Structures de données (LU2IN006)

Nawal Benabbou

Licence Informatique - Sorbonne Université

2020-2021

Parcours et recherche d'accessibilité

Recherche d'accessibilité

Dans un graphe (orienté ou non-orienté), on se pose les questions suivantes :

- Existe-t-il un chemin (ou une chaîne) entre deux sommets ?
- Quels sont les sommets accessibles à partir d'un sommet donné ?

Remarque

Soient r et u deux sommets. Rechercher si u est accessible depuis r revient à explorer tous les sommets accessibles à partir de r. En effet, comme l'on ignore quel chemin peut mener vers u depuis r, on doit explorer toutes les possibilités de chemins issus de r. Les pires-cas dans cette exploration sont :

- de ne pas trouver u.
- de trouver u en dernier.

Cette recherche revient donc à parcourir le graphe à partir de r. On appelle alors r la *racine* de la recherche.

Parcours et recherche d'accessibilité

Un *sous-parcours* partant d'un sommet est une exploration des "sommets de proche en proche", c'est-à-dire un déplacement de sommet en sommet en utilisant les arcs (ou les arêtes) du graphe. Plus formellement :

Sous-parcours à partir d'un sommet

On appelle *sous-parcours* d'un graphe à partir d'un sommet r une liste de sommets $L = (u_1, \ldots, u_k)$ telle que :

- $u_1 = r$.
- $u_1, \ldots, u_k$ est l'ensemble des sommets accessibles à partir de r.

Remarque : Pour tout $i \in \{2, \ldots, k\}$, il existe donc un arc (u_j, u_i) (ou une arête $\{u_j, u_i\}$) dans le graphe, avec $j \in \{1, \ldots, i-1\}$ (on parlera de bordure).

Sous-parcours et exploration

Le mot *sous-parcours* désigne une liste de sommets, mais désigne aussi le fait de parcourir le graphe. Parcourir un graphe depuis r, c'est explorer un à un les sommets de manière à former un sous-parcours.

Parcours et recherche d'accessibilité

Terminologie

Au cours d'un (sous-)parcours, on appelle :

- **Sommet visité** : un sommet qui a été ajouté au sous-parcours.
- **Sommet non-visité** : un sommet qui n'a pas (encore) été ajouté au sous-parcours.
- **Sommet ouvert** : un sommet visité dont tous les descendants (ou sommets accessibles) n'ont pas encore été visités.
- **Sommet fermé** : un sommet visité dont tous les descendants (ou sommets accessibles) ont été visités.
- **Bordure d'une liste de sommet** : étant donné une liste L de sommets visités, la bordure $B(L)$ est l'ensemble des sommets non-visités qui sont successeurs (ou voisins) d'au moins un sommet dans la liste. Formellement, dans le graphe $G = (S,A)$, on a :

$$B(L) = \{s \in S \backslash L : \exists (s',s) \in A \text{ avec } s' \in L\} \text{ (cas orienté)}$$

$$B(L) = \{s \in S \backslash L : \exists \{s',s\} \in A \text{ avec } s' \in L\} \text{ (cas non-orienté)}$$

Parcours et recherche d'accessibilité

Pseudo-code : sous-parcours à partir d'un sommet r

$L = (r)$
Tant que $B(L) \neq \emptyset$:
    Choisir un sommet $s \in B(L)$
    $L = L \cup \{s\}$
Fin Tant que

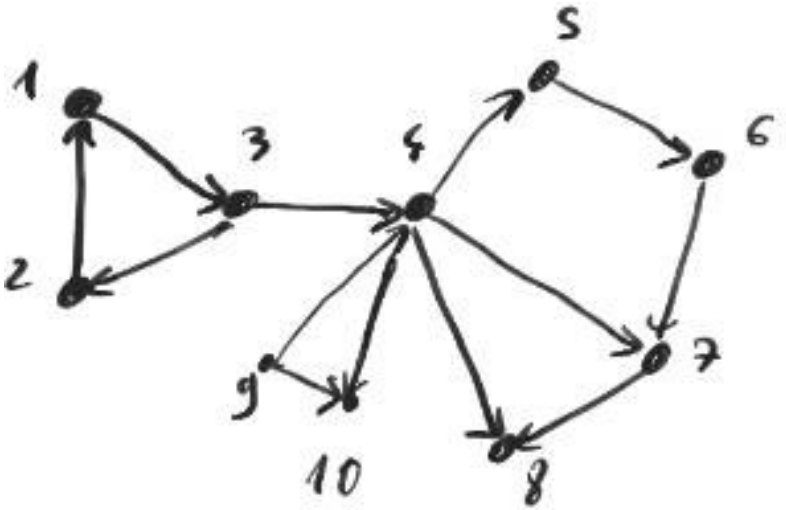
Remarque

À la fin du (sous-)parcours :

- tous les sommets visités sont fermés.
- un sommet non-visité (c'est-à-dire qui n'est pas dans le sous-parcours) est inaccessible depuis r.

Le deuxième point donne comment répondre à la question de la recherche d'accessibilité à l'aide d'un (sous-)parcours de graphe.

Exemple : graphe orienté



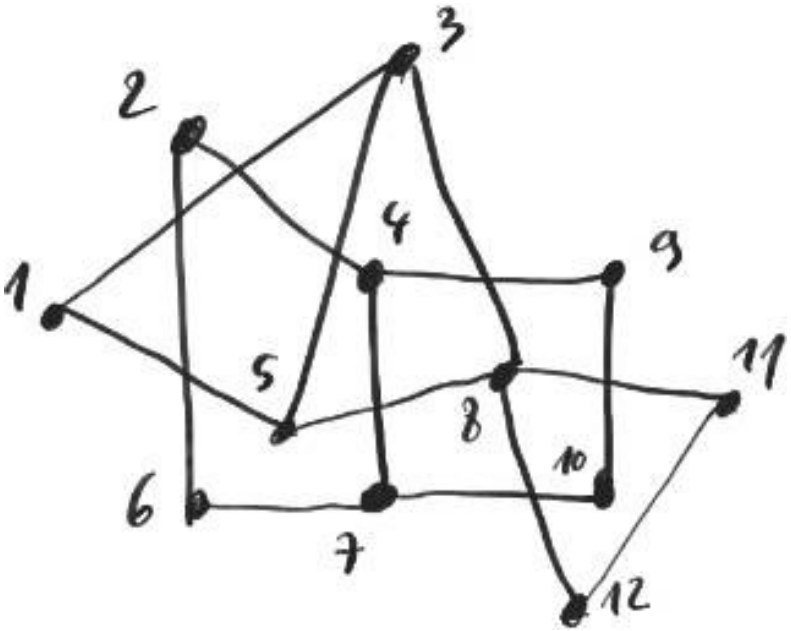
Parcours à partir du sommet s_9 :

Itération	Liste des sommets visités
1	$L_1 = (s_9)$
2	$L_2 = (s_9, s_4)$
3	$L_3 = (s_9, s_4, s_8)$
4	$L_4 = (s_9, s_4, s_8, s_5)$
5	$L_5 = (s_9, s_4, s_8, s_5, s_6)$
6	$L_6 = (s_9, s_4, s_8, s_5, s_6, s_7)$
7	$L_7 = (s_9, s_4, s_8, s_5, s_6, s_7, s_{10})$

A la fin de la procédure :

- on obtient un sous-parcours où les sommets s_1, s_2 et s_3 n'ont pas été visités.
- tous les sommets accessibles à partir de s_9 ont été trouvés.

Exemple : graphe non-orienté



Parcours à partir du sommet s_1 :

Itération	Liste des sommets visités
1	$L_1 = (s_1)$
2	$L_2 = (s_1, s_3)$
3	$L_3 = (s_1, s_3, s_8)$
4	$L_4 = (s_1, s_3, s_8, s_{11})$
5	$L_5 = (s_1, s_3, s_8, s_{11}, s_5)$
6	$L_6 = (s_1, s_3, s_8, s_{11}, s_5, s_{12})$

A la fin de la procédure :

- on obtient un sous-parcours où six sommets n'ont pas été visités.
- tous les sommets accessibles à partir de s_1 ont été trouvés.

Parcours et recherche d'accessibilité

A la fin d'un sous-parcours, on peut ne pas avoir visité tous les sommets (on s'est limité aux sommets accessibles à partir de la racine du parcours).

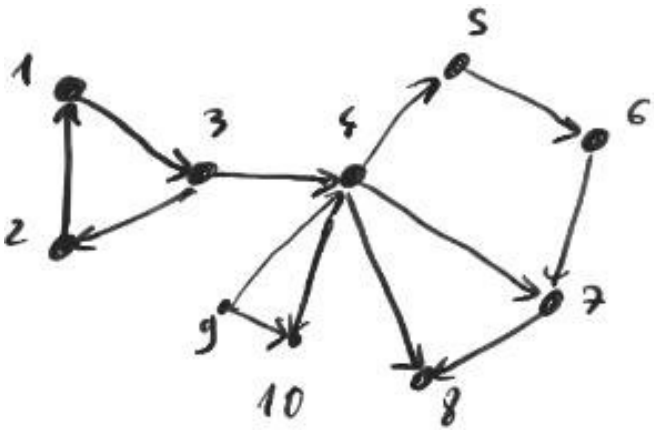
Parcours de graphe et points de régénération

Dans un *parcours*, on doit rencontrer tous les sommets du graphe. Quand ils ne sont pas tous accessibles depuis la racine, on doit "relancer" un sous-parcours à partir d'une nouvelle racine (non-visitée). Les racines de tous les sous-parcours sont appelés des *points de régénération*.

Pseudo-code : parcours de graphe à partir d'un sommet r

$L = (r)$
Tant qu'il existe un sommet non-visité :
    Si $B(L) \neq \emptyset$:
        Choisir un sommet $s \in B(L)$.
    Sinon :
        Choisir un sommet s non-visité.
    Fin Si
    $L = L \cup \{s\}$
Fin Tant que

Exemple : graphe orienté



Parcours à partir du sommet s_9 :

Itération	Liste des sommets visités
1	$L_1 = (s_9)$
2	$L_2 = (s_9, s_4)$
3	$L_3 = (s_9, s_4, s_8)$
4	$L_4 = (s_9, s_4, s_8, s_5)$
5	$L_5 = (s_9, s_4, s_8, s_5, s_6)$
6	$L_6 = (s_9, s_4, s_8, s_5, s_6, s_7)$
7	$L_7 = (s_9, s_4, s_8, s_5, s_6, s_7, s_{10})$
8	$L_8 = (s_9, s_4, s_8, s_5, s_6, s_7, s_{10}, s_2)$
9	$L_9 = (s_9, s_4, s_8, s_5, s_6, s_7, s_{10}, s_2, s_1)$
10	$L_{10} = (s_9, s_4, s_8, s_5, s_6, s_7, s_{10}, s_2, s_1, s_3)$

A la fin du parcours :

- tous les sommets ont été visités (pas de surprise).
- on a deux points de régénération : s_9 et s_2.

Stratégie de parcours

Stratégie de parcours : choix du sommet dans la bordure

Il existe plusieurs stratégies pour visiter les sommets d'un graphe lors d'un parcours :

- parcours quelconque (choix "aléatoire" dans la bordure).
- parcours en profondeur.
- parcours en largeur.
- parcours en cherchant le sommet "le plus proche" (au sens d'une distance).
- etc.

Ces différentes stratégies permettent :

- chacune de trouver les sommets accessibles depuis un sommet r.
- mais les trois dernières permettent de mettre en évidence des caractéristiques du graphe.

$\Rightarrow$ Dans le cadre de ce cours, nous allons nous intéresser à l'implémentation du parcours en profondeur et en largeur.

Parcours en profondeur

Définition

On appelle *parcours en profondeur* ou DFS (pour "Deep First Search" en anglais) le fait d'effectuer un parcours en choisissant systématiquement d'explorer un successeur (ou un voisin) du dernier sommet visité ouvert.

Pseudo-code : sous-parcours en profondeur à partir de r

$L = (r)$
Tant que $B(L) \neq \emptyset$:
    Choisir un sommet $s \in B(L)$ qui est successeur (ou voisin) du dernier sommet ouvert de L.
    $L = L \cup \{s\}$.
Fin Tant que

Implémentation

Dans un parcours en profondeur, la bordure est en fait une pile ! En effet, en empilant les sommets successeurs (ou voisins) non encore visités, on aura en dépilant les sommets en profondeur d'abord. On peut considérer deux versions :

- Une version récursive : la pile n'a pas besoin d'être codée, il suffit d'utiliser implicitement celle la pile d'exécution des fonctions récursives.
- Une version itérative : qui correspond à la dérécursivation du code récursif, où on explicite la pile des sommets à explorer.

(Sous-)parcours en profondeur : version récursive

```
1  void parcours_profondeur_rec(Graphe *G,int r, int* visit){
2
3      visit[r]=1;
4      printf("%d ",r); /* ou ajout de r dans une liste */
5
6      ElementListeA* cour = G->tabS[r]->L_adj;
7      while (cour!=NULL){
8          int v = cour->a->j;
9          if (visit[v]==0){
10             parcours_profondeur_rec(G,v,visit);
11         }
12         cour = cour->suiv;
13     }
14 }
```

Notes :

- Le code est donné pour un graphe orienté (c'est pour cela que l'on choisit l'extrémité j de l'arc à la ligne 8).
- `visit` est un tableau de "booléen" initialisé à false (0) pour tous les sommets.
- Cette fonction affiche la liste des sommets résultant du parcours, mais on aurait pu la stocker dans un tableau ou dans une liste chaînée.

(Sous-)parcours en profondeur : version itérative

```
1  void parcours_profondeur_ite(Graphe *G, int r){
2
3      int* visit=(int*)malloc(G->nbS*sizeof(int));
4      int i;
5      for (i=0;i<G->nbS;i++){
6          visit[i]=0;
7      }
8      visit[r]=1;
9      Pile* P = creerPile();
10     empile(P,r);
11
12     while (!(estPileVide(P))){
13         int u=depile(P);
14         printf("%d ",u); /* ou ajout de u dans une liste */
15
16         ElementListeA* cour = G->tabS[u]->L_adj;
17         while (cour!=NULL){
18             int v=cour->a->j;
19             if (visit[v]==0){
20                 visit[v]=1;
21                 empile(P,v);
22             }
23             cour=cour->suiv;
24         }
25     }
26     printf("\n");
27 }
```

Complexité temporelle et spatiale

Complexité temporelle

Dans les deux versions, chaque ligne de code est soit :

- exécutée une seule fois.
- répétée au maximum une fois par sommet (soit n fois au total).
- répétée au maximum une fois par arc (ou arêtes) (soit m ou $2m$ fois).

De plus chaque ligne de code correspond à des opérations en $O(1)$. Donc la complexité du parcours en profondeur est en $O(m+n)$.

Occupation mémoire

Les versions récursives ou itératives sont similaires sur l'occupation mémoire :

- la version récursive va empiler au pire n appels récursifs dans la pile d'exécution.
- la version impérative va contenir au pire n sommets dans la pile.

Si n n'est pas trop grand, les deux versions sont similaires. Par contre, si n est grand, comme la version récursive n'est pas terminale, la mémoire peut être très occupée : on préférera donc la version itérative dans ce cas.

Parcours en profondeur pour la détection de circuits/cycles

Rappel

Un circuit est un chemin dont le premier sommet est le même que le dernier. Un cycle est une chaîne dont le premier sommet est le même que le dernier.

Problème de détection d'un circuit (ou d'un cycle)

Détecter dans un graphe s'il existe un circuit (ou un cycle) et, le cas échéant, donner un tel circuit (ou cycle). Ce problème se résout en adaptant l'algorithme de parcours en profondeur et donc se résout aussi en $O(n+m)$.

Principe

Au cours d'un parcours en profondeur, lorsque l'on visite le sommet v, on teste s'il existe un arc (appelé *arc retour*) reliant v à un sommet ouvert u. En effet, si u est ouvert et que l'on est en train de visiter v, c'est qu'il existe un chemin de u à v, ainsi en ajoutant l'arc (v,u) à ce chemin, on obtient un circuit.

Parcours en largeur

Parcours en largeur

On appelle *parcours en largeur* ou BFS (pour "Breadth-First Search" en anglais) le fait d'effectuer un parcours en choisissant systématiquement d'explorer un successeur (ou un voisin) du premier sommet visité ouvert.

Pseudo-code : sous-parcours en largeur à partir de r

$L = (r)$
Tant que $B(L) \neq \emptyset$:
    Choisir un sommet $s \in B(L)$ qui est successeur (ou voisin) du premier sommet ouvert de L.
    $L = L \cup \{s\}$.
Fin Tant que

Implémentation

Pour un parcours en largeur, il suffit de coder la bordure par une `file` (à la place d'une pile pour le parcours en profondeur). En terme d'implémentation, il n'y pas de version récursive possible. Le code suivant est exactement le même que la version itérative du parcours en profondeur où "file" a remplacé "pile". La complexité est ainsi la même en $O(n+m)$.

(Sous-)parcours en largeur

```
1  void parcours_largeur_ite(Graphe *G, int r){
2
3      int* visit=(int*)malloc(G->nbS*sizeof(int));
4      int i;
5      for (i=0;i<G->nbS;i++){
6          visit[i]=0;
7      }
8      visit[r]=1;
9      File* F = creerFile();
10     enfile(P,r);
11
12     while (!(estFileVide(F))){
13         int u=deFile(F);
14         printf("%d ",u); /* ou ajout de u dans une liste */
15
16         ElementListeA* cour = G->tabS[u]->L_adj;
17         while (cour!=NULL){
18             int v=cour->a->j;
19             if (visit[v]==0){
20                 visit[v]=1;
21                 enfile(F,v);
22             }
23             cour=cour->suiv;
24         }
25     }
26     printf("\n");
27 }
```

Parcours en largeur pour le chemin le plus court

Plus court chemin en nombre d'arcs (ou arêtes)

Un parcours en largeur à partir d'un sommet r permet de connaître le nombre minimal d'arcs/arêtes permettant d'atteindre tout sommet u à partir de r.

Principe

Il suffit de mettre à jour un tableau D tel que :

- $D[r] = 0$ (initialisation),
- $D[v] = D[u] + 1$ pour chaque nouveau sommet v ajouté à la file depuis le sommet prédécesseur u.