Université Clermont Auvergne
BUT Info 1A
R207 - Graphes
Année 2021-2022

Florent Foucaud
Malika More
Thibault Ralet
Carine Simon
Thierry Trévisan

TP6 - colorations de graphes

Fonctions de base dans NetworkX (graphes non-orientés)

Tutoriel et documentation de NetworkX en ligne (en anglais).
Dans ce TP quelques fonctions de bases pourraient être utiles :

```
H=nx.Graph()                      #crée un graphe
H.add_edge(0,1)                   #ajoute une arête entre les sommets 0 et 1
H.add_edges_from([(3,0),(3,4)]) #ajoute les arêtes d'une liste donnée
H.add_node("toto")                #ajoute un sommet nommé "toto"
H.remove_node(s)                  #supprime le sommet s
H.nodes                           #sommets du graphe (attention, pour en faire
                                   une vraie liste Python, écrire: list(H.nodes))
H.edges                           #arêtes du graphe (attention, pour en faire
                                   une vraie liste Python, écrire: list(H.edges))
H.edges(s)                        #les arêtes qui touchent le sommet s

H.neighbors(s)                    #un itérateur sur les voisins du sommet s dans H
                                   pour obtenir une liste, écrire: list(H.neighbors(s))
H.nodes[s]["attri"]               #accède à l'attribut nommé "attri" du sommet s
                                   (tant en lecture qu'en écriture)
                                   exemple: H.nodes[s]["attri"]=2
                                   ou alors print(H.nodes[s]["attri"])
```

Exercice 1 (Algorithme glouton simple).

On considère l'algorithme glouton de coloration de graphes suivant. On représente chaque couleur par un nombre entier (au moins 1). On parcourt les sommets, et pour chaque sommet, on lui attribue la plus petite couleur non encore présente parmi ses voisins. Pour stocker les couleurs, on utilisera un attribut “couleur”, par exemple pour colorer le sommet `s` avec la couleur 5 on écrira : `G.nodes[s]["couleur"]=5`.

On peut tester avec ce graphe `H` :

```
H=nx.Graph()
H.add_edges_from([(1,2),(1,5),(2,3),(2,5),(3,4),(4,5),(4,6),(5,0),(0,1),
(0,6),(3,6),(3,5),(5,6)])
```

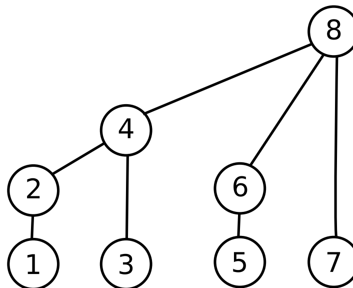
1. Écrire une fonction `plus_petite_couleur_disponible(G,s)` qui prend un graphe `G` et un sommet `s` en entrée, et renvoie la plus petite couleur disponible parmi les voisins de `s`.
2. Écrire une fonction `coloration_gloutonne(G)` qui prend en entrée un graphe `G`, calcule une coloration de façon gloutonne en stockant les couleurs dans l'attribut "couleur" des sommets, et renvoie le nombre de couleurs du graphe. Au début de l'algorithme, on mettra l'attribut "couleur" à 0 pour chaque sommet (couleur non encore définie).
3. Écrire une fonction `afficher_graphe_colore(G)` qui affiche le graphe et les couleurs (sous forme d'entiers) en utilisant l'option `labels=nx.get_node_attributes(G,"couleur")` de la fonction `nx.draw(...)`.

   En option, on pourrait aussi afficher des vraies couleurs, prises au hasard. Pour tirer une couleur au hasard, il faut tirer au hasard trois flottants représentant la proportion de rouge, vert, et bleu, entre 0 et 1 chacun, avec la fonction `random.random()` (il faut faire au préalable un `import random`). Il faudra ensuite associer à chaque numéro de couleur de la coloration trouvée, une couleur différente. Pour dessiner les sommets d'une liste `L` d'une couleur de valeurs `r`, `v`, `b`, on écrit :

```
dico_pos = nx.spring_layout(G)
nx.draw(G,pos=dico_pos,with_labels=True)
nx.draw_networkx_nodes(G, dico_pos, nodelist=L, node_color=[[r,v,b]])
plt.show()
```

Exercice 2 (Algorithme de Welsh-Powell)**.**

L'algorithme glouton vu dans l'exercice précédent peut renvoyer une très mauvaise coloration si on n'a pas de chance sur l'ordre des sommets. C'est le cas sur l'exemple suivant, quand on suit l'ordre donné par les numéros des sommets.

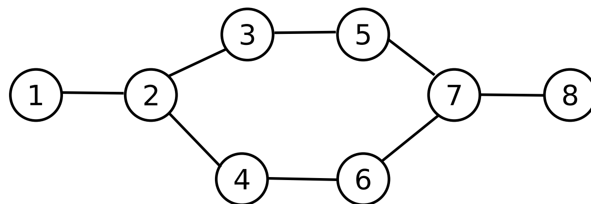


L'algorithme de Welsh-Powell améliore cela en colorant les sommets par ordre décroissant de leur degré.

1. Avec combien de couleurs l'algorithme glouton de l'exercice 1 colorie-t-il l'arbre de la figure ci-dessus ? Et l'algorithme de Welsh-Powell ? Quel est le nombre optimal de couleurs ?
2. Coder une fonction `prochain_sommet_WP(G)` qui, étant donné un graphe partiellement coloré, renvoie le prochain sommet coloré par l'algorithme de Welsh-Powell.
3. Écrire la fonction `WelshPowell(G)` en faisant appel aux fonctions `prochain_sommet_WP(G)` et `plus_petite_couleur_disponible(G,s)`.

Exercice 3 (Algorithme DSATUR de Brélaz)**.**

L'algorithme de Welsh-Powell de l'exercice précédent peut renvoyer une mauvaise coloration pour certains graphes. C'est le cas sur l'exemple suivant.

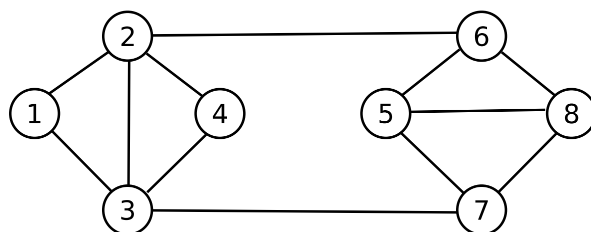


L'algorithme de Brélaz est une modification de celui de Welsh-Powell vu dans l'exercice précédent. Cette fois-ci, on ne considère plus les sommets dans l'ordre décroissant selon les degrés, mais selon l'ordre décroisant du nombre de couleurs différentes qu'ils ont dans leur voisinage. Cette valeur est appelée la *saturation* du sommet. (En cas d'égalité, on privilégie les sommets de plus grand degré.)

1. Avec combien de couleurs l'algorithme de Welsh-Powell de l'exercice 2 colorie-t-il le graphe de la figure ci-dessus ? Et l'algorithme DSATUR ? Quel est le nombre optimal de couleurs ?
2. Écrire la fonction `prochain_sommet_Brelaz(G)` qui correspond à cet algorithme : on calcule itérativement les saturations pour trouver le sommet avec la saturation maximale. En cas d'égalité entre plusieurs sommets à saturations égales (où si aucun sommet n'est encore colorié), on prend un sommet de plus grand degré parmi les sommets possibles. Pour calculer la saturation d'un sommet, on pourra d'abord calculer la liste des couleurs de ses voisins.
3. Écrire et tester la fonction `Brelaz(G)` qui calcule la coloration obtenue par l'algorithme de Brélaz, et renvoie la plus grande couleur utilisée. Utiliser les fonctions `plus_petite_couleur_disponible(G,s)` et `prochain_sommet_Brelaz(G)`.

Exercice 4 (Coloration par recherche exhaustive)**.**

L'algorithme DSATUR de l'exercice précédent peut renvoyer une mauvaise coloration pour certains graphes. C'est le cas sur l'exemple suivant.



En réalité, trouver une coloration optimale de n'importe quel graphe donné en paramètre est un problème difficile et il n'existe aucun algorithme *efficace*[1] pour le faire (on conjecture

1. Un algorithme est communément considéré être effiace si le nombre d'étapes exécutées dans le pire des cas est proportionnel à un polynôme en la taille du graphe.

même qu'un algorithme efficace qui décide si un graphe est 3-coloriable ou pas, n'existe pas : c'est un problème de recherche à 1 million de dollars).

Cependant, un algorithme correct mais très lent [2] consiste à générer exhaustivement toutes les k-colorations possibles d'un graphe (pour un nombre de couleurs k fixé), et à les tester une par une. On en trouvera forcément une si elle existe, sinon, on peut incrémenter k et recommencer.

1. Avec combien de couleurs l'algorithme DSATUR de l'exercice 3 colorie-t-il le graphe de la figure ci-dessus ? Quel est le nombre optimal de couleurs ?
2. Écrire la fonction `est_coloration_valide(G)` qui renvoie `True` si la coloration du graphe `G` est valide, et `False` sinon (il suffit de vérifier si les sommets de chaque arête ont des couleurs différentes).
3. On représente une k-coloration de `G` par une liste de taille n (où n est le nombre de sommets de `G`) où chaque élément a une valeur entre `1` et `k`, représentant la couleur du sommet correspondant. Pour énumérer toutes les colorations possibles, on commence par la coloration `[1,...,1]` pour finir à la coloration `[k,...,k]`. Écrire une fonction `prochaine_coloration(C,k)` qui prend en paramètre la liste `C` et essaie d'augmenter de 1 la couleur la plus à droite de la liste. Si celle-ci était égale à `k`, il faut la passer à `1` et incrémenter la couleur à gauche de celle-ci, et ainsi de suite (un peu comme dans l'algorithme d'incrémentation d'un nombre écrit en base `k`, sauf qu'ici on n'utilise pas le chiffre `0`).
4. Écrire et tester la fonction `existe_coloration(G,k)` qui renvoie `True` s'il existe une coloration valide avec `k` couleurs du graphe `G` (et qui colorie le graphe dans ce cas). On utilisera les fonctions `prochaine_coloration(C,k)` et `est_coloration_valide(G)`.
5. Écrire et tester la fonction `coloration_optimale(G)` qui trouve une coloration optimale du graphe `G` en utilisant la fonction `coloration(G,k)` en faisant varier `k` de `1` au nombre de sommets `G.order()` de `G`.
6. Testez votre fonction sur un graphe aléatoire à `n` sommets obtenu de la façon suivante : `H=nx.gnp_random_graph(n,0.5,seed=None,directed=False)` en faisant varier `n` de 8 à 13. Que constatez-vous ? Comparer avec les algorithmes précédents.

Exercice 5 (Deux couleurs)**.**

Contrairement au cas général, colorier un graphe avec deux couleurs peut être fait de façon efficace. Les sommets d'un graphe peuvent être colorés proprement avec deux couleurs si et seulement le graphe est *biparti*, c'est-à-dire qu'on peut partitionner l'ensemble de sommets en deux parties X et Y de telle façon que toutes les arêtes relient un sommet de X à un sommet de Y. De plus, un graphe est biparti si est seulement tous ses cycles sont de longueur paire.

Le but de cet exercice est de coder un algorithme qui détermine si un graphe donné est biparti.

1. Écrire une fonction `est_biparti(G)` qui renvoie `True` si le graphe `G` est biparti, et `False` sinon. Pour cela on peut parcourir le graphe (avec un parcours de notre choix, de préférence un parcours en largeur pour faciliter la question 2) et successivement

2. Le nombre d'étapes exécutées sera exponentiel en la taille n du graphe, environ k^n.

mettre les sommets dans une liste X et une liste Y (si on considère un sommet X, tous ses voisins seront dans Y, et vice-versa). Si on détecte une arête $X - X$ ou $Y - Y$, la réponse est `False`. Testez-la avec le graphe `G` de l'exercice 1, ainsi qu'avec un graphe biparti aléatoire obtenu avec `nx.bipartite.random_graph(10,10,0.3, seed=None, directed=False)`

2. Modifier la fonction pour qu'elle affiche un *cycle impair* dans le cas où la réponse est `False`. Pour cela, lors du parcours en largeur, on peut se souvenir du prédécesseur de chaque sommet en créant un dictionnaire des prédécesseurs (comme dans l'exercice 5 du TP 3). Lorsqu'on détecte une arête qui connecte deux sommets u, v tous les deux dans X ou tous les deux dans Y, on remonte les prédécesseurs en parallèle, jusqu'à tomber sur le même sommet. L'ensemble des prédécesseurs et les sommets u, v forment un cycle impair.