ÉTUDE ET MODÉLISATION DE LA PROPAGATION D'UNE RUMEUR DANS UN RÉSEAU SOCIAL

THÈME TIPE 2018-2019: TRANSPORTS

Problème de maximisation d'influence:

Comment trouver les personnes amenant à la plus grande diffusion de la rumeur ?

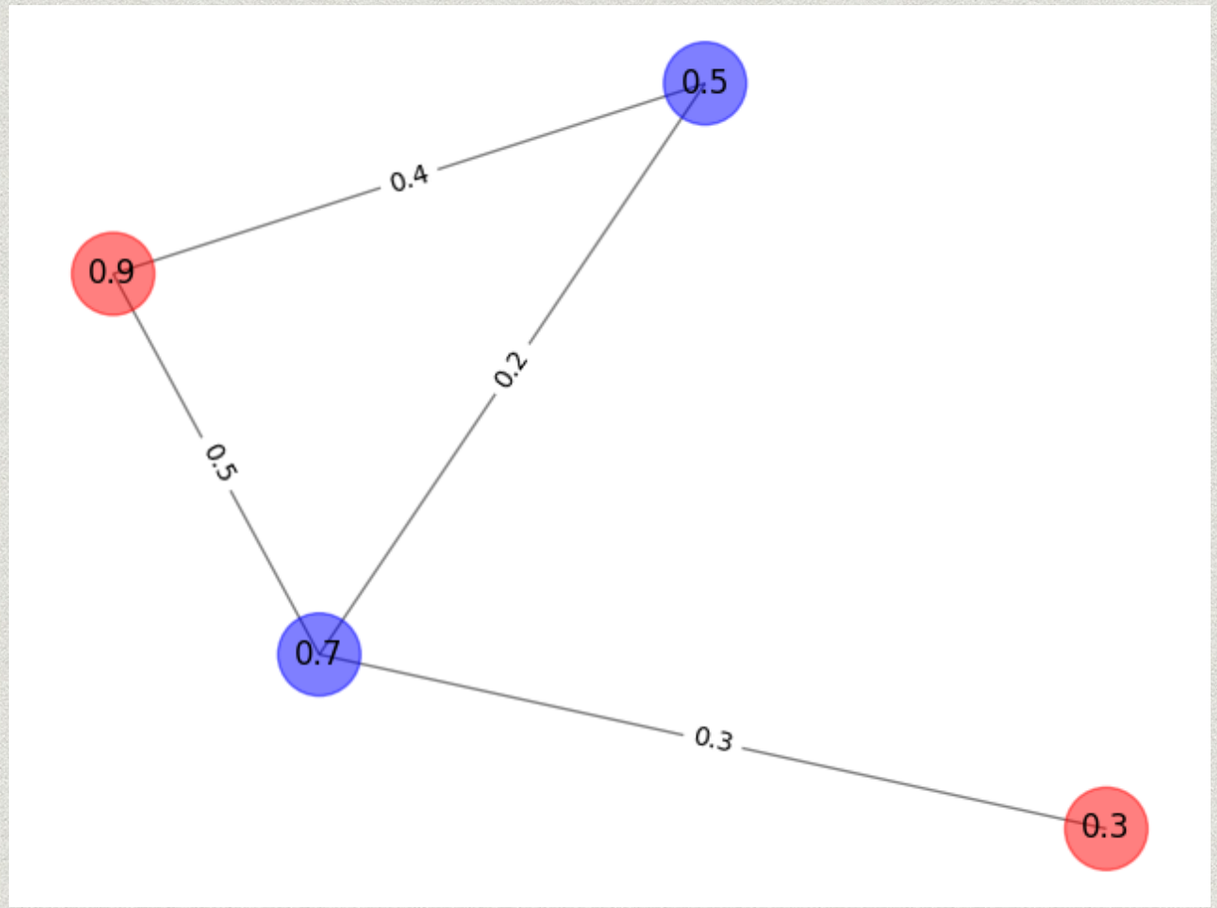
Plan

1. Modélisation du problème
2. L'algorithme glouton
3. Amélioration de l'algorithme glouton
4. Une approche différente
5. Résultats expérimentaux

Modélisation du problème

Propagation

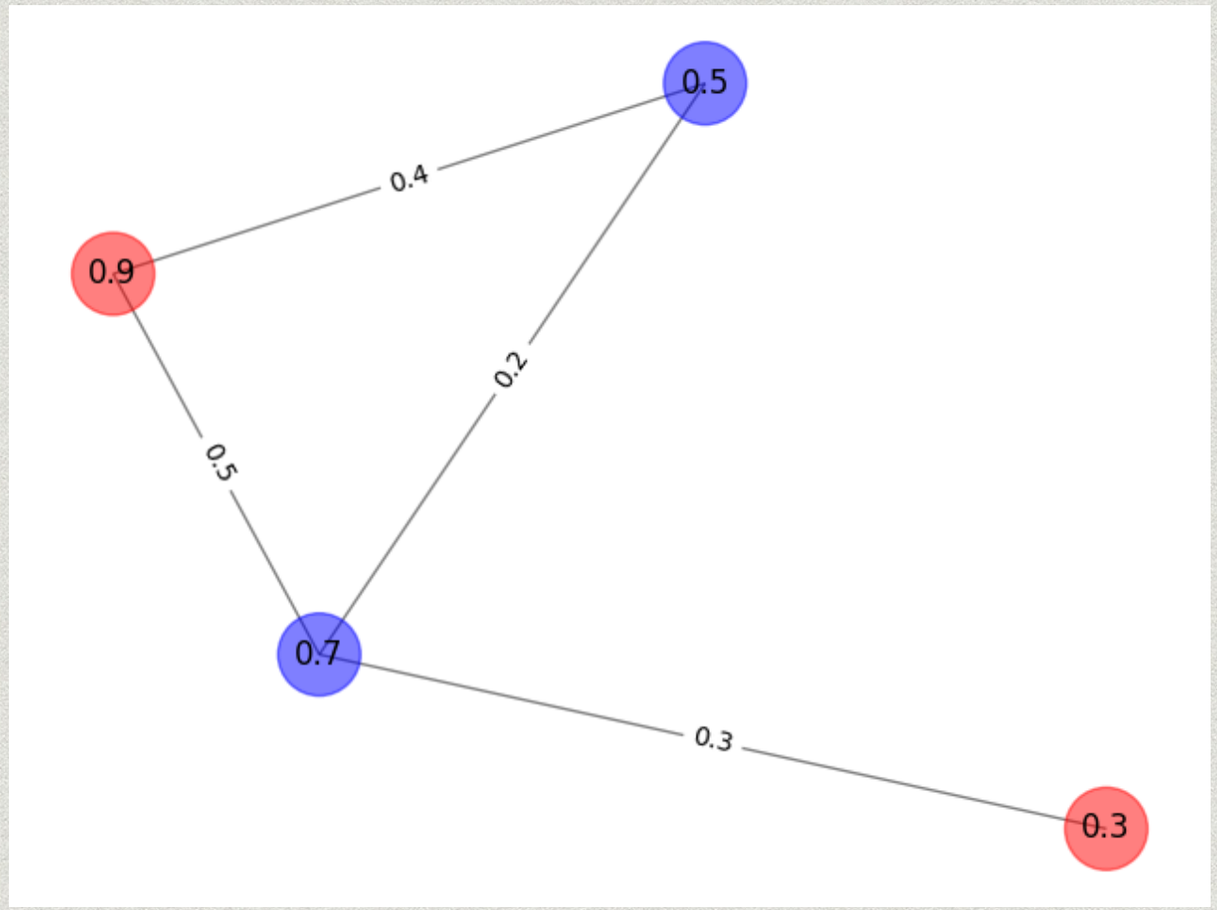
- Linear Threshold (LT) :



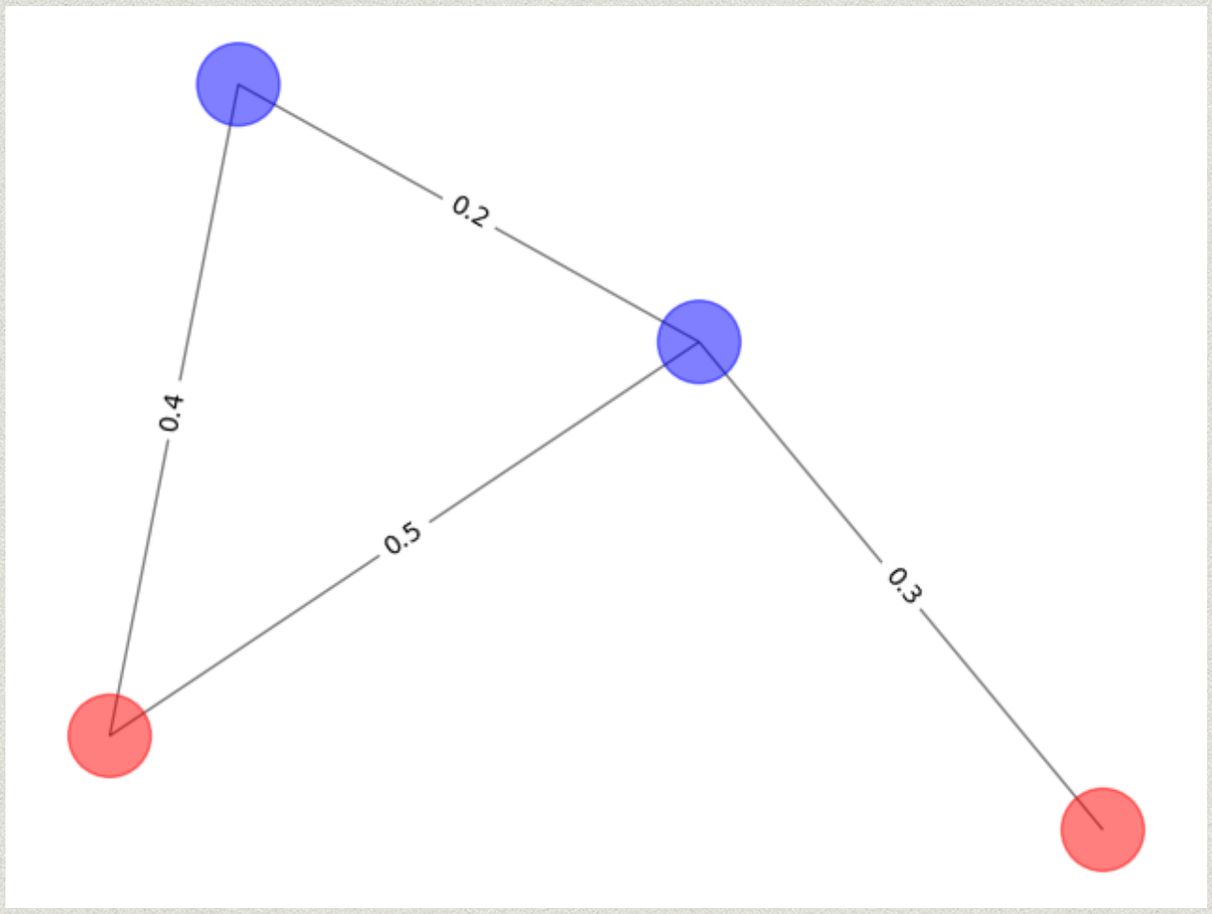
Modélisation du problème

Propagation

- Linear Threshold (LT) :



- Independent Cascade (IC) :



Modélisation du problème

Formalisation

- $G = (V, E)$, $V = [\![1,n]\!]$
- Noeuds actifs à l'instant i : A_i
- Fonction d'influence : $\boxed{\sigma(A) = E(|A_t|)}$, t instant final
- Problème : **Trouver A de taille k qui maximise σ**

Algorithme glouton

Présentation

```
Initialiser S = ∅
Tant que |S| < k:
    Choisir v tel que σ(S ∪ {v}) - σ(S) soit maximal
     S = S ∪ {v}
```

Théorème 1: *σ(S) ≥ (1 - 1/e) σ(O) ≈ 0,63 σ(O)*

Algorithme glouton

Preuve de $\sigma(S) \geq (1 - 1/e)\ \sigma(O)$

f sous-modulaire : $X \subset Y \Rightarrow f(X \cup \{e\}) - f(X) > f(Y \cup \{e\}) - f(Y)$

Théorème 2: f sous-modulaire, positive et croissante $\Rightarrow$

$f(S_{glouton}) \geq (1 - 1/e)\ f(O)$

Théorème 3: σ est sous-modulaire

Amélioration de l'algorithme glouton

Complexité temporelle

```
Initialiser S ← ∅, u ← 0, max ← 0
Tant que |S| < k:                          ← k
   Pour v allant de 1 à n, v∉S:            ← n
      gain ← σ(S ∪ {v}) - σ(S)             ← r × m
      Si gain > max alors u ← v, max ← gain
S ← S ∪ {u}
```

$$O(knrm)$$

Amélioration de l'algorithme glouton

Algorithme 2 : CELF

```
Initialiser S ← ∅, tas ← ∅
Pour v allant de 1 à n:
    insérer ((σ(v), v), tas)
S ← retirer_max(tas)
Tant que |S| < k:
    (gain, noeud_max) ← retirer_max(tas)
    insérer (σ(S ∪ {noeud_max}) - σ(S), tas)
    (nouveau_gain, nouveau_noeud_max) ← max(tas)
    Tant que noeud_max != nouveau_noeud_max :
        (gain, noeud_max) ← retirer_max(tas)
        insérer (σ(S ∪ {noeud_max}) - σ(S), tas)
        nouveau_gain, nouveau_noeud_max) ← max(tas)
    (gain, u) ← retirer_max(tas)
    S ← S ∪ {u}
```

Une approche différente

Algorithme 3 : Degré_max

```
Initialiser S ← ∅, tas ← ∅
Pour v de 1 à n:
    insérer((degre(v), v), tas)
Tant que |S| < k:
    d, u = retirer_max(tas)
    S ← S ∪ {u}
```

Une approche différente

Une amélioration : Degree Discount

- Si v voisin de u actif ⇒ moins intéressant
- Quand u devient actif, $dv \leftarrow 2tv + (dv - tv)tv \times p$
  (tv = voisins actifs, dv = degré de v)

Résultats expérimentaux

Implémentation

Opérations nécessaires CELF , Degree Discount

	Liste	Tas
Ajouter	O(1)	O(log n)
Rechercher max	O(n)	O(1)
Supprimer max	O(n)	O(log n)

⇒ Tas

Résultats expérimentaux

Comparaison des modèles

influence(k), temps(k) sur graphe fb des étudiants d'une université US

Résultats expérimentaux

Comparaison des algorithmes

influence(k), temps(k) pour CELF et greedy => étude uniquement de CELF dans la suite

Résultats expérimentaux

Comparaison des algorithmes

influence(k), temps(k) pour CELF, DegreeDiscount, DegreMax

Conclusion

	CELF	Degree Discount
Besoin d'un résultat optimal	✔	✘
Grand réseau/ temps limité	✘	✔

- **Autres modèles :** General Cascade, General Threshold
- **Paramètres :**

Théorème 2

Lemme : Soient $S_1, \ldots, S_k$ les ensembles produits par l'algorithme glouton
On a $\sigma(S_{i+1}) - \sigma(S_i) \geq 1/k \times (\sigma(O) - \sigma(S_i))$

Preuve : **Par croissance,** $\sigma(S_i \cup O) \geq \sigma(O)$

Or $\sigma(S_i \cup O) = \sigma(S_i) + \sum_{j=1}^{k} \sigma(S_i \cup \left\{o_1, \ldots, o_j\right\}) - \sigma(S_i \cup \left\{o_1, \ldots, o_{j-1}\right\})$

et $\sigma(S_i \cup \left\{o_1, \ldots, o_j\right\}) - \sigma(S_i \cup \left\{o_1, \ldots, o_{j-1}\right\}) \leq \sigma(S_i \cup o_j) - \sigma(S_i)$

Comme $\sigma(S_i \cup o_j) \leq \sigma(S_{i+1})$, **on a** $\sigma(O) \leq \sigma(S_i) + k \times (\sigma(S_{i+1}) - \sigma(S_i))$

Preuve du théorème 2 :

On a donc $\sigma(O) - \sigma(S_i) - (\sigma(O) - \sigma(S_{i+1})) \geq 1/k \times (\sigma(O) - \sigma(S_i))$

$$\sigma(O) - \sigma(S_{i+1}) \leq (1 - 1/k) \times (\sigma(O) - \sigma(S_i))$$

$$\sigma(O) - \sigma(S_k) \leq (1 - 1/k)^k \times \sigma(O)$$

$$\sigma(O) - \sigma(S_k) \leq 1/e \times \sigma(O)$$

et finalement $\sigma(S_k) \geq (1 - 1/e) \times \sigma(O)$

Théorème 3

Triggering Set Model : Pour chaque v, T_v contient les noeuds capables d'activer v

Sous modularité dans ce modèle
IC et LT sont des instances de ce modèle

Diffusion : Linear threshold

```
def diffusion(L, A):
    """
    L : liste d'adjacence contenant les couples (voisins,poids)
    A : ensemble de depart
    renvoie la liste des noeuds actifs à la fin du processus
    """
    n = len(L)
    seuils = sp.random.rand(n)
    somme = [0 for i in range(n)]
    actifs = copy.deepcopy(A)
    est_actif = [False for i in range(n)]
    for u in actifs:
        est_actif[u] = True
    nouveaux_actifs = copy.deepcopy(A)
    #Tant que des noeuds deviennent actifs on continue:
    while len(nouveaux_actifs) > 0:
        prochains_actifs = []
        """Pour chaque noeud qui vient d'être activé
        on essaye d'activer ses voisins"""
        for u in nouveaux_actifs:
            for v, p in L[u]:
                if not est_actif[v]:
                    somme[v] += p
                    if somme[v] >= seuils[v]:
                        est_actif[v] = True
                        actifs.append(v)
                        prochains_actifs.append(v)
        nouveaux_actifs = copy.deepcopy(prochains_actifs)
    return actifs
```

Diffusion : Independent Cascade

```python
def diffusion(L, p, A): #p : probabilité de propagation
    n = len(L)
    actifs = copy.deepcopy(A)
    nouveaux_actifs = copy.deepcopy(A)
    est_actif = [False for i in range(n)]
    for u in actifs:
        est_actif[u] = True
    while len(nouveaux_actifs) > 0:
        prochains_actifs = []
        for u in nouveaux_actifs:
            for v in L[u]:
                if not est_actif[v]:
                    if random.random() <= p:
                        est_actif[v] = True
                        actifs.append(v)
                        prochains_actifs.append(v)
        nouveaux_actifs = copy.deepcopy(prochains_actifs)
    return actifs
```

```python
def sigma(L, p, A, iterations):
    s = 0
    for i in range(iterations):
        influence = len(diffusion(L, p, A))
        s += influence
    return s/iterations
```

```python
def algo_glouton(L, p, k, iterations):
    A = []
    for i in range(k):
        x, influence = noeud_optimal(L, p, A, iterations)
        A.append(x)
    return A
```

```python
def degre_max(L, k):
    n = len(L)
    tas = []
    indices = [0 for i in range(n)]
    choisis = []
    for v in range(n):
        inserer(tas, indices, (degre(L, v), v))
    for i in range(k):
        _, u = retirer_max(tas, indices)
        choisis.append(u)
    return choisis
```

CELF

```
def celf(L, k, p, iterations):
    n = len(L)
    gains_marginaux = []
    indices = [0 for i in range(n)]
    for v in range(n):
        s = (sigma(L, p, [v], iterations), v)
        inserer(gains_marginaux, indices, s)
    influence, noeud_max = retirer_max(gains_marginaux, indices)
    choisis = [noeud_max]
    for i in range(k-1):
        _, noeud_courant = retirer_max(gains_marginaux, indices)
        nouvelle_influence = sigma(L, p, choisis + [noeud_courant], iterations)
        inserer(gains_marginaux, indices, (nouvelle_influence - influence, noeud_courant))
        while gains_marginaux[0][1] != noeud_courant: #Tant que la racine ne reste pas racine
            _, noeud_courant = retirer_max(gains_marginaux, indices)
            nouvelle_influence = sigma(L, p, choisis + [noeud_courant], iterations)
            inserer(gains_marginaux, indices, (nouvelle_influence - influence, noeud_courant))
        influence = nouvelle_influence
        choisis.append(noeud_courant)
        retirer_max(gains_marginaux, indices)
    return choisis
```