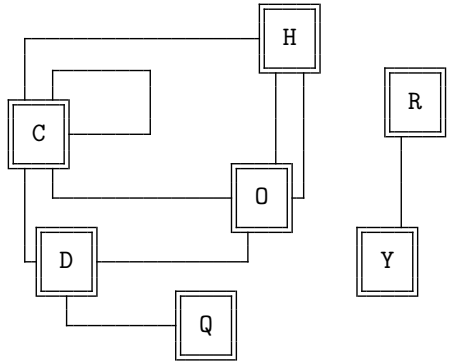
I. Introduction et définitions

Notion de Graphe et d'arbre

On appelle graphe un ensemble de sommets reliés par des arrêtes. Le dessin ci-dessous est un exemple de graphe :

G_0 :



Dans cet exemple, le graphe G_0 a 7 sommets, notés C, D, H, O, Q, R et Y. Les arrêtes qui les relient ne sont pas nécessairement des lignes droites (sur cet exemple elles sont représentées avec des angles). On note en particulier que sur cet exemple, il y a une arrête qui relie le sommet C à lui même, et qu'il y a deux arrêtes qui relient le sommet H au sommet O.

Mathématiquement, on identifie un graphe à un couple (S,A) où S est l'ensemble des sommets et A est la donnée des arrêtes. L'exemple ci-dessus correspond ainsi à $S = \{C, D, H, O, Q, R, Y\}$ et $A = (\{C,C\},\{C,D\},\{C,H\},\{C,O\},\{D,O\},\{D,Q\},\{H,O\},\{H,O\},\{R,Y\})$.

Un arbre est un cas particulier de graphe qui est connexe et n'a pas de boucle, c'est à dire que pour tous couple (S_1,S_2) de sommets, il existe un unique chemin (sans demi-tour) qui va de S_1 à S_2.

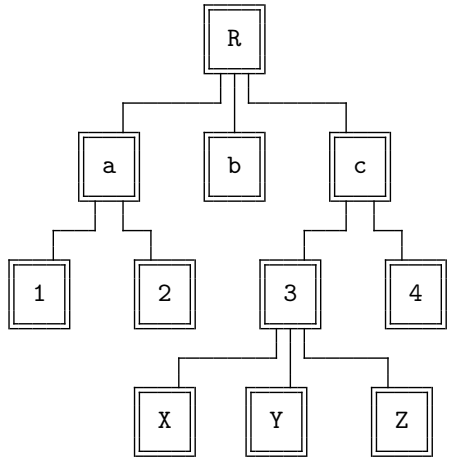
Le graphe G_0 ci dessus n'est pas un arbre car il n'est pas connexe (par exemple il n'y a pas de chemin qui aille de H à R), et qu'il a des boucles (par exemple il y a plein de façons d'aller de C à O : de manière directe, ou en passant par D, ou en passant par H, etc).

Parcours de graphes "en profondeur" et "en largeur"

De nombreux problèmes concrets se ramènent à des "parcours de graphes", c'est à dire des algorithmes qui considèrent successivement les sommets (et les arrêtes) du graphe.

Il se trouve que l'on distingue deux grandes familles d'algorithmes : les algorithmes de parcours en profondeur et les algorithmes de parcours en largeur. Par exemple, si on considère l'arbre ci-dessous :

A:



- Un parcours "en largeur" signifierait typiquement qu'on considère d'abord le sommet R, puis les sommets a, b, et c, puis les sommets 1, 2, 3 et 4, et enfin les sommets X, Y et Z, c'est à dire qu'on regarde les lignes l'une après l'autre.
- Un parcours en profondeur signifie qu'on va jusqu'au bout d'une branche, puis jusqu'au bout de la branche suivante, etc. Sur cet exemple, cela signifie qu'on commence par les sommets R, a, et 1 puis on revient en arrière (en repassant par a) pour aller à 2, puis (en repassant passe par a et R,) on arrive à b, puis on (re)passe en R, en c, 3, X, 3, Y, 3, Z, 3, c, puis 4 (et à nouveau c et R).
  Bien sur on peut aussi considérer des variantes où on ne compte pas les passages multiples au même sommet.

Implémentation en python : dictionnaire

Notion de dictionnaire

On ne cherchera pas à utiliser de module python spécifique pour la manipulation de graphes, et en choisit de représenter en python un graphe par deux dictionnaires.

Un dictionnaire contient un ensemble d'identifiants associés à des valeurs, il se manipule comme ci dessous :

```
d={"hirondelle":"oiseau","python":"serpent","pomme":"fruit"}
d["papillon"]="insecte"
print(d)
print('d["pomme"]='+d["pomme"])
```

Dans cet exemple les identifiants sont `"hirondelle"`, `"python"`, `"pomme"`, puis `"papillon"` (ajouté en exécutant `d["papillon"]="insecte"`). `"oiseau"` est la valeur associée à l'identifiant `"hirondelle"`, tandis que `"serpent"` est la valeur associée à l'identifiant `"python"`, etc.

Si on exécute par exemple `d["oiseau"]`, on obtiendra un message d'erreur car `"oiseau"` n'est pas un des identifiants.

Il est utile de savoir qu'avec un dictionnaire, on peut faire une boucle `for`, mais que dans ce cas on parcourt uniquement les identifiants (comme ci-dessous, où `i` prend pour valeurs successives les différents identifiants) :

```
for i in d:
    print(i+" → "+d[i])
```

Choix d'implémentation des graphes

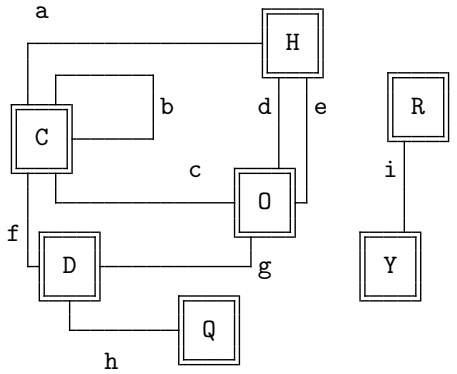
On choisit d'implémenter les graphes par deux dictionnaires : un dictionnaire qui a pour identifiants les sommets et pour valeurs la liste des arrêtes adjacentes à ce sommet, et un autre dictionnaire qui a pour identifiants les arrêtes et pour valeurs les sommets adjacents à chaque arrête.

Par exemple, Pour le graphe G_0 défini précédemment, ces dictionnaires sont :

```
S={"C":["a","b","b","c","f"],"D":["f","g","h"],"H":["a","d","e"],
   "O":["c","d","e","g"],"Q":["h"],"R":["i"],"Y":["i"]}
A={"a": ["C", "H"], "b": ["C", "C"], "c": ["C", "O"], "d": ["H", "O"],
   "e": ["H", "O"], "f": ["C", "D"], "g": ["D", "O"], "h": ["D", "Q"],
   "i": ["R", "Y"]}
```

Cela correspond à numéroter les arrêtes de la façon suivante :

G_0 :



1. Écrire une fonction qui construit le dictionnaire des sommets à partir de celui des arrêtes .
2. Écrire de même une fonction qui construit le dictionnaire des arrêtes à partir du dictionnaire des sommets.

II. Exemples de parcours de graphes

Parcours de labyrinthe : algorithme de Trémaux

On considère les graphes comme une façon de modéliser un labyrinthe : les sommets représentent les intersections, et les arrêtes sont les couloirs du labyrinthe.

L'algorithme de Trémaux est une méthode, permettant à quelqu'un qui explore un labyrinthe d'en visiter toutes les pièces (et le cas échéant de trouver la sortie) sans se perdre :

- La première fois qu'on parcourt un couloir on dessine une marque à chacune de ses extrémités (on sait que c'est un couloir que l'on n'avait pas encore parcouru s'il n'y a pas encore de marque à l'entrée de ce couloir).
- Quand on arrive à une intersection où l'on s'est déjà rendu, on revient en arrière par le couloir par lequel on est arrivé.
  - Dans cette situation, et plus globalement à chaque fois qu'on parcours un couloir pour la deuxième fois, on appose une deuxième marque à l'entrée et à la sortie du couloir.
  - Après qu'un couloir ait ainsi été marqué en double, on n'emprunte plus jamais ce couloir.

- Quand on arrive à une intersection qui n'a jamais été explorée (c'est à dire que l'on ne voit de marque à l'entrée d'aucun couloir), alors on dessine une triple marque à la sortie du couloir par lequel on est arrivé à cet intersection.
- À chaque fois qu'on arrive à intersection, on choisit d'emprunter un couloir qui n'a jamais été exploré (s'il y en a), sinon un couloir qui a une seule marque (s'il y en a plusieurs on choisit au hasard), ou en dernier recours le couloir qui a une triple marque.

3. Écrire une fonction qui implémente cet algorithme. Vérifie que pour le graphe G_0, Si l'on commence en O, alors cet algorithme permet d'aller à n'importe quel sommet sauf R et Y.
4. Cet algorithme correspond-il à un parcours de graphe en largeur ou en profondeur ?

Plus court chemin

On considère désormais que les différentes arrêtes n'ont pas toutes la même longueur. On ajoute donc un troisième dictionnaire qui indique la longueur de chaque arrête. Par exemple

```
l={"a": 3, "b": 9, "c": 1, "d": 2, "e": 9, "f": 9, "g": 3, "h": 9, "i": 2}
```

qui indique que l'arrête `"a"` est de longueur 3, alors que `"b"` est de longueur 9, etc. Ces longueur peuvent par exemple correspondre à la distance à parcourir entre des villes, ou à des temps de trajet (ce qui ne revient pas au même si l'on ne vas pas à la même vitesse sur toute les routes). On suppose en tout cas que toutes les arrêtes ont un longueur positive.

On désigne par $\ell(S,T)$ la distance entre deux sommets, c'est à dire la longueur du plus court chemin entre ces deux sommets (où la longueur d'un chemin est la somme des longueurs des arrêtes qui le compose). S'il n'existe aucun chemin entre S et T, alors on considère que $\ell(S,T) = +\infty$.

Pour déterminer la distance $\ell(S_1,S_2)$, on utilise l'algorithme suivant :

- Au fur et a mesure du calcul, on garde en mémoire une estimation de la distance $\ell(S_1,T)$ pour chaque sommet T. Initialement on sait juste que $\ell(S_1,S_1) = 0$ et que $\forall T \neq S_1 : 0 \leqslant \ell(S_1,T) \leqslant +\infty$.
  On note donc $m_S = 0$ pour chaque sommet S, $M_T = +\infty$ pour chaque sommet $T \neq S_1$ et $M_{S_1} = 0$, de sorte que l'information dont l'on dispose est $\forall S : m_S \leqslant \ell(S_1,S) \leqslant M_S$.
  De plus, on peut affirmer dès à présent que dans la suite, on ne considérera pas les arrêtes qui reviennent au même sommet que celui dont elles sont parties (comme l'arrête `"b"` du graphe G_0).
- On commence par considérer le sommet S_1 :
  - Pour chaque arrête partant de S_1, si on note l sa longueur et T le sommet auquel elle mène, on peut affirmer que $\ell(S_1,T) \leqslant l$, donc on réactualise M_T en le remplaçant par $\min(M_T,l)$.
  - Si la plus courte arrête qui part de S_1 mène au sommet T_0 et est de longueur l_0, alors on peut être sur que $\forall S \neq S_1 : \ell(S_1,T) \geqslant l_0$.
    En effet tout chemin de S_1 à S commence par parcourir une arrête qui part de S_0 et a une longueur supérieure ou égale à l_0.
    On peut donc réactualiser en remplaçant m_S par $\max(m_S,l_0)$, pour chaque $S \neq S_1$.
  - Combiné à l'inégalité précédente, cela donne en particulier que $\ell(S_1,T_0) = l_0$.
- On considère ensuite le sommet T_0 obtenu à l'étape précédente :
  - Pour chaque arrête partant de T_0, si on note l sa longueur et T le sommet auquel elle mène, on peut affirmer que $\ell(S_1,T) \leqslant l + l_0$, donc on réactualise M_T en le remplaçant par $\min(M_T,l + l_0)$.
  - Les valeurs $V = \{M_S | S \neq S_0, S \neq T_0\}$ sont les longueurs des plus petits chemins qui partent de S_1 et vont à un sommet distinct de T_0. On peut donc affirmer que $\forall S \notin \{S_1,T_0\}, \ell(S_1,S) \geqslant \min(V)$.
    En effet, le début du plus court chemin entre S_1 et S (jusqu'au premier sommet distinct de S_1 et de T_0) est forcément un des chemins dont la longueur appartient à V.
    On peut donc réactualiser en remplaçant m_S par $\max(m_S, \min(V))$, pour chaque $S \notin \{S_1,T_0\}$.
  - On note T_1 tel que $M_{T_1} = \min(V)$. Comme $M_{T_1} = \min(V) = M_{T_1}$, on sait que $\ell(S_1,T_1) = \min(V)$, et on va considérer le sommet T_1 à la prochaine étape.
- Plus généralement à la $n^{\text{ème}}$ étape, on connaît déjà les longueurs $\ell(S_1,S_1) = 0$, $\ell(S_1,T_0) = l_0$, $\ell(S_1,T_1) = l_1$, …, $\ell(S_1,T_{n-1}) = l_{n-1}$.
  - Pour chaque arrête partant de T_{n-1}, si on note l sa longueur et T le sommet auquel elle mène, on peut affirmer que $\ell(S_1,T) \leqslant l + l_{n-1}$, donc on réactualise M_T en le remplaçant par $\min(M_T,l + l_{n-1})$.
  - Les valeurs $V = \{M_S | S \notin \{S_0,T_0,\ldots,T_{n-1}\}\}$ sont les longueurs des plus petits chemins qui partent de S_1 et vont à un sommet distinct des T_i (pour les $i < n$).
    On peut donc affirmer que $\forall S \notin \{S_1,T_0,\ldots,T_{n-1}\}, \ell(S_1,S) \geqslant \min(V)$.
    En effet, le début du plus court chemin entre S_1 et S (jusqu'au premier sommet distinct de S_1 et des T_i) est forcément un des chemins dont la longueur appartient à V.
    On peut donc réactualiser en remplaçant m_S par $\max(m_S, \min(V))$, pour

chaque $S \notin \{S_1, T_0, \ldots, T_{n-1}\}$.
- On note T_n tel que $M_{T_n} = \min(V)$, on sait désormais que $\ell(S_1, T_n) = \min(V)$, et on va considérer le sommet T_n à la prochaine étape.

- Si à un moment $T_n = S_2$, alors on renvoie comme réponse que $\ell(S_1, S_2) = l_n$.
- Sinon, le procédé s'interrompt dès que $V = \emptyset$. Cela signifie qu'il n'existe pas de chemin qui aille plus loin que le sommet T_{n-1}, et qu'on a calculé la distance de tous les sommets atteignables depuis S_1.
  Dans ce cas on renvoie $+\infty$ pour signifier qu'il n'y a pas de chemin entre S_1 et S_2.

Remarque : En fait dans cet algorithme il est inutile de retenir la valeur des m_S.

5. Écrire une fonction qui implémente cet algorithme et calcule la distance entre des sommets.
6. Cet algorithme s'apparente -t-il à un parcours en profondeur ou à un parcours en largeur ?

Logique booléenne

7. Lors du chapitre 2, on a écrit une fonction `eval_arbre` qui évaluait la fonction booléenne correspondant à un arbre.
   Argumenter que cette fonction effectue elle aussi un parcours d'arbre, et déterminer s'il s'agit d'un parcours "en profondeur" ou d'un parcours "en largeur".

Problème "du sac à dos"

On considère un situation où il existe différents objets : chaque objet x_i a une valeur v_i et un encombrement e_i. On a un sac à dos qui est trop petit pour emporter tous les objets : il permet uniquement de transporter un (sous-)ensemble d'objets dont l'encombrement total vaut au maximum C.

On cherche donc un ensemble E_0 tel que

$$\begin{cases} \sum_{i \in E_0} e_i & \leqslant C \\ \sum_{i \in E_0} v_i & = \max\left\{ \sum_{i \in E} v_i \middle| \sum_{i \in E} e_i \leqslant C \right\} \end{cases} \tag{1}$$

8. Argumenter que les différents ensembles d'éléments peuvent être vus comme les branches d'un arbre.
9. Écrire un programme qui trouve l'ensemble E_0 par un parcours d'arbre, en essayant de ne pas explorer certaines régions inutiles de l'arbre.