

Algorithmique et structures de données

le 12 décembre 2008, durée 3h, aucun document autorisé.

Question de cours:

1) Qu'est-ce qu'un graphe connexe?

Un graphe est connexe si pour tout couple de sommets (x,y) il existe une chaîne de x à y

2) Qu'appelle-t-on numérotation topologique d'un graphe sans circuit ? Vous appellerez la définition et vous donnerez un exemple à 6 sommets.

Une numérotation topologique N d'un graphe orienté G sans circuit associe à chaque sommet x un numéro $N(x)$ de sorte que pour tout arc (x,y) de G , $N(x) < N(y)$.

Exercice 1: listes

On considère dans cet exercice des listes simplement chaînées d'entiers, avec les types suivants:

```
typedef struct maillon{int val; struct maillon *suiv;}maillon,*liste;
```

Question 1: Ecrire une fonction dont le prototype est `int recherche(liste l,int x);` qui renvoie 1 si la valeur x est dans la liste l , et 0 sinon. Calculer sa complexité.

```
int recherche(liste l, int x)
{while ((l)&&(l->val!=x)) l=l->suiv;
if (l) return(1); else return(0);
}
```

Complexité: au pire, la boucle while fait un parcours total de la liste l , et chaque itération est de complexité constante donc si la liste l a n éléments, on obtient $O(n)$.

Question 2: Ecrire une fonction dont le prototype est `liste intersect(liste p,liste q);`, qui renvoie une liste chaînée comportant les valeurs qui se trouvent à la fois dans p et dans q . Cette fonction ne doit pas détruire les listes initiales.

```
liste ajout_tete(liste l, int x)
{liste p=(liste) malloc(sizeof(maillon));
p->val=x; p->suiv=l; return p;
}

liste intersect(liste p, liste q)
{liste inter;
while (p)
{if recherche(q,p->val) {inter=ajout_tete(inter;p->val);}
p=p->suiv;
}
return(inter);
}
```

Question 3: Calculer la complexité d'intersect en fonction du nombre d'éléments n_p de p et n_q de q .

La boucle sur p comporte np itérations. Dans chaque itération, il y a un appel à recherche de complexité $O(nq)$ et au plus un appel à ajout_tête, plus une affectation, les deux ont une complexité constante. D'où une complexité $O(np*nq)$

Question 4: Supposons maintenant que les deux listes p et q sont triées par ordre croissant des valeurs. Ecrire une fonction `liste_intersect_croissant(liste p, liste q)` qui renvoie une liste chaînée comportant les valeurs en ordre croissant qui se trouvent à la fois dans p et dans q. Cette fonction ne doit pas détruire les listes initiales.

```
void ajout_queue(liste *tete, liste *queue, int x)
{liste p=(liste) malloc(sizeof(maillon));
p->val=x; p->suiv=null;
if(!(*tete)) *tete=*queue=p;
else {(*queue)->suiv=p; *queue=p;}
}

liste intersect_croissant(liste p, liste q)
//si l'une des deux listes est vide, il ne peut y avoir d'élément commun.
//ici on tire parti de l'ordre des listes: si le premier el de p (resp.a) est plus petit que le premier el de
//q (resp.p) alors il est certain que le premier el de p (resp q) ne peut être dans la liste q (resp p).
//il reste donc à calculer l'intersection entre la liste p->suiv et q (resp p et q->suiv)
//En cas d'égalité des plus petits éléments, cet élément fait partie de l'intersection, il est donc dans
//la liste résultat, et l'on n'a plus qu'à calculer l'intersection entre les listes p->suiv et q->suiv
//les éléments sont insérés dans la liste resultat du plus petit au plus grand, donc on doit insérer en
//queue.
{liste tete,queue;
while (p&&q)
{if(p->val == q->val) {ajout_queue(&tete,&queue,p->val);p=p->suiv;q=q->suiv;}
else {if (p->val<q->val) p=p->suiv;
else q=q->suiv;}
}
return(tete);
}
```

Question 5: Calculer la complexité d'`intersect_croissant`.

La complexité est de $O(np+nq)$, car à chaque itération de la boucle while la somme des longueurs des deux listes décroît d'au moins 1, et que la complexité d'une itération est constante.

Question 6: Supposons qu'on dispose d'une fonction `liste_trie(liste l)` qui renvoie la liste des maillons de l en ordre croissant, et qui a la meilleure complexité dans le pire des cas possible, parmi les algorithmes vus en cours. Quelle est cette complexité?

La meilleure complexité d'un algo de tri vu en cours est celle du tri fusion $O(n\log(n))$

Question 7: Discuter des avantages et inconvénients entre les deux alternatives suivantes en partant de deux listes p et q non triées

- a) calculer leur intersection avec la fonction `intersect`, puis trier la liste avec la fonction `trie`.
 b) trier `p`, trier `q` avec la fonction `trie`, puis appeler la fonction `intersect_croissant` sur les listes triées.

En terme de complexité dans le pire des cas:

solution a). L'intersection des deux listes a une taille au plus égale à $nr = \min(np, nq)$. On a donc dans le pire des cas une complexité $O(np*nq + nr\log(nr))$ notons que $nr\log(nr) < np*nq$, c'est donc ce dernier terme qui domine la complexité globale.

solution b) en terme de complexité, on a $O(np\log(np) + nq\log(nq))$ pour les deux tris, puis $O(nr)$ pour l'intersection. A chaque fois ces trois quantités sont inférieures strictement à $np*nq$, donc l'approche b) a une meilleure complexité dans le pire des cas.

Exercice 2: Graphes, listes et parcours

On considère la représentation d'un graphe donnée par les types suivants:

```
typedef struct G{int n; liste *tabsom;} *graphe;
(Le type liste est celui de l'exercice 1)
```

Pour un graphe `g`, une fois alloué le tableau `g->tabsom`, `g->tabsom[i]` contient la liste chaînée des successeurs du sommet `i`. `g->n` est le nombre des sommets de `g`.

Dans cet exercice, vous pouvez utiliser l'une des fonctions de l'exercice 1 (ou des questions précédentes) lorsque cela vous paraît nécessaire.

On donne les fonctions de parcours suivantes (vues en cours et en TD):

```
int largeur(graphe g, int depart, int file[g->n])
{ int N=g->n;
  bool M[N];
  int io=0, jo=0;
  sommet x, y;
  liste p;
  for(x=0; x<N; x++) M[x]=0;
  M[file[jo++]=depart]=1; // file[jo]=depart; M[depart]=1; jo++;
  //on met depart dans la file d'attente et on le marque
  while(io<jo)
    for(p=g->tabsom[x=file[io++]]; p; p=p->suiv)
      if(!M[y=p->val])
        M[file[jo++]=y]=1;
  return jo;
  /*jo : nombre de sommets que l'on peut atteindre
   à partir de depart y compris lui même*/
}
```

```
void Profondeur(graphe g, int depart, int pere[g->n])
{
  void parcours(int s)
  {
    liste p;
    for(p=g->tabsom[s]; p; p=p->suiv)
      if(pere[p->val]<0)
        pere[p->val]=s,
        parcours(p->val);
  }
  int j;
  for(j=0; j<g->n; j++) pere[j]=-1;
  pere[depart]=depart;
  parcours(depart);
}
```

Question 1: Dessiner un graphe orienté à 8 sommets et 15 arcs, et appliquer les deux parcours à partir d'un sommet de votre choix. Vous indiquerez pour chaque itération du parcours en largeur l'état de la file et les sommets marqués, ainsi que l'arborescence du parcours. Pour le parcours en profondeur, vous indiquerez la suite des appels récursifs.

Question 2: Modifier les deux fonctions pour qu'elles renvoient une liste chaînée des sommets marqués et calculer leur complexité. Par hypothèse, les fonctions s'appelleront `liste_largeur` et `liste_profondeur`.

```
//On indique en rouge les modifs apportées au code
Liste liste_largeur(graphe g,int depart,int file[g->n])
//les sommets sont ajoutés au fur et à mesure en queue de la liste résultat, dès qu'ils sont sortis
//de la file.
{ int N=g->n;
  bool M[N];
  int io=0, jo=0;
  sommet x, y;
  liste p;
  liste largtete,largqueue;
  for(x=0;x<N;x++) M[x]=0;
  M[file[jo++]=depart]=1; // file[jo]=depart; M[depart]=1; jo++;
  //on met depart dans la file d'attente et on le marque
  while(io<jo)
  x=file[jo++]; ajout_queue(&largtete,&largqueue,x);
  for(p=g->tabsom[x];p;p=p->suiv)
  if(!M[y=p->val])
    M[file[jo++]=y]=1;
  return largtete;
}

liste liste_profondeur(graphe g, int depart,int pere[g->n])
{liste proftete, profqueue;
//les sommets paramètres de la fonction parcours sont ajoutés en queue de la liste résultat en début
// d'appel. On a ainsi les sommets dans l'ordre des appels.
void parcours(int s)
{liste p;
ajout_queue(&proftete,&profqueue,s);
for(p=g->tabsom[s];p;p=p->suiv)
if(pere[p->val]<0)
  pere[p->val]=s,
```

```

    parcours(p->val);
}
int j;
for(j=0;j<g->n;j++) pere[j]=-1;
pere[depart]=depart;
parcours(depart);
return(proftete);
}

```

Soit G un graphe orienté. On appelle $I(G)$ le graphe construit à partir de G en inversant le sens des arcs.

Question 3: Ecrire une fonction qui à partir d'un graphe G , alloue l'espace nécessaire et calcule le graphe $I(G)$. La fonction ne devra pas détruire la structure du graphe G , et renvoyer une représentation de $I(G)$. Le prototype s'écrit donc: `graphe inverse(graphe g)`.

```

graphe inverse(graphe g)
graphe inv=(graphe) malloc(sizeof(struct G));
{inv->n=g->n;
inv->tabsom=(*liste) malloc(g->n*sizeof(liste))
liste p;
int i;
for(i=0;i<g->n;i++)
{for(p=g->tabsom[i]; p; p=p->suiv)
// p->val est un successeur de i, donc dans le graphe inversé, i est un succ de p->val. On l'ajoute à la
//liste des succ de p->val, à l'aide de la fonction définie en début de sujet.
        {inv->tabsom[p->val]=ajout_tete(inv->tabsom[p->val], i);}
}
return inv;}

```

La complexité est en fait celle d'un parcours de toute la structure représentant g , soit $O(n+m)$.

Question 4: On dit que x est un ancêtre de y s'il existe un chemin de x à y . Ecrire une fonction qui, à partir d'un graphe, renvoie la liste des ancêtres d'un sommet donné: `lanc(graphe g, int y)`

Calculer sa complexité.

Soient x et y deux sommets. Remarquons que s'il existe un chemin de x à y dans G , alors il existe un chemin de y à x dans $I(G)$ et réciproquement. Par conséquent x est un ancêtre de y dans G si et seulement si c'est un descendant de y dans $I(G)$. Or les descendants d'un sommet y sont obtenus à l'aide d'un parcours à partir de y .

par conséquent, le code qui répond à cette question est le suivant:

```

liste lanc(graphe g, int y)
{graphe inv=inverse(g); return (liste_largeur(inv, y));} //(ça marcherait aussi avec liste_profondeur)

```

La complexité de cette fonction est celle de inverse + celle d'un parcours en largeur, soit $O(n+m)$.

Question 5: Ecrire une fonction qui, à partir d'un graphe G et deux sommets x et y , renvoie la liste des sommets z tels qu'il existe un chemin de x à y passant par z . Calculer sa complexité.

```
Liste intermediaire(graphe g, int x,inty)
```

```
{liste desc=liste_largeur(g,x);
```

```
liste anc=lanc(g,y);
```

```
desc=trie(desc);anc=trie(anc); return(intersect_croissant(desc,anc));
```

```
}
```

La complexité de cette fonction est en fait $O(n\log(n)+n+m)$. Selon le nombre d'arcs, ce terme est dominé par $n\log(n)$ (lorsque m n'est pas trop gros) ou par m .

On peut trouver un algorithme plus efficace pour faire la même chose, mais le sujet n'en demandait pas plus. Par contre, vous pouvez tâcher de le trouver en exercice de révision.

Question 6: Appliquer votre fonction à votre graphe et deux sommets de votre choix.