Cours du vendredi 19 octobre 2007

Durant ce cours, nous avons travaillé sur l'implémentation du parcours en largeur selon la représentation du graphe. Pour chaque représentation, nous avons examiné des éléments permettant d'améliorer la complexité de l'algorithme.

Rappel de l'algorithme de parcours en largeur à partir d'un sommet s dans un graphe non orienté.

Initialiser l'état de chaque sommet à libre
enfiler s sur la file des sommets ouverts FO
P=s % il s'agit de la liste du parcours
Tant que FO est non vide
    Soit x le sommet en tête de FO (le plus ancien, donc)
    Si x a un voisin libre y, alors enfiler y sur FO, P=P.y
    Sinon, défiler x de FO.

Implémentation 1 :

le graphe est représenté par une matrice d'adjacence M et le nombre de sommets n.
Rappel : M[i][j] vaut 1 s'il y a une arête entre i et j, 0 sinon.

Une première version :

On utilise un tableau complémentaire permettant de stocker pour chaque sommet l'état (libre, ouvert, fermé) de chaque sommet. Etat[i] vaudra 0 si le sommet est libre, 1 s'il est ouvert, -1 s'il est fermé). La fonction retourne la liste du parcours ici sous forme d'un tableau. La file est représentée avec une liste circulaire comme vu dans un cours précédent.

```
Typedef struct maillon {int som ; struct maillon*suiv ;} maillon,
*filesom ;
Filesom enfiler(int x, filesom f)
{filesom p= (filesom) malloc(sizeof(maillon)) ;
p->som=x;
if (f) {p->suiv=f->suiv; f->suiv=p; return(p);}
p->suiv=p; return(p);
}
int tetefile(filesom f)
{return (f->suiv->som);}
filesom defiler(filesom f)
{if (f->suiv= =f)
      {free(f); return(0);}
filesom p=f->suiv;
f->suiv=p->suiv;
free(p);
return(f);
}
int * parcourslarg1(int **M, int n, int s)
{int i,k=0,x; filesom fo=0; int *etat=( int*) malloc(n*sizeof(int));
int* parc=(int *) malloc(n*sizeof(int))
for (i=0;i<n;i++) {parc[i]=-1; etat[i]=0;}
fo=enfiler(s,fo); etat[s]=1; parc[k++]=s;
while (fo)
{x=tetefile(fo);
      i=0;
      while((i<n)&& ((M[x][i]! =1)||(  etat[i]))) { i++;}
```

```
      % la condition d'arrêt de cette boucle c'est que soit i=n, et x n'a
pas de voisin libre, soit M[x][i] vaut 1 , i est donc voisin de x, et
etat[i] vaut 0, i est donc libre.%
      if (i<n)
      {etat[i]=1 ; fo=enfiler(i,fo) ; parc[k++]=i;}
      else
      {etat[x]=-1; fo=defiler(fo);}
}
return(parc);
}
```

Analyse de la complexité.

Tout d'abord analysons la complexité en dehors de la boucle while (fo): on a deux boucles d'intialisation, donc une constante fois n: O(n)

Complexité d'une itération de la boucle while (fo) :
Le while interne comporte au maximun n itérations. Les autres instructions ont une complexité constante (enfiler, défiler, tests + affectation).
Par conséquent chaque itération de la boucle while (fo) a une complexité O(n)
La question est : combien y-a-t-il d'itérations de la boucle while(fo) ?

On observe, dans le fonctionnement de l'algorithme, qu'un sommet x qui est en tête de file va le rester tant qu'il lui reste un voisin libre. On observe également qu'à chaque itération de la boucle while(fo), ce même sommet a un voisin libre de moins (puisqu'on l'ouvre). Par conséquent, le nombre d'itérations où un sommet x est en tête de file est au maximum égal au nombre de voisins de x d°(x). Comme un sommet fermé ne peut plus réapparaître dans la file, le nombre d'itérations de while (fo) est majoré par n*(somme des degrés de sommets).

Notons m le nombre d'arêtes du graphe. On voit facilement que somme des degrés des sommets=2*m.
On peut donc dire que le nombre d'itérations de la boucle while est majoré par 2*m.
D'où une complexité globale de O(m*n) :a*n+2b*n*m, où a et b sont des constantes est majoré par une constante (par ex a+2b) fois m*n.

Analyse des défauts de cette implémentation

On observe une perte de temps lors de la recherche d'un voisin libre : en effet, à chaque fois qu'un sommet x reste en tête de file, on va systématiquement rechercher son premier voisin libre à partir du début de la ligne de M : en repassant donc par les voisins qu'on a ouvert lors des itérations précédentes.

Solution proposée par un étudiant : ne pas remettre i à 0 à chaque fois, seulement lorsqu'on change de sommet en tête de file (et donc lorsqu'on défile). D'où le texte :

```
int * parcourslarg2(int **M, int n, int s)
{int i,k=0,x; filesom fo=0; int *etat=( int*) malloc(n*sizeof(int));
int* parc=(int *) malloc(n*sizeof(int))
for (i=0;i<n;i++) {parc[i]=-1; etat[i]=0;}
fo=enfiler(s,fo); etat[s]=1; parc[k++]=s; i=0;
while (fo)
{x=tetefile(fo);
      while((i<n)&& ((M[x][i]! =1)||(  etat[i]))) { i++;}
      % la condition d'arrêt de cette boucle c'est que soit i=n, et x n'a
pas de voisin libre, soit M[x][i] vaut 1 , i est donc voisin de x, et
etat[i] vaut 0, i est donc libre.%
      if (i<n)
```

```
      {etat[i]=1 ; fo=enfiler(i,fo) ; parc[k++]=i;}
      else
      {etat[x]=-1; i=0;
      fo=defiler(fo);}
}
return(parc);
}
```

Analyse de l'efficacité de cette modification:

Elle a un impact sur la complexité de la boucle while la plus interne. En effet, tant qu'un sommet x reste en tête de file, i va parcourir une seule fois tous les indices de 0 à n.

Par conséquent, comme un sommet arrive en tête de file, puis y reste un certain nombre d'itérations, puis ressort de la file et n'y entre plus jamais, la contribution globale à la complexité de la boucle while interne est donc de a*n*n (une constante fois n pour chaque sommet).

La contribution du reste est d'une constante fois le nombre d'itérations du while (fo), qui n'a pas changé : O(m).
D'où une complexité de $O(m+n^2)$.

Est-ce que cela améliore la complexité précédente ?
Oui de toute manière, puisque pour chaque graphe, on passe moins de temps dans la boucle while interne.

Mais sur de gros graphes, est-ce que l'ordre de grandeur change ?
Oui lorsque le graphe est très dense, c'est-à-dire lorsque le nombre d'arêtes est une certaine fraction de n^2 : dans ce cas, m*n est de l'ordre de n^3 tandis que l'amélioration a une complexité n^2.

Cela a moins d'impact lorsque le graphe est peu dense m=cste*n.

Autre problème :

En fait, dans cette implémentation, le résultat du parcours est un tableau dont on ne renvoie pas le nombre d'éléments : pour savoir quels sont les sommets accessibles à partir de s, on doit donc parcourir le tableau jusqu'à trouver un -1. Une liste chaînée serait peut-être plus appropriée si ce nombre est faible par rapport à l'ensemble des sommets.

Implémentation 2 :

Le graphe est représenté par des listes d'adjacence.
Typedef struct graphe{int n ; filesom * lsucc;} graphe, *pgraphe;
Pgraphe g;

Si g est une variable de type pgraphe, alors g->n contient le nombre de sommets du graphe, et g->lsucc est un tableau de n cases, tq g->lsucc[i] est la tête d'une liste chaînée des sommets voisins du sommet i.

On garde les types et fonctions de manipulation de file déjà vus plus haut.

```
filesom  parcourslarg3(pgraphe g, int s)
{int i,k=0,x; filesom parc,p,fo=0; int *etat=( int*) malloc(n*sizeof(int));
for (i=0;i<n;i++) { etat[i]=0;}
fo=enfiler(s,fo); etat[s]=1; parc=enfiler(s,0);

while (fo)
{x=tetefile(fo);
p=g->lsucc[x];
      while(p)
      {if (!etat[p->som]) %le voisin de x est libre
      {etat[p->som]=1 ; fo=enfiler(p->som,fo) ; parc=enfiler(p->som,parc);}
      p=p->suiv;
      }
      % a la fin de ce while, x n'a plus de voisin libre: il faut donc le
fermer
      fo=defiler(fo); etat[x]=-1;
}
return(parc);
}
```

Analyse :

Cette version tire parti du fonctionnement du parcours en largeur, puisque cette fois, une itération de la boucle while(fo) contient toutes celles des versions précédentes où un même sommet est en tête de file jusqu'à ce qu'il soit défilé.

Par conséquent dans cette version, le nombre d'itérations de la boucle While (fo) est au plus égal au nombre de sommets n.
Comme pour chaque sommet x, la boucle while(p) a une complexité qui est une constante fois le degré du sommet x, en tout, la contribution de cette boucle while au temps de calcul sera donc une constante a fois 2*m (somme des degrés).
La contribution des autres éléments (défiler, changement de l'état de x) est du type b*n (une constante fois le nb d'itération de while(fo)).

D'où une complexité globale en O(n+m).

Cela permet donc d'améliorer considérablement l'efficacité de l'algorithme, surtout lorsque le graphe est peu dense (si m est de l'ordre de n^2, l'ordre de grandeur est le même).