Projet Chaînes de Markov : *PageRank*

6 avril 2016

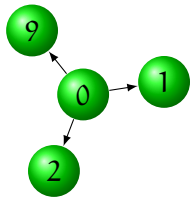
L'une des applications célèbres du modèle des Chaînes de Markov est l'algrithme *PageRank* de Google.
PageRank est l'algorithme d'analyse des liens concourant au système de classement des pages Web utilisé par le moteur de recherche Google. Il mesure quantitativement la popularité d'une page web. Le *PageRank* n'est qu'un indicateur parmi d'autres dans l'algorithme qui permet de classer les pages du Web dans les résultats de recherche de Google.
Son principe de base est d'attribuer à chaque page une valeur (ou score) proportionnelle au nombre de fois que passerait par cette page un utilisateur parcourant le graphe du Web en cliquant aléatoirement, sur un des liens apparaissant sur chaque page. Ainsi, une page a un *PageRank* d'autant plus important qu'est grande la somme des *PageRanks* des pages qui pointent vers elle (elle comprise, s'il y a des liens internes) [1].
Le but de ce projet est de simuler les calculs de *PageRank* en Python sur de petites instances : des nanoWebs.
Remarques sur le projet Le rendu de ce projet sera un ensemble de classes python et un document pdf zippés et nommé suivant les noms des auteurs.
Remarques sur le rapport Le rapport (pdf) contiendra les réponses aux questions qui ne sont pas de l'implémentation ainsi qu'un rapide tour de l'implémentation et des points particuliers soulevés par celle-ci.
Remarques sur l'implémentation Une attention particulière sera portée à la documentation, la qualité et à la robustesse du code fournie. Lors de l'évaluation, le code sera testé sur d'autres exemples que ceux fournis dans l'énoncé.

Principes

On considère le Web comme une collection de N pages ($N > 10^{10}$) reliées entre elles par des liens hyper-texte (On dit qu'une page "pointe" vers ces autres pages). L'idée de base utilisée par les moteurs de recherche pour classer les pages par ordre de pertinence décroissante consiste à considérer que plus une page est la cible de liens venant d'autres pages, c'est-à-dire plus il y a de pages qui pointent vers elle, plus elle a de chances d'être fiable et intéressante pour l'utilisateur final, Il s'agit donc de quantifier cette idée, i.e. d'attribuer un score de pertinence à chaque page.
Attention : On ne tient pas compte des "auto-références" : une page qui pointe vers elle-même.
Pour chaque page i, on appelle $r_i \geq 0$ le score de pertinence de la page. *PageRank* propose comme hypothèse que plus une page est pertinente, plus de pages pointent vers elle. De plus, un lien vers la page contribuera d'autant plus à sa pertinence si il provient lui-même d'une page pertinente. Autrement dit, les scores (r_i) servent (a) à ranger les pages de la plus pertinente à la moins pertinente et (b) à se calculer eux-mêmes récursivement : chaque page distribue son score, uniformément vers sur tous ces liens dont elle est l'origine vers les pages sur lesquelles elle pointe.

1. From `fr.wikipedia.org`

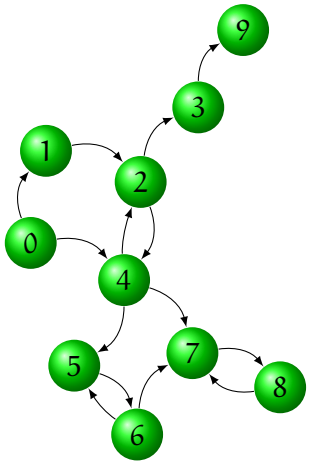
Exemple : soit le sous-graphe suivant d'un nanoWeb. Il indique que sur la page 0, il existe des liens vers les pages 1, 2 et 9.
Supposons que $r_0 = 9$ alors la contribution du nœud 0 aux scores des nœuds 1, 2, et 9 sera de $\frac{1}{3} \cdot r_0 = 3$.



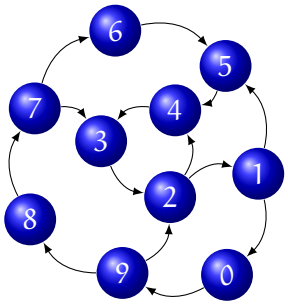
Structures de données

semaine 1

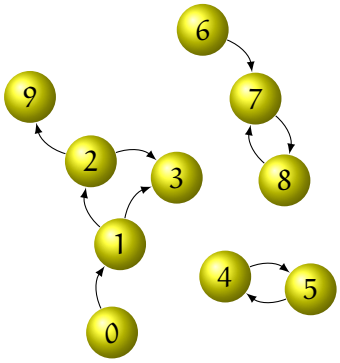
Il s'agit d'implémenter une structure de données qui permettra de représenter correctement ces structures de pages. Par exemple :



nanoWeb 1



nanoWeb 2



nanoWeb 3

Comme nous nous intéresserons aux problèmes de petite taille, nous nous permettons de ne pas choisir entre représentation par graphe ou matrice de transition : nous aurons les 2 représentations simultanément dans nos objets `SimpleWeb`.
On supposera que les graphes sont composés de nœuds (identifié par un entier) et d'arcs (orientés, identifiés par un entier `tail` et un entier `head` : `tail`→`head`).

Question1 – Proposer des implémentations des classes python suivantes : `Node`, `Arc`, `SimpleWeb` :

- `Arc` contiendra au moins une référence vers ses 2 `Node` ainsi que la représentation de la probabilité associé à l'arc,
- `Node` contiendra au moins son identifiant, ainsi qu'une liste d'`Arc` entrants et une liste d'`Arc` sortants du nœud,
- `SimpleWeb` contiendra au moins la liste des nœuds du graphe de transition et une représentation de la matrice de transition.

Ces classes doivent permettre de construire (entre autre) les 3 exemples de nanoWebs comme des instances de `SimpleWeb` :

```
from sys import path
path.append('.')

from datastructures import SimpleWeb

def creeNanoWeb1():
    n=SimpleWeb(10) # 10 noeuds de 0 a 9
    n.addArc(0, 1);n.addArc(0, 4);
    n.addArc(1, 2)
    n.addArc(2, 3);n.addArc(2, 4);
    n.addArc(3, 9)
    n.addArc(4, 2);n.addArc(4, 5);n.addArc(4, 6);
    n.addArc(5, 6)
    n.addArc(6, 5);n.addArc(6, 7);
    n.addArc(7, 8)
    n.addArc(8, 7)

    n.updateTransitionMatrix()
    return n;


if __name__ == "__main__":
    n1=creeNanoWeb1()
    print(n1) # affiche la representation texte
    n1.getGraph("nano1.png") # cree la representation image
```

Les méthodes nécessaires dans `SimpleWeb` sont donc les suivantes :

- `SimpleWeb(int max)` qui construit un CMTD de `max` nœuds (pour l'instant sans arc),
- `AddArc(int tail,int head` qui construit cet arc (sans probabilité !) et qui met à jour les différentes références nécessaires,
- `updateProbas()` qui calcule l'ensemble des probabilités de transition des arcs suivant la proposition faite par *PageRank*,

Remarques sur la robustesse Par exemple, deux `addArc` sur les même nœuds devraient lever une exception ; rajouter un arc doit invalider les probabilités et la matrice de transition, etc.

Question2 – Il existe beaucoup de méthodes pour visualiser un graphe (le format dot (cf. wikipedia par exemple) , les packages networkx, pydotplus, etc.). Proposer des solutions de visualisations en mode texte et en mode graphique (génération d'image) du graphe.
remarque La méthode `__str__()` peut être utilisé pour la représentation en mode texte : elle est appelée par exemple quand on emploie `print(x)`.

Simulations

semaine 2

Le principe du *PageRank* consiste donc, à partir de ces transitions et de leur poids, de définir quelles pages sont les plus "importantes". Il s'agit donc de se demander quelle chance a un internaute se baladant au hasard de tomber sur chacune des pages du web. En effet, plus une page est "importante" dans ce sens là, plus ses propres liens vont rendre les pages accessibles "importantes".
Dans un premier temps, on voudrait donc simuler le déplacement d'un internaute dans notre `SimpleWeb`. Le principe est simple : on place l'internaute dans un nœud initial, puis on utilise les distributions de probabilités de transition pour décider à chaque pas de temps où il se rend. L'idée est alors de garder le nombre de fois où l'internaute passe dans chaque nœud, ce qui permettrait de construire une distribution de probabilités (si convergence).

Question3 – Quels sont les différents comportements que vous pouvez attendre de la part de cette simulation ? En particulier, que pouvez-vous dire des problèmes que rencontrerait une telle simulation sur les 3 exemples de nanoWebs.

Question4 – Implémenter un nouvelle classe `Internaute` qui, à partir d'un `SimpleWeb`, va lancer une navigation. Cette navigation peut prendre fin pour deux raisons différentes : soit on a atteint un nombre limite de navigation soit on a atteint une convergence pour l'estimation de la distribution de probabilité. En notant π_t et π_{t+1} les estimations aux temps t et $t+1$, on peut proposer comme seuil :

$$\epsilon = \max_i |\pi_t(i) - \pi_{t+1}(i)|$$

Question5 – Garder certaines valeurs de ϵ (une toutes les 100 itérations par exemple) afin de produire des courbes indiquant les convergences au cours du temps pour les 3 nanoWebs.

```
from sys import path
path.append('.')

from nanowebs import creeNanoWeb1
from internautes import Internaute

#creation du SimpleWeb
nanoweb=creeNanoWeb1()

# Bob se ballade dans le nanoweb
bob=Internaute(nanoweb)

# bob est dans le noeud 3
bob.goTo(3)


# bob conserve les valeurs de epsilon
# toutes les 100 iterations
# dans ce fichier
bob.trace(100,"epsilons.txt")

# bob se ballade 10000 fois
# ou jusque espilon<0.01
bob.walk(10000,0.01)

# bob affiche la frequence de sa presence
# dans chaque noeud durant sa promenade
bob.showFrequencies()
```

remarque Il sera certainement nécessaire d'enrichir au moins la classe `SimpleWeb`.

Vecteurs-matrices

semaine 2-3

Question6 – La simulation utilisant la classe `Internaute` n'est pas complètement satisfaisante. Pour quelles raisons à votre avis ?

Question7 – Soit π_t la probabilité de se trouver dans chacun des nœuds du graphe ? Expliquer (voire démontrer) la formule suivante (avec P étant la matrice de transition) :

$$\pi_{t+1} = \pi_t \cdot P$$

Question8 – Écrire dans `SimpleWeb` une méthode `nestStep(pi_t)` implémentant ce calcul de π_{t+1} à partir de π_t.
Puis écrire une nouvelle classe de simulation qui, à chaque pas de temps, calcule π_t à partir d'un π_0, d'un nombre d'itérations limite, et toujours d'un seuil ϵ (même API que internaute donc). Afficher les courbes d'ϵ également. Commenter les résultats.
Au lieu de la fonction `goTo`, pourquoi est-il intéressant de pouvoir choisir directement une distribution de probabilité π_0 ?

Puissances de matrices

semaine 2-3

Dans ce cadre, que représente P^2, P^3, P^n, $\lim_{n\to\infty} P^n$?

Question9 – Écrire une méthode dans `SimpleWeb` permettant de vérifier la convergence de la suite des puissances de P en définissant un nouveau type de seuil ϵ pour des matrices. Afficher les courbes d'ϵ également. Commenter les résultats

Génération de Webs

semaine 3

Question10 – Proposer un générateur de `SimpleWeb` ergodique, comparer les temps de calcul des différents algorithmes d'estimation de la distribution stationnaire.