INSAT

Projet Résolveur de problèmes Généraux

Rapport du projet

CHAABENE Amal & HAMMOUDA Hajer

RT4 Groue2

17/12/2017

Nous interprétons dans ce rapport le code de l'application qui correspond au premier sujet proposé.

Introduction :

Dans un souci d'implémenter les connaissances en matière d'intelligence artificielle, nous avons pris en charge le premier projet proposé qui consiste à développer une interface qui permet, non seulement, la visualisation des graphes décrits sous formats texte mais aussi l'application des algorithmes de recherche sur le graphe choisi, l'affichage des résultats (parcours, chemin, temps d'exécution, cout) et le sauvegarde de ces derniers dans un fichier trace.

Interprétation du code :

Nous avons réalisé l'application avec le langage « python ».

Nous avons utilisé la bibliothèque graphique pour créer l'interface graphique, les bibliothèques « networkx » et « graphviz » pour manipuler et dessiner les graphes.

I. Définition des nœuds :

Nous avons défini une classe pour décrire les nœuds (valeur, nœuds voisins, valeur heuristique et cout de chaque nœud voisin)

```
class def_node:
    val=0
    succW=[]
    def __init__(self, val,succ):
        self.val=val
        self.succW=succ
```

II. Implémentation des algorithmes de recherche :

1. La recherche en largeur d'abord :

Nous avons utilisé la méthode prédéfinie de la bibliothèque networkx « bfs_edges » qui retourne les arcs parcourus à partir d'un sommet donné en appliquant l'algorithme de recherche en largeur d'abord.

Nous avons, ainsi, utilisé la méthode prédéfinit « shortest_path » de la bibliothèque networkx pour déterminer le chemin du point de départ vers le but.

Puis on a extrait les nœuds amenant au but :

```
def bfs_algo (self,debut,fin):
    listbfs=list(nx.bfs_edges(self.gx,debut))
    nodeBfs = []
    nodeBfs.append((listbfs[0])[0])
    for c in listbfs :
        nodeBfs.append(c[1])
    print (nodeBfs)
    if fin in nodeBfs :
        self.parcours=nodeBfs[:nodeBfs.index(fin)+1]
        self.chemin=nx.shortest_path(self.gx,debut,fin,1)
        self.cout=self.calcChemin(self.chemin)
```

La méthode calcChemin permet de calculer le cout du chemin suivit. Elle prend en paramètre le chemin sous forme de list des nœuds et retourne le cout de ce dernier. Elle permet de parcourir les nœuds deux à deux et ajouter à chaque fois le cout de l'arc associé.

Elle définit comme suit :

```
def calcChemin(self,listC):
    l=len(listC)
    k=0
    v=[]
    valeur=0
    print(l)
    while k < l :
        for arc in self.allArcs :
            v=arc[1:-1].split(",")
            if listC[k]==v[0] and listC[k+1]==v[1] :
                valeur+=int(v[2])
        k=k+1

    return valeur
```

Remarque : Cette méthode est appelée dans les algorithmes de recherche «en largeur d'abord », « en profondeur d'abord », « en profondeur limité itératif » , « SMA »

2. La recherche en profondeur d'abord :

Nous avons utilisé la méthode prédéfinie de la bibliothèque networkx « dfs_preorder_nodes » qui retourne les nœuds parcourus à partir d'un sommet donné en appliquant l'algorithme de recherche en profondeur d'abord, puis on retourne l'ensemble des nœuds amenant au but.

```
def dfs_algo (self,debut,fin):
    parcours=list(nx.dfs_preorder_nodes(self.gx,debut))
    if fin in parcours:
        self.parcours=parcours[:parcours.index(fin)+1]
        self.chemin=nx.shortest_path(self.gx,debut,fin,1)
        self.cout=self.calcChemin(self.chemin)
```

3. La recherche en profondeur limité itératif :

Pour l'implémentation de cet algorithme on a eu l'idée de construire un nouveau graph de type networkx par niveau, autrement dit, on a construit d'abord un graphe de niveau 1, on applique l'algorithme de recherche en profondeur sur ce graphe, si le but n'est pas atteint on ajoute les nœuds appartenant au niveau suivant et on applique l'algorithme et ainsi de suite jusqu'à atteindre la limite donnée.

```
def profondeurLimité(self,debut,fin):
    limite=int(self.entre_val.get())
    v=[]
    p=[]
    tmp=[]

    while (self.cst<limite):
        if self.a ==0 :
            for n in self.nodeObj :
                if n.val == debut :
                    self.a=1
                    self.nCible=n
                    self.succ.append(n)
                    self.gx1.add_node(n.val)

        for j in self.succ:
            for a in self.allArcs :
                v=a[1:-1].split(",")
                for k in j.succW :
                    if j.val==v[0] and k[0]==v[1]:
                        self.gx1.add_node(j.val)
                        self.gx1.add_edge(j.val,k[0])
                        print("on a ajouté ")
                        print(j.val)
                        print(k[0])

        p=list(nx.dfs_preorder_nodes(self.gx1,debut))
        print(p)
        print("nouvelle liste")
        self.cst+=1
        self.parcours.extend(p)

        for ajout in self.succ :
            for suc in ajout.succW :
                for n in self.nodeObj :
                    if suc[0]==n.val :
                        tmp.append(n)
        self.succ=tmp

    if fin in self.parcours :
        print("hhh")
        self.cst=limite
        self.parcours=self.parcours[:self.parcours.index(fin)+1]
        self.chemin=nx.shortest_path(self.gx,self.debA,fin)
        self.cout=self.calcChemin(self.chemin)
        self.succ=[]
        self.gx1.clear()

    else :
        self.profondeurLimité(debut,fin)
```

Cette partie du code permet de choisir le nœud de départ dans la liste self.nodeObj une seule fois lors du lancement de l'algorithme et de l'ajouter au graphe.

Cette partie du code permet ajouter les nœuds successeur et les arcs au graphe.

Application de l'algorithme de recherche ne profondeur sur le graph construit

Si on atteint le but, on définit le parcours le cout et le chemin, sinon on relance l'algorithme.

4. La recherche en cout uniforme :

Pour l'implémentation de cet algorithme, on a défini une méthode « lenNode » qui permet de calculer la valeur heuristique de chaque nœud voisin comme suit : f(n)=g(n) afin de choisir par suite la prochaine destination.

Remarque: chaque nœud visité est sauvegardée dans une liste « visited ».

```
def lenNode (self,n):
    for i in self.visited :
        print(i[0])
        if (n == i[0]) :
            return i[1]
    return 0
```

Le code de recherche est comme suit :

```
def coutUniforme(self,debut,fin):
    mini=999
    val = 0

    for n in self.nodeObj :
        if n.val == debut :
            self.nCible=n
            self.parcours.append(debut)

    if len(self.nCible.succW)!=0 :
        val =self.lenNode(self.nCible.val)
        print(self.nCible.val)
        for i in self.nCible.succW :
            print(i)
            self.succ.append((i[0],int(i[1])+val ))

    for s in self.succ :
        print("les succ sont :")
        print(s)
        if int(s[1])< mini :
            mini = int(s[1])
            resultat = (s[0],s[1])
    print(resultat)
    print("resultat")
    print(resultat[0])
    self.visited.append(resultat)

    self.succ.remove(resultat)
    if (resultat[0] == fin) :
        print("finn")
        self.chemin=nx.shortest_path(self.gx,self.debA,fin)
        self.cout=resultat[1]
        self.parcours.append(resultat[0])
    else :
        self.coutUniforme(resultat[0],fin)
```

On choisit le nœud « objet » parmi la liste des nœuds sauvegardés dans la liste « nodeObj » avec lequel on va lancer la fonction.

On parcourt la liste des voisins du nœud courant et on ajoute à la liste « succ » le couple (valeur du nœud, cout).

On choisit le nœud ayant le cout minimal.

On supprime le nœud visité.

Si le but est atteint on définit le cout, le parcours et le chemin sinon on relance l'algorithme avec le nœud choisi précédemment

5. A* :

Pour l'implémentation de cet algorithme, on a défini une méthode « lenNodeA » qui permet de calculer la valeur heuristique de chaque nœud voisin comme suit : f(n)=g(n)+h(n) afin de choisir par suite la prochaine destination.

Pour cet algorithme on suit les mêmes étapes de l'algorithme de recherche en cout uniforme sauf le calcul de f(n) change.

Remarque: chaque nœud visité est sauvegardée dans une liste « visited » sous la forme **(valeur, cout de chemin + h(n) associé)** et les successeurs du nœud visité sont sauvegardé dans la liste « vist » sous la forme d'un couple **(valeur, cout de chemin).**

```
def lenNodeA (self,n):
    for i in self.visited :
        if i[0] == n :
            for j in self.vist :
                if i[0]==j[0]:
                    return j[1]
    return 0
```

Le code de recherche est le suivant :

```
def aEtoile (self,debut,fin):
    val = 0
    mini=999
    rm=[]

    for n in self.nodeObj :
        if n.val == debut :
            self.nCible=n
            self.parcours.append(debut)

    if len(self.nCible.succW)!=0 :
        val =self.lenNodeA(self.nCible.val)
        for i in self.nCible.succW :
            self.succ.append((i[0],int(i[1])+int(i[2])+val))
            self.vist.append((i[0],int(i[1])+val))
    for s in self.succ :
        print(s)
        if int(s[1])< mini :
            mini = int(s[1])
            resultat = (s[0],s[1])

    self.visited.append(resultat)
    for t in self.succ :
        if resultat[0]==t[0] :
            rm.append(t)
        if t[0] in self.parcours :
            rm.append(t)
    self.succ = [x for x in self.succ if x not in rm]
    #self.succ.remove(resultat)

    if (resultat[0] == fin) :
        self.chemin=nx.shortest_path(self.gx,self.debA,fin)
        self.cout=resultat[1]
        self.parcours.append(resultat[0])
        self.succ=[]

    else :
        self.aEtoile(resultat[0],fin)
```

Seulement f(n) change : f(n)=h(n)+g(n)

6. Recherche meilleur d'abord gloutonne :

Pour cet algorithme on suit les mêmes étapes de l'algorithme de recherche en cout uniforme sauf le calcul de f(n) change : f(n)=g(n).

L'algorithme est aussi récursif et le cout final vaut la valeur h associé au nœud but.

```
def gloutonne (self,debut,fin):
    val = 0
    mini=999
    rm=[]

    for n in self.nodeObj :
        if n.val == debut :
            self.nCible=n
            self.parcours.append(debut)

    if len(self.nCible.succW)!=0 :
        for i in self.nCible.succW :
            self.succ.append((i[0],int(i[2])))
    for s in self.succ :
        print(s)
        if int(s[1])< mini :
            mini = int(s[1])
            resultat = (s[0],s[1])
    for t in self.succ :
        if resultat[0]==t[0] :
            rm.append(t)
        if t[0] in self.parcours :
            rm.append(t)
    self.succ = [x for x in self.succ if x not in rm]

    if (resultat[0] == fin) :
        self.chemin=nx.shortest_path(self.gx,self.debA,fin)
        self.cout=resultat[1]
        self.parcours.append(resultat[0])
        self.succ=[]

    else :
        self.gloutonne(resultat[0],fin)
```

7. SMA :

On a essayé d'implémenter un algorithme qui cherche avec la méthode de recherche en profondeur d'abord par niveau et qui prend fin soit lorsque le but est trouvé soit lorsque tout le graphe est parcouru.

Cet algorithme se base sur le même principe de l'algorithme de recherche en profondeur limitée itérative, en fait en construit chaque niveau à chaque itération et on effectue la recherche en profondeur d'abord.

```
def sma(self,debut,fin):
    v=[]
    p=[]
    tmp=[]

    if self.a ==0 :
        for n in self.nodeObj :
            if n.val == debut :
                self.a=1
                self.nCible=n
                self.succ.append(n)
                self.gx1.add_node(n.val)

    for j in self.succ:
        for a in self.allArcs :
            v=a[1:-1].split(",")
            for k in j.succW :
                if j.val==v[0] and k[0]==v[1]:
                    self.gx1.add_node(j.val)
                    self.gx1.add_edge(j.val,k[0])
                    print("on a ajouté ")

    p=list(nx.dfs_preorder_nodes(self.gx1,debut))
    print(p)
    print("nouvelle liste")

    self.parcours.extend(p)

    for ajout in self.succ :
        for suc in ajout.succW :
            for n in self.nodeObj :
                if suc[0]==n.val :
                    tmp.append(n)
    self.succ=tmp

if fin in self.parcours :
    print("hhh")
    self.parcours=self.parcours[:self.parcours.index(fin)+1]
    self.chemin=nx.shortest_path(self.gx,self.debA,fin)
    self.cout=self.calcChemin(self.chemin)
    self.succ=[]
    self.gx1.clear()

else :
    self.sma(debut,fin)
```

On choisit le nœud de départ dans la liste self.nodeObj une seule fois lors du lancement de l'algorithme et on l'ajoute au graphe.

On ajoute les nœuds successeur et les arcs au graphe.

On applique l'algorithme de recherche ne profondeur sur le graph déjà construit

On ajoute les successeurs des nœuds à la liste « succ »

Si on atteint le but, on définit le parcours le cout et le chemin, sinon on relance l'algorithme.

III. Choix de l'algorithme :

Pour le choix de l'algorithme on ajoute une fonction au « comboBox » et selon l'entré en applique l'algorithme adéquat et à chaque fois on réinitialise les paramètres partagés par tous les fonctions (parcours, chemin, cout).

Exemple :

```
def choixAlgo(self,event):
    algo = self.TCombobox1.get()
    s=self.entre_source.get()
    e=self.entre_end.get()

    if self.TCombobox1.get()== "En profondeur d abord":
        start_time = time.time()
        self.resultat.delete('1.0', END)
        self.cout=0
        self.dfs_algo(s,e)
```

Cette méthode permet de calculer le temps d'exécution de l'algorithme.

IV. Affichage du résultat :

Le résultat est affiché dans la zone « Text » du Frame appelé resultat.

Ce résultat est sauvegardé par suite sous un fichier trace « algoRecherche.txt ».

```
self.resultat.insert(END," ".join(map(str, ["Parcours :", self.parcours,'\n\n'])))
self.resultat.insert(END," ".join(map(str, ["Chemin :", self.chemin,'\n\n'])))
self.resultat.insert(END," ".join(map(str, ["Cout : ", self.cout,'\n\n'])))
self.resultat.insert(END," ".join(map(str, ["Temps d'execution : ","%.4f" %(time.time() - start_time),"secondes"])))
ch= self.resultat.get("1.0",END)
fich = open('algoRecherche.txt','w')
fich.write(str(ch))
fich.close()
ch=""
```

⇨ Vous trouveriez avec ce rapport une copie du code et une vidéo démo montrant le fonctionnement de l'application.