TP 13 : Piles et files, parcours en largeur de graphe

Télécharger l'annexe sur le site web. Elle contient des importations de modules et des exemples de graphes.

1 La structure `deque`

Présentation. Le module `collections` propose notamment la structure `deque`, pour *double ended queue*, c'est-à-dire en français *file à deux bouts*. Concrètement, c'est une structure de stockage linéaire des éléments (comme une liste !), mais qui fournit la possibilité d'effectuer l'insertion ou la suppression des deux côtés en temps constant.

Opérations. Dans la suite, `f` désigne une `deque`.

- Création : `deque()` pour une deque vide, `deque(x)` pour convertir un objet `x` (une liste par exemple) en deque vide.
- `len(f)` : nombre d'éléments. `len(f)==0` pour tester que la deque est vide.
- `f[i]` accès au i-ème élément, comme pour une liste standard : attention, l'accès est de coût constant aux extrémités mais de coût linéaire au milieu !
- `f.append(x)`, `f.appendleft(x)` : ajout de `x` à droite ou à gauche dans une deque, en temps constant.
- `f.pop()`, `f.popleft()` : retrait de l'élément à droite ou à gauche d'une deque non vide, en temps constant. S'évalue en l'élément supprimé.

Piles et files avec `deque`. Une pile suit le principe LIFO (last in, first out) : utiliser une deque avec seulement `append` et `pop` donne une structure de pile. Inversement, une file suit le principe FIFO (first in, first out) : on peut l'implémenter avec une deque en utilisant `append` et `popleft`, par exemple.

Exercice 1. *Temps d'accès.*

1. La fonctionne `randrange(a,b)` du module `random` (importée dans l'annexe) fournit un entier aléatoire de $[\![a, b[\![$. Écrire une fonction `randL(N)` générant une liste aléatoire de taille N, dont les entiers sont dans $[\![0, 10^6[\![$. Écrire de même une fonction `randD(N)` générant une deque aléatoire de taille N, dont les entiers sont dans $[\![0, 10^6[\![$.
2. On rappelle que `process_time` (déja importée) permet de mesurer le temps d'exécution d'une suite d'instructions comme ceci :

```
t = process_time()
[instructions]
t2 = process_time() - t #temps d'exécution de la suite d'instructions.
```

En utilisant `process_time`, écrire une fonction `test_acces(N)` générant aléatoirement une `deque` de taille N, et mesurant le temps nécessaire pour accéder un million de fois à l'élément central, d'indice $\lfloor N/2 \rfloor$ (attention à ne pas compter le temps de création de la deque !). Vérifier que ce temps est proportionnel à N en testant avec $N = 10^3, 10^4$ et 10^5.

```
>>> test_acces(10**3)
0.08014732499999999
>>> test_acces(10**4)
0.14801614199999996
>>> test_acces(10**5)
3.0611087969999997
```

Modifier votre fonction pour vérifier qu'avec une liste, ce temps reste constant. Vérifier toutefois qu'avec une `deque` et l'indice 0 ou l'indice $N-1$, le temps reste constant.

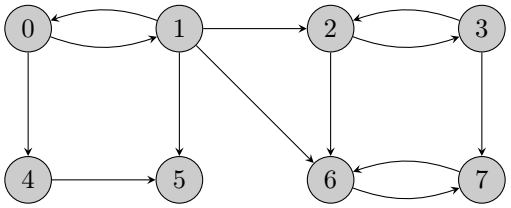
2 Parcours en largeur

On rappelle en pseudo-code le parcours en largeur depuis un sommet source s, il renvoie toutes les distances depuis le sommet s sous la forme d'une liste.

Algorithme 1 : Parcours en largeur

Entrée : Un graphe G donné par listes d'adjacence, un sommet de départ s
`a_traiter` $\leftarrow \{s_0\}$# une file;
`D` $\leftarrow [\infty, \dots, \infty]$# n fois $+\infty$;
`B`$[s] \leftarrow 0$;
tant que *`a_traiter` est non vide* **faire**
  $u \leftarrow$ sortir un élément de `a_traiter`;
  pour *tout voisin v de u tel que `D`$[v]$ est $+\infty$* **faire**
    `a_traiter` $\leftarrow$ `a_traiter` $\cup \{v\}$;
    `D`$[v] \leftarrow 1 +$ `D`$[u]$
Renvoyer D

Le graphe `G1` défini dans l'annexe est le suivant.



Exercice 2. *Distances depuis s.* Compléter le squelette de l'annexe pour implémenter le parcours en largeur. Pour gérer la file `a_traiter`, on utilisera une deque sous forme de file, comme expliqué plus haut. L'infini est encodé par le flottant infini, qu'on peut obtenir via `float('inf')`. Tester avec le graphe de l'annexe.

```
>>> parcours_largeur(G1, 0)
[0, 1, 2, 3, 1, 2, 2, 3]
>>> parcours_largeur(G1, 2)
[inf, inf, 0, 1, inf, inf, 1, 2]
```

Exercice 3. *Calcul des prédecesseurs.* Reprendre votre parcours en largeur pour calculer en même temps une *liste de prédecesseurs* : on dit que u est le prédecesseur de v dans le parcours en largeur si v est découvert pour la première fois dans la liste d'adjacence de u (et on a donc $D[v] = D[u] + 1$). Votre fonction renverra donc un couple (D, P), où D est la liste des distances, et P la liste des prédecesseurs : à la fin du parcours, $P[v]$ est le prédecesseur de v (on pourra initialiser une liste contenant uniquement des -1, à la fin du parcours on aura toujours $P[v] = -1$ si $v = s$ ou si v n'est pas accessible depuis s).

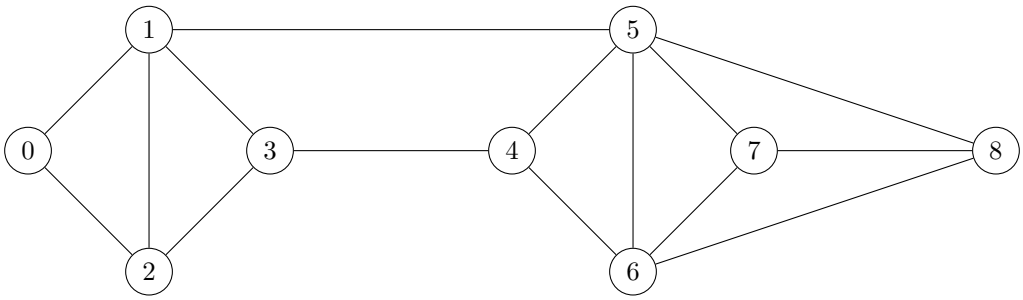
```
>>> parcours_largeur(G1, 0)
([0, 1, 2, 3, 1, 2, 2, 3], [-1, 0, 1, 2, 0, 4, 1, 6])
>>> parcours_largeur(G1, 2)
([inf, inf, 0, 1, inf, inf, 1, 2], [-1, -1, -1, 2, -1, -1, 2, 6])
```

Exercice 4. *Calcul effectif de plus courts chemins.* À partir de la liste P précédente, il est possible de calculer effectivement des plus courts chemins : pour calculer le plus court chemin de s à v (avec v accessible depuis s), il suffit de remonter de prédecesseur en prédecesseur de v jusqu'à s grâce à la liste P : on obtient alors le chemin à l'envers. Écrire une fonction `chemin(P,s,v)` prenant en entrée la liste des prédecesseurs dans un parcours en largeur d'un graphe depuis s, les sommets s et v, et renvoyant (dans l'ordre !) un plus court chemin de s à v.

```
>>> chemin(parcours_largeur(G1,0)[1], 0, 7)
[0, 1, 6, 7]
```

3 Application du parcours en largeur

Pour l'exercice suivant, le graphe G_2 de l'annexe pris en exemple est celui-ci :



Exercice 5. *Excentricité, diamètre, rayon, centre, périphérie*. Soit $G = (V, E)$ un graphe non orienté, qu'on suppose connexe (c'est-à-dire qu'il existe toujours un chemin entre deux sommets quelconques du graphe). Pour $(u, v) \in V^2$, on note $\delta(u, v)$ la longueur d'un plus court chemin entre u et v, qui existe bien car G est connexe. On introduit les définitions suivantes :

— l'*excentricité* d'un sommet u est $\max\{\delta(u, v) \mid v \in V\}$.

— le *rayon* d'un graphe est l'excentricité minimale de ses sommets. Les sommets ayant une excentricité égale au rayon en forme le *centre*.

— le *diamètre* d'un graphe est l'excentricité maximale de ses sommets. Les sommets ayant une excentricité égale au diamètre en forme la *peripherie*.

On suppose le graphe représenté par listes d'adjacence. On utilisera la fonction `parcours_largeur(G,s)` précédente.

1. Écrire `excentricite(G,s)` calculant l'excentricité du sommet s. Quelle est sa complexité ?

```
>>> excentricite(G2, 1)
2
>>> excentricite(G2, 7)
3
```

2. En déduire `rayon_centre(G)` calculant un couple (r, C) où r est le rayon et C le centre du graphe (sous forme de liste de sommets). Quelle est sa complexité ?

```
>>> rayon_centre(G2)
(2, [1, 5])
```

3. Même chose avec `diametre_peripherie(G)`.

```
>>> diametre_peripherie(G2)
(3, [0, 2, 3, 4, 6, 7, 8])
```

4 Manipulation de piles : expressions bien parenthésées

Exercice 6. *Expression bien parenthésée*. Une expression bien parenthésée est une expression qui comporte une succession cohérente de parenthèses ouvrantes et fermantes. Par exemple, l'expression suivante est bien parenthésée :

$$(2 + 5) \times (((11 - 2)/3) \times (1 + 4))$$

et celle-ci ne l'est pas :

$$(2 + (5 \times (11)))) - 2)/3(\times(1 + 4))$$

Afin de vérifier si une expression est bien parenthésée, on souhaite construire une fonction prenant comme argument la chaîne de caractères représentant cette expression. La chaîne de caractère est parcourue de gauche à droite : lorsqu'une parenthèse ouvrante est découverte, on empile un jeton sur une pile créée à cet effet ; lorsqu'une parenthèse fermante est découverte, on tente de dépiler un jeton de la pile.

1. Appliquer cet algorithme aux expressions proposées ci-dessus. Quel est le contenu de la pile à la fin de l'algorithme pour une expression bien parenthésée ? Cette condition est-elle nécessaire pour qu'une expression soit bien parenthésée ? Cette condition est-elle suffisante pour qu'une expression soit bien parenthésée ?
2. Écrire une fonction `bien_parenthese(s)` prenant comme argument une chaîne de caractères `s` (représentant une expression), et retournant le booléen `True` si l'expression est bien parenthésée, `False` sinon. On pourra utiliser une deque ou une liste classique pour simuler une pile (en utilisant les méthodes `append` et `pop`).

```
>>> bien_parenthese('(2 + 5) * (((11 - 2)/3)*(1 + 4))')
True
>>> bien_parenthese('(2 + (5 * (11)))) - 2)/3(*(1 + 4))')
False
>>> bien_parenthese('(()')
False
>>> bien_parenthese('(())')
True
>>> bien_parenthese('())(')
False
```

3. La notion de parenthésage n'est pas limitée aux seules parenthèses, mais englobe également l'utilisation des crochets et des accolades. Écrire une fonction `bien_parenthese_complet(s)` prenant comme argument une chaîne de caractères `s` (représentant une expression pouvant contenir des parenthèses et/ou des crochets et/ou des accolades), et retournant le booléen `True` si l'expression est bien parenthésée, `False` sinon. Naturellement une expression est bien parenthésée si les couples sont bien imbriqués.
Indication : lorsqu'on rencontre un caractère ouvrant (parenthèse, crochet, accolade), on pourra l'empiler. Lorsqu'on rencontre un caractère fermant, on pourra vérifier que la pile est non vide, et que le sommet est le caractère ouvrant correspondant.

```
>>> bien_parenthese_complet('{}()[()]')
True
>>> bien_parenthese_complet('({})')
True
>>> bien_parenthese_complet('({)}')
False
```