TP5: Graphes

MP2I Option SI: Informatique tronc commun

1 Dictionnaire d'adjacence

Dans tout ce TP, on manipulera des graphes représentés par liste d'adjacence. On rappelle que pour un graphe $G = (S, A)$, avec les sommets numérotés $s_0, \ldots, s_{n-1}$, la représentation de G par listes d'adjacence est un tableau L de taille n tel que la case $L[i]$ contient les voisins de s_i pour $i \in [\![0, n-1]\!]$. Cependant, utiliser directement des tableaux nécessite de passer sans cesse des indices aux sommets et inversement. En pratique, on peut utiliser un dictionnaire plutôt qu'un tableau. Par exemple, considérons le graphe suivant :

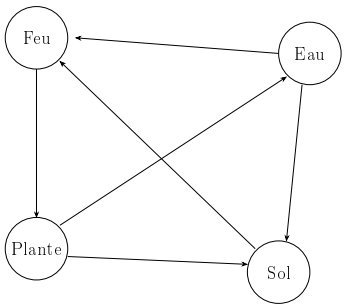


FIGURE 1 – Efficacité des types Pokémon

On peut utiliser un dictionnaire pour représenter ce graphe en Python :

```
g_pkmn = {
  "Feu":["Plante"],
  "Plante":["Eau", "Sol"],
  "Eau":["Feu", "Sol"],
  "Sol":["Feu"]
}
```

Ainsi, `g_pkmn[t]` donne la liste des successeurs de `t` (ou des voisins pour un graphe non-orienté).

Il est fréquent lorsque l'on utilise des graphes d'avoir seulement à disposition la liste des arêtes/arcs. Par exemple, sur le graphe suivant, la liste des arcs serait :

```
aretes_pkmn = [
  ("Feu", "Plante"),
  ("Plante", "Eau"),
  ("Plante", "Sol"),
  ("Eau", "Feu"),
  ("Eau", "Sol"),
  ("Sol", "Feu")
]
```

Cette représentation est cependant bien moins maniable que les listes d'adjacences, car moins structurée.

Q1. Écrire une fonction `construire_adj(sommets, aretes, oriente)` qui prend en entrée une liste `sommets`, une liste de couples `aretes` et un booléen `oriente`, et construit le dictionnaire d'adjacence correspondant au graphe dont les sommets sont `sommets`, dont les arêtes sont celles de la liste `aretes` et étant orienté ou non selon la valeur du booléen `oriente`. Votre fonction fera attention à ne pas créer de doublons dans la liste d'adjacence, même s'il y en a dans la liste initiale des arêtes.

Dans la suite, on suppose que les graphes que l'on manipule sont toujours représentés par dictionnaires d'adjacence.

Q2. Implémenter les deux fonctions suivantes :

- `liste_voisins(g, u)` qui renvoie la liste des successeurs / voisins de u dans le graphe g ;
- `est_arete(g, u, v)` qui renvoie un booléen indiquant si (u, v) est une arête de g.

Q3. Écrire une fonction qui calcule le degré moyen des sommets d'un graphe non-orienté.

Q4. Dessiner sur une feuille un graphe satisfaisant les conditions suivantes :

1. Non-orienté
2. Exactement 10 sommets, nommés $A, B, C, \ldots$
3. Exactement 12 arêtes
4. Plusieurs composantes connexes
5. Pas de boucle (i.e. pas d'arête de la forme (x, x))

Q5. Représenter ce graphe dans votre programme. Il vous servira pour tester les fonctions que nous allons implémenter.

Q6. Dessiner une copie **orientée** de votre graphe, en choisissant pour chaque arête une orientation quelconque. Représenter ce graphe dans votre programme.

2 Parcours de graphe

On rappelle que le parcours en profondeur d'un graphe à partir d'un sommet source s'effectue à l'aide d'une pile, selon l'algorithme suivant :

Algorithme 1 : `Parcours_profondeur_source`(G, V)

Entrée(s) : $G = (S, A)$ graphe orienté, $s \in S$ sommet de départ, V ensemble des sommets déjà traités.

Sortie(s) : Traite tous les sommets accessibles à partir de s dans G, et les rajoute à V.

```
1 P ← pile_vide() ;
2 P.empiler(s) ;
3 V.ajouter(s);
4 tant que P non vide faire
5     u ← P.depiler();
      // Traiter u (dépend de l'application)
6     pour v voisin de u faire
7         si V ne contient pas v alors
8             V.ajouter(v);
9             P.empiler(v);
```

Ensuite, le parcours en profondeur d'un graphe s'effectue en lançant l'algorithme précédent depuis chaque sommet, en gardant en mémoire les sommets déjà parcourus afin de ne pas lancer de parcours depuis un sommet vu précédemment :

Algorithme 2 : `Parcours_profondeur`(G)

Entrée(s) : $G = (S, A)$ graphe orienté.

Sortie(s) : Traite tous les sommets de G.

```
1 V ← ∅// sommets vus
2 pour s sommet de G faire
3     si V ne contient pas s alors
4         Parcours_profondeur_source(G, s);
```

Notons que les listes python peuvent directement servir de piles : si `L` est une liste, alors `L.pop()` dépile et renvoie le dernier élément de `L`, et `L.append(x)` empile `x` à la fin de `L`.

Enfin, on rappelle l'existence des **ensembles** en Python :

```python
e = set() # création d'un ensemble vide
e.add(5) # ajout d'un élément
if 5 in e: # vérifier si l'ensemble contient un élément
  print("ok")

e1 = set([1, 2, 3]) # création d'un ensemble à partir d'une liste
e2 = set([2, 4, 6])
e3 = e1.union(e2) # union ensembliste
```

Q7. Écrire deux fonctions `afficher_sommets_source(g, s)` et `afficher_sommets(g)` utilisant un parcours en profondeur, affichant les sommets du graphe g dans l'ordre de visite. Les tester sur des graphes non-orientés et orientés.

Q8. Selon le même principe de parcours en profondeur, écrire une fonction permettant de calculer les composantes connexes d'un graphe non-orienté. Votre fonction renverra un dictionnaire associant à chaque sommet le numéro de sa composante connexe. On pourra passer par la fonction suivante de parcours à partir d'une source :

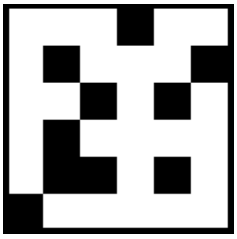
```
def parcours_composante(G, s, C, k):
    """
    Entrées:
    - G graphe
    - s sommet de G
    - C dictionnaire associant à chaque sommet de G un entier
    - k un entier
    Sortie:
    Parcours G à partir de s, et pour chaque sommet u visité,
    fixe C[u] = k.
    """
```

3 Résolution de labyrinthes, plus court chemin

On considère des labyrinthes constitués de grilles rectangulaires. Dans un tel labyrinthe, chaque case (i, j) peut être libre où occupée par un mur. On peut passer d'une case libre à n'importe quelle autre case libre adjacente. On représentera un labyrinthe par un tableau de booléens T, avec $T[i][j]$ indiquant si la case contient un mur. Une case libre sera donc représentée par la valeur `False`.

On stockera les labyrinthes sous format texte en représentant les espaces libres par des espaces et les murs par des "`X`". On indiquera également l'entrée du labyrinthe par un "`E`" et la sortie par un "`S`". Des exemples sont donnés dans l'archive du TP.

Voici un exemple de labyrinthe (l'entrée est en haut à gauche et la sortie en bas à droite), et le contenu du fichier texte correspondant :



```
E  X
 X   X
  X X
 X
 XX X
X    S
```

Nous allons utiliser les parcours de graphe pour résoudre des labyrinthes, et même y trouver des chemins les plus courts possible.

Lecture depuis un fichier

Tout comme en C, on peut ouvrir, lire, écrire, et fermer des fichiers en Python. Nous allons utiliser les 3 fonctions suivantes :

— `f = open(fn, "r")` ouvre le fichier de nom `fn` en mode lecture, et stocke dans f un pointeur permettant de lire/écrire dans le fichier

— `f.close()` ferme le fichier f

— `l = f.readlines()` renvoie la liste des lignes du fichier f. Attention, cette instruction avance jusqu'à la fin du fichier : si vous l'utilisez deux fois d'affilée, la deuxième lecture renverra une liste vide.

Par exemple, le code suivant affiche le contenu d'un fichier, ligne par ligne :

```
1 fn = "mon_fichier.txt"
2 f = open(fn, "r")
3 lignes = f.readlines()
4 for l in lignes:
5   print(l)
6 f.close()
```

Q9. Écrire une fonction `lire_labyrinthe(filename)` prenant en entrée un nom de fichier et allant y lire un labyrinthe sous le format précisé plus haut. La fonction renverra un triplet `(lab, e, s)` contenant le labyrinthe lu (sous forme d'un tableau de booléens), et les coordonnées de l'entrée et de la sortie. Attention, l'entrée et la sortie sont aussi des cases libres !

Q10. Écrire une fonction `graphe_labyrinthe(lab)` qui prend en entrée un tableau de booléens représentant un labyrinthe et renvoie le graphe correspondant, dont les sommets sont les cases libres du labyrinthes, et dont les arêtes représentent les chemins que l'ont peut emprunter à partir d'une case (pensez à utiliser la fonction `construire_adj`).

Recherche de chemin Passons maintenant à la résolution de labyrinthe. Le problème de graphe sous-jacent est la recherche d'un chemin dans un graphe : étant donné un graphe $G = (S, A)$ et $u, v \in S$ deux sommets , existe t-il un chemin entre u et v, c'est à dire une suite de sommets $u_0 u_1 \dots u_k$ avec $u_0 = u$, $u_k = v$ et $(u_i, u_{i+1}) \in A$?

On peut même s'intéresser à la recherche d'un chemin le plus court possible entre ces deux points, c'est à dire passant par le moins d'arêtes possible. On rappelle que le parcours en largeur à partir d'un sommet permet de résoudre ce problème. Plus précisément, lancer un parcours en largeur à partir d'un sommet s permet de calculer le chemin le plus court depuis s jusqu'à n'importe quel sommet :

Algorithme 3 : Parcours en largeur : calcul des distances

Entrée(s) : $G = (S, A)$ graphe non-orienté, $s \in S$ sommet de départ
Sortie(s) : **Pred** dictionnaire des prédécesseurs dans les chemins depuis s dans G

```
1  Pred ← dictionnaire_vide() // dictionnaire des prédécesseurs
2  F ← file_vide() ;
3  F.enfiler(s) ;
4  Pred[s] ← None ;
5  tant que F non vide faire
6      u ← F.defiler();
7      pour v voisin de u faire
8          si Pred ne contient pas v alors
9              Pred[v] ← u;
10             F.enfiler(v);
```

A la fin de l'exécution de cet algorithme, **Pred**$[u]$ contient le sommet précédant u dans un plus court chemin depuis s. On peut donc l'utiliser pour reconstruire un plus court chemin de s à u : on part de u, et on remonte vers s en suivant les arêtes indiquées par **Pred**.

Plutôt que de recoder nos propres files, nous pouvons utiliser celles fournies par Python. Le module `collections` contient plusieurs structures de données, dont la `deque` (**d**ouble-**e**nded **que**ue), dans la quelle on peut ajouter et extraire des éléments à gauche et à droite. Voici un exemple d'utilisation de cette structure :

```
1 from collections import deque
2 F = deque() # création d'une deque vide: []
3 F.append(5) # ajout à droite: [5]
4 F.appendleft(6) # ajout à gauche: [6, 5]
5 F.append(8) # ajout à droite: [6, 5, 8]
6 x = F.popleft() # extrait à gauche: donne 6 [5, 8]
7 y = F.popleft() # extrait à gauche: donne 5 [8]
8 z = F.pop() # extrait à droite: donne 8 []
```

En n'utilisant que `popleft` et `append`, on peut donc simuler une file !

Q11. Implémentez l'algorithme de parcours en largeur donné plus haut, et écrivez une fonction `predecesseurs(g, s)` qui calcule et renvoie le dictionnaire des prédecesseurs **Pred**.

Q12. Implémentez une fonction `reconstruire_chemin(s, u, pred` qui calcule un plus court chemin entre s et u, en utilisant le tableau **Pred** calculé par le parcours en largeur.

Q13. Vérifiez vos fonctions en trouvant des chemins les plus courts possibles sur les labyrinthes donnés dans l'archive du TP.

4 Visualisation de graphe, fichier CSV

Q14. Écrire une fonction `plot_graph(g, positions)` prenant en entrée un graphe `g` représenté par listes d'adjacences, et un dictionnaire `positions` dont les clés sont les sommets du graphe et dont les valeurs sont les positions des sommets dans le plan, et qui affiche le graphe (en utilisant `matplotlib.pyplot`).

Q15. Testez votre fonction avec les labyrinthes. Afin de respecter l'orientation, on mettra la case (i, j) à la position $(j, -i)$ dans le plan.

Le fichier "communes_france.csv" de l'archive du TP est un tableur contenant le nom et les coordonnées GPS de 568 communes françaises de plus de 12000 habitants [1]. Un fichier CSV (Comma Separated Values) contient des valeurs séparées par des virgules, chaque colonne correspondant à un attribut, et chaque ligne correspondant à un enregistrement. Par exemple, dans le fichier fourni, chaque ligne correspond à une ville et contient 3 valeurs : le nom, la latitude, la longitude. Il existe des modules python spécialisés permettant de lire de tels fichiers (le module `csv` par exemple), mais il n'est pas très dur de le faire à la main.

Q16. Écrire une fonction `infos_ville(l)` prenant en entrée une ligne de texte contenant un nom de commune et ses deux coordonnées, le tout séparé par des virgules, et qui renvoie un triplet `(nom, lat, lon)`. Vous pouvez vous renseigner sur la fonction `split` qui permet de séparer une chaîne de caractères.

Q17. Écrire une fonction `lire_csv(fn)` qui lit dans le fichier nommé `fn`, dont on suppose qu'il contient des données formatées comme "communes_france.csv", et qui renvoie un dictionnaire, dont les clés sont les noms des villes, et les valeurs les positions.

Nous allons utiliser ces données pour former un graphe. Les sommets seront les différentes villes, et deux villes seront reliées par une arête si leur distance à vol d'oiseau est inférieure à un certain seuil.

Calcul de distance Étant donné deux points A, B de coordonnées GPS (φ_A, λ_A) et (φ_B, λ_B) (avec φ la latitude et λ la longitude), la distance en kilomètres entre A et B s'obtient par la formule suivante :

$$d(A, B) = 6371 \arccos(\sin(\varphi_A)\sin(\varphi_B) + \cos(\varphi_A)\cos(\varphi_B)\cos(\lambda_A - \lambda_B))$$

Q18. Écrire une fonction `graphe_villes(villes, seuil)` qui prend en entrée un dictionnaire `villes` ayant pour clés des noms de ville et pour valeurs leurs coordonnées GPS, ainsi qu'une valeur de distance `seuil`, et qui construit le graphe dont les sommets sont les villes du dictionnaire `villes`, et dont les arêtes correspondent aux villes dont la distance est inférieure au seuil. Attention, les fonctions trigonométriques du module `math` manipulent des **radians**.

Q19. Afficher les graphes obtenu en prenant comme seuil 3km, 10km, 60km puis 150km.

Q20. Déterminer à partir de quel seuil le graphe est connexe (pensez à utiliser le principe de dichotomie !).

Q21. Un gastronome veut aller de Toulouse à Lille. Il veut s'arrêter chaque soir dans une ville assez grande, afin d'avoir un grand choix de restaurants. Il part en vélo, et peut pédaler 60 kilomètres par jour. Déterminer le trajet idéal et l'afficher. Combien de jours prendra-t-il ?

Q22. Et s'il ne peut faire que 40 kilomètres par jour ?

1. La quasi-totalité des villes proches de Paris ont été retirées pour des raisons de performances.