Structures de données (LU2IN006)

Cours 8 : Parcours en longueur/largeur et composantes connexes

Nawal Benabbou

Licence Informatique - Sorbonne Université

2022-2023

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

```c
void parcours_profondeur_ite(Graphe *G, int r){

    int* visit=(int*)malloc(G->nbS*sizeof(int));
    int i;
    for (i=0;i<G->nbS;i++){
        visit[i]=0;
    }
    visit[r]=1;
    Pile* P = creerPile();
    empile(P,r);

    while (!(estPileVide(P))){
        int u=depile(P);
        printf("%d ",u); /* ou ajout de u dans une liste */

        ElementListeA* cour = G->tabS[u]->L_adj;
        while (cour!=NULL){
            int v=cour->a->j;
            if (visit[v]==0){
                visit[v]=1;
                empile(P,v);
            }
            cour=cour->suiv;
        }
    }
    printf("\n");
}
```

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

```c
void parcours_largeur_ite(Graphe *G, int r){

    int* visit=(int*)malloc(G->nbS*sizeof(int));
    int i;
    for (i=0;i<G->nbS;i++){
        visit[i]=0;
    }
    visit[r]=1;
    File* F = creerFile();
    enfile(P,r);

    while (!(estFileVide(F))){
        int u=deFile(F);
        printf("%d ",u); /* ou ajout de u dans une liste */

        ElementListeA* cour = G->tabS[u]->L_adj;
        while (cour!=NULL){
            int v=cour->a->j;
            if (visit[v]==0){
                visit[v]=1;
                enfile(F,v);
            }
            cour=cour->suiv;
        }
    }
    printf("\n");
}
```

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.

Théorie des graphes

La théorie des graphes est un domaine des mathématiques discrètes qui consiste à étudier les propriétés des graphes. Par exemple :

- Comment colorier les sommets d'un graphe non orienté de manière à ne pas utiliser la même couleur pour deux sommets adjacents et en utilisant le moins de couleurs possibles ? (problème de coloration de graphes).
- Comment représenter un graphe sur un plan de manière à avoir le moins d'arêtes (ou d'arcs) qui se croisent dans le plan ?
- Comment peut-on caractériser des graphes ayant des particularités, par exemple ne contenant pas de cycles impairs ou ne contenant pas de chemins de plus de 3 arêtes ?

Théorie des graphes et algorithmique des graphes

Algorithmique des graphes

On s'intéresse plutôt ici à l'algorithmique des graphes, qui cherche à proposer des algorithmes résolvant des problèmes où l'on doit répondre à une question, et ce quelle que soit l'instance (c'est-à-dire la donnée, le graphe). Exemple :

- Quels sont les sommets accessibles à partir d'un sommet ?
- Existe-t-il un circuit dans un graphe ?
- Quel est le plus court chemin reliant deux sommets donnés ?
- Comment transférer des données d'un sommet à un autre ?

Remarque : Toutes les questions de ce type dans les graphes n'ont pas toujours de réponses satisfaisantes à ce stade de la connaissance scientifique ! Par exemple, il y a des questions pour lesquelles on ne connaît pas d'algorithmes efficaces capables de répondre à la question quelle que soit l'instance de graphe. C'est le cas du célèbre *problème du voyageur de commerce* qui consiste à trouver un plus court circuit (appelé alors tournée) passant par tous les sommets et revenant à son point de départ. On ne connaît, à ce jour, que des algorithmes de complexité exponentielle capables de répondre à cette question.

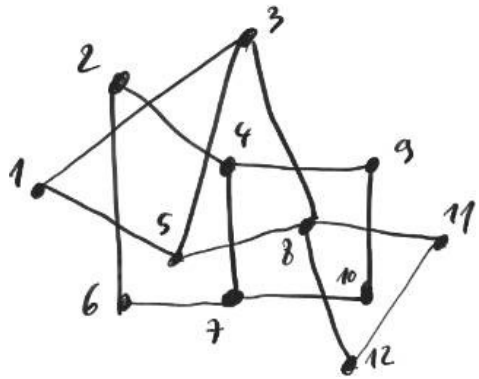
Calcul des composantes connexes

Définition : composante connexe

Dans un graphe non orienté, une composante connexe est un sous-graphe induit maximal connexe. Cela correspond à un ensemble de sommets tels que :

- tous les sommets de l'ensemble sont accessibles les uns des autres.
- aucun sommet en dehors de l'ensemble n'est accessible depuis les sommets de cet ensemble.

Exemple : Dans ce graphe, il y a exactement deux composantes connexes.



Question :

Dans un graphe non orienté, comment déterminer efficacement ses composantes connexes ? Une manière efficace de répondre à la question est d'utiliser un parcours, ce qui nous permet de découvrir une nouvelle composante connexe à chaque point de résurrection. Par contre, quand le graphe est construit incrémentalement, il est plus efficace de mettre à jour les composantes connexes grâce à la structure de données *Union-Find*.

Type de données abstrait : Partition

La structure de données Union-Find est une implémentation particulière du type abstrait de données appelé partition.

Type abstrait de données : partition

Ce type de données représente une partition d'un ensemble fini d'éléments X. Une partition de X est un ensemble de sous-ensembles de X, deux à deux disjoints, et dont l'union est égale à l'ensemble X. Chaque sous-ensemble de la partition est communément appelée une *classe d'équivalence*, et on choisit arbitrairement un de ses éléments pour la représenter. Opérations classiques sur ce type de données :

- CreerClasse(x) : crée une classe dont le seul membre et représentant est x.
- Union(x,y) : fusionne les classes d'équivalence des éléments x et y.
- Find(x) : retourne le représentant de la classe de x.

Pour le calcul des composantes connexes, l'ensemble X que l'on souhaite partitioner est S (l'ensemble des sommets du graphe). L'objectif est de le partitioner de manière à ce que les classes d'équivalence correspondent aux composantes connexes du graphe.

Partition et calcul des composantes connexes d'un graphe

Pour construire la partition correspondant aux composantes connexes du graphe, il suffit de commencer par créer une classe différente pour chaque élément de S. Puis, pour chaque arête du graphe, il s'agit de fusionner les classes de ses extrémités si elles sont différentes, et ne rien faire sinon.

Pseudo-code pour le problème du calcul des composantes connexes

Pour chaque sommet $s \in S$:
    CreerClasse(s)
Pour chaque arête (i,j) du graphe :
    Si Find(i) != Find(j) :
        Union(i,j)
    Fin Si
Fin Pour

Il s'agit maintenant de choisir une implémentation du type abstrait partition pour que cet algorithme soit le plus efficace possible.

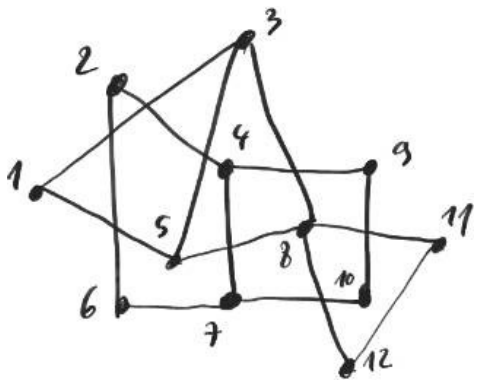
Une première implémentation possible : table de hachage

Implémentation avec une table de hachage

On peut implémenter un partition à l'aide d'une table de hachage qui à tout élément associe un entier désignant le numéro de sa classe d'équivalence.

Exemple : Dans un graphe, on peut associer au sommet i la case i de la table, qui contiendrait le numéro du sommet qui représente la classe. Pour cet exemple, on pourrait utiliser la table suivante :

1	2	1	2	1	2	2	1	2	2	1	1



Complexité des opérations

Avec cette implémentation, la complexité des opérations :

- Find(x) est en $O(1)$ car il suffit d'accéder à une case de la table.
- Union(x,y) est en $O(n)$ car il faut parcourir toute la table pour trouver les éléments qui ont le même représentant que x, pour indiquer que leur représentant est maintenant celui de y.

Une deuxième implémentation : listes doublement chaînées

Implémentation avec une liste doublement chaînée par classe

On peut implémenter une partition à l'aide d'une liste doublement chaînée pour chaque classe, et on peut prendre la tête de la liste comme représentant. Dans ce cas, la complexité des opérations :

- Find(x) est en $O(n)$ car il faut remonter toute la liste pour connaître le représentant d'un élément.
- Union(x,y) est en $O(1)$ car il suffit de concaténer les deux listes en mettant l'une à la fin de l'autre (mise à jour de deux pointeurs).

Remarques : On peut améliorer la complexité de Find en ajoutant à chaque élément un pointeur "représentant" qui pointe vers la tête de liste, ce qui conduit à une complexité en $O(1)$. Cependant, cela pénalise Union car, après avoir concaténer deux listes, il faut maintenant changer le pointeur représentant pour tous les éléments de la liste qui a été mise à la fin (complexité en $O(n)$). Pour améliorer cela, on pourrait en plus utiliser l'heuristique d'union pondérée, qui consiste à concaténer deux listes en mettant toujours la plus courte à la fin de la plus longue. Dans ce cas, on peut montrer que dans le pire des cas, le coût de k opérations successives (créer, union et/ou find) est en $O(k+nlogn)$.

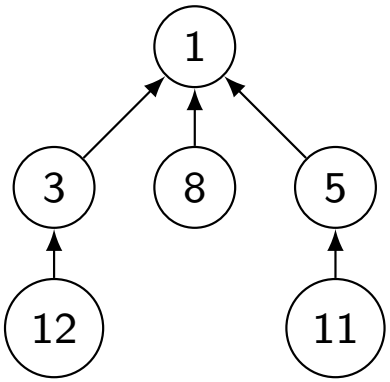
Une troisième implémentation : forêt

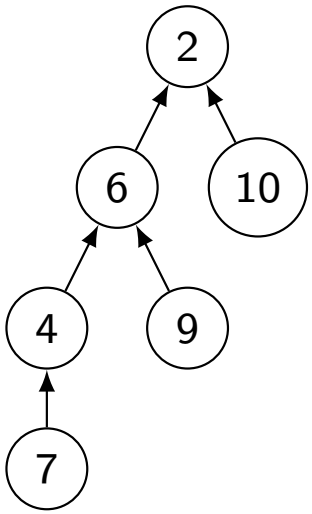
Pour représenter optimalement une partition, il faut utiliser une implémentation par forêt (c'est-à-dire un ensemble d'arbres).

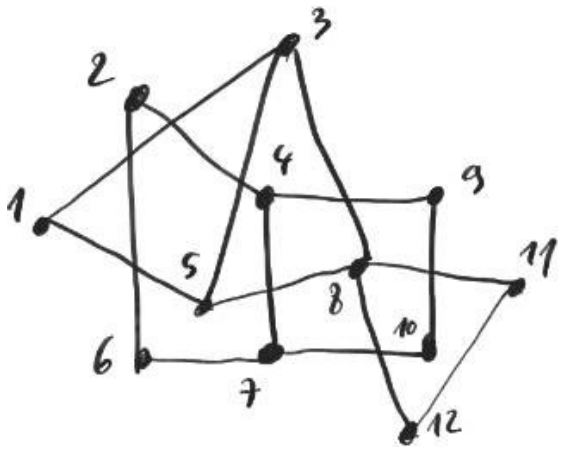
Implémentation avec une forêt

On peut représenter une partition avec une forêt, où chaque arbre correspond à une classe d'équivalence. Le représentant de chaque classe est la racine de l'arbre, et chaque noeud de l'arbre contient une référence vers son père.

Exemple : La forêt suivante permet de représenter les composantes connexes de notre graphe exemple.







Implémentation naïve des opérations avec une forêt

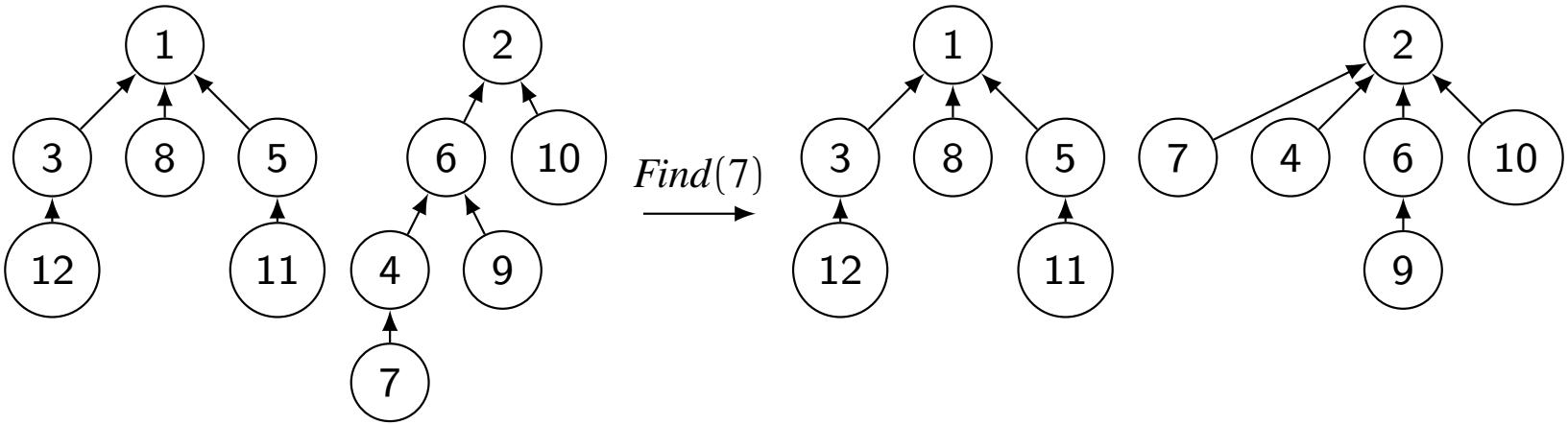
```
1  typedef struct noeud{
2      int elem;
3      struct noeud* pere;
4  } Noeud;
5
6  Noeud* creerClasse(int elem){
7      Noeud* n = (Noeud*)malloc(sizeof(Noeud));
8      n->elem = elem;
9      n->pere= n;
10     return n;
11 }
12 Noeud* find(Noeud* n){
13     if (n->pere != NULL)
14         return find(n->pere);
15     else
16         return n;
17 }
18 void union(Noeud* n1, Noeud* n2){
19     if (find(n1) != find(n2))
20         n1->pere=n2;
21 }
```

Remarque : Cette implémentation n'améliore pas la complexité des opérations par rapport à l'implémentation avec des listes doublement chaînées, car les arbres obtenus peuvent être très déséquilibrés.

Vers une implémentation optimale avec une forêt

Pour améliorer ces opérations, il faut utiliser les deux heuristiques suivantes :

- *Find avec compression de chemin :* l'idée est de profiter de chaque appel à Find pour aplatir l'arbre. Plus précisément, tous les nœuds traversés quand on remonte jusqu'à la racine doivent être modifiés en chemin pour que leur père devienne la racine à la fin de l'appel.



- *Union par rang :* On stocke dans chaque nœud un majorant de la hauteur de son sous-arbre que l'on appelle rang (le rang d'un arbre contenant un seul noeud vaut 1). Lorsque l'on réalise une union, on décide que c'est la racine de plus grand rang qui devient le père de la racine de plus petit rang. Si les deux racines ont le même rang, on fait un choix arbitraire, et la racine qui devient père de l'autre doit augmenter son rang de 1.

Implémentation optimale des opérations avec une forêt

```
1  typedef struct noeud{
2      int elem;
3      int rang;
4      struct noeud* pere;
5  } Noeud;
6
7  Noeud* creerClasse(int elem){
8      Noeud* n = (Noeud*)malloc(sizeof(Noeud));
9      n->elem = elem;
10     n->pere= n;
11     n->rang= 1;
12     return n;
13 }
14
15 Noeud* find(Noeud* n){
16     if (n->pere != n)
17         n->pere = find(n->pere);
18     return n->pere;
19 }
```

Implémentation optimale des opérations avec une forêt

```
1  void union(Noeud* n1, Noeud* n2){
2      Noeud* racine1 = find(n1);
3      Noeud* racine2 = find(n2);
4      if (racine1 != racine2){
5          if (racine1->rang < racine2->rang){
6              racine1->pere = racine2;
7          }else{
8              racine2->pere = racine1;
9          }
10         if (racine1->rang == racine2->rang){
11             racine1->rang = racine1->rang+1;
12         }
13     }
14 }
```

Théorème

L'implémentation par forêt avec les deux heuristiques correspond à la structure de données appelée Union-Find. On peut montrer que dans le pire des cas, le coût de k opérations successives (créer, union et/ou find) est en $O(k\alpha(n))$ où $\alpha(n)$ est l'inverse de la fonction d'Ackermann (et $\alpha(n) \leq 5$ dans tous les cas pratiques). En conséquence, on peut considérer que le *coût amorti* par opération est une constante. Par ailleurs, il a été prouvé que ce n'est pas possible d'obtenir une meilleure complexité pour implémenter le type abstrait partition.