Projet Résolveur de Graphes - PYTHON
Sujet 1

Binôme: Binous Mohammed Chedly & Bouker Rania
RT4-Groupe1

Objectif :

Mise en place d'une application interactive pédagogique montrant le fonctionnement des algorithmes de recherche pour la résolution de problèmes généraux.

Partie 1 : Implémentation des algorithmes suivants :

Les algorithmes de recherche non informés	*Les algorithmes de recherche informés*
1. La recherche en coût uniforme, 2. La recherche en largeur d'abord, avec et sans but 3. La recherche en profondeur d'abord, avec et sans but 4. La recherche en profondeur limitée itérative	1. A* avec f(n)=g(n)+h(n) 2. Recherche meilleur d'abord gloutonne avec f(n)=h(n)

Task 1 : Modélisation du graphe : définition des classes :

--- **Un graphe**, est un modèle abstrait de dessins de réseaux reliant des objets. Il est constitué de nœuds et d'arcs, reliant ces nœuds entre eux, Nous allons donc modéliser une classe pour les nœuds et une pour le graphe, nommées respectivement **Vertex** et **Graph**.

--- La classe Vertex : Chaque nœud, a principalement, un nom, une liste des noms de voisins directement connectés au nœud, une distance le séparant du nœud source et une couleur afin d'identifier l'état du nœud (exploré, non exploré, ouvert, fermé …).

Les méthodes de cette classe sont **: add_neighbor(v)** qui permettra d'ajouter un voisin à la liste des voisins du nœuds, **add_neighbor_cout(v)** pour ajouter un voisin v à un nœud u avec le cout de l'arc [uv].

--- La classe Graph : Un graphe est composé d'un ensemble de nœuds, il aura donc comme attribut principal un dictionnaire d'objets nœuds, qu'on appellera **Vertices {},** et un attribut **Trouver pour la recherche avec BUT.**

Tout au long de ce projet, nous allons considérer des graphes simples, des graphes valués et des arbres (graphes non cycliques).

Pour cela, nous aurons comme méthodes dans la classe Graph : **add_vertex :** permettant d'ajouter un nœud à un graphe, **add_edge** permettant de créer un arc non valué, **add_edge_cout** pour créer un arc valué et finalement **add_edge_arbre** pour la création d'arbres (utile dans l'algorithme profondeur itérative).

La méthode **print_graph()** permet d'afficher chaque nœud du graphe, sa liste de voisins et la distance le séparant de la source du graphe.

Outil de développement utilisé : Visual Studio Code 2012 ------- Langage de programmation PYTHON

La méthode **lire_graph** permettant de créer à partir d'un fichier un graphe, en initialisant les nœuds et en créant les arcs. Nous aurons bien évidement deux extensions de cette méthode pour la création des graphes valués et d'arbres : **lire_graph_cout** et **lire_graph_arbre.**

La méthode **visualiser_graph** permettant de visualiser un graphe et **visualiser_graph_cout** pour les graphes valués. Le fonctionnement de cette méthode sera mis en évidence dans la deuxième partie.

Finalement, la classe Graph contient les méthodes de résolution des graphes, à savoir : **bfs, dfs, bfs_but, dfs_but, iddfs (Profondeur itératif), a_star, glutonne, bfs_cu** …

Task2 : Implémentation des algorithmes dans des méthodes :

Les détails de l'implémentation des algorithmes se trouvent dans les commentaires du code.

Partie 2 : Création de l'interface :

Afin de développer l'interface, nous allons utiliser le module Tkinter qui est un module de base intégré dans Python, il nous permettra de créer des fenêtres, des boutons, de gérer les actions sur les boutons …

```
fenetre = Tk()
label = Label(fenetre, text="Bienvenue dans l'interface RBBC")
label.pack()
bouton=Button(fenetre, text="Welcome", command=affichage)
bouton.pack()
fenetre.mainloop()
```

Création de la fenêtre avec Tk interface, la fonction prédéfinie mainloop() pour l'afficher

```
bouton1 =Button(fenetre2, text="valider", command=lambda:g.dfs(g.vertices[saisie.get()]))
bouton1.pack()
```

Passage d'une fonction avec paramètres comme action d'un bouton avec la fonction prédéfinie LAMBDA

L'interface développée permettra de :

- Choisir l'algorithme à utiliser.
- Choisir le fichier décrivant le graphe et le générer :

```
A0,B2,C3,D0,Y5
AB,AC,BD,BY
2,6,7,3,8

Noeud:heuristique
Arcs
Couts des arcs
```

Graphe.txt

Format du fichier TXT

```
def lire_graph(self,filename):
    f = open(filename,'r')
    noeuds = list()
    arcs = list()
    liste = f.readlines()
    for i in range(0,len(liste)):
        liste[i] = liste[i].replace('\n',"")
    noeuds = liste[0].split(',')
    for i in range(0,len(noeuds)):
        self.add_vertex(Vertex(noeuds[i][:1], noeuds[i][1:]))
    arcs = liste[1].split(',')
    for i in range(0,len(arcs)):
        self.add_edge(arcs[i][:1], arcs[i][1:])
```

Génération du graphe == graphe.txt

Avec askopenfilename, qui permet d'ouvrir la fenetre de dialogue pour choisir

```
filename = askopenfilename(title="Ouvrir votre document",
        filetypes=[('txt files','.txt'),('all files','.*')])
g = Graph()
g.lire_graph(filename)
g.visualiser_graphe(filename)
```

Choix du fichier

- De le visualiser, de choisir l'état initial, l'état final : La visualisation du graphe a été utilisée avec l'outil GRAPHIZ 0.8.1 [pour créer des nœuds, utiliser *node* et *edge* pour les arcs, voir plus de détails dans le code ...]
- Le temps d'exécution a été calculé grâce au module TIME de python, la built-in méthode time.clock().
- La sauvegarde des fichiers a été effectuée par une simple ouverture d'un fichier trace en mode écriture. Nous avons aussi utilisé le module sys pour copier la sortie de la console dans un fichier.