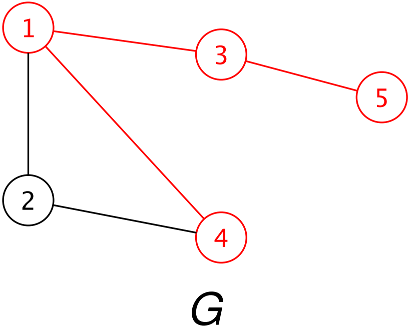
A walk in a graph G is a sequence of vertices $v_1, v_2, \dots, v_{k+1}$ where $v_i \in V$ and $(v_i, v_{i+1}) \in E$ for $1 \leq i \leq k$



Walk: 1 → 2 → 4 → 1 → 3

Graph Preliminaries

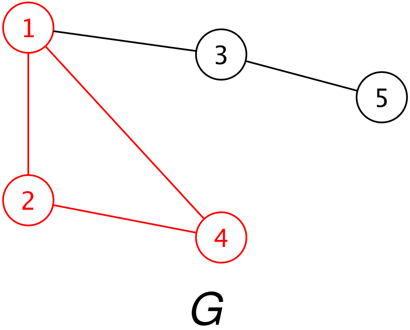
A walk in which $v_i \neq v_j \Leftrightarrow i \neq j$ is called a path



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

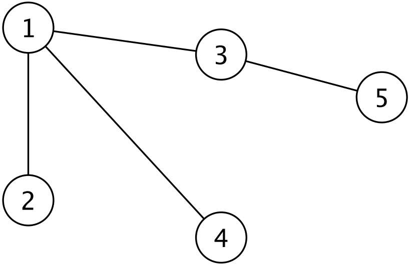
Graph Preliminaries

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



Cycle: 1 → 2 → 4

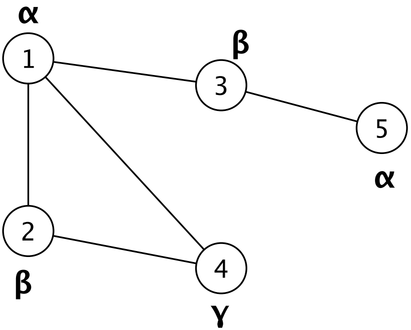
A subtree is an acyclic subgraph in which there is a path between any two vertices



G

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



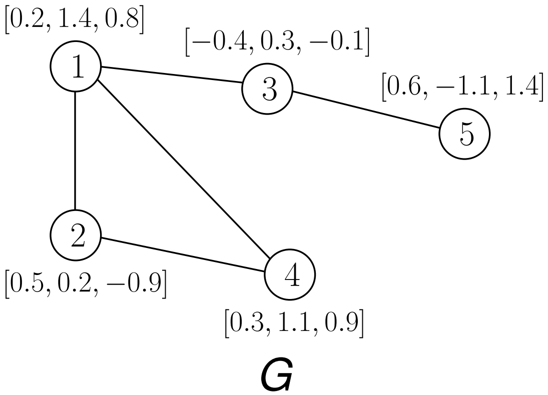
G

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

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

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



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

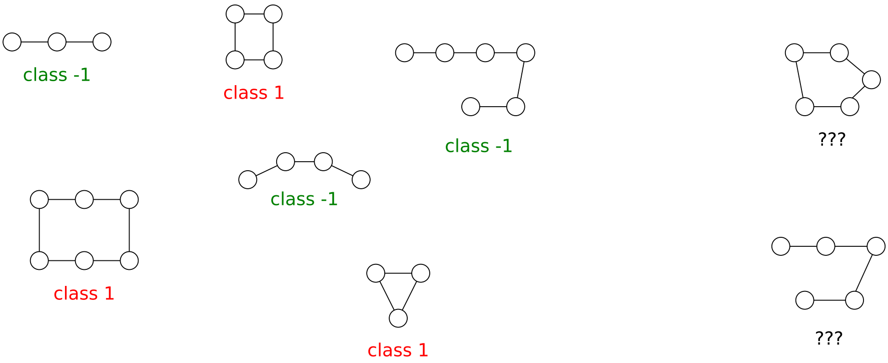
$h_1 = [0.2, 1.4, 0.8]^\top \quad h_3 = [-0.4, 0.3, -0.1]^\top$

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



- 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

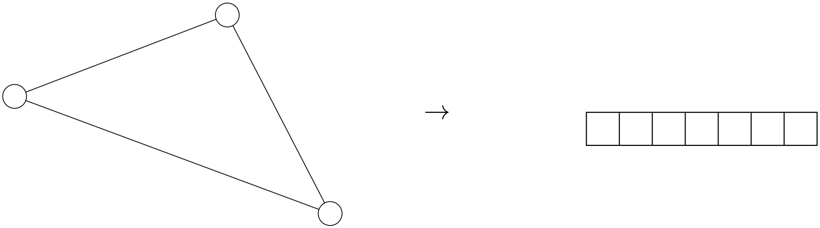
Many problems related to it are **NP-complete**

- 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

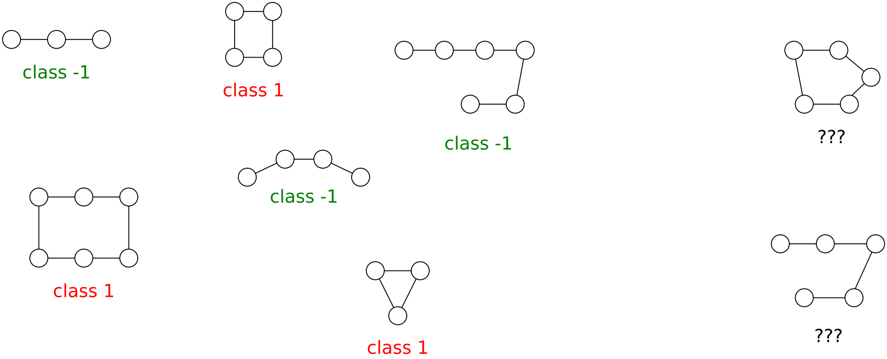
Outline

1 Intro to graphs - ML for graphs tasks

2 Graph Kernels

3 Deep Learning for Graphs - Node Embeddings

Graph Classification

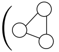


- Input data $x \in \mathcal{X}$
- Output $y \in \{-1, 1\}$
- Training set $\mathcal{S} = \{(x_1, y_1), \dots, (x_n, y_n)\}$
- Goal: estimate a function $f : \mathcal{X} \to \mathbb{R}$ to predict y from f(x)

Graph Comparison

Graph classification very related to graph comparison

Example

$f($  , $)$

$$+$$

$$k{-}nn$$

$$=$$

graph classification

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

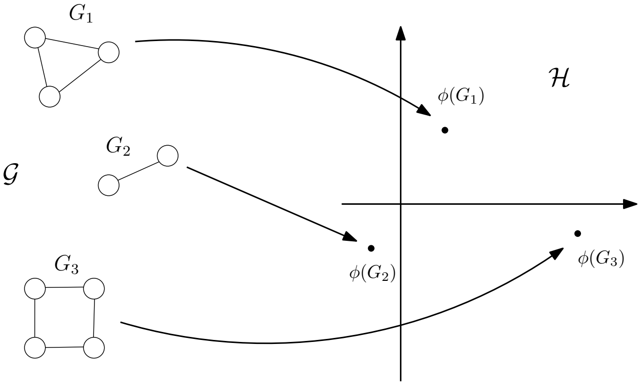
We are interested in algorithms capable of measuring the similarity between two graphs in **polynomial** time

Graph Kernels

Definition (Graph Kernel)

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

- It is equivalent to an inner product of the embeddings $\phi : \mathcal{X} \rightarrow \mathbb{H}$ 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

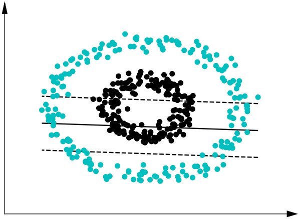
- Let $\phi(G_1), \phi(G_2)$ be vector representations of graphs G_1, G_2 in a very high (possibly infinite) dimensional feature space

- 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

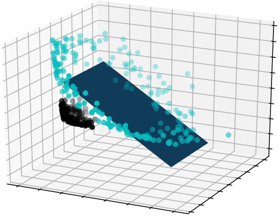
Example

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
$\phi : \mathbb{R}^2 \rightarrow \mathbb{R}^3$ defined as:

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



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

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

Applications

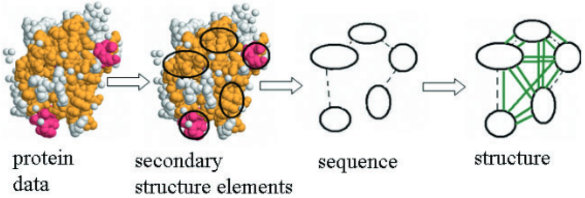
- Bioinformatics [Borgwardt et al., Bioinformatics 21(suppl_1); Borgwardt et al., PSB'07; Sato et al., BMC bioinformatics 9(1)]
- Chemoinformatics [Swamidass et al., Bioinformatics 21(suppl_1); Ralaivola et al., Neural Networks 18(8); Mahé et al., JCIM 45(4); Ceroni et al., Bioinformatics 23(16); Mahé 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]

⋮

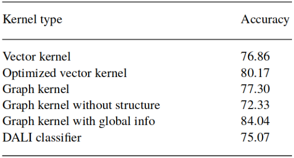
Protein Function Prediction

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

- structure
- sequence
- chemical properties



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

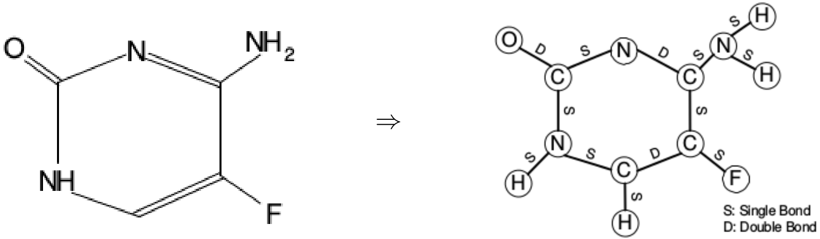


Kernel type	Accuracy
Vector kernel	76.86
Optimized vector kernel	80.17
Graph kernel	77.30
Graph kernel without structure	72.33
Graph kernel with global info	84.04
DALI classifier	75.07

[Borgwardt et al., Bioinformatics 21(suppl 1)]

Chemical Compound Classification

Represent each chemical compound as a graph



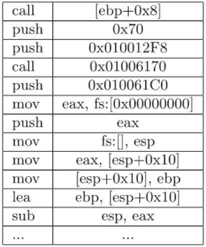
Perform **graph classification** to predict if a chemical compound displays the desired behavior against the specific biomolecular target or not

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%

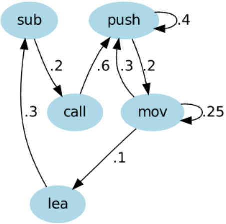
[Mahé et al., JCIM 45(4)]

Malware Detection

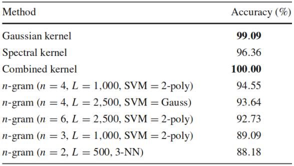
Given a computer program, create its control flow graph



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



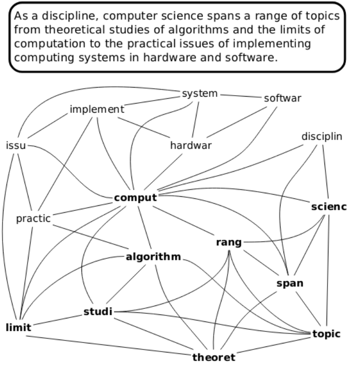
Method	Accuracy (%)
Gaussian kernel	**99.09**
Spectral kernel	96.36
Combined kernel	**100.00**
n-gram (n = 4, L = 1,000, SVM = 2-poly)	94.55
n-gram (n = 4, L = 2,500, SVM = Gauss)	93.64
n-gram (n = 6, L = 2,500, SVM = 2-poly)	92.73
n-gram (n = 3, L = 1,000, SVM = 2-poly)	89.09
n-gram (n = 2, L = 500, 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 → unique terms
- edges → co-occurrences within a fixed-size sliding window
- no edge weight
- no edge direction



As a discipline, computer science spans a range of topics from theoretical studies of algorithms and the limits of computation to the practical issues of implementing computing systems in hardware and software.

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

$\vdots$

These kernels are instance of the R-convolution framework

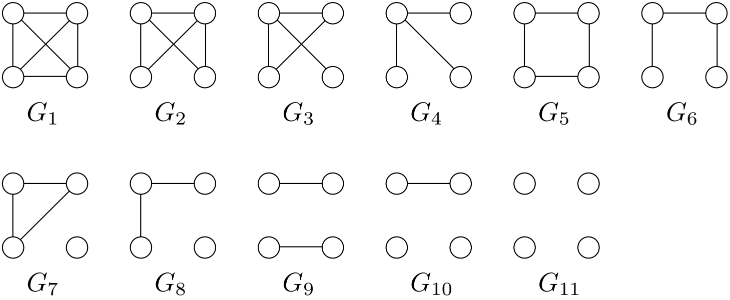
Graphlet Kernel

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

The graphlet kernel is defined as:

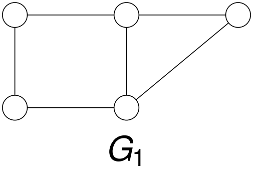
$$k(G_1, G_2) = \langle f_{G_1}, f_{G_2} \rangle$$

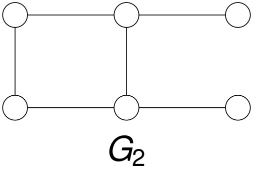
Problems:

- 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





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

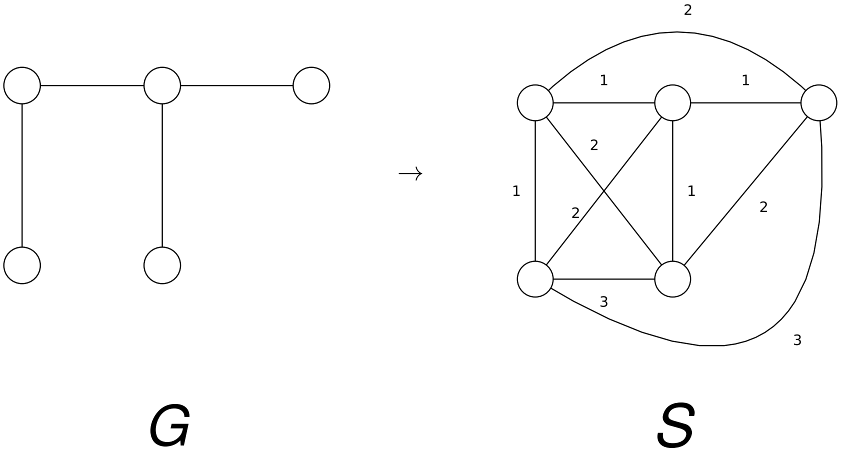
Transforms the original graphs into shortest-paths graphs

- 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]

Example

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

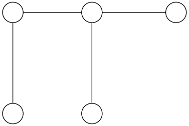
- For labeled graphs, it can be:

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

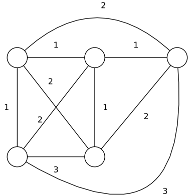
  where e^1, e^2 are the two endpoints of e

Example

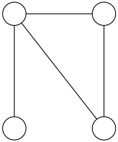
Floyd-transformations



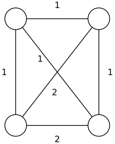
$\Rightarrow$



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

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

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]

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]

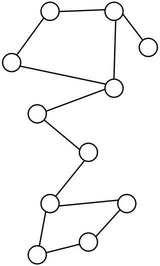
Outline

1. Intro to graphs - ML for graphs tasks

2. Graph Kernels

3. Deep Learning for Graphs - Node Embeddings

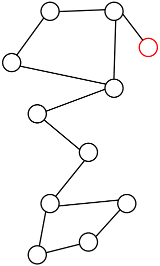
Traditional Node Representation
Representation: row of adjacency matrix



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

Deep Learning for Graphs - Node Embeddings

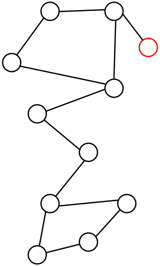
Traditional Node Representation
Representation: row of adjacency matrix



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

Deep Learning for Graphs - Node Embeddings

Traditional Node Representation
Representation: row of adjacency matrix



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

However, such a representation suffers from:

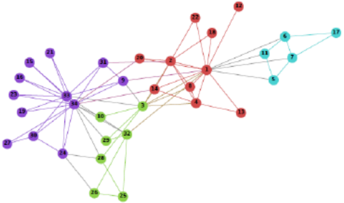
- data sparsity
- high dimensionality

$\vdots$

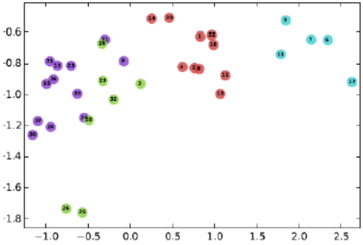
Node Embedding Methods

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

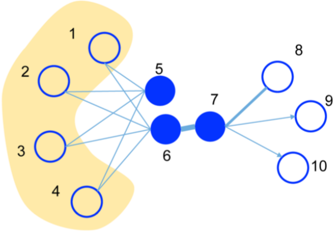
Most methods belong to the following groups:

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

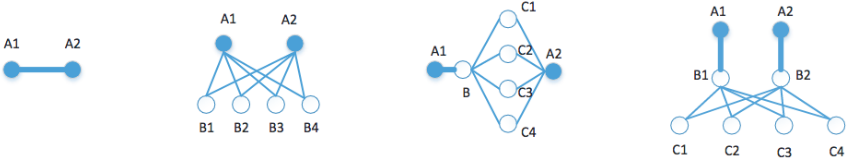
Second-order proximity: shared neighborhood structures



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

Proximities

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

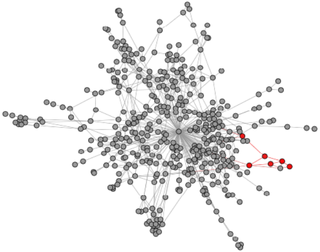


- Second-order and high-order proximities capture similarity between vertices with similar structural roles

- Higher-order proximities capture more global structure

DeepWalk

Inspired by recent advances in language modeling [1]



→

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

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

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

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

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

$\vdots$

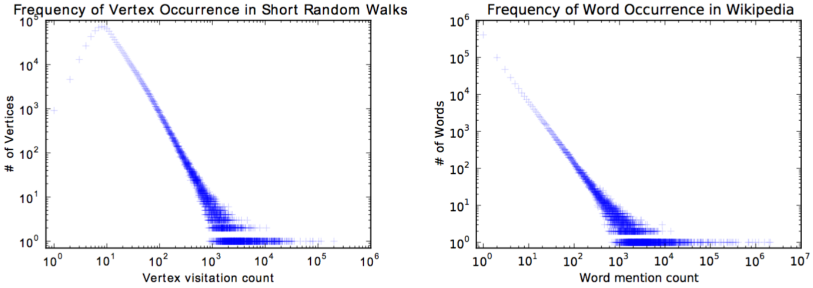
- Simulates a series of short random walks

[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

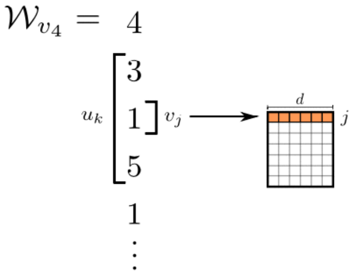
[2] Perozzi et al. DeepWalk: Online Learning of Social Representations. In KDD'14

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:



- 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:

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

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

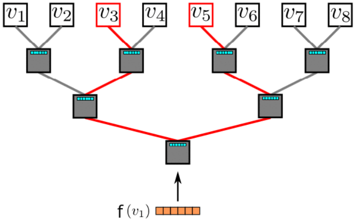
$$minimize_f \quad -\frac{1}{T}\sum_{i=1}^{T}\sum_{\substack{j=i-w\\ j\neq i}}^{i+w}\log P(v_j|f(v_i))$$

- 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

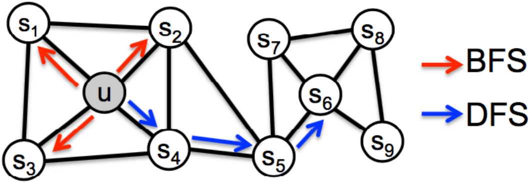
Conversely, node2vec simulates a family of biased random walks which

- 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

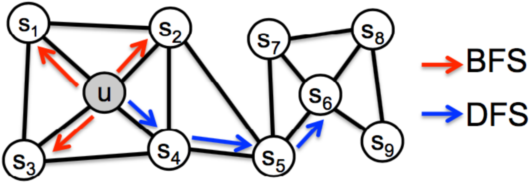
The *breadth-first sampling* (BFS) and *depth-first sampling* (DFS) represent extreme scenarios in terms of the search space



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

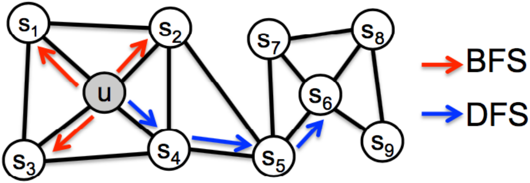


In most applications, we are interested in two kinds of similarities between vertices:

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:

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

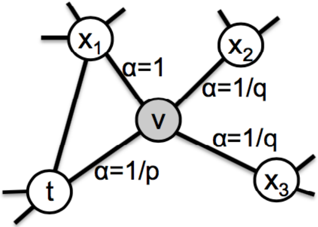
where π_{vx} is the unnormalized transition probability between v and x, and Z is a normalizing factor

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



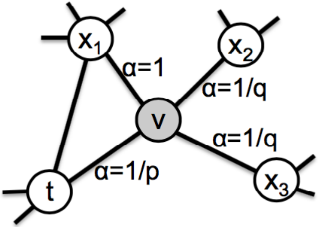
The unnormalized transition probability is $\pi_{vx} = w_{vx}\alpha_{pq}(t, x)$, where:

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

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*

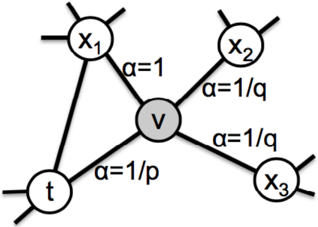


The *return parameter p* controls the likelihood of immediately revisiting a node in the walk

- 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

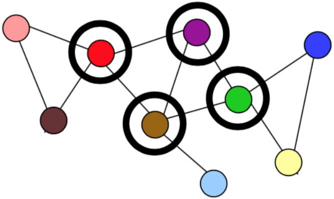
The objective function thus becomes:

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

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

Structural Identity

- Nodes in networks have specific roles
    - e. g., individuals, web pages, proteins, etc
- Structural identity
    - identification of nodes based on network structure (no other attribute)
    - often related to role played by node
- Automorphism: strong structural equivalence



Red, Green: structurally identical
Purple, Brown: structurally similar

struc2vec

- Learns node representations based on structural identity
  - structurally similar nodes close in space

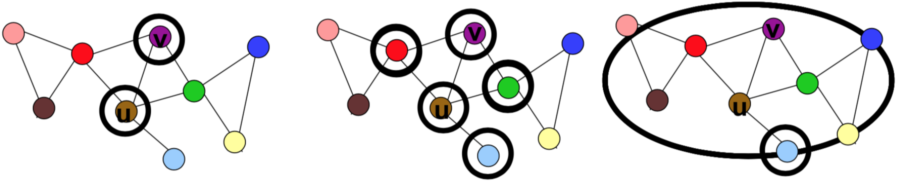
Key ideas:

- Structural similarity does not depend on hop distance
  - neighbor nodes can be different, far away nodes can be similar
- Structural identity as a hierarchical concept
  - depth of similarity varies
- Flexible four step procedure
  - operational aspect of steps are flexible

[1] Ribeiro et al. struc2vec: Learning Node Representations from Structural Identity. In KDD'17

Step 1: Structural Similarity

- Hierarchical measure for structural similarity between two nodes
- $R_k(v)$: set of nodes at distance k from v (ring)
- $s(S)$: ordered degree sequence of set S



$s(R_0(u)) = 4$ $s(R_1(u)) = 1, 3, 4, 4$ $s(R_2(u)) = 2, 2, 2, 2$

$s(R_0(v)) = 3$ $s(R_1(v)) = 4, 4, 4$ $s(R_2(v)) = 1, 2, 2, 2, 2$

Step 1: Structural Similarity

- $g(D_1, D_2)$: distance between two ordered sequences
  - cost of pairwise alignment: ${}^{\max(a,b)}/_{\min(a,b)} - 1$
  - optimal alignment by Dynamic Time Warping in our framework

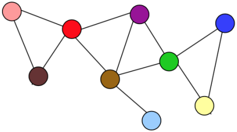
$s(R_0(u)) = 4$	$s(R_1(u)) = 1, 3, 4, 4$	$s(R_2(u)) = 2, 2, 2, 2$
$s(R_0(v)) = 3$	$s(R_1(v)) = 4, 4, 4$	$s(R_2(v)) = 1, 2, 2, 2, 2$
$g(\cdot, \cdot) = 0.33$	$g(\cdot, \cdot) = 3.33$	$g(\cdot, \cdot) = 1$

- $f_k(v, u)$: structural distance between nodes v and u considering first k rings
  - $f_k(v, u) = f_{k-1}(v, u) + g(s(R_k(v)), s(R_k(u)))$

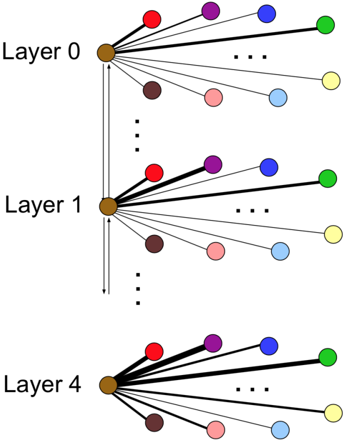
$f_0(v, u) = 0.33$ $\quad$ $f_1(v, u) = 3.66$ $\quad$ $f_2(v, u) = 4.66$

Step 2: Multi-layer graph

- Encodes structural similarity between all node pairs



- Each layer is a weighted complete graph
  - corresponds to similarity hierarchies
- Edge weights in layer k
  - $w_k(v,u) = e^{-f_k(v,u)}$
- Connect corresponding nodes in adjacent layers



Step 3: Generate Context

- Context generated by biased random walk
  - walking on multi-layer graph

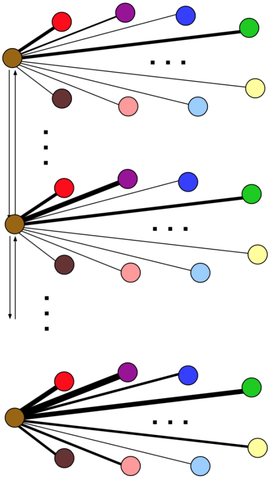
- Walk in current layer with probability p
  - choose neighbor according to edge weight
  - RW prefers more similar nodes

- Change layer with probability $1 - p$
  - choose up/down according to edge weight
  - RW prefers layer with less similar neighbors

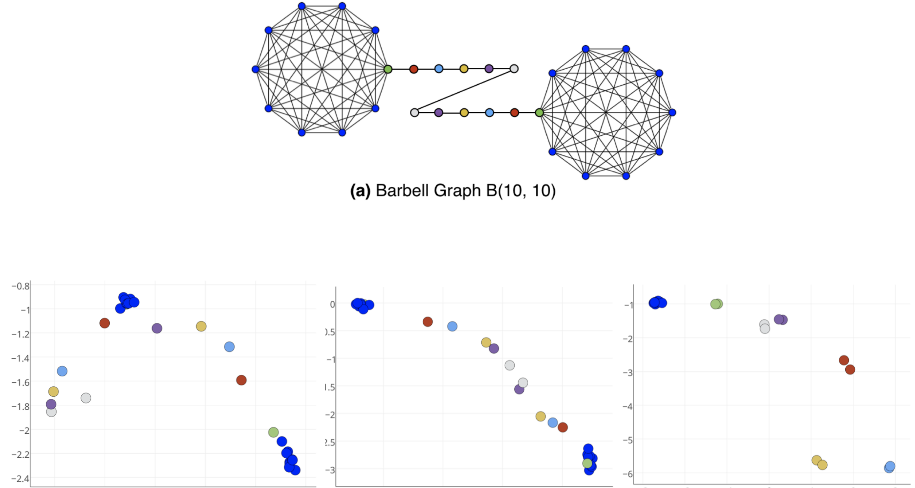
Step 3: Learn Representation

- For each node, generate set of independent and relative short random walks
  - context for node → sentences of a language

- Train a neural network to learn latent representation for nodes
  - maximize probability of nodes within context
  - Skip-gram (Hierarchical Softmax) adopted



Barbell Network



(a) Barbell Graph B(10, 10)

(c) *DeepWalk*

(d) *node2vec*

(e) *struc2vec*

- struc2vec embeds isomorphic nodes very close to each other in space

GCN

Given the adjacency matrix $\mathbf{A}$ of a graph, GCN first computes:

$$\hat{\mathbf{A}} = \tilde{\mathbf{D}}^{-\frac{1}{2}} \, \tilde{\mathbf{A}} \, \tilde{\mathbf{D}}^{-\frac{1}{2}}$$

where
$\tilde{\mathbf{A}} = \mathbf{A} + \mathbf{I}$
$\tilde{\mathbf{D}}$: a diagonal matrix such that $\tilde{\mathbf{D}}_{ii} = \sum_j \tilde{\mathbf{A}}_{ij}$

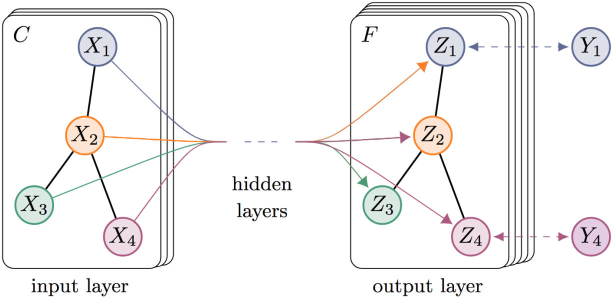
Then, the output of the model is:

$$\mathbf{Z} = \mathit{softmax}(\hat{\mathbf{A}} \; \mathit{ReLU}(\hat{\mathbf{A}} \; \mathbf{X} \; \mathbf{W}^0) \; \mathbf{W}^1)$$

where
$\mathbf{X}$: matrix whose rows contain the attributes of the nodes
$\mathbf{W}^0, \mathbf{W}^1$: trainable weight matrices

[1] Kipf and Welling. Semi-supervised Classification with Graph Convolutional Networks. In ICLR'17

GCN



To learn node embeddings, GCN minimizes the following loss function:

$$\mathcal{L} = -\sum_{i \in I} \sum_{j=1}^{|\mathcal{C}|} \mathbf{Y}_{ij} \log \hat{\mathbf{Y}}_{ij}$$

I: indices of the nodes of the training set
$\mathcal{C}$: set of class labels

Experimental Evaluation

Experimental comparison conducted in [1]

Compared algorithms:

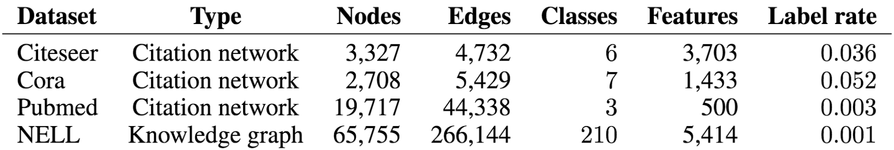
- DeepWalk
- ICA [2]
- Planetoid
- GCN

Task: node classification

[1] Kipf and Welling. Semi-supervised Classification with Graph Convolutional Networks. In ICLR'17

[2] Lu and Getoor. Link-based classification. In ICML'03

Datasets



Dataset	Type	Nodes	Edges	Classes	Features	Label rate
Citeseer	Citation network	3,327	4,732	6	3,703	0.036
Cora	Citation network	2,708	5,429	7	1,433	0.052
Pubmed	Citation network	19,717	44,338	3	500	0.003
NELL	Knowledge graph	65,755	266,144	210	5,414	0.001

Label rate: number of labeled nodes that are used for training divided by the total number of nodes

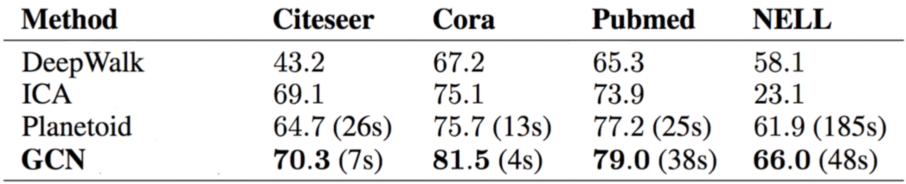
Citation network datasets:

- nodes are documents and edges are citation links
- each node has an attribute (the bag-of-words representation of its abstract)

NELL is a bipartite graph dataset extracted from a knowledge graph

Results

Classification accuracies of the 4 methods



Method	**Citeseer**	**Cora**	**Pubmed**	**NELL**
DeepWalk	43.2	67.2	65.3	58.1
ICA	69.1	75.1	73.9	23.1
Planetoid	64.7 (26s)	75.7 (13s)	77.2 (25s)	61.9 (185s)
GCN	**70.3** (7s)	**81.5** (4s)	**79.0** (38s)	**66.0** (48s)

Observation: DeepWalk → unsupervised learning of embeddings

↪ fails to compete against the supervised approaches

THANK YOU !

Acknowledgements
Dr. I. Nikolentzos

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

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