Apprentissage sur graphes

GraphML

Raphaël Fournier-S'niehotta

CNAM Paris, fournier@cnam.fr

IP Paris
CES-IA
2023-2024

le cnam

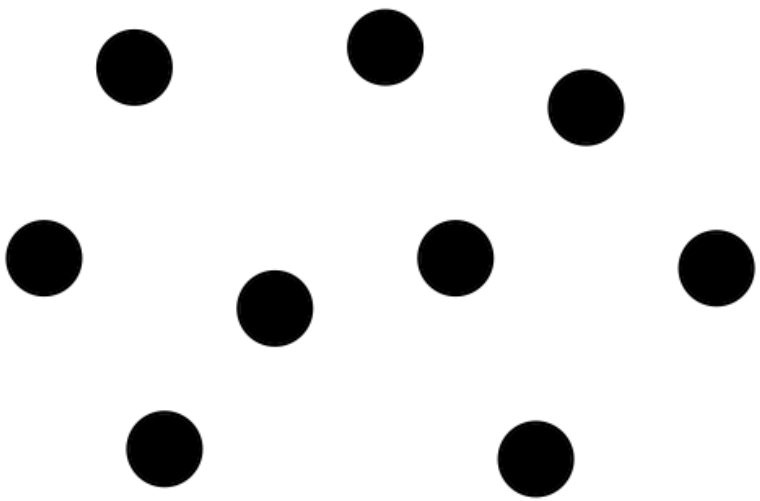
Plan

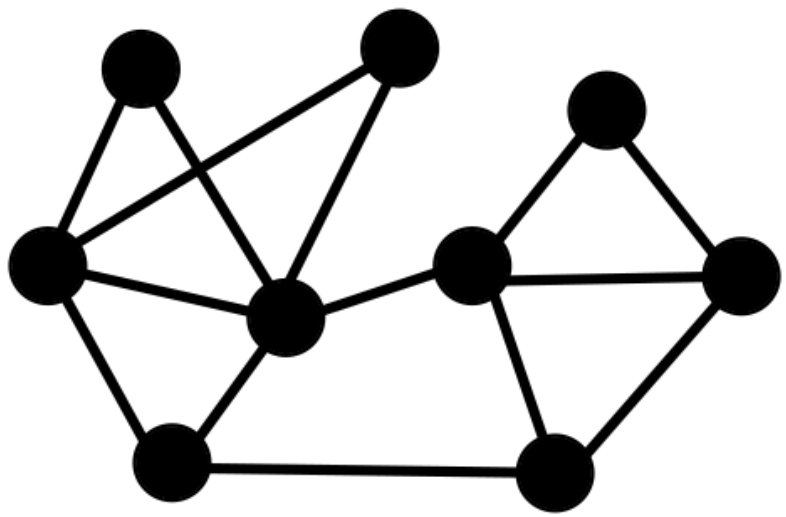
Introduction

Plan

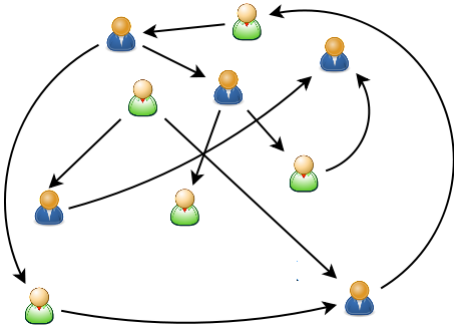
Les graphes

- Les graphes proposent un **langage général** pour décrire des **entités** qui ont des **interactions**



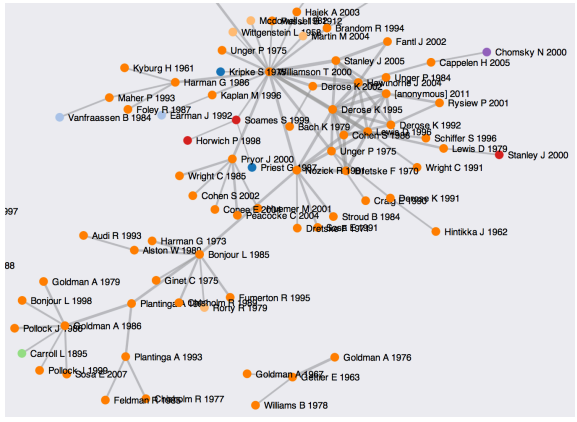


Types de graphes

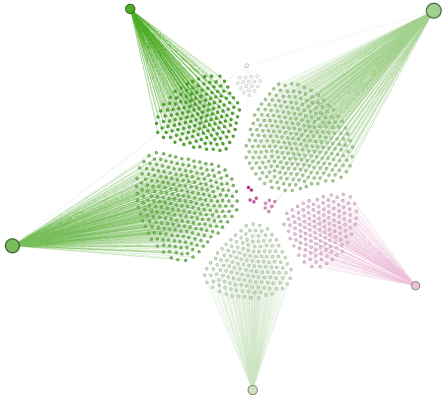


Réseau social

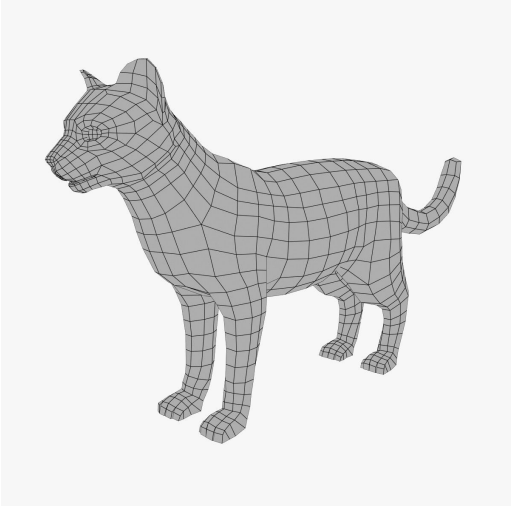
Réseau de transport



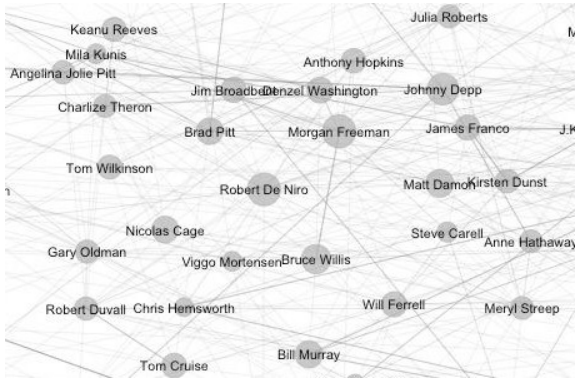
Réseau de citations



Réseau de communications



Formes 3D



Réseau de films

Domaines

- Réseaux sociaux
- Réseaux informatiques
- Communications
- Bio-informatique / médecine
- Cerveau

- Graphes de connaissances
- Dépendances logicielles
- Formes 3D

Objectif

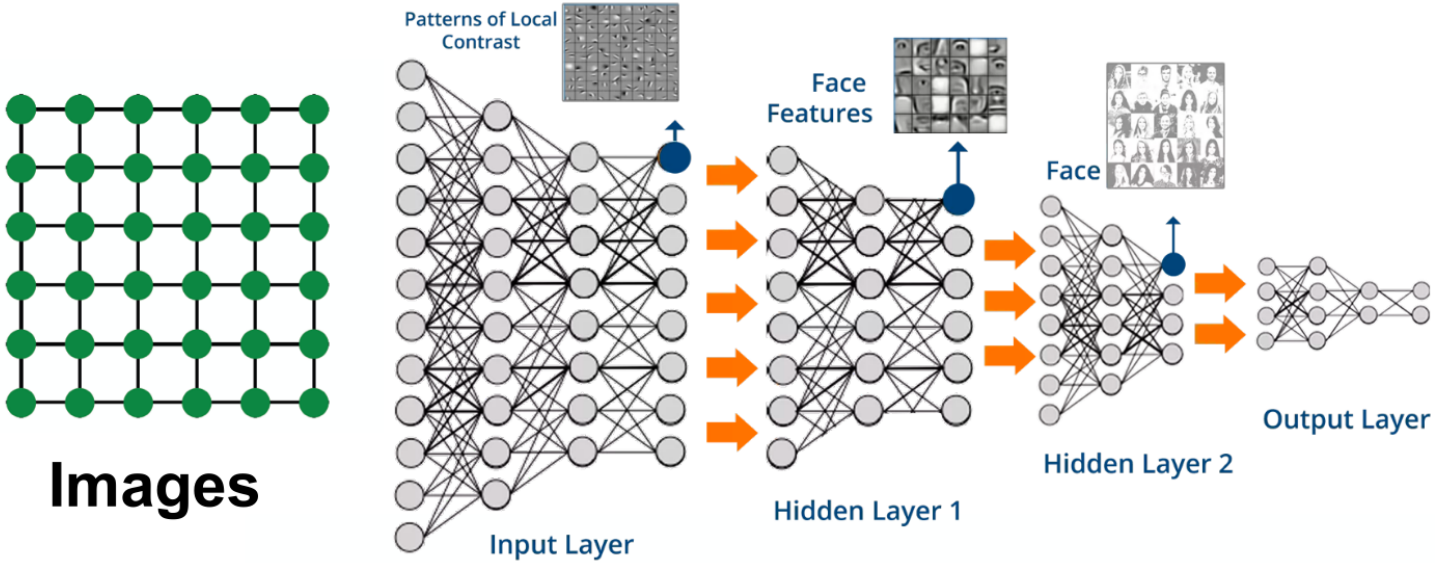
Tirer partie de la structure (relations) du graphe,
pour améliorer les prédictions.

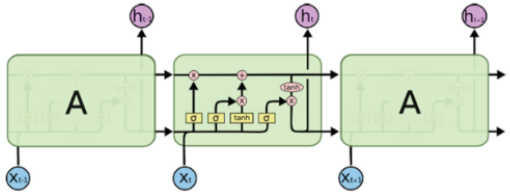
Peut-on développer des réseaux de neurones
pour ces structures ?

Plan

Motivations, challenges

Machine learning classique :





Challenges :

- Complexité topologique : pas de régularité locale (grille)
- Pas d'ordre évident pour les nœuds, pas de points de référence
- Souvent dynamique

Tâches en Graph ML

- Classification de nœuds
- Prédiction de liens
- Classification de graphes
- Clustering

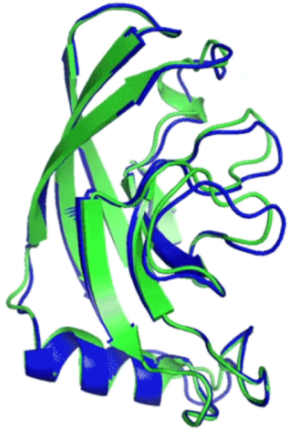
- Génération de graphes
- Évolution de graphes

Réussites

Tâches sur les nœuds

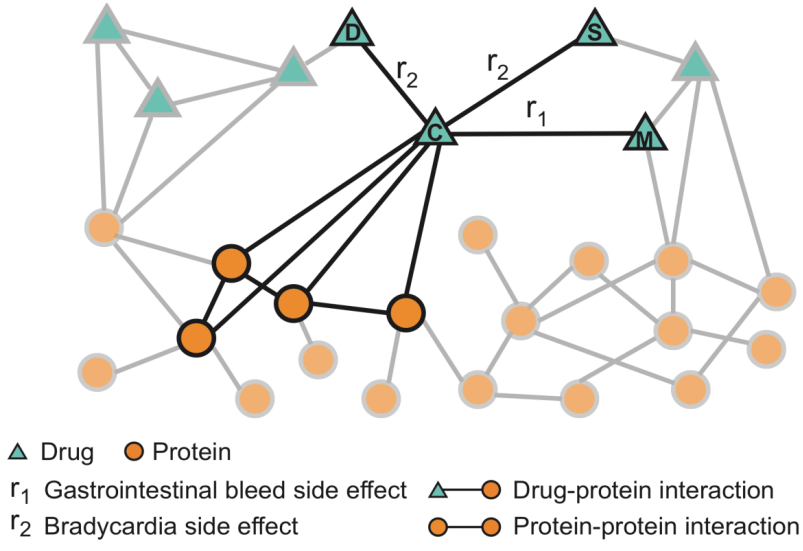
- Protein-folding (Alpha-Fold, Google)

T1037 / 6vr4
90.7 GDT
(RNA polymerase domain)



T1049 / 6y4f
93.3 GDT
(adhesin tip)

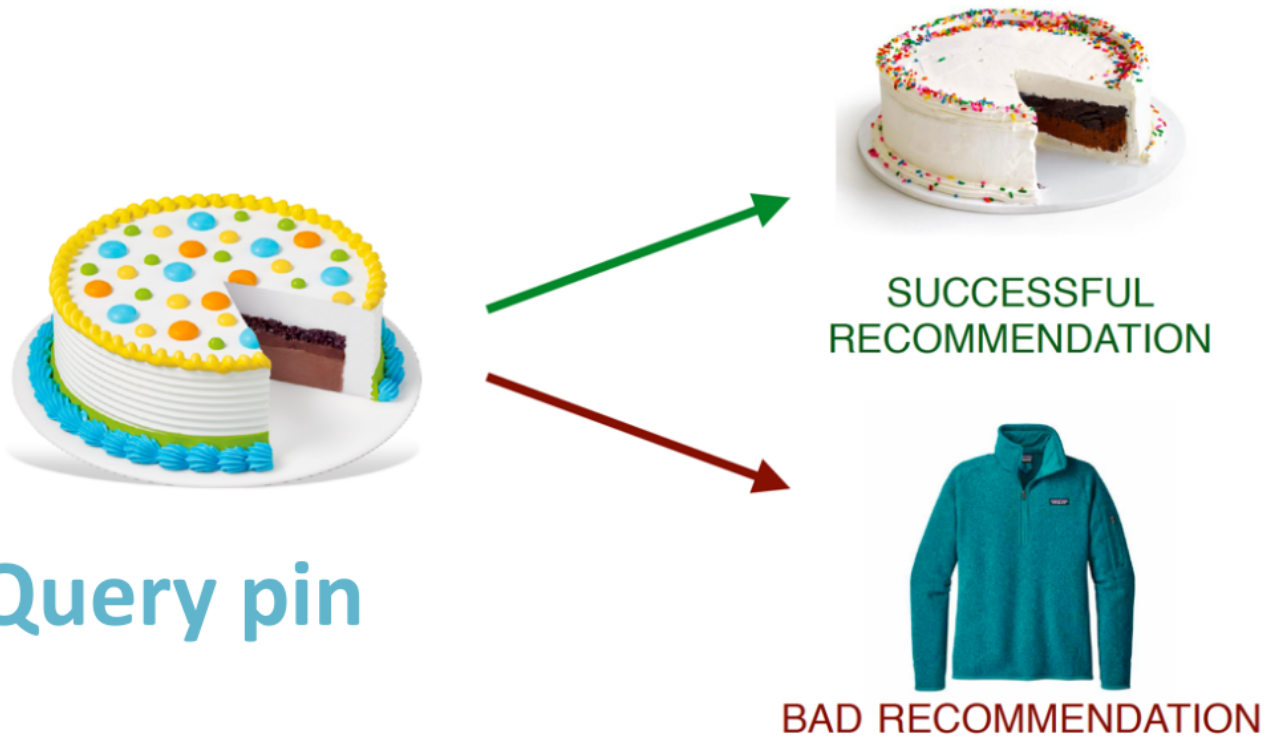
- Effets secondaires



Réussites : RecSys

Tâche sur les arêtes :

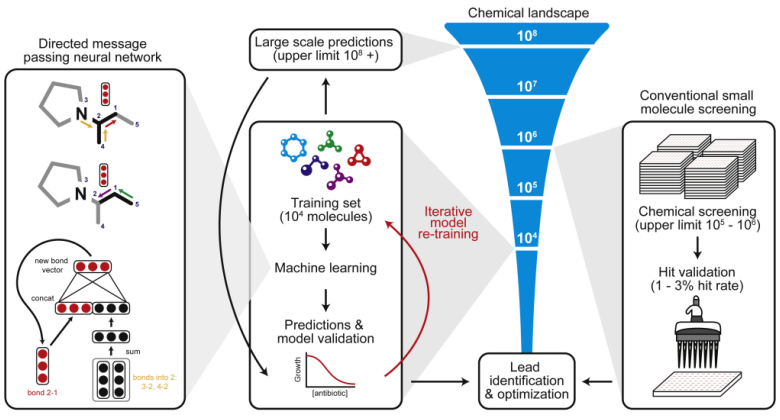
- Recommandation d'images chez Pinterest
- Très large échelle
- Proposer des images similaires à une autre
- Features multiples



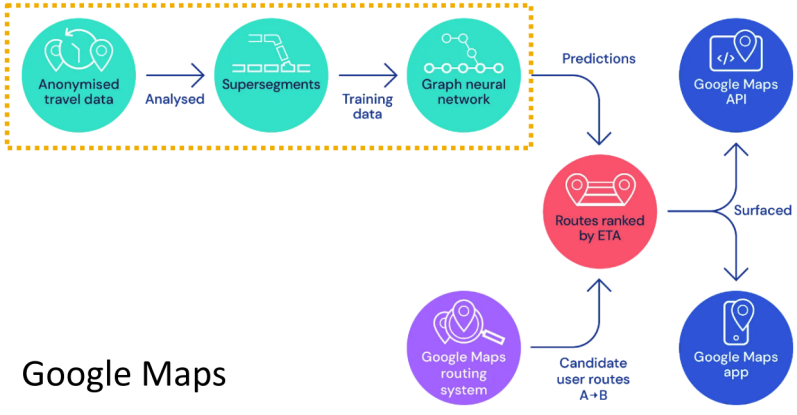
Réussites

Tâches sur les (sous-)graphes

- Sélection de molécule pour antibiotiques



- prédiction de trafic routier



Google Maps

Plan

Terminologie

Un graphe est défini par un couple G = (V,E) tel que :

- V (pour l'anglais *vertices*) est un ensemble fini de sommets
- E (pour l'anglais *edges*) est un ensemble fini d'arêtes

Un graphe peut être orienté, ou non :

- si oui, les couples $(v_i, v_j) \in E$ sont ordonnés, v_i est le sommet initial, v_j est le sommet terminal.
- on appelle alors le couple (v_i, v_j) un *arc*, représenté graphiquement par $v_i \rightarrow v_j$.
- si non, les couples ne sont pas orientés et (v_i, v_j) est équivalent à (v_j, v_i), et on l'appelle *arête*, représenté par $v_i - v_j$

Terminologie

- l'**ordre** d'un graphe, c'est son nombre de sommets (souvent désigné par n).
- une **boucle** est un arc/une arête reliant un sommet à lui-même
- un graphe dépourvu de boucle est dit **élémentaire**
- un graphe **simple** ne comporte pas de boucle et au plus une arête entre deux sommets
- un graphe **partiel** est le graphe obtenu en supprimant certains arcs ou arêtes
- un **sous-graphe** est le graphe obtenu en supprimant certains sommets et tous les arcs/arêtes incidents aux sommets supprimés.
- un sommet v_i est dit **adjacent** à un autre s'il existe une arête entre eux (on parle de **voisins**).
- le **degré** d'un sommet est le nombre d'arêtes incidentes à ce sommet.
- un graphe est dit **complet** s'il comporte une arête (v_i, v_j) pour toute paire de sommets $(v_i, v_j) \in E^2$.

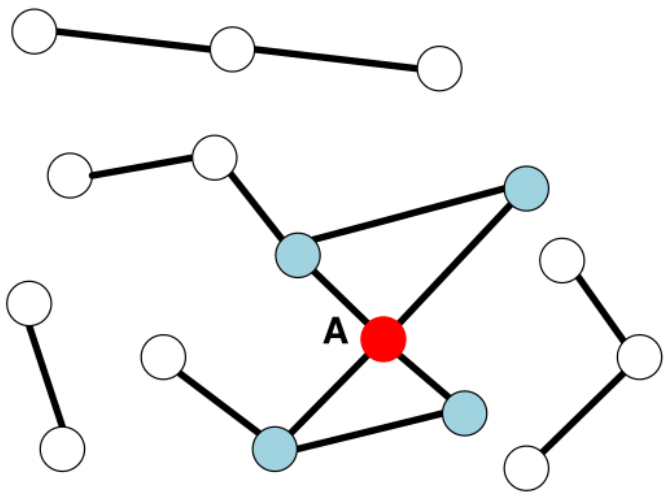
Représentation

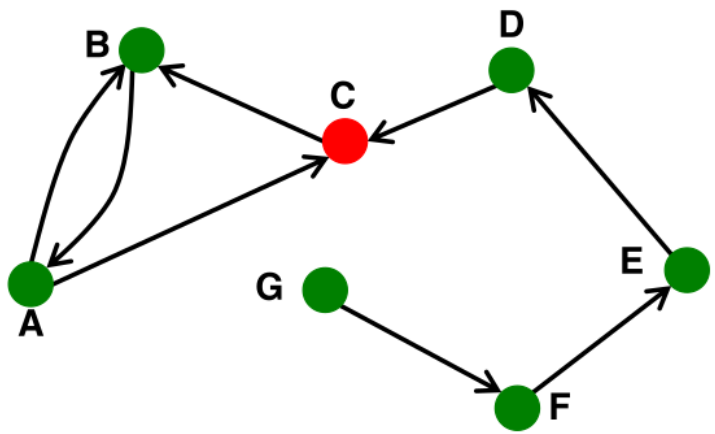
- Un graphe de gens qui travaillent ensemble, c'est un réseau professionnel
- Un graphe d'articles qui se citent, c'est un réseau de citations
- Un graphe de livres qui contiennent les mêmes mots dans leur titre : pas de nom, mais c'est un réseau...

- Importance de bien choisir nœuds et arêtes !
- Parfois, c'est évident, souvent pas
- Le choix des liens détermine la question à laquelle on répond

Degrés des nœuds

- Le degré d'un nœud, c'est le nombre de ses voisins qui lui sont directement liés
- Dans les graphes dirigés, on définit un degré entrant et un degré sortant (et le degré total est la somme des deux)

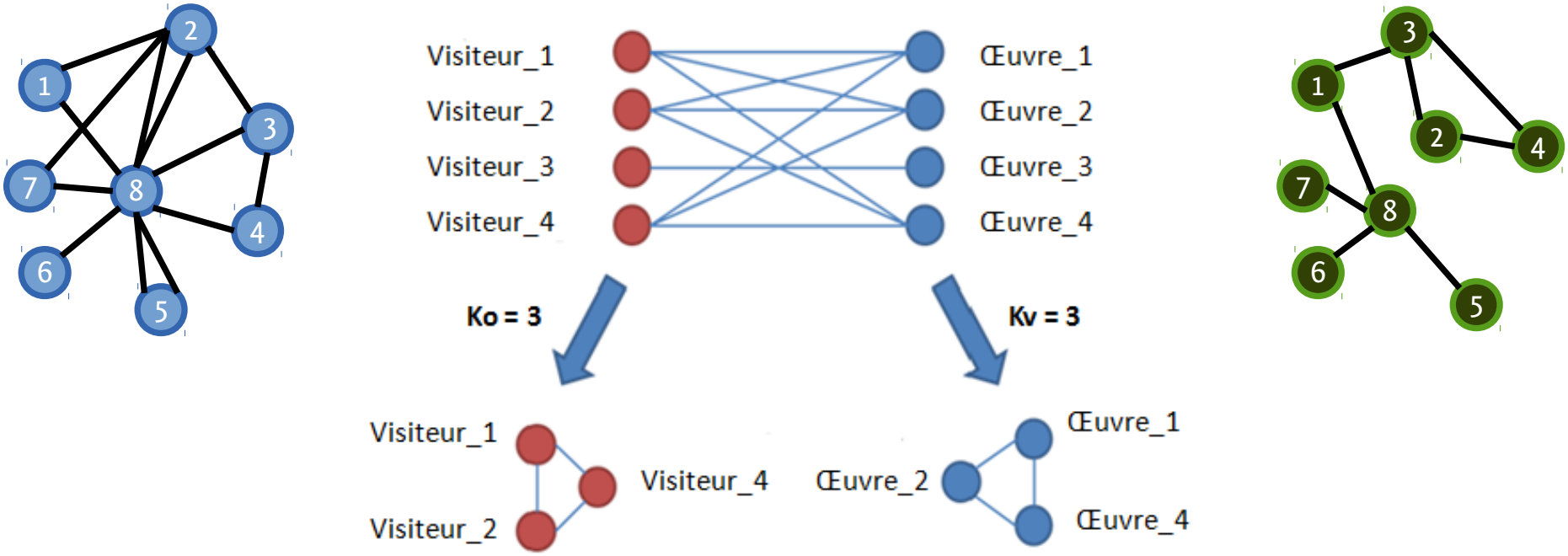




Graphe biparti

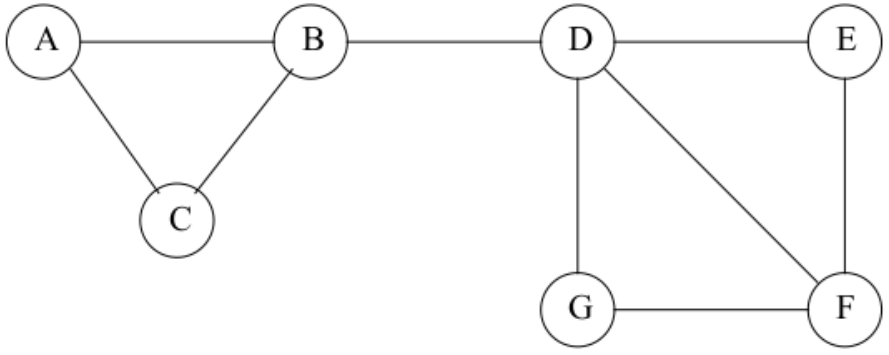
- Un graphe est dit biparti si l'on peut diviser en deux ensembles disjoints U et V ses sommets, de façon que chaque arête ait un sommet dans U et un dans V
- Exemples :
  - utilisateurs-articles (recommandation : films, titres, etc.)
  - auteurs-œuvres (recherche)
  - recettes-ingrédients
- on peut ”projeter” ces réseaux, et analyser la projection (attention au seuil)

Projection de biparti



Représentation de graphes

- Matrice d'adjacence : $m_{ij} = 1$ si l'arête (v_i, v_j) existe, 0 sinon.



$$\begin{array}{c} \\ A \\ B \end{array} \begin{array}{c} \begin{array}{ccccccc} A & B & & & & & \end{array} \\ \begin{bmatrix} 0 & 1 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 & 1 & 0 \end{bmatrix} \end{array}$$

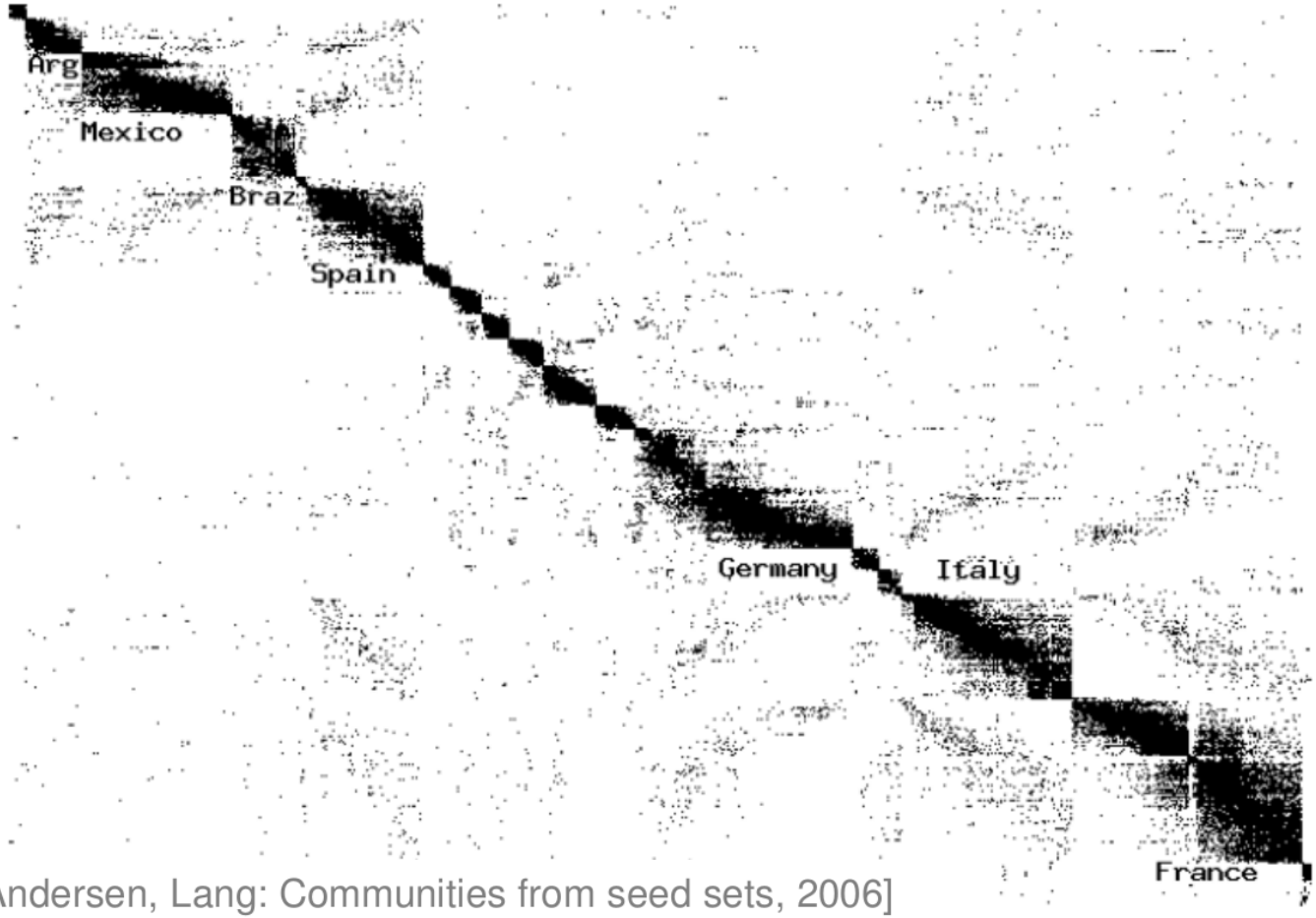
- Matrice des degrés : $m_{ij} = d(v_j)$ pour $i = j$, 0 sinon.

$$\begin{bmatrix} 2 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 3 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 2 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 4 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 2 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 3 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 2 \end{bmatrix}$$

- Matrices de Laplace (ou "laplacienne") : L = D - A (attention : nombreuses variantes)

Sparsity

Attention : les matrices d'adjacence sont souvent très creuses.

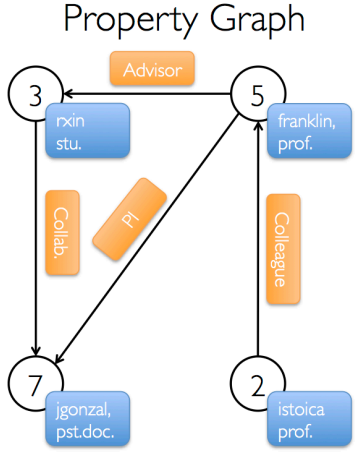


[Andersen, Lang: Communities from seed sets, 2006]

Attributs

Pour les nœuds et les liens :

- poids (fréquence de communication)
- rang (collaborateur préféré, 2e préféré)
- type (ami, collaborateur, famille)
- signé ou non : confiance, amitié/inimitié
- propriétés liées au graphe : nombre de voisins en commun



Vertex Table

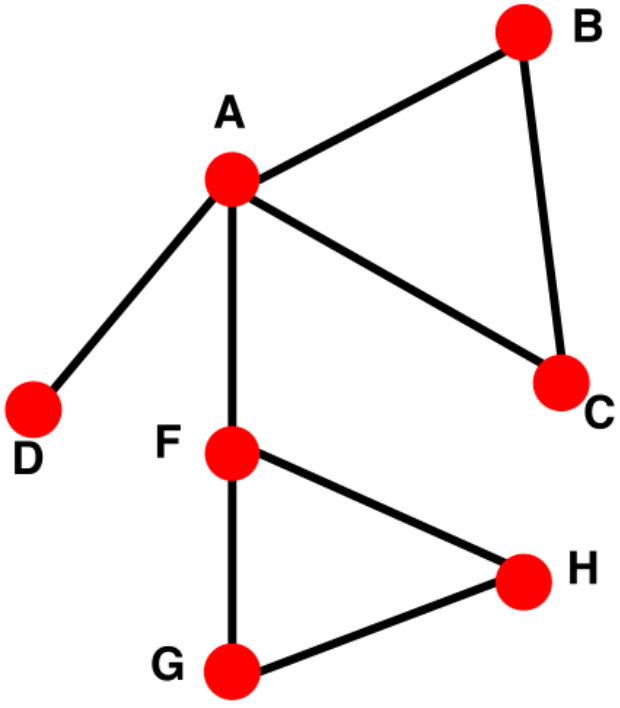
Id	Property (V)
3	(rxin, student)
7	(jgonzal, postdoc)
5	(franklin, professor)
2	(istoica, professor)

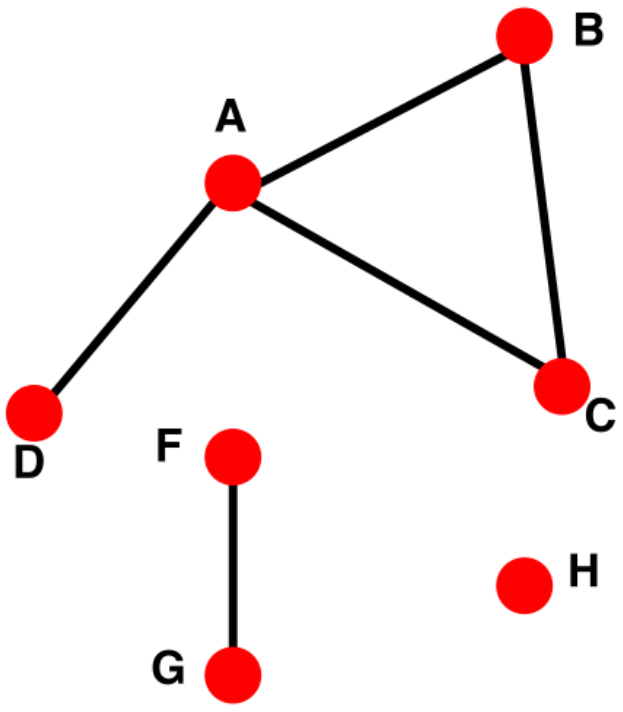
Edge Table

SrcId	DstId	Property (E)
3	7	Collaborator
5	3	Advisor
2	5	Colleague
5	7	PI

Connexité

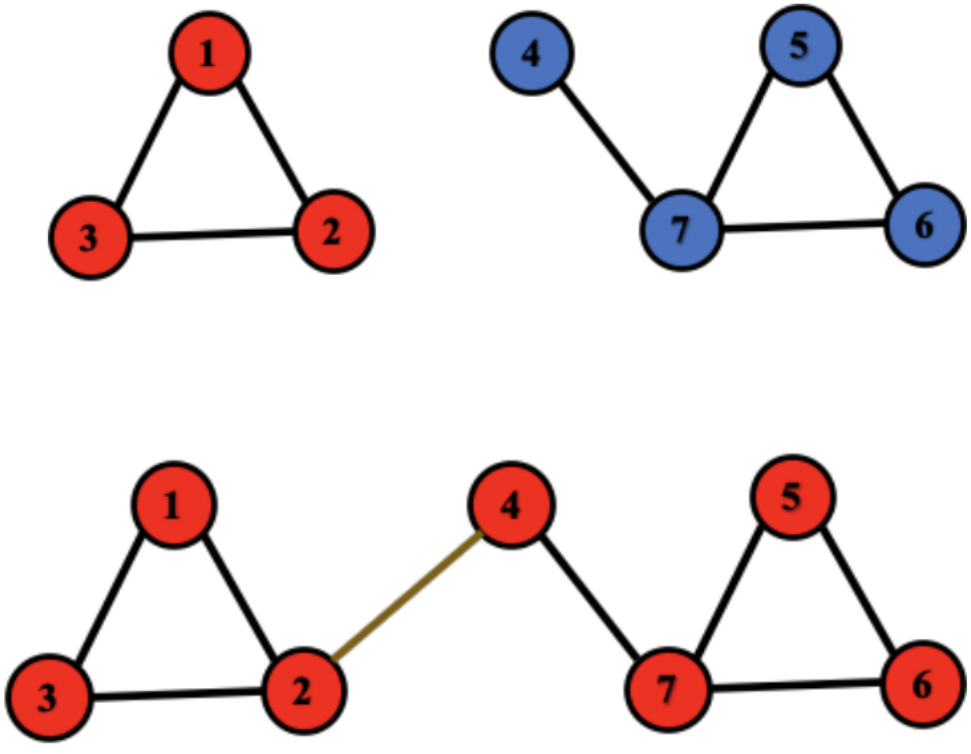
- Un graphe est connecté si deux sommets quelconque peuvent être liés par un chemin
- Un sous-graphe connecté est appelé une composante connexe
- Un graphe est déconnecté s'il a deux composantes connexes ou plus





Connexité et matrice d'adjacence

La matrice d'adjacence d'un graphe avec plusieurs composantes connexes peut être ré-écrite pour être diagonale par blocs.

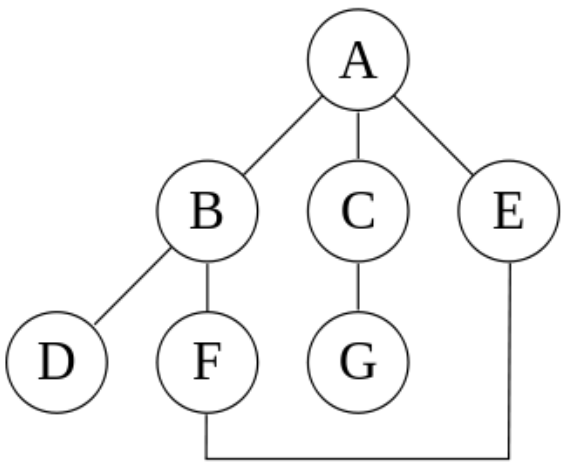


$$\begin{pmatrix} 0 & 1 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 1 & 1 & 0 \end{pmatrix}$$

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

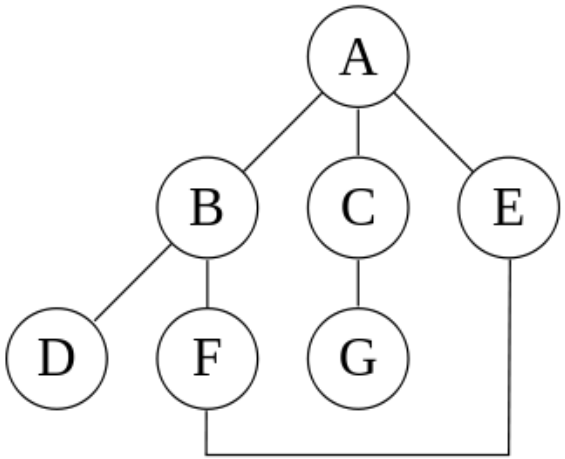
Parcours en profondeur

- En anglais, DFS, pour Depth First Search
- progresse à partir d'un sommet S en s'appelant récursivement pour chaque sommet voisin de S.
- pour chaque sommet, on prend le premier sommet voisin, on explore tous ses voisins (non marqués) avant de revenir au "père"
- Ordre de visite : A, B, D, F, E, C, G
- s'implémente avec une pile (LIFO)



Parcours en largeur

- En anglais, BFS, pour Breadth First Search
- pour chaque sommet, on repère tous ses voisins, on stocke ceux qui ne sont pas marqués dans une file (queue)
- Ordre de visite : A, B, C, E, D, F, G
- s'implémente avec une file (FIFO)
- on obtient les plus courts chemins à la racine

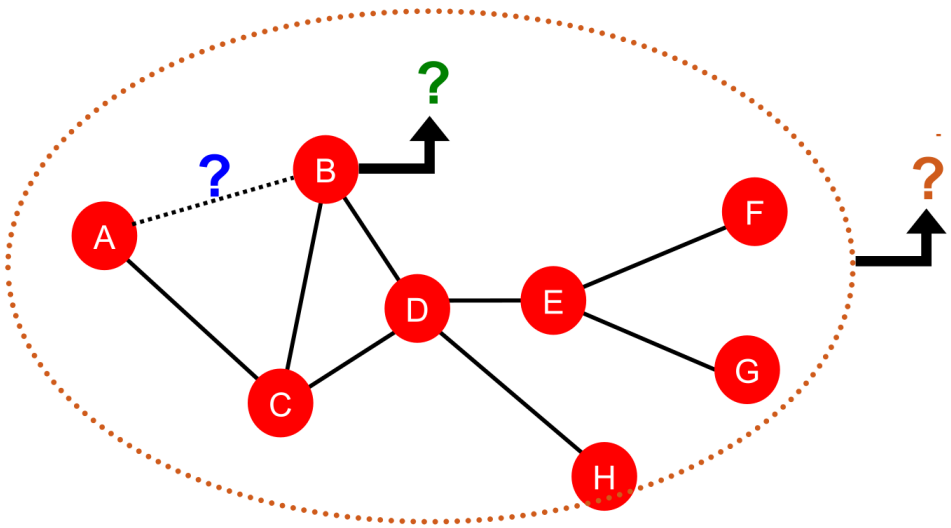


Apprentissage classique sur des graphes

Apprentissage classique sur graphes

Trois types de tâches :

- prédiction sur les nœuds
- prédiction sur les arêtes
- prédiction au niveau graphe entier



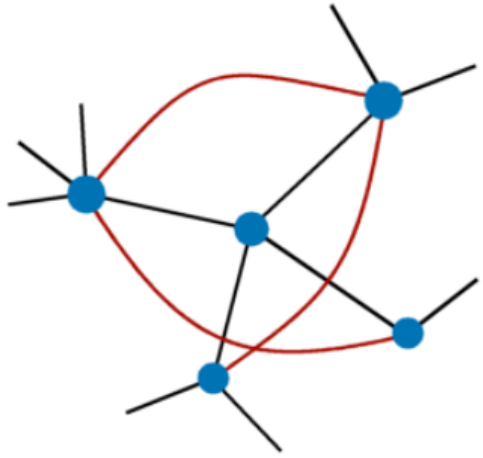
Schéma/pipeline classique

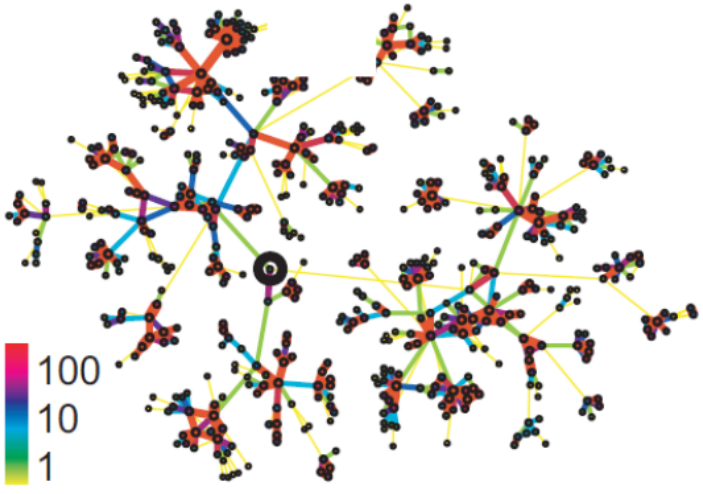
- On conçoit des caractéristiques pour les nœuds/arêtes/sommets
- On obtient les valeurs de ces caractéristiques pour toutes les données d'apprentissage
- On entraîne le modèle
- On l'applique : pour un nouveau nœud/lien/graphe, on utilise ses caractéristiques pour faire une prédiction

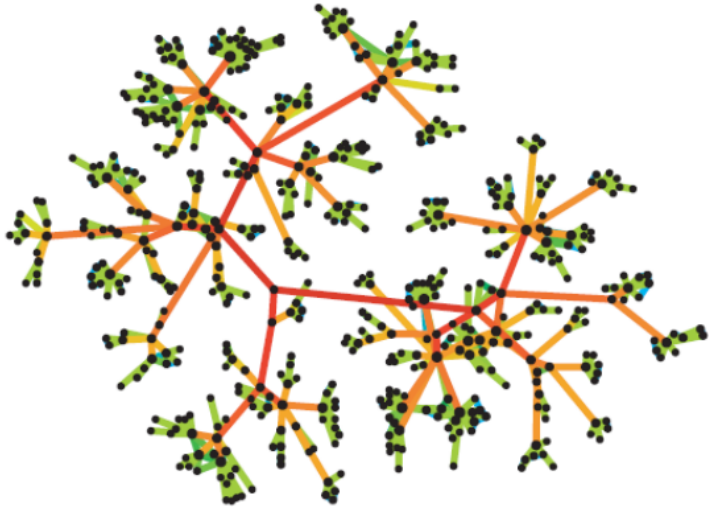
Plan

Attributs de nœuds

- degré (voisinage, sans l'importance)
- centralité (diversement calculée)
- coefficient de clustering (densité locale)
- graphlets

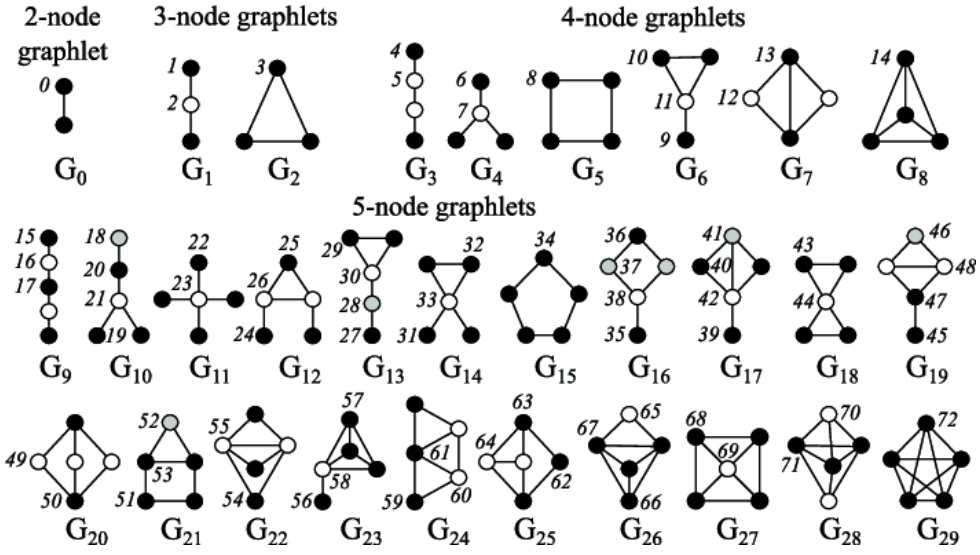




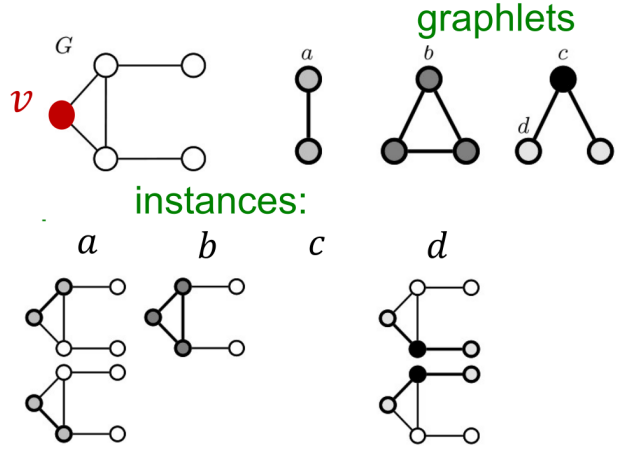


Graphlets

- sous-graphes connectés non isomorphes



- GDV de v : $[2, 1, 0, 2]$
- Graphlet Degree Vector : mesure de topologie locale (dim 73 !)



Attributs de liens

- Métrique reposant sur la distance
  - plus court chemin entre deux nœuds
- Voisinage local (cf cours Reco)
  - Common Neighbors
  - Indice de Jaccard
  - Adamic-Adar
- Structure globale
  - Indice/centralité de Katz (cf Reco)

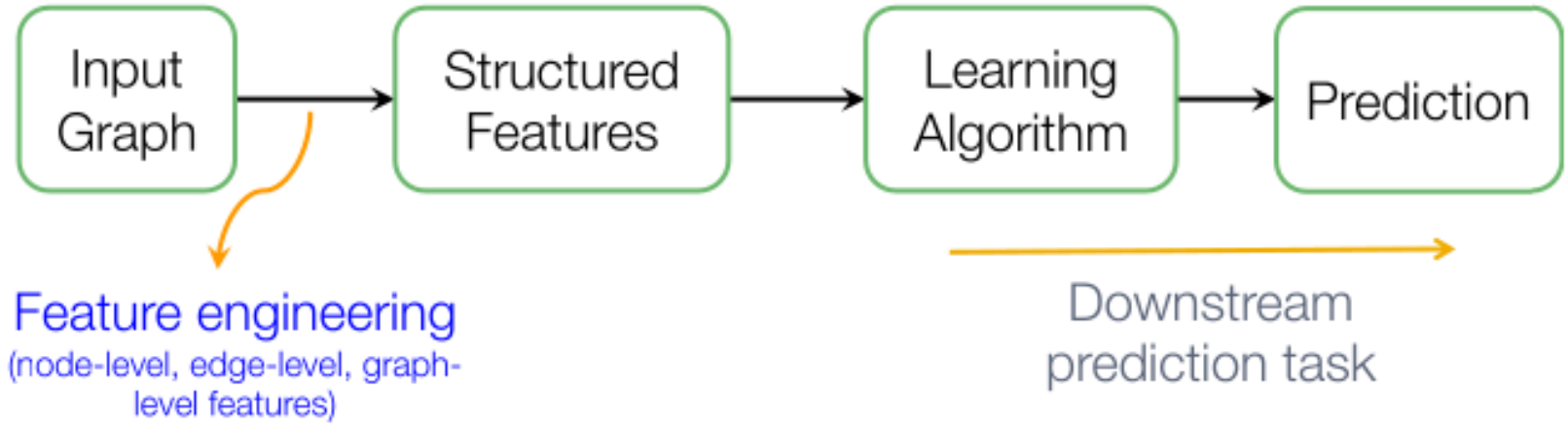
Attributs de graphes entiers

- Méthodes à noyaux.
  L'idée : trouver des noyaux plutôt que des vecteurs de caractéristiques. Une fois le noyau défini, on prend un modèle sur étagères (SVM) pour prédire
- Idée commune : généraliser les "bag-of-words" (IR/NLP) aux graphes
- Quelques modèles :
  - Noyau à Graphlets (généralisés, avec nœuds isolés)
  - Noyau de Weisfeiler-Lehman (Bag-of-degrees subtil)
  - Noyau à marche aléatoire, à plus court chemin, etc.

Résumé

Le ML classique sur les graphes consiste donc à :

- extraire les features (attributs, puis métriques) pour les nœuds, les liens, voire le graphe
- apprendre un modèle



Plan

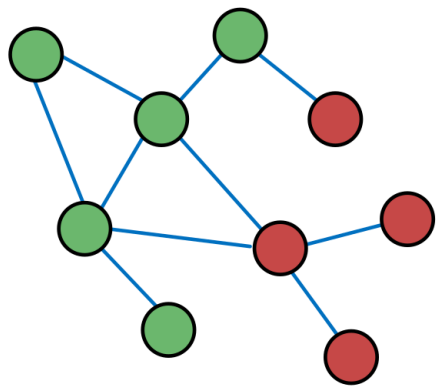
Passage de messages

Le passage de message est un framework pour faire de la classification de nœuds.

- Cela repose sur les corrélations dans le graphe : des nœuds similaires sont connectés (homophilie, cf RecSys)
- Classification collective : on cherche à affecter les étiquettes de tous les nœuds du graphe
- On suppose que le label Y_v d'un nœud v dépend des labels de ses voisins :

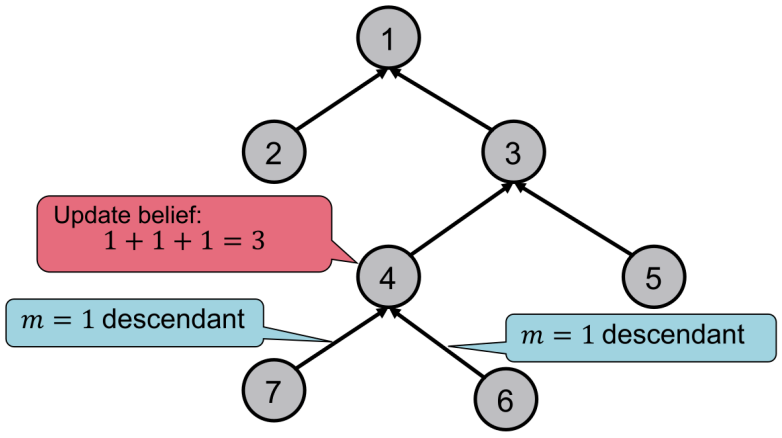
$$P(Y_v) = P(Y_v|N_v)$$

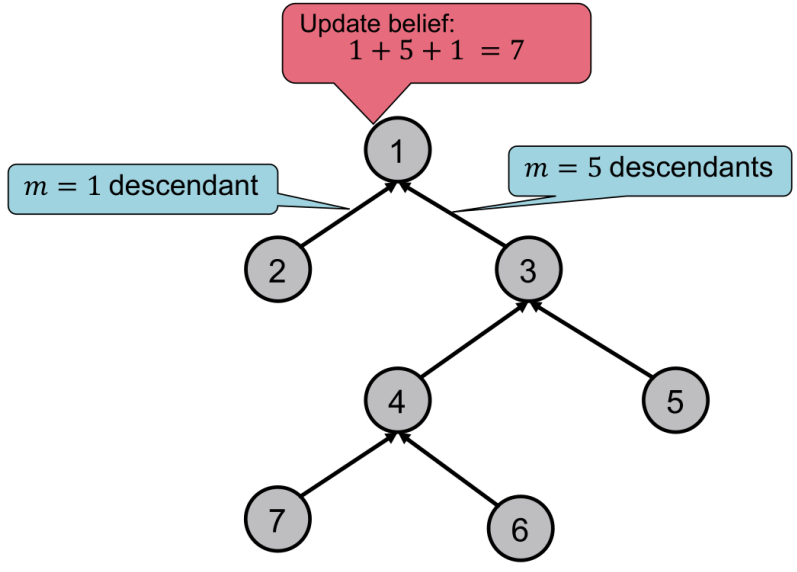
- 3 étapes :
  1. on affecte des étiquettes
  2. on capture les corrélations entre nœuds
  3. on propage ces corrélations dans le réseau



Techniques de passage de message

- Classification relationnelle
  - probabilité qu'un nœud ait une classe
  - mise-à-jour : moyenne (pondérée) du voisinage
  - risque de non convergence et pas d'usage des attributs
- Classification itérative (utilisation d'attributs)
  - résumé à partir des attributs les plus fréquents dans le voisinage
- Propagation de croyance (*belief propagation*, avec programmation dynamique)

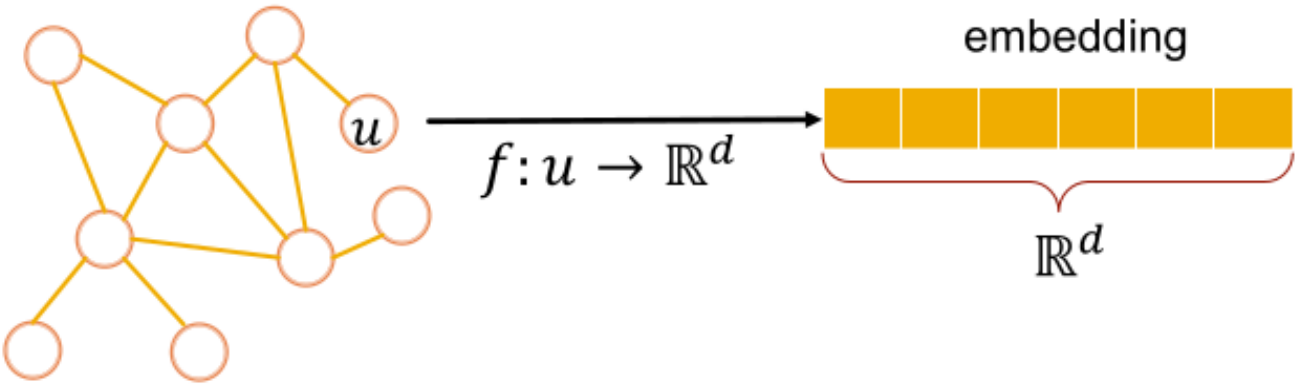




Embeddings

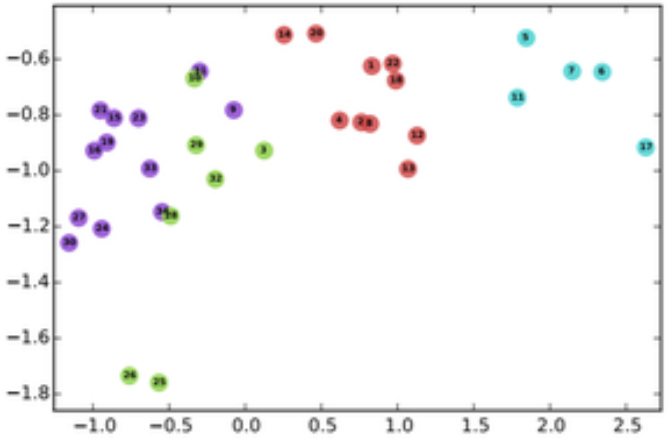
Embeddings

- Embeddings = plongements
- Objectif : apprendre des représentations indépendantes des tâches
- On veut une représentation vectorielle des nœuds
  - similarité d'embedding = similarité dans le réseau (proximité, symétrie)
  - encoder l'information structurelle



Exemple : Karaté-club

(a) Input: Karate Graph



(b) Output: Representation

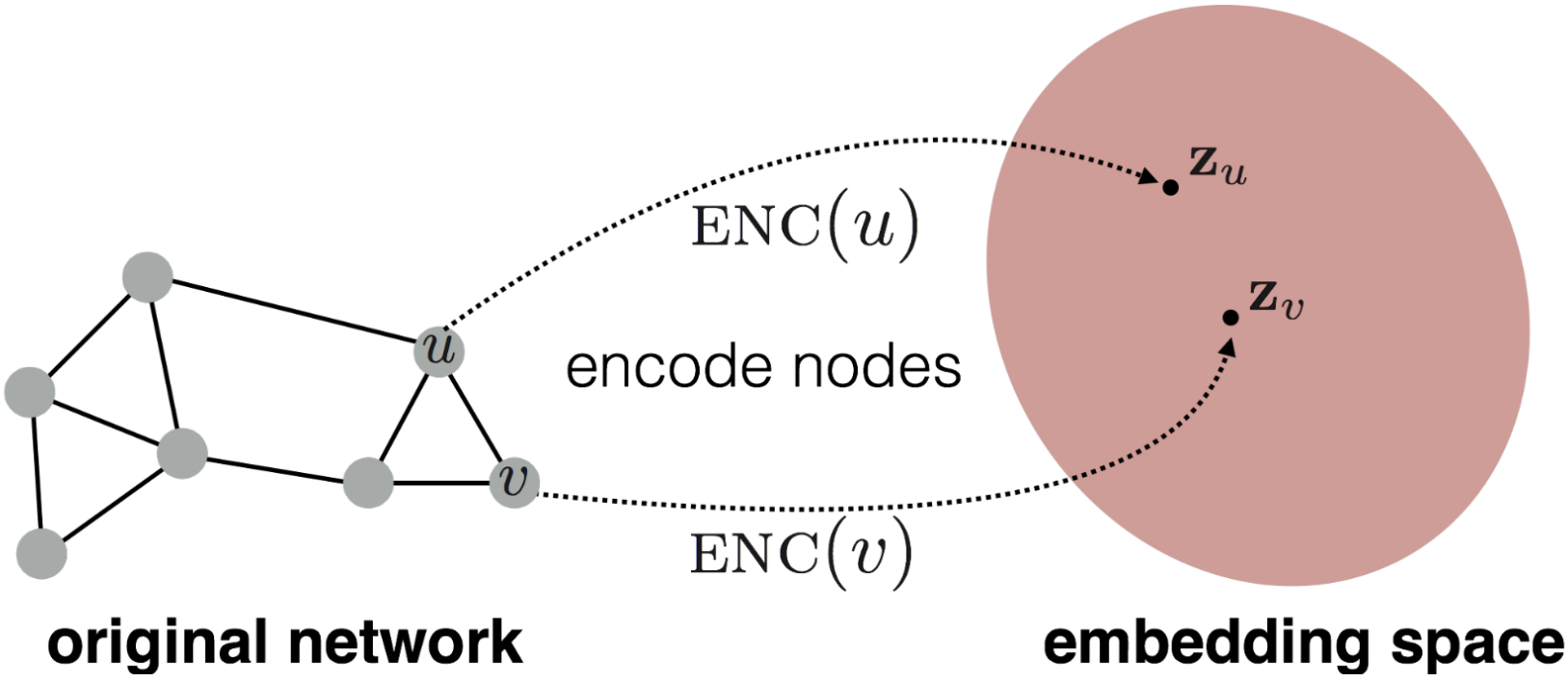
Perozzi et al, 2014. ”DeepWalk”

Objectif

On voudrait :

$$sim(u,v) \approx z_v^T z_u$$

- sim(u,v) concerne le graphe initial
- $DEC(z_v, z_u) = z_v^T z_u$ calcule la similarité d'embeddings



Apprendre les embeddings

- L'encodeur ENC fait la correspondance des nœuds vers les embeddings

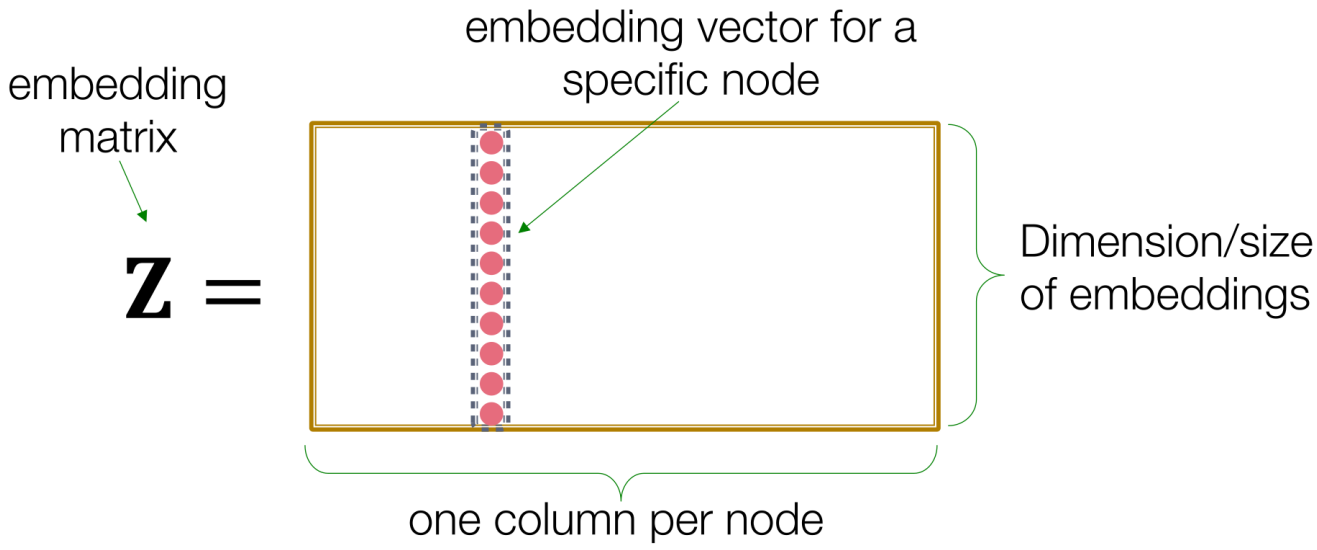
$$\text{ENC}(v) = z_v \qquad \text{(de dimension d)}$$

- On définit une similarité entre nœuds dans le réseau original
- Le décodeur DEC fait la correspondance entre embeddings et similarité
- On cherche à optimiser les paramètres (encodeur, décodeur) tels que :

$$\text{sim(u,v)} \approx DEC(z_v, z_u) = z_v^T z_u$$

Encodeur superficiel

- L'encodeur est juste un "lookup"
- $ENC(v) = z_v = Z \cdot v$ où
  - $Z \in \mathbb{R}^{d \times |V|}$ est une matrice (chaque colonne est un embedding (appris)
  - $v \in \mathbb{I}^{|V|}$ est un vecteur-indicateur, nul sauf sur la ligne correspondant à v



- En fait, on optimise/apprend "directement" les embeddings de chaque nœud (via la matrice Z).
- La fonction objectif doit maximiser $z_v^T z_u$ pour les paires de nœuds similaires

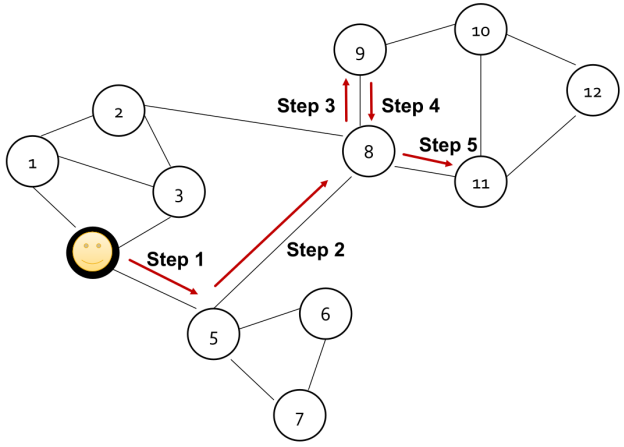
Similarités

On veut ”plonger” les nœuds dans un espace tel que les distances dans celui-ci reflètent les similarités dans le graphe original. Comment faire ?

- Naïvement : similarité si connexion entre nœuds
- Indicateurs de voisinage (Jaccard, Adamic-Adar)
- Approches par **marches aléatoires**
  - expressivité : incorpore de l’information locale et globale sur le graphe
  - plus efficace : on ne considère pas toutes les paires dans l’entraînement, seulement celles qui sont co-apparaissent dans une marche aléatoire
  - Méthodes DeepWalk, node2vec
- On peut aussi faire de l’embedding de graphes entiers (hors scope)

Similarités avec marches aléatoires

- Objectif : estimer directement des valeurs d'embedding pour un nœud de façon à préserver des aspects de la structure du graphe



On veut : $z_v^T z_v \approx$ P ("u,v" dans la même marche aléatoire)
On estime la probabilité de visiter le nœud v quand on fait une marche

aléatoire depuis u :

- on fait une marche aléatoire de longueur fixée, à partir de chaque nœud dans le graphe
- on collecte l'ensemble des nœuds visités
- on optimise avec une SGD :

$$L = \sum_{v \in V} \sum_{v \in N_R(u)} -log(P(v|z_u))$$

Similarités avec marches aléatoires

Comment faire la marche aléatoire :

- marche aléatoire non biaisée, depuis chaque nœud (DeepWalk. Perozzi et al., 2013)
- node2vec : marche aléatoire ”flexible”, avec 2 paramètres :
  - possibilité de retourner au nœud précédent
  - ratio d’examen par BFS ou DFS
  - linéaire en temps et parallélisable

Embeddings : à quoi ça sert

Une fois qu'on a les z_i, on peut faire :

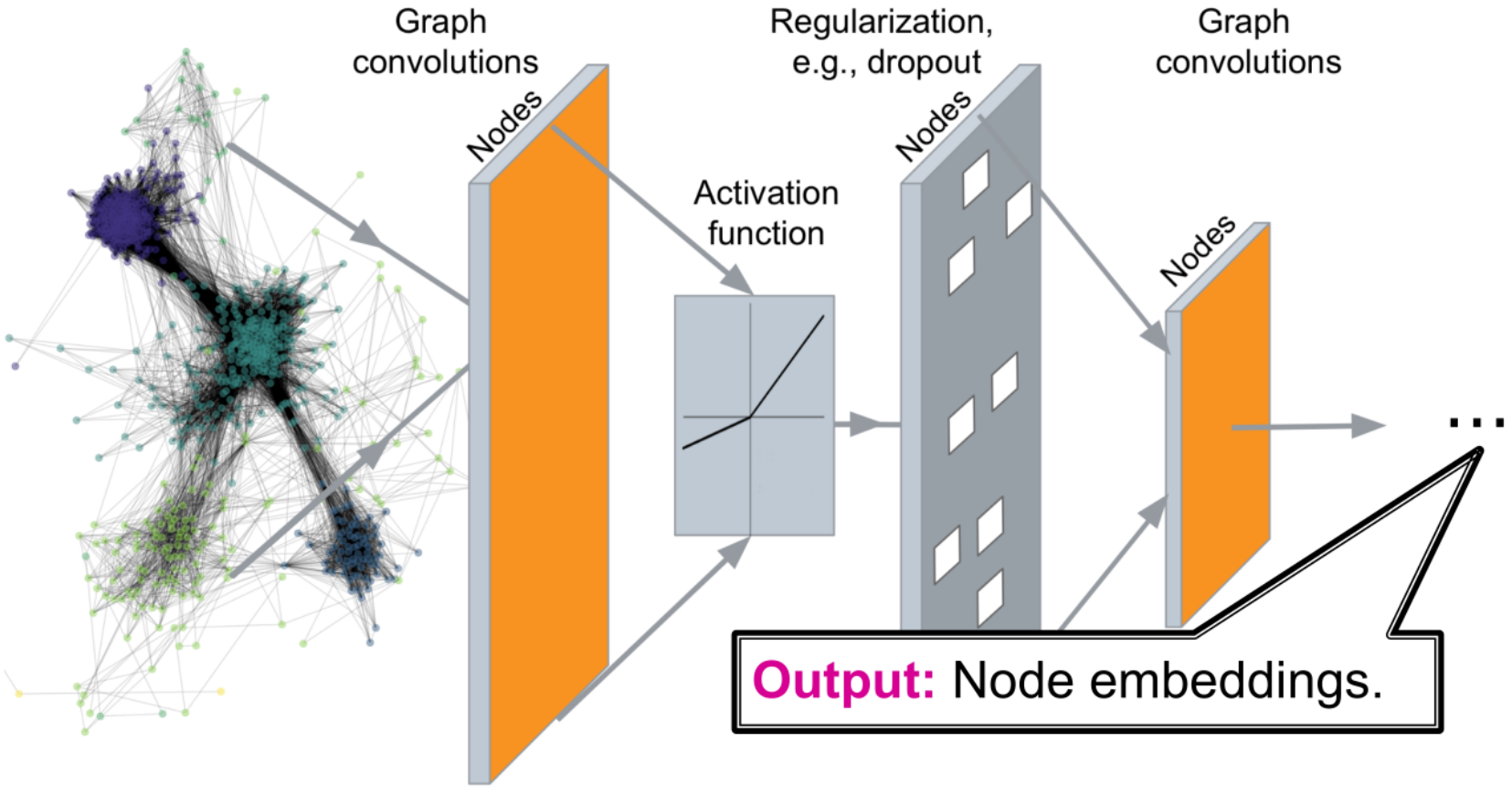
- du clustering/de la détection de communautés
- classer des nœuds, en prédisant une étiquette avec z_i
- prédire des liens (i, j) à l'aide de (z_i, z_j)
  - concaténation, somme, moyenne d'embeddings
- classer des graphes entiers, en agrégeant des z_i

Graph neural networks

Limite du Shallow encoding

- beaucoup de paramètres ($\mathcal{O}(|V|)$) sont nécessaires
  - pas de partage de paramètres entre nœuds
  - chaque nœud a un embedding unique
- transductif uniquement : impossible de générer des embeddings pour des nœuds non vus lors de l'entraînement
- pas d'utilisation des attributs des nœuds

Encodeurs profonds pour graphes

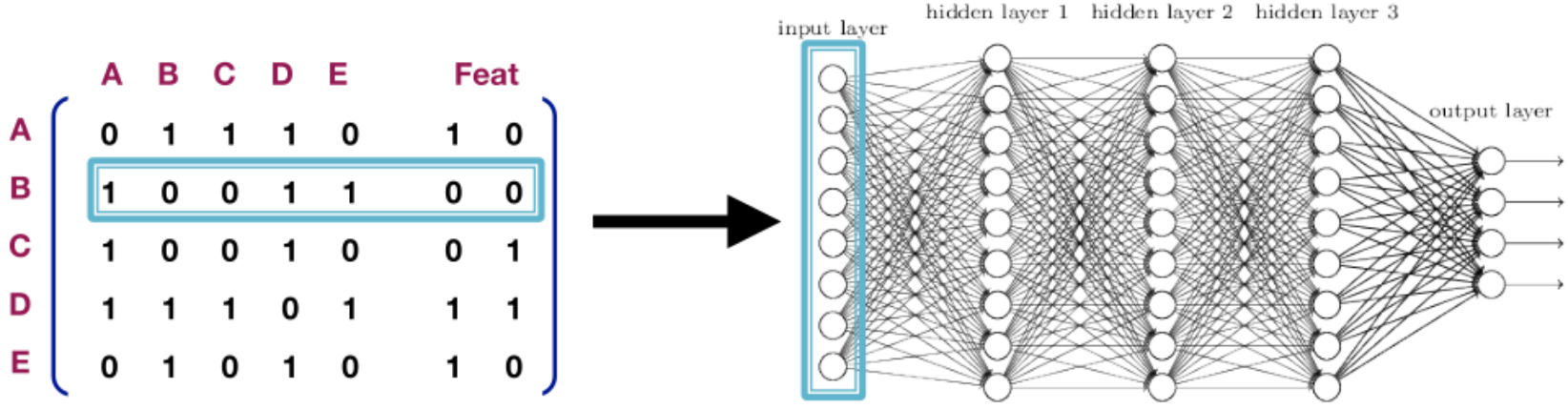


Apprentissage profond et graphes

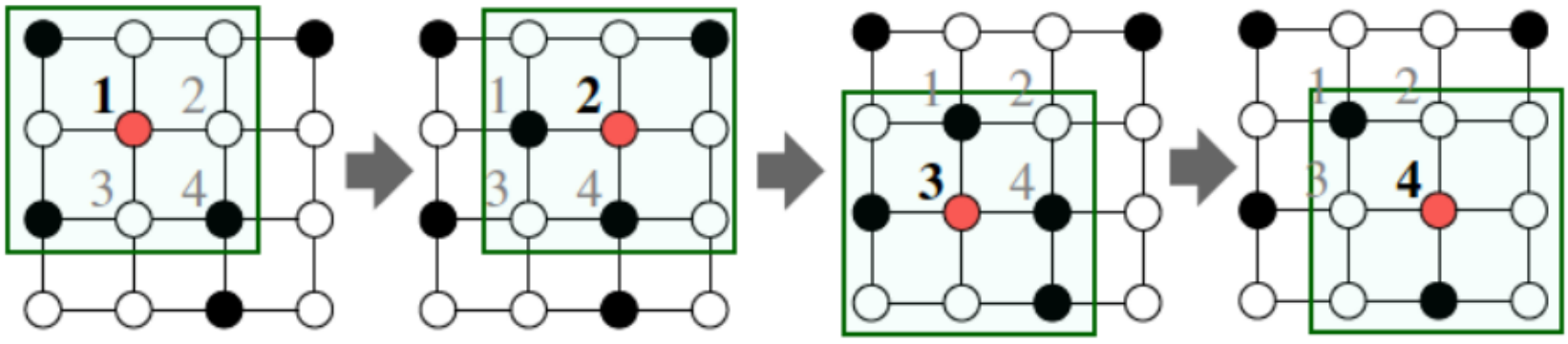
- Graphe G
- Matrice d'adjacence A (binaire)
- $v \in V$, $N(v)$ ses voisins
- $X \in \mathbb{R}^{m \times |V|}$ matrice d'attributs
  - profil de réseau social, image, etc.

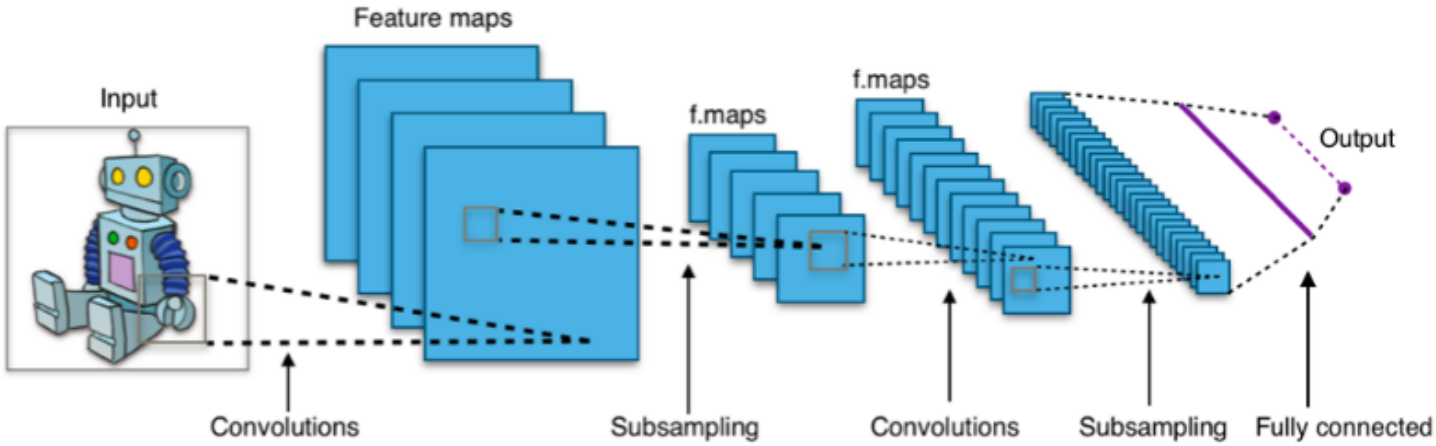
Approche naïve

- fusionner / concaténer la matrice d'attributs et celle d'adjacence, la passer dans un réseau de neurones classique
- problèmes :
  - toujours $O(|V|)$ paramètres
  - peu applicable aux graphes de différentes tailles
  - sensibilité à l'ordre des nœuds

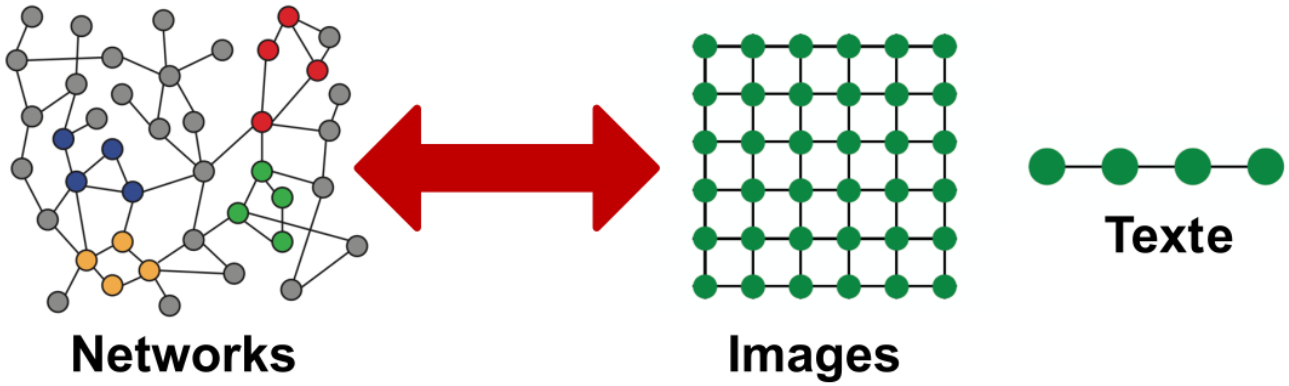


CNN et images





Localité et graphes

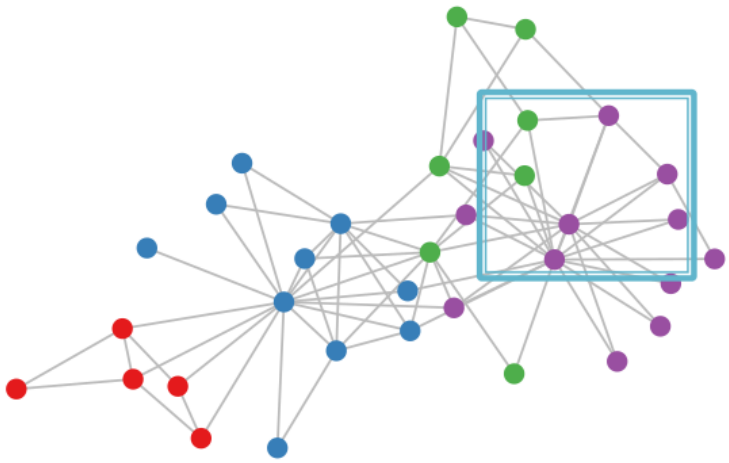


- Les graphes n'ont pas la notion classique de "localité", ou de fenêtre glissante

- Les graphes sont "invariants par permutation" :

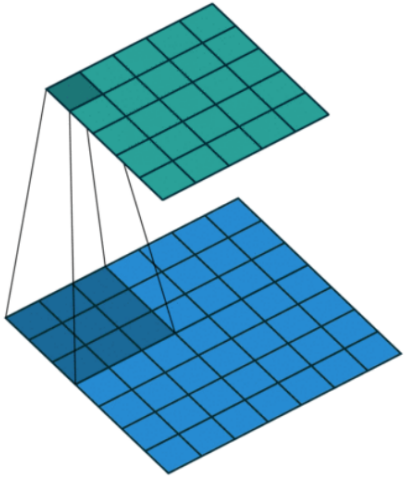
$$f(PAP^T) = f(A) \qquad \text{invariance}$$

$$f(PAP^T) = Pf(A) \qquad \text{equivariance}$$

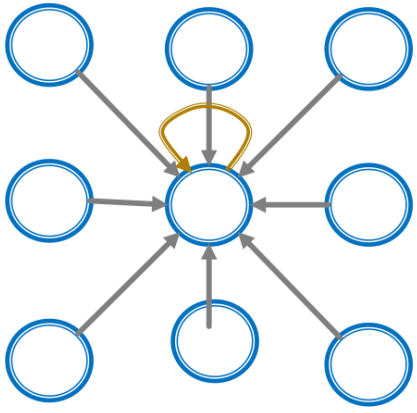


Des images aux graphes

- L'idée est de "transformer" l'information des voisins :
  - les embeddings h_i des voisins sont transformés $W_i \cdot h_i$
  - Puis sommés : $\sum_i W_i h_i$



Image

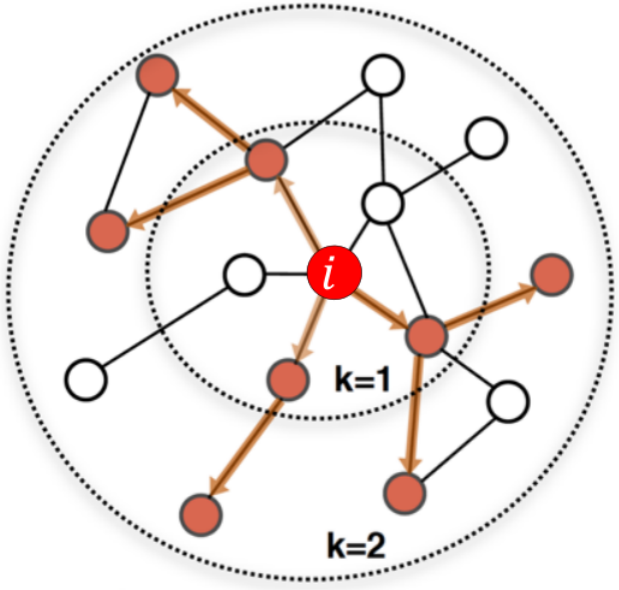


Graph

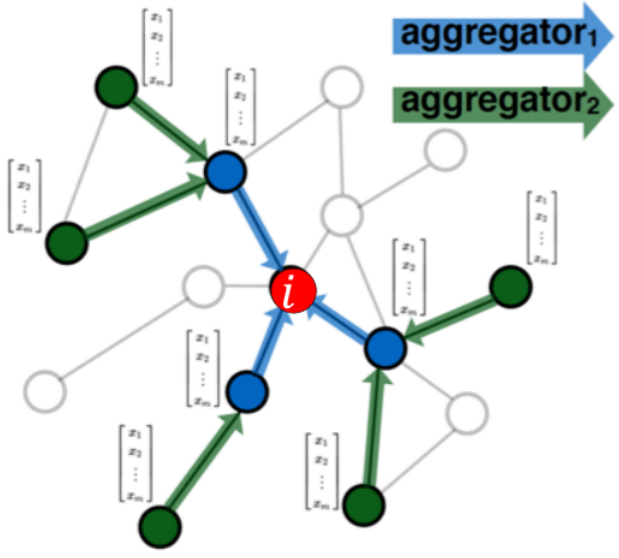
Plan

Graph Convolutional Networks

- Kipf & Welling, ICLR 2017
- le voisinage d'un nœud définit un "graphe de calcul", à l'aide duquel on va propager et transmettre l'information



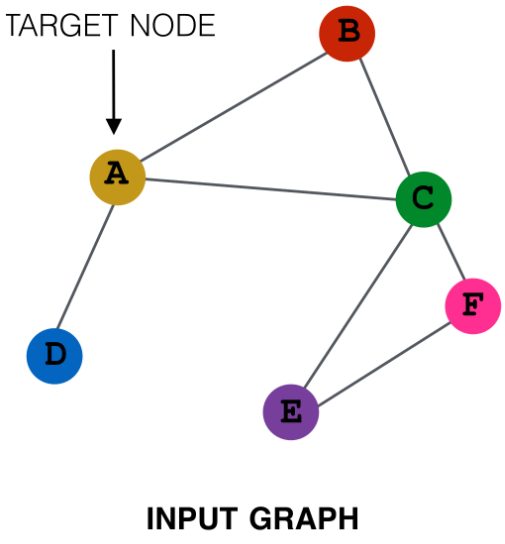
Determine node computation graph

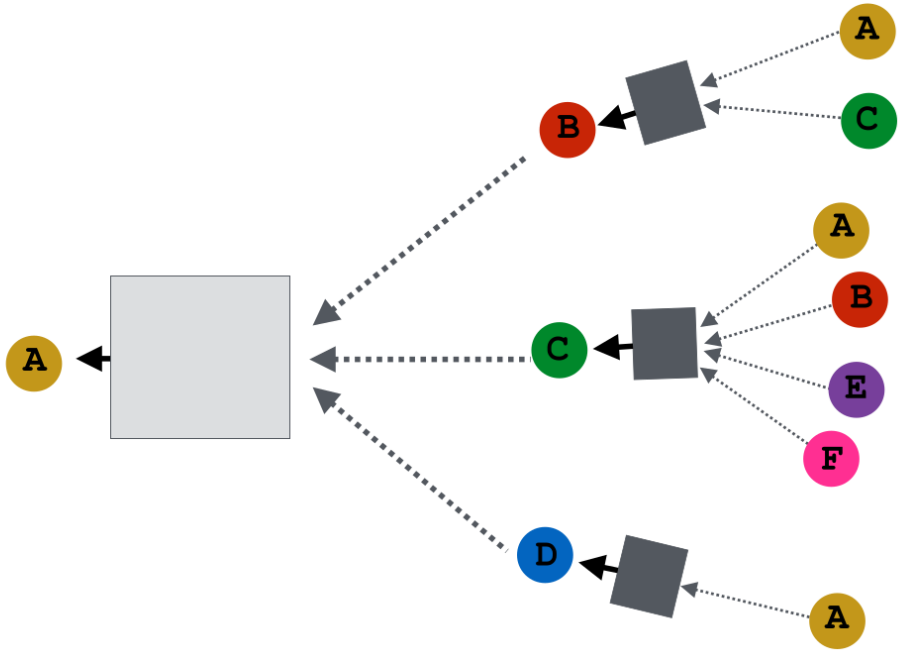


Propagate and transform information

GCN : agrégation de voisinage

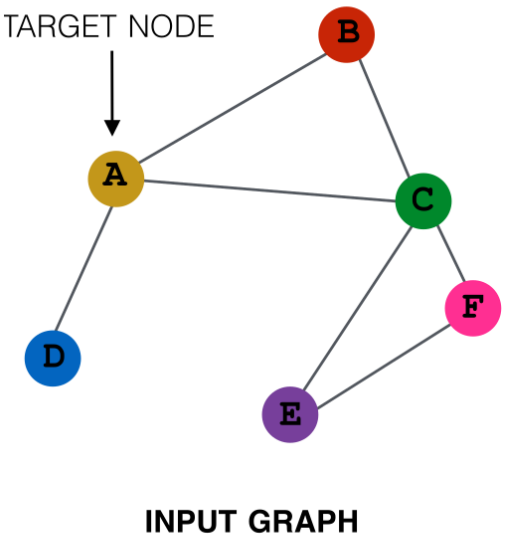
- Embeddings générés à partir des voisinages (localité)

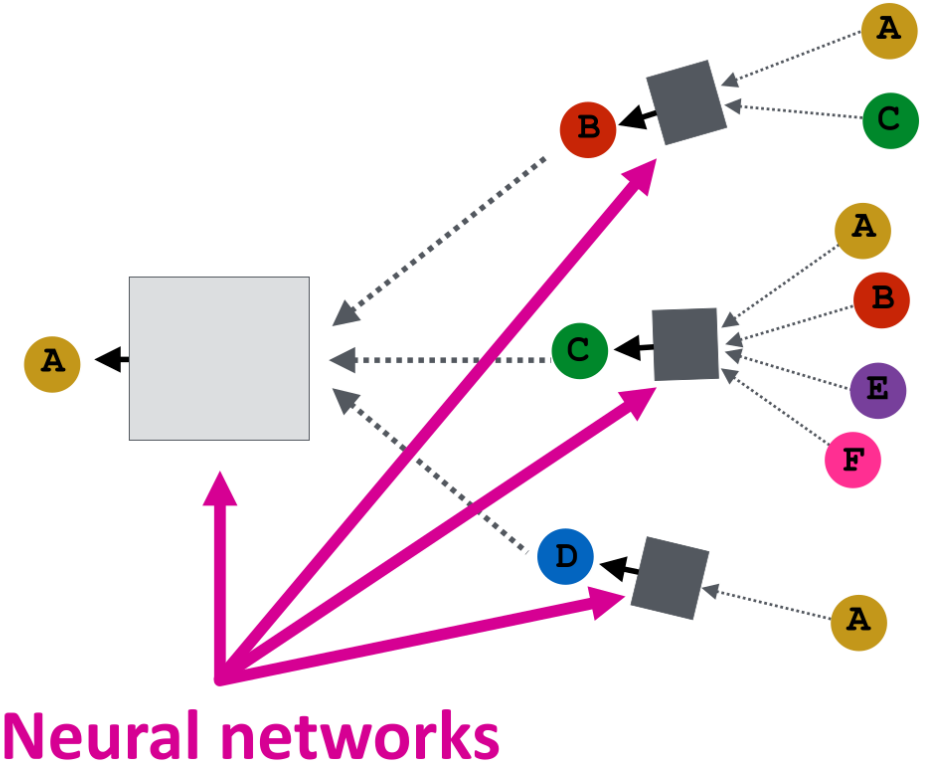




GCN : agrégation de voisinage

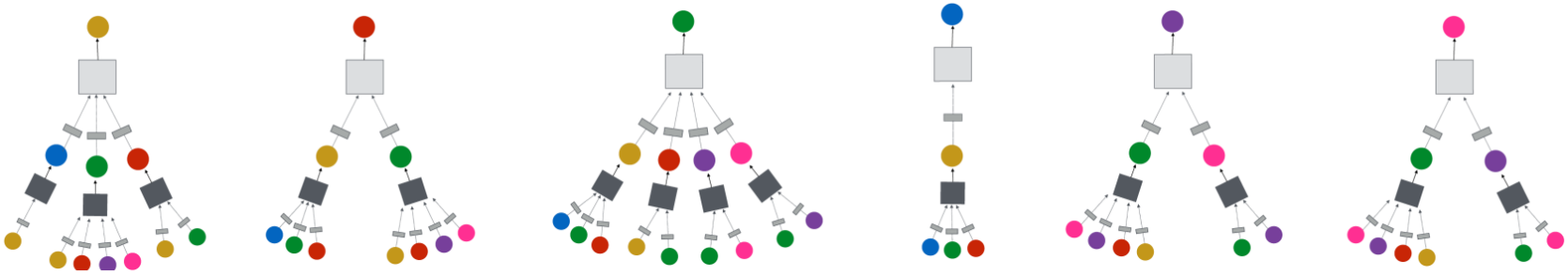
- Les nœuds agrègent l'information des voisins avec des réseaux de neurones





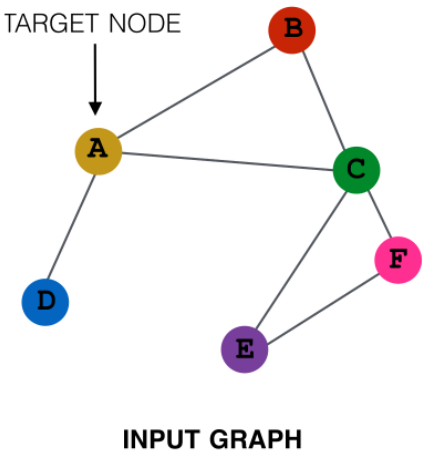
GCN : agrégation de voisinage

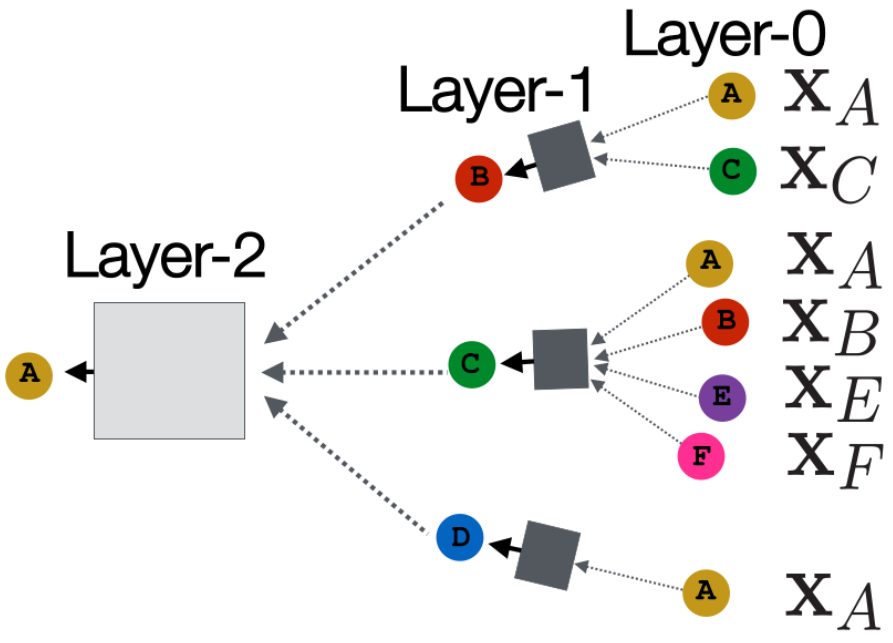
- Chaque nœud définit un ”graphe de calcul” (computation graph) en fonction de la structure de son voisinage



Nombre de couches

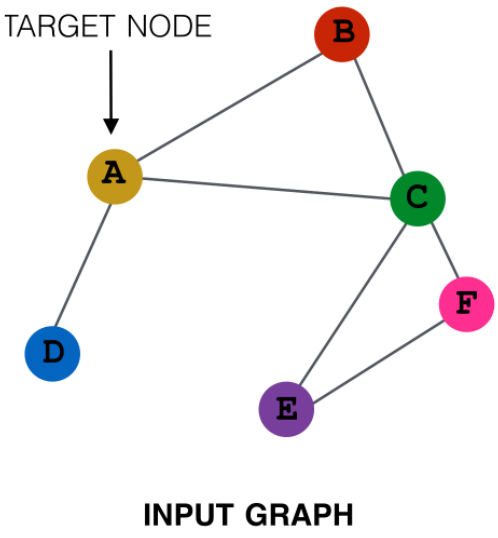
- Profondeur arbitraire (!)
  - les nœuds ont des embeddings à chaque couche
  - la couche 0 : les attributs d'entrée x_v
  - la k-ième couche : de l'information des sommets à distance k

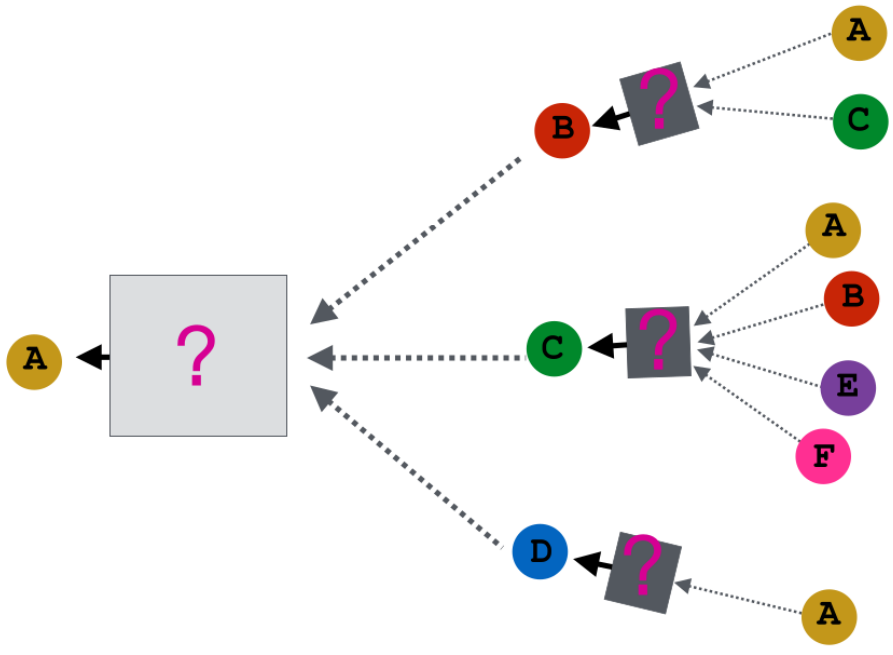




Quel réseau de neurones ?

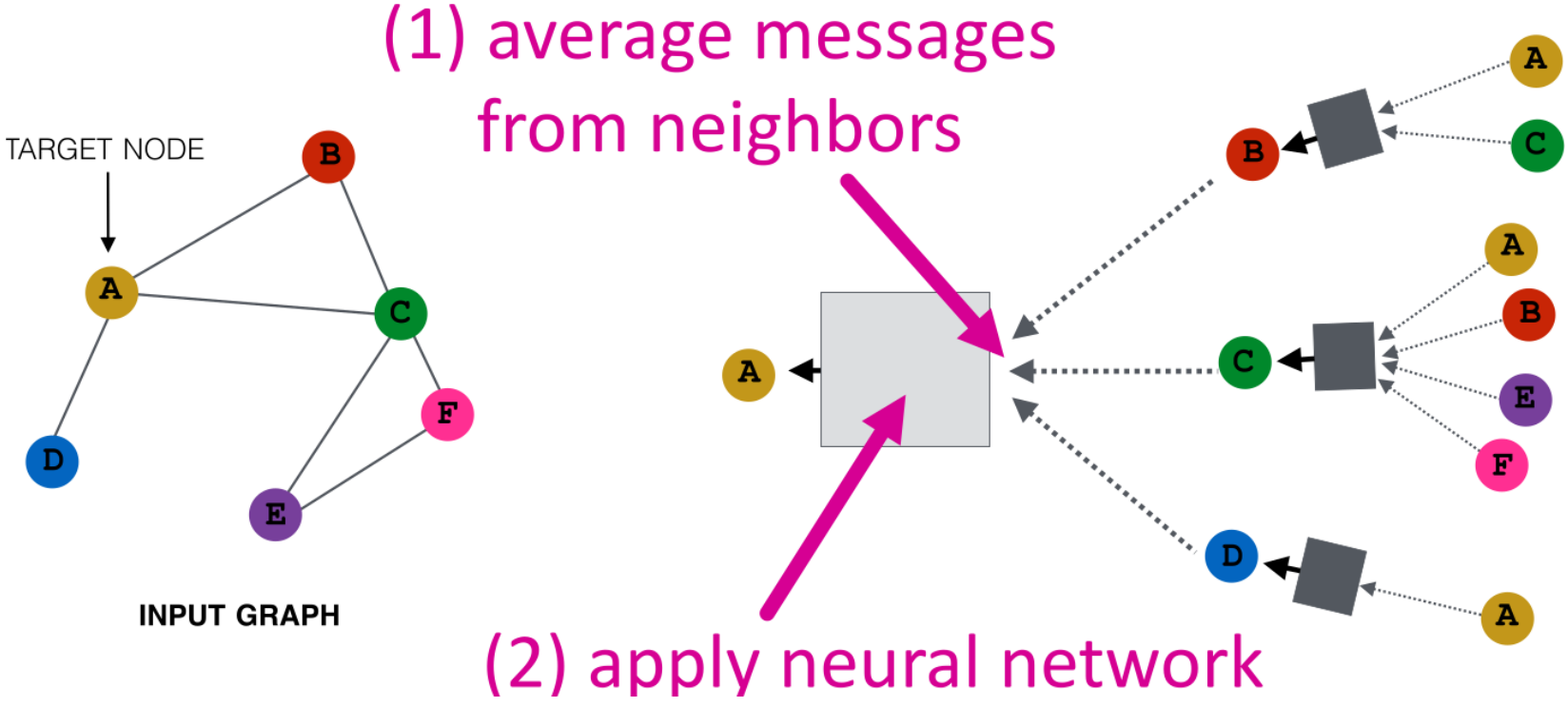
- Les approches diffèrent surtout dans la manière d'agréger l'information des différentes couches





Quel réseau de neurones ?

- Approche simple : une moyenne des voisins



Mathématisation

- $h_v^0 = x_v$
- $h_v^{l+1} = \sigma(W_l \sum_{u \in N(v)} \frac{h_u^{(l)}}{|N(v)|} + B_l h_v^{(l)}), \qquad \forall l \in \{0, ..., L-1\}$
- $z_v = h_v^L$

- h_v^l est la représentation cachée de v pour la couche l
- W_l et B_l sont des matrices de poids (qu'on apprend). W pour l'agrégation du voisinage, B pour la transformation du vecteur lui-même
- on peut placer ces embeddings dans une fonction de loss et utiliser une SGD pour apprendre ces poids

Entraînement

Supervisé

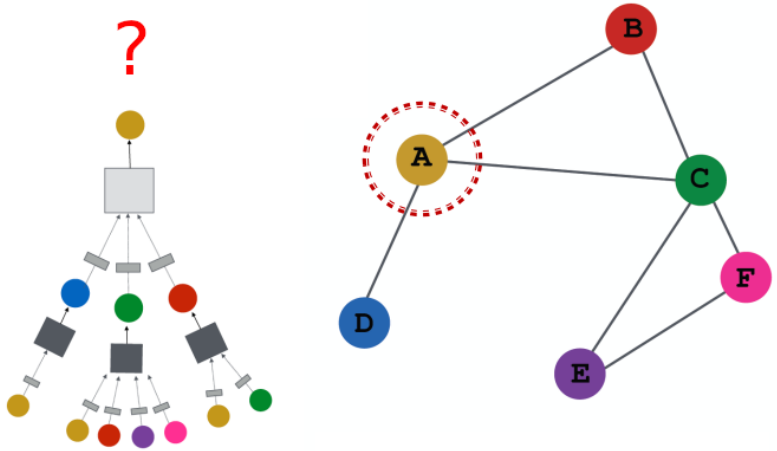
- on cherche à minimiser la loss $\mathscr{L}$:

$$\min_{\theta} \mathscr{L}(y, f(z_v))$$

- y est l'étiquette du nœud
- $\mathscr{L}$ peut être L2 si y est réel, Cross-entropy sinon :

$$\mathscr{L} = \sum_{v \in V} y_v log(\sigma(z_v^T \theta)) + (1 - y_v) log(1 - \sigma(z_v^T \theta))$$

On entraîne directement, par exemple pour de la classification



- on peut entraîner en non supervisé, en prenant une similarité structurelle comme label

Inductivité

- Les paramètres d'agrégation sont partagés entre les nœuds : le modèle a un nombre de paramètres sous-linéaire en $|V|$
- Capacité de généralisation aux nœuds non vus !
  - application : recommandation à des nouveaux utilisateurs !
- On peut même généraliser à des graphes entièrement "non vus"
  - à partir de graphes d'interaction de protéines d'un organisme A, on peut générer des embeddings à partir de données collectées pour l'organisme B

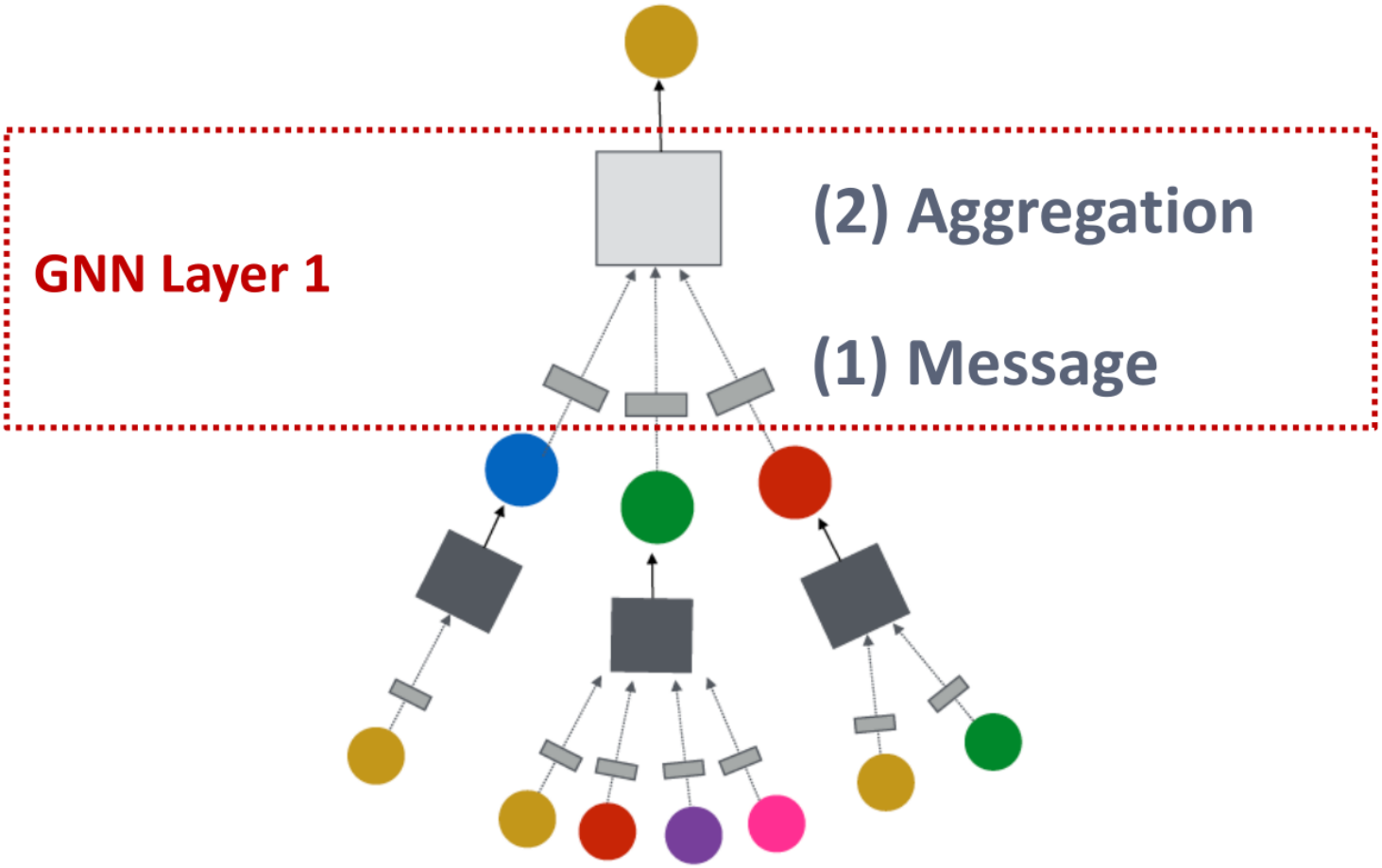
Résumé

- On définit une fonction d'agrégation de voisinage (on passe des "messages"
- On définit une loss sur les embeddings
- On s'entraîne sur un ensemble de sommets (ie : des graphes de calcul)
- On génère des embeddings pour les sommets... et même pour ceux que l'on n'a pas vu (à l'aide des embeddings/des valeurs dans les couches)

Les composants du framework GNN

Framework GNN

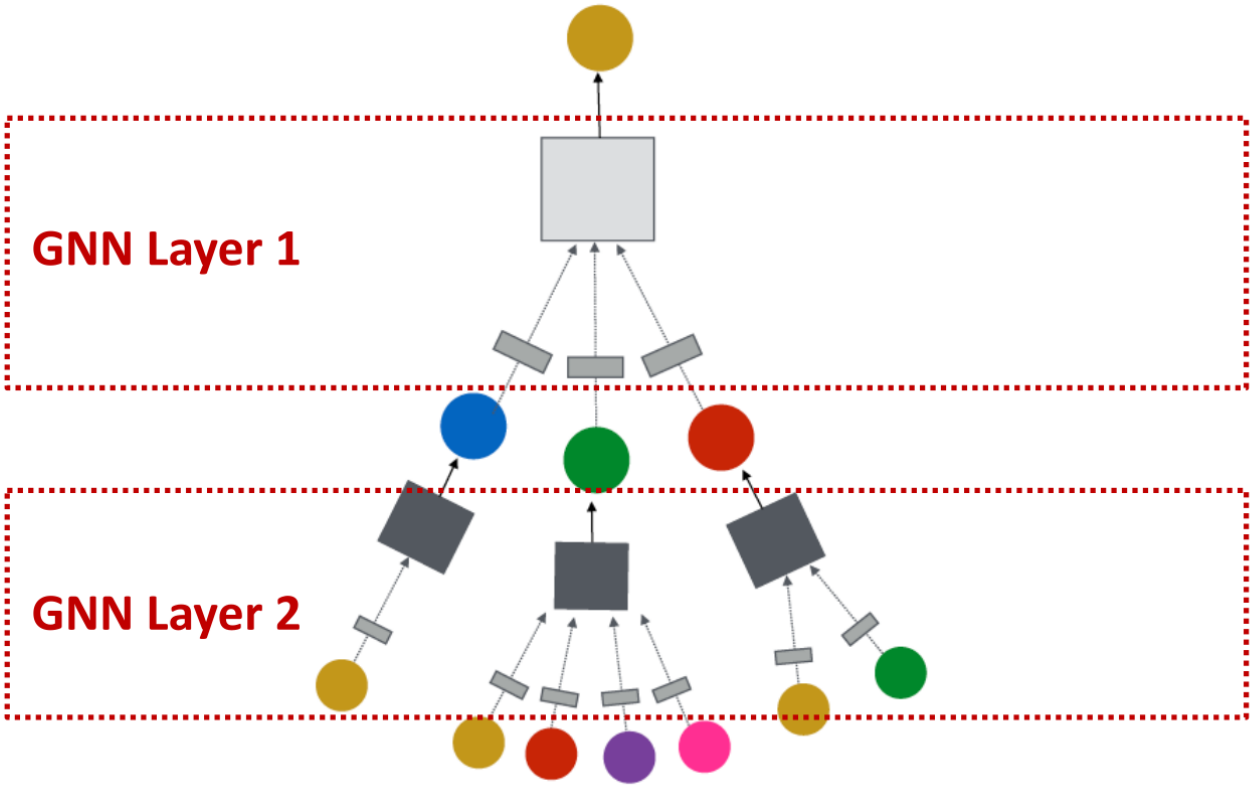
- Une couche GNN, c'est :
  - des "messages"
  - leur agrégation



Framework GNN

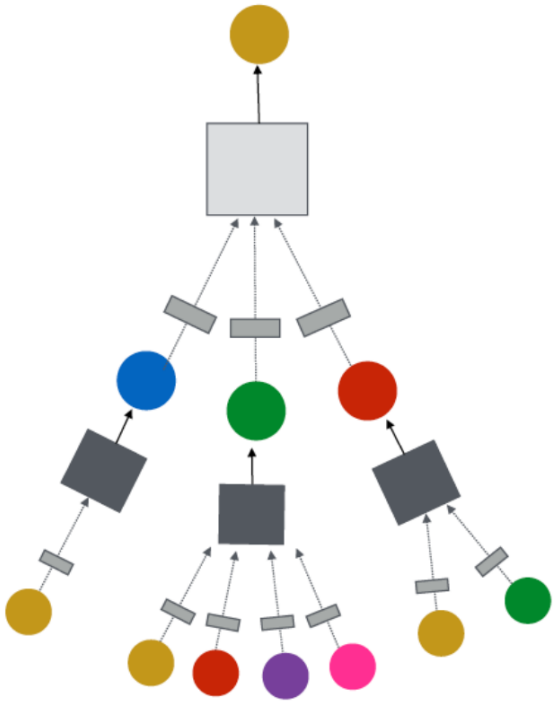
- Des connexions entre couches GNN
  - différentes manières de les empiler
  - des raccourcis (skip connections)

(3) Layer connectivity



Framework GNN

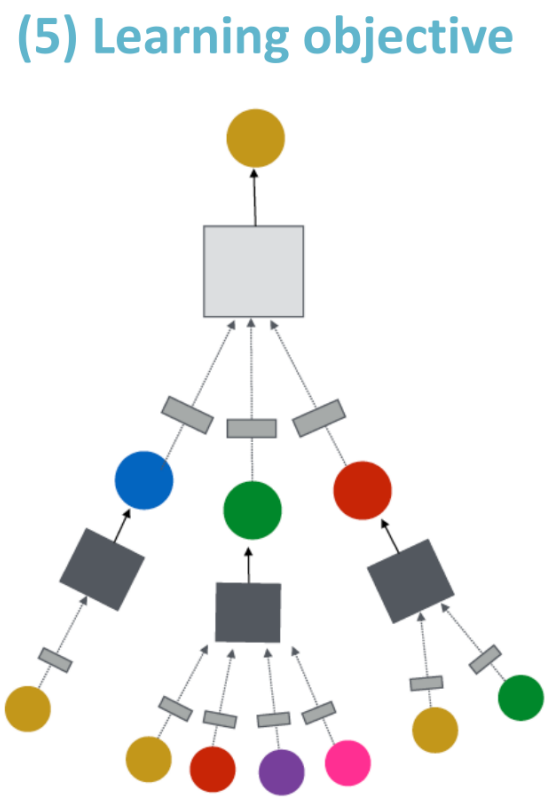
- le "graphe de calcul", à partir duquel sont calculés les messages, peut être différent du graphe "brut"
  - ajout d'attributs
  - modification de la structure



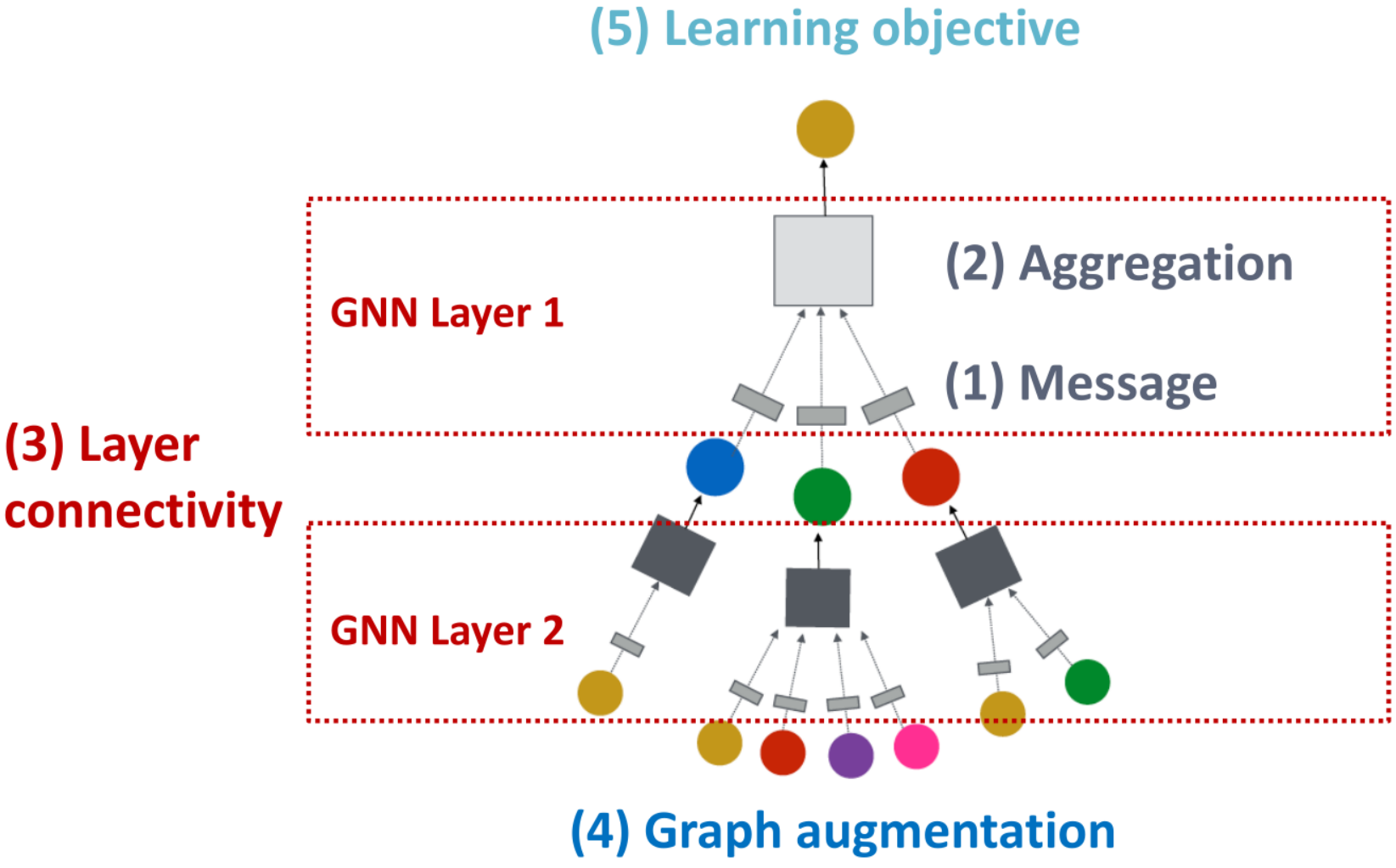
(4) Graph augmentation

Framework GNN

- l'entraînement du GNN peut se faire
  - en supervisé/non supervisé
  - à plusieurs niveaux : Noeud/Lien/Graphe



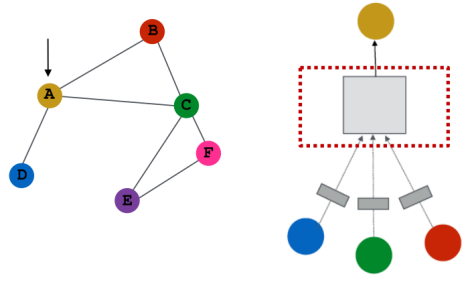
Framework GNN : synthèse



Plan

Message

- L'idée d'une couche GNN, c'est de compresser un ensemble de vecteurs (celui d'un nœud, de ses voisins) en un seul
- 2 étapes : Message + Agrégation



Message

- Un *message* est créé par chaque nœud, passés aux autres ensuite

$$m_u^{(l)} = MSG^{(l)}(h_u^{(l-1)})$$

- En linéaire : $m_u^{(l)} = W^{(l)} h_u^{(l-1)}$

Agrégation

- Chaque nœud agrège les messages de ses voisins

$$h_u^{(l)} = AGG^{(l)}(\{m_v^{(l)}, v \in N(u)\})$$

- Somme, Moyenne, Max

Information du nœud

Problème de l'agrégation

- Perte de l'information sur chaque nœud (si le calcul de $h_u^{(l)}$ ne dépend pas de $h_u^{(l-1)}$)

- Solution : on inclut $h_u^{(l-1)}$, dans le message et l'agrégation
- Dans le message, avec un message spécifique $m_u^{(l)}$
- Mais aussi dans l'agrégat, en concaténant avec l'agrégation des voisins :

$$h_u^{(l)} = \text{CONCAT}(AGG^{(l)}(\{m_v^{(l)}, v \in N(u)\}), m_u)$$

Plan

GNN classiques : GCN

- Graph Convolutional Networks (Kipf Welling 2017)

$$h_v^{(l)} = \sigma\left(W^l \sum_{u \in N(v)} \frac{h_u^{(l-1)}}{|N(v)|}\right)$$

- Message (chaque voisin) : $m_u^{(l)} = \frac{1}{|N(v)|} W^{(l)} h_u^{(l-1)}$
- Agrégation : Somme sur tous les voisins, puis activation

GraphSage

$$h_v^{(l)} = \sigma\big(W^{(l)} \quad \text{CONCAT}\big(h_u^{(l-1)}, \text{AGG}\big(\{h_u^{(l-1)}, u \in N(v)\}\big)\big)\big)$$

- Message : AGG (variable)
- Agrégation en 2 phases :
  - on agrège le voisinage
  - on agrègre avec le nœud lui-même
- AGG peut être :
  - Moyenne pondérée : $\text{AGG} = \sum_{u \in N(v)} \frac{h_u^{(l-1)}}{|N(v)|}$
  - Pooling : $\text{AGG} = \text{Mean}(\text{MLP}(h_u^{(l-1)}, \forall u \in N(v)))$
- Optionnel : L2 normalisation à chaque couche

Graph Attention Networks

$$h_v^{(l)} = \sigma\Big(\sum_{u \in N(v)} \alpha_{vu} W^l h_u^{(l-1)} \Big)$$

- Dans GCN/GraphSage, $\alpha_{vu} = \frac{1}{|N(v)|}$
- Ces poids sont définis à partir du degré
- Tous les voisins de v sont également importants pour v
- L'idée est d'améliorer l'agrégation du voisinage, en allouant des poids variés (et appris) à différents voisins

Mécanisme attentionnel

- α_{uv} est le résultat d'un calcul de coefficients d'attention bruts e_{vu} :
  - $e_{vu} = \text{ATT}\left(W^l h_u^{(l-1)}, W^l h_v^{(l-1)}\right)$
  - les e_{vu} sont normalisés par softmax, pour que $\sum_{u \in N(v)} \alpha_{vu} = 1$

$$\alpha_{uv} = \frac{exp(e_{vu})}{\sum_{k \in N(v)} exp(e_{vk})}$$

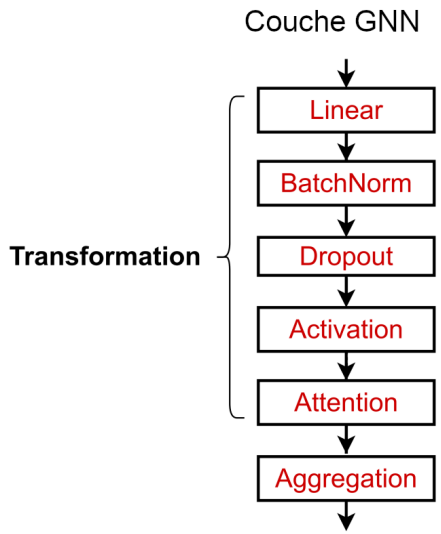
- le mécanisme d'attention ATT peut être un NN simple couche, dont les paramètres sont appris en même temps que W^l

Bénéfice de l'attention

- Efficacité computationnelle : parallélisation sur les liens / les nœuds
- Efficacité en espace : $\mathcal{O}(V+E)$ pour les matrices à stocker, nombre fixé de paramètres (ne dépend pas de n)
- Inductif : ne dépend que de l'arête (localité)

Améliorations : des couches GNN augmentées

- On a vu les blocs à la base d'une couche GNN
- On peut aussi inclure d'autres éléments dans les couches :
  - Batch Normalization (stabilise l'apprentissage).
    Centrage des embeddings pour avoir une moyenne 0 et une variance unitaire
  - Dropout (évite le sur-apprentissage).
    Probabilité de désactiver quelques neurones dans l'entraînement (couche linéaire)
  - Activation.
    ReLU, Sigmoïde, ReLU paramétrique ($y = ax, x < 0$ avec a appris)
  - Attention (contrôle l'importance des messages)

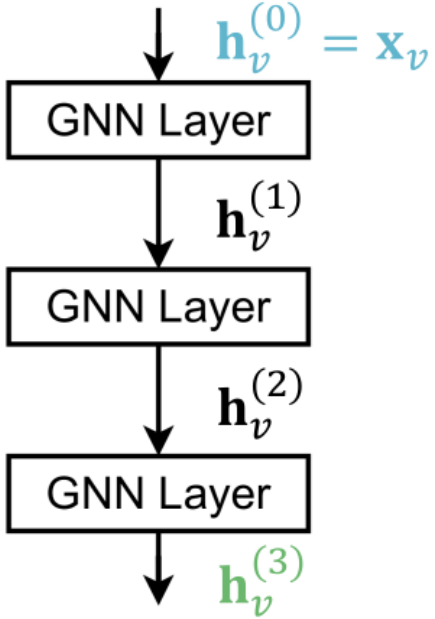


Plan

Empiler des couches GNN

Manière standard de construire un GNN :

- on place les couches les unes à la suite des autres
- entrée : les attributs du nœud dans le vecteur x_v
- sortie : les embeddings $h_v^{(L)}$ après L couches



Problème :

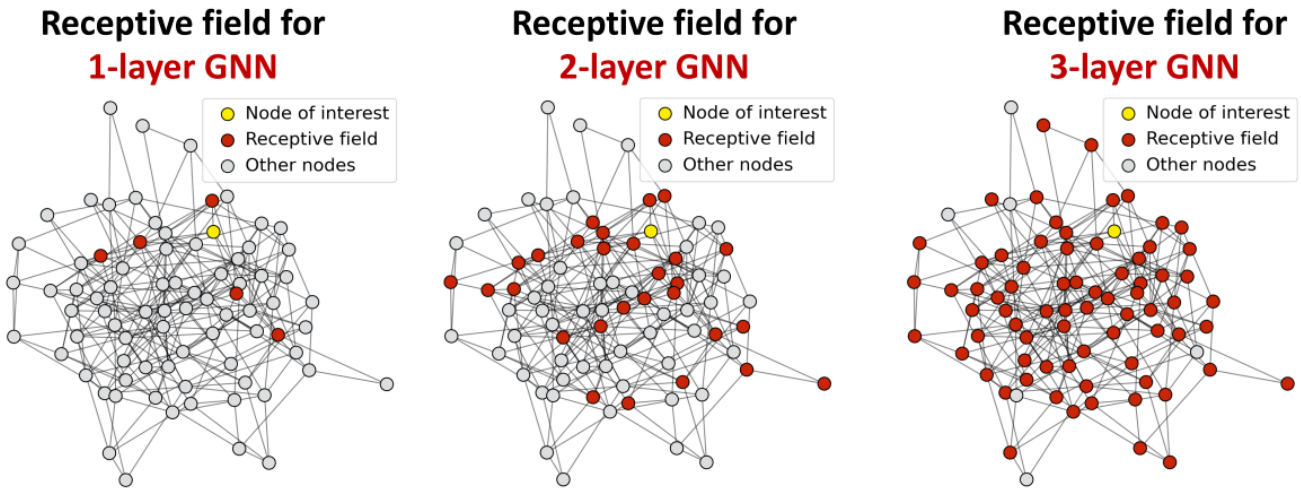
- les embeddings convergent vers des valeurs similaires
- gênant car on veut les utiliser pour différencier des nœuds !

Champ réceptif

Que se passe-t-il ?

- On parle du champ réceptif pour désigner l'ensemble des nœuds qui contribuent à l'embedding d'un nœud donné
- Dans un GNN de K couches, chaque nœud a un champ réceptif correspondant à son voisinage à distance K (!)
- Les voisins en commun deviennent rapidement nombreux (quand on augmente le nombre de couches)
- À distance 3, presque tous les sommets

- Conséquence : ces voisinages avec beaucoup de recouvrement induisent des embeddings très similaires !



Champ réceptif

- Il faut donc être prudent dans l'ajout/empilement des couches GNN
- Contrairement aux images (CNN), ajouter des couches n'aide pas toujours
  - Il faut analyser le champ réceptif nécessaire au problème (par exemple, en analysant le diamètre du graphe)
  - Choisir L pas trop grand !

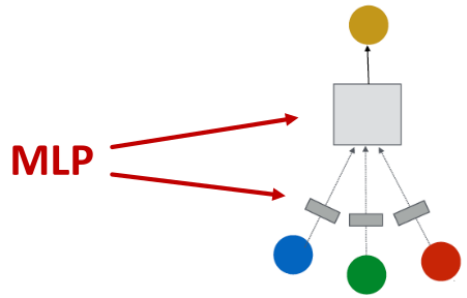
3 solutions possibles :

- Accroître l'expressivité de chaque couche (complexifier)
- Ajouter des couches qui ne font pas de passage de messages
- Ajouter des raccourcis entre couches (*skip-connections*)

Solutions

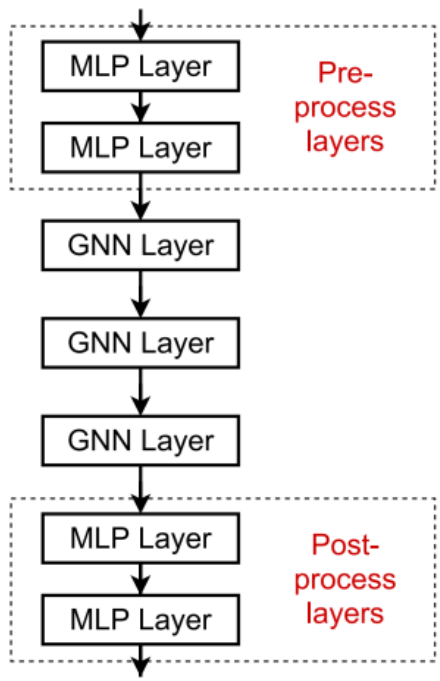
Expressivité

- dans nos exemples, on n'avait qu'une couche linéaire pour l'agrégation/transformation
- chaque agrégation/transformation peut être un réseau de neurones (par exemple : un MLP de 3 couches)



Autres couches

- on peut avoir des couches "non GNN" dans un "GNN"
- des couches de pré-traitement, par exemple pour encoder les features visuelles/textuelles associées à un nœud
- des couches de post-traitement, pour raisonner sur les embeddings eux-mêmes (classification de graphes)



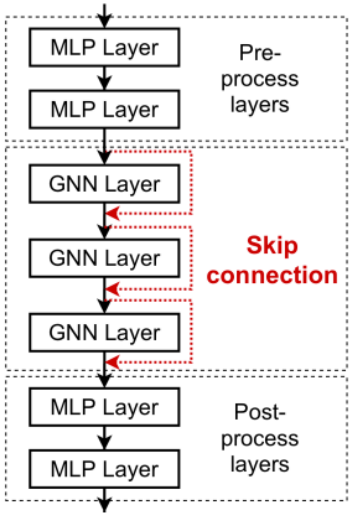
Raccourcis (*skip connections*)

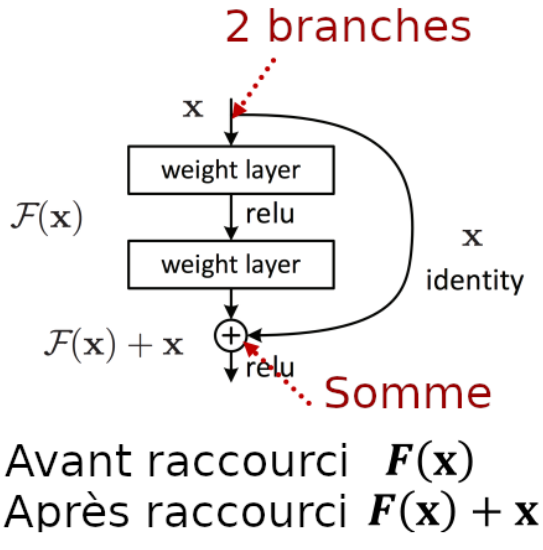
Si on a, malgré tout, besoin de nombreuses couches GNN, on peut ajouter des raccourcis.

- Pourquoi ? Les embeddings des premières couches différencient mieux les nœuds les uns des autres

$\Rightarrow$ On augmente leur influence avec des raccourcis dans le réseau

- Ça marche en créant, automatiquement, un mélange entre GNN "superficiel" et GNN profond (puisque l'information peut prendre divers chemins)
- N raccourcis $\Rightarrow 2^N$ chemins
- On peut aussi faire les raccourcis vers la dernière couche





Avant raccourci $\boldsymbol{F}(\mathbf{x})$
Après raccourci $\boldsymbol{F}(\mathbf{x}) + \mathbf{x}$

Plan

Ajouts

Jusqu'à présent, on a considéré que le graphe de calcul était basé sur le graphe brut. Pourtant :

- le graphe peut manquer d'attributs $\Rightarrow$ on peut en ajouter
- le graphe peut être trop creux $\Rightarrow$ passage de message peu efficace
- le graphe peut être trop dense $\Rightarrow$ passage de message trop coûteux
- le graphe peut être trop grand $\Rightarrow$ mémoire (GPU)

Le graphe brut est probablement pas optimal pour le calcul des embeddings.

Modifications du graphe

- ajout d'attributs
- Ajouts de nœuds/arêtes
- Échantillonnage de voisinage
- Échantillonnage par sous-graphes

Ajout d'attributs

- Pas d'attributs (ex. : on n'a que la matrice d'adjacence)
- 2 approches :

Valeur constante ajoutée aux nœuds

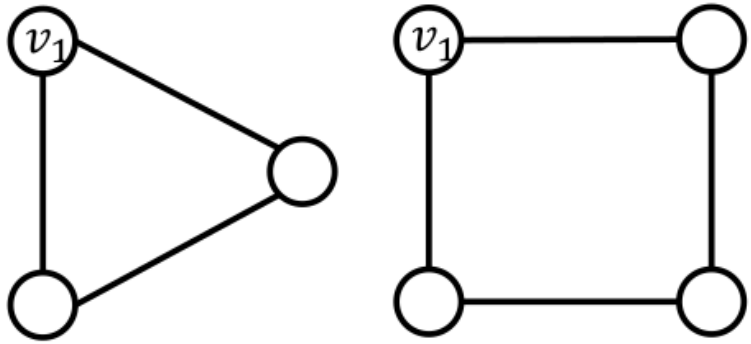
- les nœuds sont identiques, mais on peut tout de même apprendre de la structure
- généralisation aux nouveaux nœuds
- coût de calcul faible

Identifiants unique pour chaque nœud (1-hot encoding)

- on peut préserver l'information de chaque nœud
- généralise mal aux nouveaux nœuds
- taille élevée, donc coût de calcul plus important
- contexte : petits graphes, sans nouveaux nœuds

Autres ajouts d'attributs

- On peut aussi essayer de compléter les attributs existants, s'il y en a peu.
- Exemple : essayer de savoir si le nœud est dans un cycle, et quelle en est la longueur



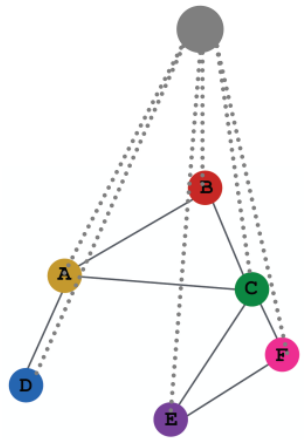
- D'autres caractéristiques classiques :
  - PageRank
  - Coefficient de clustering
  - Centralité

Modifications structurelles

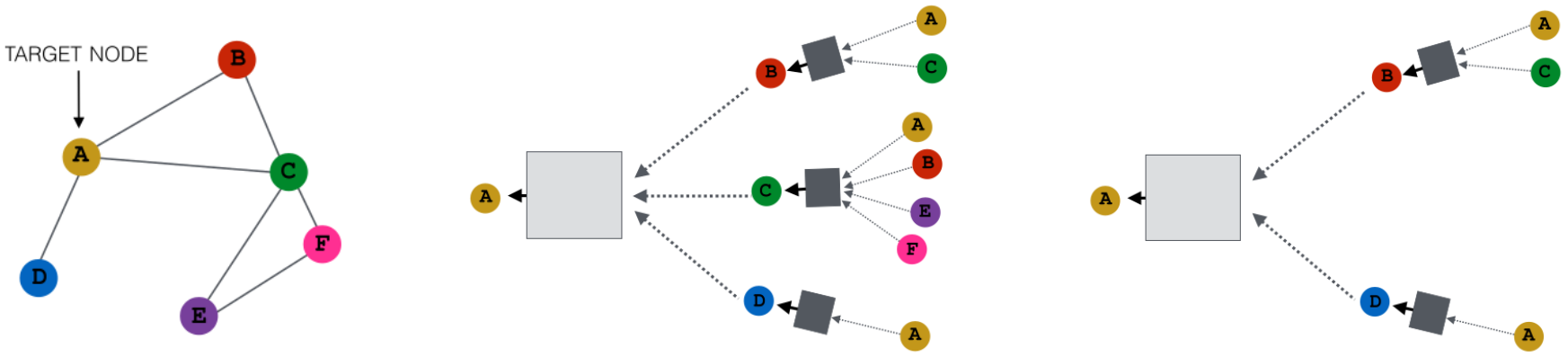
On peut souhaiter augmenter les graphes un peu creux

- Ajout de liens
  - Idée simple : ajout de liens directs pour les voisins à distance 2
  - traduction matricielle : au lieu de calculer avec la matrice d'adjacence A, on travaille avec $A + A^2$
- Ajouts d'arêtes
  - nœud virtuel connecté à tous les sommets
  - dans un graphe creux, où les plus courts chemins sont de taille 10 $\Rightarrow$ taille 2
  - A $\Rightarrow$ nœud virtuel $\Rightarrow$ B

  $\Rightarrow$ Améliore le passage de message



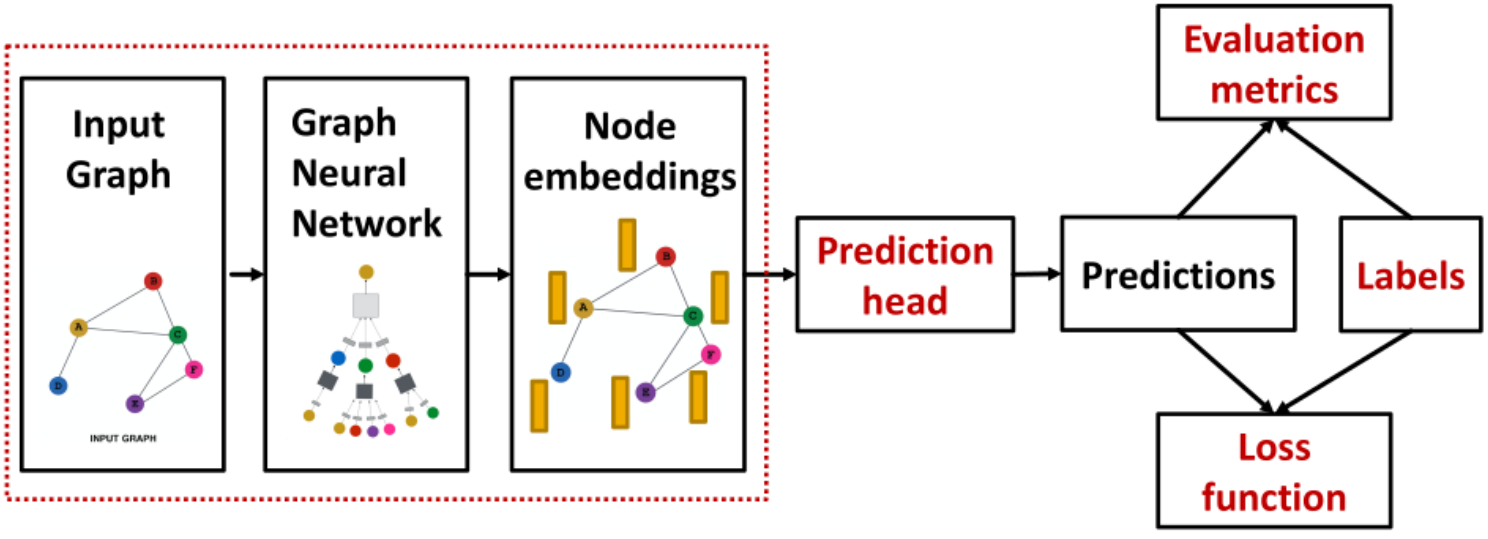
Échantillonnage de voisinage



- On peut changer de voisins à chaque calcul
- En pratique, ça marche bien

Plan

Entraînement



- Différentes manières de faire des prédictions
  - Niveau nœud
  - Niveau lien
  - Niveau graphe

Niveau nœud

- Usage direct des embeddings possible
- On dispose d'embeddings de dimension d : $\{h_v^{(L)} \in \mathbb{R}^d, \forall v \in G\}$
- On veut prédire en k catégories
- Notre "tête de prédiction" est :

$$\hat{y} = \text{Head}_{node}(h_v^{(L)}) = W^{(H)} h_v^{(L)}$$

- $W^{(H)} \in \mathbb{R}^{k \times d}$

Niveau lien

- On utilise les embeddings de la paire de sommets qui sont liés par le lien
- 2 options principales pour $\text{Head}_{edge}(h_v^{(L)}, h_u^{(L)})$
  - Concaténation et couche linéaire :

$$\hat{y}_{uv} = \text{Linear}(\text{Concat}(h_u^{(L)}, h_v^{(L)}))$$

  - Linear doit mapper de dimension 2d vers k.
  - Produit scalaire :

$$\hat{y}_{uv} = {h_u^{(L)}}^T h_v^{(L)}$$

  - Marche pour la prédiction vers dimension 1 (link prediction)

Niveau graphe

- On veut utiliser tous les embeddings de notre graphe
- On veut

$$\hat{y} = \text{Head}_{graph}(\{h_v^{(L)} \in \mathbb{R}^d, \forall v \in G\})$$

- C'est comme l'agrégation dans une couche GNN, donc mêmes idées :
  - Mean, max, sum pooling
  - Attention, le pooling global perd de l'information
  - Pooling hiérarchique, par exemple (basé sur clusters)

Supervisé - non-supervisé

- Entraînement supervisé : les labels viennent de sources externes
- Entraînement non-supervisé : les signaux viennent du graphe lui-même
- parfois la différence est mince (si on entraîne pour prédire le coefficient de clustering)
- Idée : réduire la tâche à de la prédiction de label (ex. : du clustering)
- on parle de self-supervision. On peut utiliser des signaux classiques (stats de nœuds)

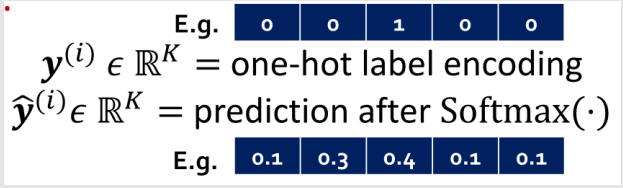
Classification, régression

- Classification : les labels que l'on prédit sont discrets
- Régression : "labels continus" (probabilité d'être une molécule viable)

- Les GNN peuvent s'utiliser dans les deux cas, la loss et la métrique d'évaluatuion changent !

Loss

- Classification : entropie croisée $CE = -\sum y_j \log(\hat{y}_j)$



- MSE : $MSE = \sum(y_j - \hat{y}_j)$

Métriques

Régression

- RMSE
- MAE

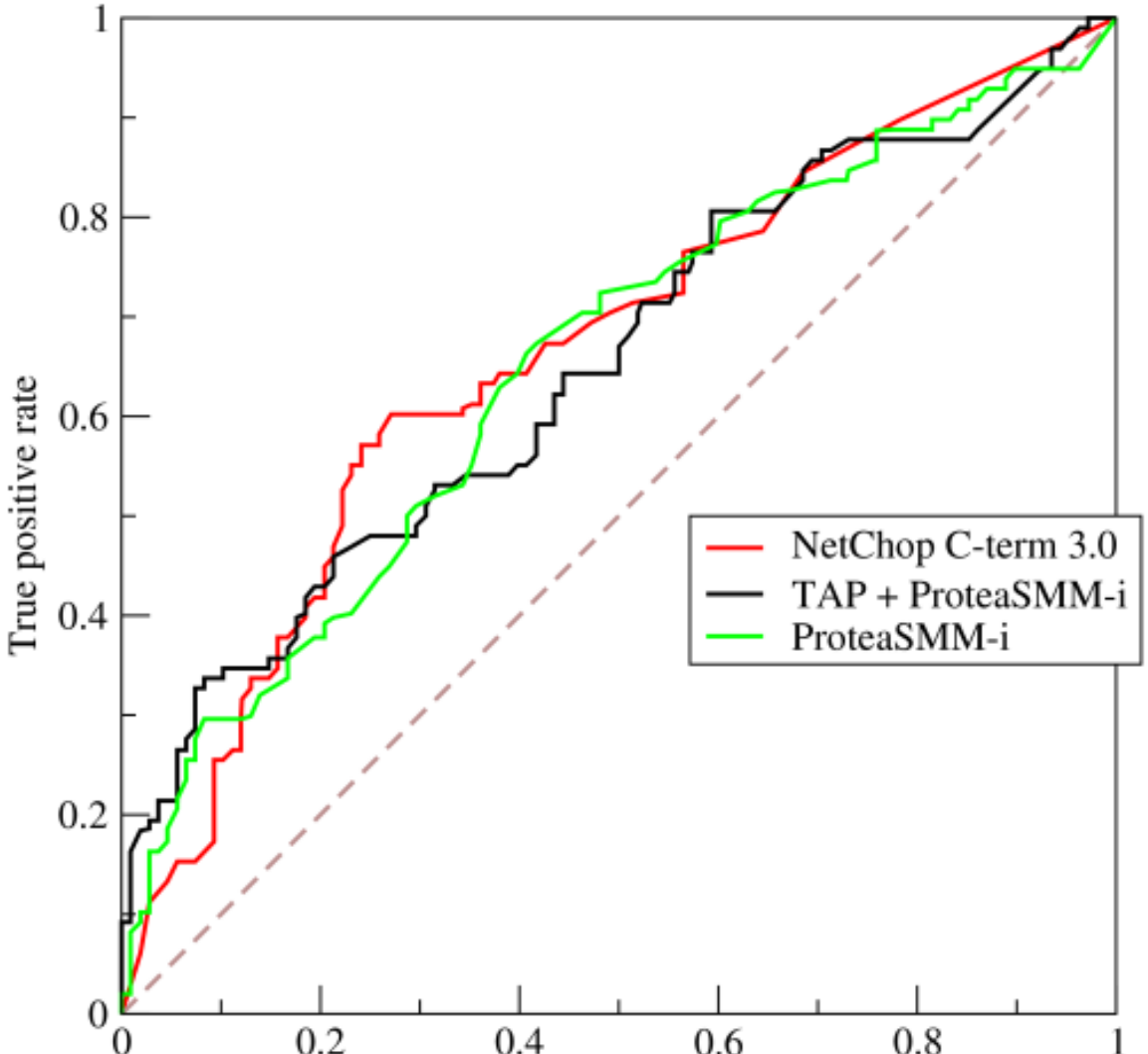
Classification

- Accuracy : $\frac{TP+TN}{TP+TN+FP+FN}$
- Précision $\frac{TP}{TP+FP}$
- Rappel $\frac{TP}{TP+FN}$ (Information Retrieval)

- ROC AUC

ROC AUC

- ROC : Receiver Operating Curve
- Affiche TPR ($\frac{TP}{TP+FN}$) comparé à FPR ($\frac{FP}{FP+TN}$)
- AUC : Area Under Roc. Probabilité qu'un classifieur classe une instance positive au-dessus d'une négative

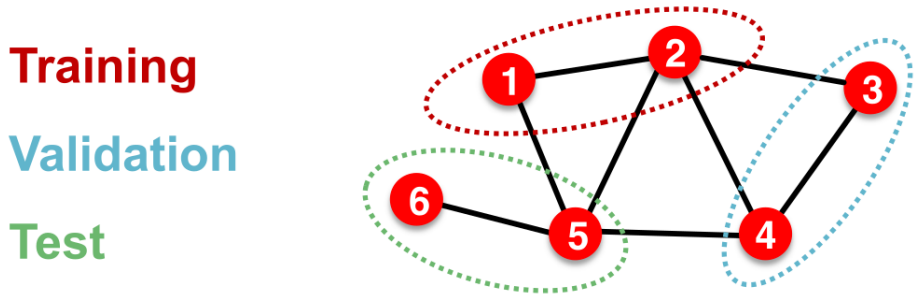


Split de graphe

Pour entraîner notre GNN, il faut découper notre dataset.
On peut faire un split fixe :

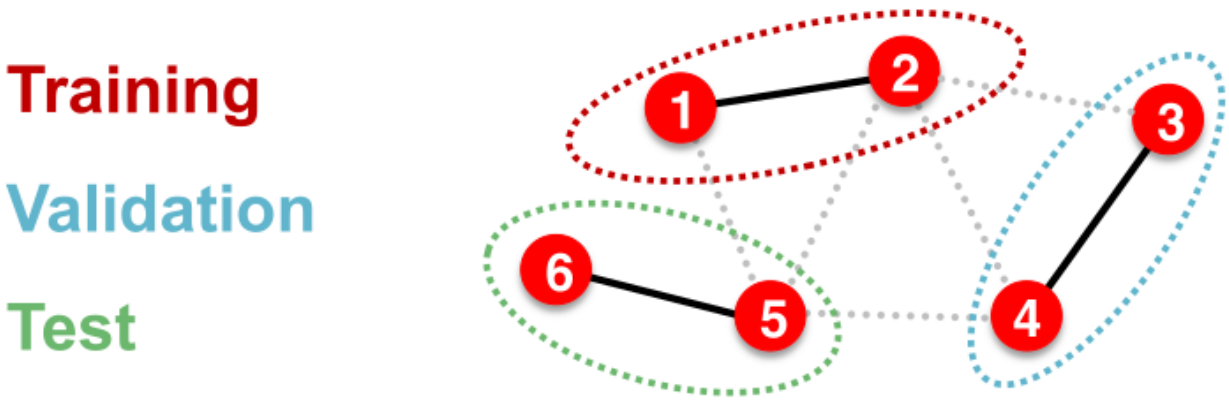
- Training set, pour optimiser les paramètres du GNN
- Validation set, pour améliorer le modèle
- Test set, pour la performance finale

- En imagerie, les instances sont indépendantes
- En graphe : non !



Idées de solution

- Transductif : on garde le graphe entier dans tous les sous-ensembles
  - on splitte seulement les labels
  - on calcule donc les embeddings avec tout le graphe
  - mais on s'entraîne sur une partie des labels
  - à la validation, re-calcul sur tout le graphe, mais évaluation sur d'autres labels
- Inductif : on casse des arêtes, pour avoir des graphes indépendants

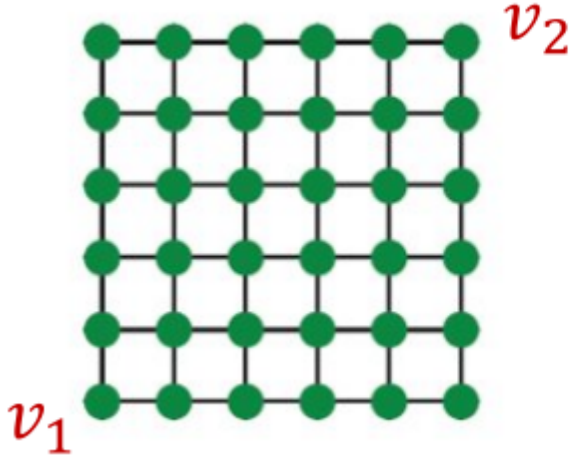


Sujets avancés

Plan

Limites des GNNs

- Deux nœuds avec le même voisinage $\Rightarrow$ même embedding
- Pas le même voisinage $\Rightarrow$ embedding différent
- Pourtant : on peut vouloir tenir compte de la position
  - *Position-aware* GNN



- Les nœuds dans des cycles ont les mêmes embeddings, malgré des longueurs variables
  - *Identity-aware* GNN

Expressivité des GNNs

- L'expressivité des GNN, c'est leur capacité à distinguer des nœuds
- Elle dépend surtout de la manière dont sont agrégés les voisinages
- Les premiers GNN ne parviennent pas à distinguer des structures simples, on peut faire mieux

- Il existe un Graph Isomorphism Network (GIN, Xu et al. ICLR'19) dont on peut montrer que la fonction d'agréagtion est injective :
  - c'est le plus "expressif" des GNN
  - réussit à distinguer la plupart des graphes réels

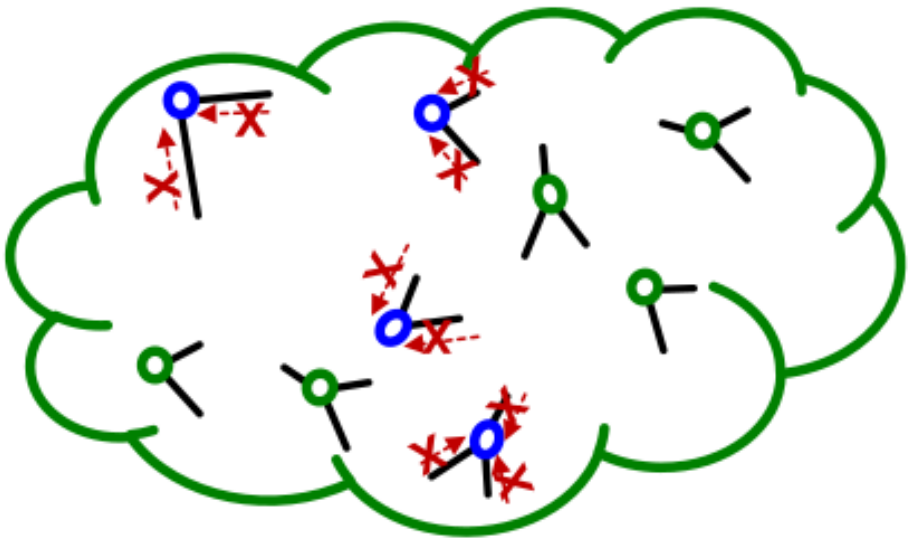
Plan

Passage à l'échelle

- Les GNNs sont utilisés dans des contextes très grande échelle aujourd'hui
  - recommandation (10^8 utilisateurs, 10^7 produits)
  - réseaux sociaux (10^8 utilisateurs)
  - *drug discovery* (10^8 molécules)

Difficulté :

- nœuds isolés si on échantillonne
- SGD standard inefficace



Passage à l'échelle

- Idée naïve : full batch
- charger tout le graphe et les attributs
- chaque couche calcule des embeddings avec tous ceux de la couche d'avant
- Mais : une GPU, c'est 10/20 Go, pas 1 To (Ram)

Approches

- Pré-traitement d'attributs : Simplified GCN
- Passage de message sur des sous-graphes de taille réduite (Cluster-GCN, NeighborSampling)

Références

Remerciements

Ce cours doit beaucoup aux ressources suivantes :

- le cours de Jure Leskovec à Stanford
- le livre Graph Representation Learning de W. L. Hamilton

Version 1.1 du 4 mars 2024