Devoir maison 2

18 mars 2019

Dans ce devoir maison, nous nous intéressons aux algorithmes des graphes. Les graphes peuvent représenter divers type de données comme Facebook ou Twitter. Nous utilisons des données publiques (missing the link).

Un graphe G est représenté en maths sous la forme $G = (V, E)$ où V est l'ensemble des nœuds, E est l'ensemble des liens de la forme (u, v) tels que $u, v \in V$. Les nœuds représentent les entités des données et les liens représentent les interactions entre elles.

En Clojure, nous représentons les graphes sous la forme d'une `Map`, où chaque clé représente un nœud et la valeur représente un dictionnaire de caractéristiques de ce nœud.

Voici une exemple :

```
{0 {:neigh #{1 3}, :close 3.5, :rank 2},
 1 {:neigh #{0 4 3}, :close 4.0, :rank 3},
 2 {:neigh #{3}, :close 2.8333333333333335, :rank 0},
 3 {:neigh #{0 1 2}, :close 4.0, :rank 3},
 4 {:neigh #{1}, :close 2.8333333333333335, :rank 0}}
```

1 Première partie : Génération de graphes

Question 1

Les données sont representées sous la forme de fichier .txt où chaque ligne, sous la forme u v, représente une interaction entre u et v. Il existe donc un lien direct entre ces deux noeuds.

Complétez la fonction `gen-graph`, qui prend en paramètre une sequence de liens sous la forme "u v" et qui renvoie une `Map` qui représente le graphe en question.

Par exemple, la séquence : `("0 1" "2 3" "0 2"])` est représentee par :
`{1 {:neigh #{0}}, 0 {:neigh #{1 2}}, 3 {:neigh #{2}}, 2 {:neigh #{0 3}}}`

Nous vous fournissons la fonction `readfile` qui lit un fichier texte et qui retourne une séquence de chaines representant les liens.

Question 2

Complétez la fonction `erdos-renyi-rnd` qui génère un graphe aléatoire. La fonction prend deux paramètres en argument : n qui est le nombre de nœuds du graphe et p qui est la probabilité d'avoir un lien entre une paire de nœuds.

Un exemple possible, `(erdos-renyi-rnd 5 0.5)` renvoie

```
{1 {:neigh #{0 4 3}}, 0 {:neigh #{1 3}},
 3 {:neigh #{0 1 2}}, 4 {:neigh #{1}}, 2 {:neigh #{3}}}
```

2 Deuxième partie : metriques de centralité

Question 1

Complétez la fonction `degrees` qui prend en argument un graphe et qui ajoute à chaque nœud sa centralité de degré i.e., le nombre de voisins directs du nœud.

```
(let
  [g {1 {:neigh #{0 4 3}},
      0 {:neigh #{1 3}},
      3 {:neigh #{0 1 2}},
      4 {:neigh #{1}},
      2 {:neigh #{3}}}]
  (degrees G))

{1 {:neigh #{0 4 3}, :degree 3}, 0 {:neigh #{1 3}, :degree 2},
 3 {:neigh #{0 1 2}, :degree 3}, 4 {:neigh #{1}, :degree 1},
 2 {:neigh #{3}, :degree 1}}
```

Question 2

La distance naturelle entre une paire de nœuds est définie par la longueur de leur(s) plus court(s) chemin(s). Complétez la fonction `distance` qui prend en arguments un graphe `g` et un nœud n et qui retourne une map avec la distance entre n et les autres nœuds du graphe G.

```

  (let
    [g {1 {:neigh #{0 4 3}},
        0 {:neigh #{1 3}},
        3 {:neigh #{0 1 2}},
        4 {:neigh #{1}},
        2 {:neigh #{3}}}]
    (distance G 1))

  {0 1.0, 4 1.0, 3 1.0, 1 0.0, 2 2.0}

```

Question 3

La centralité de proximité d'un nœud n est définie comme la somme de l'inverse de la longueur des chemins les plus courts entre le nœud n et tous les autres nœuds du graphe. Complétez la fonction `closeness` qui retourne la centralité du nœud.

```
  (let
     [g {1 {:neigh #{0 4 3}},
         0 {:neigh #{1 3}},
         3 {:neigh #{0 1 2}},
         4 {:neigh #{1}},
         2 {:neigh #{3}}}]
      (closeness g 1))
  3.5

```

Question 4

Complétez la fonction `closeness-all` qui calcule la centralité de proximité pour tous les nœuds et qui renvoie le graphe avec les valeurs de centralité.

```
(let
 [g {1 {:neigh #{0 4 3}},
     0 {:neigh #{1 3}},
     3 {:neigh #{0 1 2}},
     4 {:neigh #{1}},
     2 {:neigh #{3}}}]
   (closeness-all g))

{0 {:neigh #{1 3}, :close 3.5}, 1 {:neigh #{0 4 3}, :close 3.5},
2 {:neigh #{3}, :close 2.8333333333333335},
3 {:neigh #{0 1 2}, :close 4.0}, 4 {:neigh #{1}, :close 2.8333333333333335}}

```

Question 5

Complétez la fonction `rank-nodes` qui prend en arguments un graphe `g` et un label l et qui classe les noeuds entre eux par rapport au label l (`:close` ou `:degree`).

```
(let
  [g {1 {:neigh #{0 4 3}},
     0 {:neigh #{1 3}},
     3 {:neigh #{0 1 2}},
     4 {:neigh #{1}},
     2 {:neigh #{3}}}]
  (rank-nodes (C/closeness-all G) :close))

{0 {:neigh #{1 3}, :close 3.5, :rank 2}, 1 {:neigh #{0 4 3}, :close 4.0, :rank 3},
2 {:neigh #{3}, :close 2.8333333333333335, :rank 0},
3 {:neigh #{0 1 2}, :close 4.0, :rank 3},
4 {:neigh #{1}, :close 2.8333333333333335, :rank 0}}


```

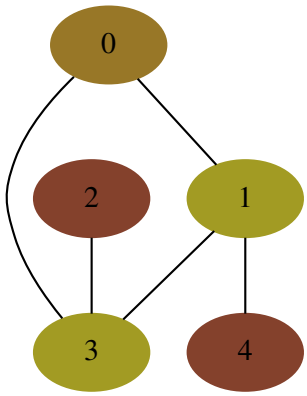
3 Program Princpal

Question 1

Complétez la fonction `to-dot` qui retourne un graphe sous la forme de chaine avec le format de `.dot`. La couleur de chaque neud est defini par sa classement d'importance, vous pouvez utiliser la fonction `generate_colors`.

```
(let
  [g {1 {:neigh #{0 4 3}},
      0 {:neigh #{1 3}},
      3 {:neigh #{0 1 2}},
      4 {:neigh #{1}},
      2 {:neigh #{3}}}]
  (rank-nodes (closeness-all g) :close))

  graph g{
    0 [style=filled color="0.11764705882352941
    0.7450980392156863 0.596078431372549"]
    1 [style=filled color="0.1568627450980392
    0.7843137254901961 0.6352941176470588"]
    2 [style=filled color="0.0392156862745098
    0.6666666666666666 0.5176470588235295"]
    3 [style=filled color="0.1568627450980392
    0.7843137254901961 0.6352941176470588"]
    4 [style=filled color="0.0392156862745098
    0.6666666666666666 0.5176470588235295"]
    0 -- 1
    0 -- 3
    1 -- 4
    1 -- 3
    2 -- 3
  }
```



Question 2

Complétez `-main` pour pouvoir utiliser les differentes fonctions. Nous utilisons le ligne de commande pour passer les differentes arugments.