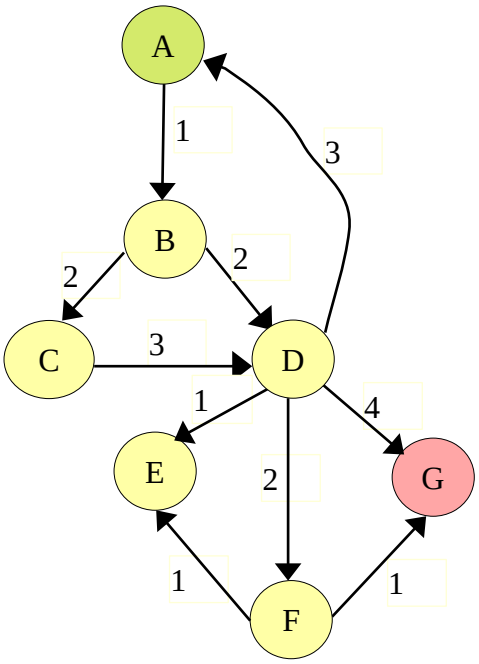
TP N 02 : Les Algorithmes de Recherche non Informés

I. But du TP

Ce TP a pour objectif de mettre en pratique les algorithmes de recherche étudiés en cours. Dans ce TP, nous allons commencer par l'implémentation de quelques algorithmes de recherche non informés (BFS, DFS, etc.). L'implémentation de ces algorithmes permet une meilleure maîtrise des concepts théoriques associés.

Partie I : Un Simple Algorithme de recherche

Soit l'espace de recherche suivant :



Cet espace de recherche peut être représenté par un dictionnaire. Dans ce dictionnaire, les clés correspondent aux états (nœuds) de l'espace, tandis que les valeurs associées à chaque clé représentent une liste des successeurs (fils) de cet état.

```
MyResearchSpace={
   'A': ['B'],
   'B': ['C','D'],
   'C': ['D'],
   'D': ['A','E','F','G'],
   'E': [],
   'F': ['E','G'],
   'G': [],
}
```

Pour réaliser l'implémentation d'un algorithme de recherche, commencez par coder les éléments suivants :

1. Écrire la fonction « **genere_successeur** » qui prend en paramètre un état, et retourne une liste de ses successeurs.

TP N 02 : Les Algorithmes de Recherche non Informés

2. Écrire la fonction «**test_etat_fianl** », qui prend en paramètre, un état 'C' et une liste d'états F représentant les états finaux. La fonction retourne 'True' si C appartient à F, et 'False' sinon.

3. Créer deux listes : '**Open**' et '**Closed**'. La liste 'Open' sera utilisée comme une **pile** (**LIFO** :**Last In, First Out**), en employant la fonction 'append' pour empiler et la fonction 'pop()' pour dépiler. En vous appuyant sur ces deux listes, implémentez le code qui **parcourt l'ensemble des états** (en commençant par 1) en respectant les étapes suivantes :

   a) Empiler l'état '1' dans la liste 'Open'.

   b) Tant que la liste 'Open' n'est pas vide :

   - Dépiler un état 'N' de 'Open', et le placer dans 'Closed' (c'est l'état visité);
   - Afficher à chaque fois l'état dépilé (c'est l'ordre de visite).
   - Générer les successeurs de 'N' (avec la fonction genere_successeur), puis empilez-les dans 'Open' seulement si ces états ne sont pas déjà présents dans 'Open' ni dans 'Closed'.

   c) Quel est le comportement de ce parcours ? (en largeur ou en profondeur)

4. Utiliser les codes précédents comme base pour implémenter une fonction de recherche appelée 'search', en suivant les directives suivantes :

   a) La fonction 'search' prend deux paramètres : un état initial 'I' et une liste d'état finaux 'F'.

   b) La fonction doit parcourir les états de l'espace de recherche depuis l'état initial 'I' et s'arrête si elle visite l'un des états finaux de 'F'.

   c) La fonction retourne True si elle parvient a trouvé un état final, et False sinon. Autrement dit, elle retourne 'True' lorsque un état dépilé depuis 'Open' appartient a F, et 'False' si la liste 'Open' devient vide sans pouvoir dépilé un état appartenant à F.

   d) Tester la fonction 'search' avec « search('A',['G']) ».

   e) Est ce que cette fonction vous permet de retourner la solution, c-à-dire , le chemin depuis l'état initial à l'état final ?

5. Pour changer le comportement de parcours de notre fonction de recherche 'search', il suffit de modifier la stratégie de gestion de la structure de données 'Open'. Au lieu d'utiliser une pile, nous utiliserons une **file** (**FIFO : First In First Out**) . En Python, cela peut être réalisé avec la structure '**deque**', comme illustré dans l'exemple suivant :

```
from collections import deque
file = deque() # Une file vide
#Ajouter les valeurs 1,3 et 4
file.append(1)
```

```
file.append(3)
file.append(4)
print(file) #Affiche deque([1, 3, 4])
v=file.popleft() # v prent la valeur 1
print(file) ##Affiche deque([3, 4])
```

NB : deque peut aussi être utiliser pour simuler une pile (avec append et pop()).

a) Modifier alors la fonction « search » avec cette nouvelle structure de données.

b) Tester la fonction 'search' avec «search('A',['G'])».

c) Quel est la nouvelle stratégie de recherche de cette fonction ?

Partie II: Un Algorithme plus évolué

Pour que notre algorithme de recherche soit plus complet et qu'il soit capable de retourner : **le résultat de recherche** (True ou False), **l'ordre de visite** (pour pouvoir déterminer la stratégie de l'algorithme) ainsi que **la solution** (le chemin entre l'état initial et l'état final), nous avons fait les modification suivantes :

- Un état sera représenté par une classe (la classe *Node*) qui stock l'information relative au parent de chaque état. Cette information permet, à partir d'un état 'C', de reconstruire le chemin depuis l'état initial jusqu'à l'état 'C'.

```
class Node:
   def __init__(self, name, parent, action):
      self.name = name #Le nom de l'état
      self.parent = parent # Le père d'un nœud
      self.action = action # L'action qui génère de nœud (optionnel )
```

- La fonction suivante implémente l'algorithme de recherche DFS, en utilisant la classe '*Node*' au lieu d'un simple état.

```
def dfs_search(I,F):
   open=[]
   closed=[]
   ordreVisite=[]
   open.append(Node(I, None, ""))
   while open:
         N = open.pop()
         ordreVisite.append(N.name)
         if test_etat_final(N, F):
            return True,ordreVisite,getSolution(N)
         else:
           closed.append(N)
           succ = gener_successur(N)
           if len(succ) > 0:
              for s in succ:
                if s.name not in [x.name for x in open] and s.name not in [x.name for x in closed]:
                   open.append(s)
   return False,ordreVisite,[]
```

Cette fonction prend en paramètres, comme les fonctions précédentes, un état initial 'I' et une liste d'états finaux 'F' (par exemple : dfs_search('A',['G']). Elle retourne trois

éléments : **un booléen** indiquant si l'algorithme a trouvé l'un des états finaux, une **liste des états visités** durant la recherche, et enfin une liste représentant **la solution** (le chemin depuis l'état initial jusqu'à l'état final).

1. Essayer de comprendre le fonctionnement de cette fonction.

2. Pour pouvoir exécuter cette fonction, compléter l'implémentation des fonctions suivantes :

a) La fonction `test_etat_final(N, F),` qui prend en paramètre un état N (un objet de type *Node*) et une liste d'états finaux F. Elle retourne `True` si le nom (*name*) de N appartient à F, et `False` sinon.

b) La fonction `gener_successur(N),` qui prend en paramètre un état N (un objet de type *Node*) et genére ses successeurs. Elle retourne une liste des successeurs (une liste d'objets Node) de l'état N.

c) La fonction `getSolution(N),` qui prend en paramètre un état N (un objet de type *Node*) et retourne une liste des noms des états depuis l'état initial I jusqu'a l'état N.

d) Tester votre implémentation avec l'exemple : `dfs_search('A',['C'])`

3. En modifiant la fonction `dfs_search,` implémenter la fonction `bfs_search,` et vérifier votre implémentation en inspectant la liste '`ordreVisite`' avec l'exemple : `dfs_search('A',['C'])`.

4. Problème :

Considérons le problème suivant :
Vous disposez de deux jarres : l'une de 4 litres et l'autre de 3 litres. L'objectif est d'obtenir exactement 2 litres d'eau, sans avoir d'unité de mesure spécifique. Pour cela, vous pouvez réaliser les opérations suivantes (sans contraintes sur la quantité d'eau disponible) :

- Vider la première jarre : V1.
- Vider la deuxième jarre : V2.
- Remplir la première jarre :R1.
- Remplir la deuxième jarre :R2.
- Transvaser le contenu la première jarre la deuxième : T12.
- Transvaser le contenu la deuxième jarre la première : T12.

Les deux jarres sont initialement vides.

Les tâches à accomplir :

1. Modéliser ce problème pour qu'il puisse être résolu par un algorithme de recherche.
2. Adapter les programmes développés dans ce TP afin de résoudre ce problème.