Examen juin 2011 - 1ere session

Durée : 2H

Documents autorisés: supports et notes manuscrites de cours et de TD/TME
– Barême indicatif –

Note importante : Les questions peuvent toutes être traitées indépendamment les unes des autres, au besoin en s'appuyant sur l'existence des fonctions des questions précédentes.

Exercice 1 (4 points) – Adressage ouvert

Soit T une table de hashage de taille $taille(T) = 15$, utilisant l'adressage ouvert avec probing linéaire. Les clés sont des entiers et la fonction de hashage est simplement :

$$h(c) = c \text{ modulo } taille(T)$$

Q 1.1 (1 point) Rappeler comment fonctionne l'insertion d'un entier dans T

Question de cours

Q 1.2 (1 point) Insérer les clés 3,27,18,30,42,33

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
30			3	18	33							27	42	

Q 1.3 (2 points) On veut insérer les clés de 101 à 110 dans T. Quel problème cela pose-t-il ? Quelle solution proposeriez-vous ?

Il y a déjà 6 clés dans T. Avec un adressage ouvert, T ne peut pas contenir plus de $taille(T)$ clés. Donc il faut augmenter la taille de T et re-dispatcher les clés déjà enregistrés, puis enfin insérer les nouvelles clés.

Exercice 2 (2 points) – Utilisation de structures

On considère une machine-outil d'une usine assemblant des cartes-mères d'ordinateur. Cette machine est capable d'effectuer toutes les opérations d'assemblage (placement d'un composant, soudure, test,...). On note $t_1, t_2, ..., t_n$ ces n opérations. On connaît la liste L des précédences entre ces opérations, c'est-à-dire que L indique si une opération t_i doit être réalisée avant une opération t_j. Dans L, chaque

operations t_i n'a au plus que 5 opérations t_j avec laquelle elle forme une précédence.

L'usine U_0 suivant illustre ces définitions : l'usine U_0 comporte 5 opérations t_1, t_2, t_3, t_4 et t_5. La liste L des précédences dans U_0 indique que t_1 doit être réalisée avant t_3, t_3 avant t_2, t_5 avant t_4, t_5 avant t_3, t_5 avant t_1 et t_2 avant t_4.

On appelle $(o_1, ..., o_n)$ un *ordre* des opérations si o_i, $i = 1, ..., n$, est une opération et $o_i \neq o_j$ si $i \neq j$ (en d'autres termes, $(o_1, ..., o_n)$ est une permutation de $(t_1, t_2, ..., t_n)$.
On dit qu'un ordre $(o_1, ..., o_n)$ est *réalisable* s'il respecte les précédences de L, c'est-à-dire que, dans cet ordre, pour tout $i < j$, il n'existe pas de précédences où o_j doit être réalisé avant o_i.

Q 2.1 (0,5 point) Proposez un ordre réalisable pour l'usine U_0.

Il n'y en a qu'un $(t_5, t_1, t_3, t_2, t_4)$.

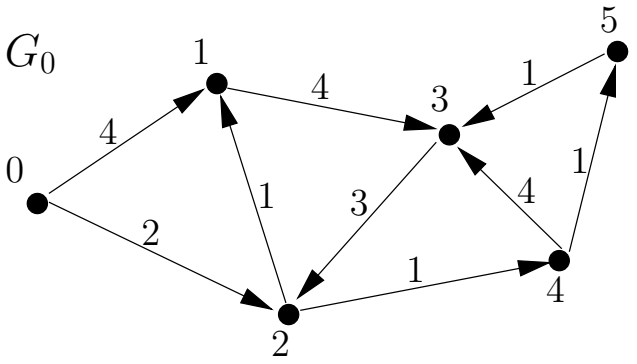
Q 2.2 (1,5 point) Quelle est la structure de de données la plus adaptée pour stocker et manipuler les données de ce problème ? Vous la décrirez sans donner le code correspondant.

Etant donné que les données à stocker concernent une liste de précédence L composée de couple d'opérations (t_i, t_j), on pourrait utiliser une matrice dont chaque ligne et chaque colonne correspondent à des opérations et chaque case à une précédence. Néanmoins, comme il y a seulement au plus $5n$ cases utiles dans ce tableau, cette structure est mal adaptée. Une structure de matrice creuse est alors la plus adaptée.
Cette structure correspond en fait à celle d'un graphe par liste d'adjacence où les arcs correspondent aux précédences : un tableau dont chaque case correspond à une opération et dans chaque case la liste chaînée des opérations qu'il précède.

Exercice 3 (8 points) – Plus court chemin

On considère ici un graphe orienté $G = (V, A)$ où $V = \{0, 1, ..., n-1\}$ est l'ensemble des n sommets et A celui des arcs. Chaque arc $(i, j) \in A$ est muni d'une longueur l_{ij}. On rappelle qu'un *chemin* P de G est une suite d'arcs $((u_0, u_1), (u_1, u_2), (u_2, u_3), ..., (u_{l-1}, u_l))$. On dit que les sommets u_0 et u_l sont les extrémités de P, les autres sommets sont dits sommets intermédiaires. On rappelle également que la *longueur* d'un chemin est la somme des longueurs des arcs de ce chemin.

Considérons le graphe G_0 suivant avec 6 sommets et 9 arcs.

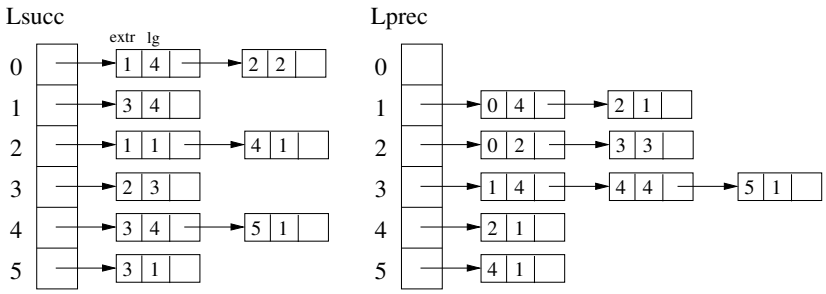


Partie 1

Nous allons tout d'abord implémenter une structure de graphe pour manipuler nos instances. Les graphes que nous allons considérer contiennent peu d'arêtes, nous choisissons une structure de liste d'adjacence. Pour implémenter l'algorithme de Bellman, nous allons avoir besoin, pour chaque sommet de connaître la liste des arcs prédécesseurs et la liste des arcs successseurs. Voici une structure permettant de définir un tel graphe.

```
1  typedef struct arc{
2      float lg;
3          /* Longueur de l'arc */
4      int extr;
5          /* Extremite opposee de l'arc */
6      struct arc *suiv;
7          /* Pointeur sur l'element-arc suivant dans la liste */
8  } Elnt_liste_arc;
9
10 typedef struct graph{
11   Elnt_liste_arc ** Lsucc;
12       /* Tableau tel que Lsucc[i] est la liste des arcs successeurs de i */
13   Elnt_liste_arc ** Lprec;
14       /* Tableau tel que Lprec[i] est la liste des arcs predecesseurs de i */
15   int nbsom;
16       /* Nombre de sommets */
17 } Graphe;
```

La figure suivante représente le codage du graphe G_0 selon la structure donnée :



Q 3.1 (1 point) Ecrivez la fonction `void alloue_graphe(Graphe *G, int n);` qui permet d'allouer une structure de graphe avec n sommets et aucun arc.

```
void alloue_graphe(Graphe *G, int n){
  int i;

  G->nbsom=n;

  G->Lsucc=(Elnt_liste_arc **) malloc(G->nbsom*sizeof(Elnt_liste_arc*));
  G->Lprec=(Elnt_liste_arc **) malloc(G->nbsom*sizeof(Elnt_liste_arc*));
  for (i=0;i<G->nbsom;i++){
    G->Lsucc[i]=NULL;
    G->Lprec[i]=NULL;
  }

}
```

Q 3.2 (1 point) Ecrivez la fonction `void ajoute_arc(Graphe G, int i, int j, float l);` qui ajoute un arc (i, j) dans la structure de graphe (attention, il faut penser qu'un arc est à la fois le prédécesseur d'un sommet et le successeur d'un autre sommet).

```
void ajoute_arc(Graphe *G, int i, int j, float l){

  Elnt_liste_arc *nouv=(Elnt_liste_arc*) malloc(sizeof(Elnt_liste_arc));
  nouv->extr=j;
  nouv->lg=l;
  nouv->suiv=G->Lsucc[i];
  G->Lsucc[i]=nouv;

  nouv=(Elnt_liste_arc*) malloc(sizeof(Elnt_liste_arc));
  nouv->extr=i;
  nouv->lg=l;
  nouv->suiv=G->Lprec[j];
  G->Lprec[j]=nouv;

}
```

Q 3.3 (0,5 point) Ecrivez une fonction `main` permettant d'entrer le graphe G_0 en mémoire.

```
int main(){

  Graphe G;

  int n=6;

  alloue_graphe(&G,n);

  ajoute_arc(G,0,1,4);
  ajoute_arc(G,0,2,2);
  ajoute_arc(G,1,3,4);
  ajoute_arc(G,2,1,1);
  ajoute_arc(G,2,4,1);
  ajoute_arc(G,3,2,3);
  ajoute_arc(G,4,3,4);
  ajoute_arc(G,4,5,1);
  ajoute_arc(G,5,3,1);
```

Partie 2

On s'intéresse ici à un algorithme de recherche d'un plus court chemin appelé "Algorithme de Bellman" ou parfois "algorithme de Programmation Dynamique pour les plus courts chemins" (qui est différent de l'algorithme de Dijkstra vu en cours). On se limitera ici à rechercher des chemins à partir du sommet 0.

L'algorithme de Bellman fonctionne en 2 étapes : la première est dite étape Directe et la seconde est dite étape Retour.

L'étape Directe est composée de $n-1$ itérations, numérotées $k = 1, ..., n-1$, qui remplissent chacune un tableau V^k de n valeurs numériques. En fait, chaque case $V^k[i]$, $i = 0, ..., n$, indique la longueur d'un plus court chemin entre 0 et le sommet i qui passe par au plus k sommets intermédiaires. Ainsi, comme un chemin dans G contient au plus $n-1$ sommets intermédiaires, la case $V^{n-1}[i]$ contient donc la longueur d'un plus court chemin de 0 à i.
Le tableau est initialement rempli de la valeur $+\infty$ dans chaque case. Le tableau V^0 est initialisé de la façon suivante : $V^0[i] = l_{0i}$ si l'arc $(0, i)$ existe dans A et $V^0[i] = +\infty$ sinon.
A chaque itération $k \subset \{1, ..., n-1\}$, l'étape Directe calcule les valeurs de V^k à partir de celles de V^{k-1}. En fait, lors de ce calcul, chaque valeur de $V^k[i]$, $i \in \{1, ...n-1\}$, est calculée par la formule de récurrence suivante :

$$V^k[i] = \min(V^{k-1}[i], \min_{(j,i)\in A}(V^{k-1}[j] + l_{ji})).$$

On peut remarquer que, lors de ce calcul, la valeur $V^k[i]$ dérive d'une unique valeur de $V^{k-1}[j]$ où $j \in \{0, ..., n-1\}$. On doit conserver tout au long des 2 étapes de quel indice j de V^{k-1} dérive chacune des valeurs de V^k.

L'étape Retour de l'algorithme de Bellman a pour but de reconstituer le plus court chemin. Pour cela, on "remonte" à l'envers la liste des indices qui ont permis de déterminer le chemin. Pour retrouver le chemin menant en i, il suffit de se déplacer à partir de la case $V^{k-1}[i]$, puis sur la case $V^{k-1}[ind]$ où ind est le sommet qui a permis de calculer $V^{k-1}[i]$ et on réitère cela, jusqu'à rencontrer le sommet 0

A titre d'illustration, on a ici utilisé l'algorithme pour le graphe G_0 de l'exemple. Le tableau V suivant indique, dans chaque case $V^k[i]$, le couple (l, j) où l est la longueur du plus court chemin de 0 à i d'au plus k sommets intermédiaires et j est le numéro du sommet qui réalise le minimum de la formule de récurrence.

	0	1	2	3	4	5
k=0	(0,-1)	(4,0)	(2,0)	(∞,-1)	(∞,-1)	(∞,-1)
k=1	(0,-1)	(3,2)	(2,0)	(8,1)	(3,2)	(∞,-1)
k=2	(0,-1)	(3,2)	(2,0)	(7,1)	(3,2)	(4,4)
k=3	(0,-1)	(3,2)	(2,0)	(5,5)	(3,2)	(4,4)
k=4	(0,-1)	(3,2)	(2,0)	(5,5)	(3,2)	(4,4)
k=5	(0,-1)	(3,2)	(2,0)	(5,5)	(3,2)	(4,4)

Dans cet exemple, la longueur contenue dans la case $V^3[3]$ est obtenue comme le minimum entre $V^2[3] = 7$, $V^2[1] + l_{13} = 3 + 4 = 7$ et $V^2[4] + l_{43} = 3 + 4 = 7$ et $V^2[5] = 4 + 1 = 5$; l'indice contenu dans la case $V^3[3]$ et donc le sommet 5 car c'est pour ce sommet que le sommet est atteint.

On applique ensuite l'étape Retour pour déterminer le plus court chemin de 0 au sommet 3 : le sommet 3 dérive du sommet 5, qui lui-même dérive du sommet 4, qui lui-même dérive du sommet 2, qui dérive du sommet 0 : le plus court chemin de 0 à 5 est donc le chemin $(0, 2), (2, 4), (4, 5), (5, 3)$.

On veut conserver toutes les étapes de l'algorithme de Bellman, c'est-à-dire toutes les vecteurs V^k de chacune des itérations. Pour cela, on va utiliser un tableau `V` de n cases pointant chacune sur un tableau V^k. Chacune des cases `V[k][i]` contient la longueur du plus court chemin de 0 à i d'au plus k arcs et le sommet dont dérive i dans ce chemin.

Q 3.4 (1 point) Proposez une structure de données pour implémenter ce tableau (contenant des éléments nommés `Elnt_v`) ainsi qu'une fonction permettant de l'allouer et de l'initialiser (c'est-à-dire de mettre dans chaque case les valeurs `#define infty 100` pour les longueurs et -1 pour les indices).

```
typedef struct{
   float val;
   int deriv_ind;
} Elnt_V;
```

dans le main

```
Elnt_V ** V;
```

```
void alloue_V(Elnt_V *** V, int n){
  int k,i;
  *V=(Elnt_V **) malloc(n*sizeof(Elnt_V*));

   for (k=0;k<n; k++)
      (*V)[k]=(Elnt_V *) malloc(n*sizeof(Elnt_V));
```

```
7
8     for (k=0;k<n;k++)
9      for (i=0;i<n;i++){
10       (*V)[k][i].val=infty;
11       (*V)[k][i].deriv_ind=-1;
12     }
13  }
```

Q 3.5 (1,5 points) Ecrivez la fonction `void EtapeDirecte_V0(Graphe G, Elnt_V ** V, int n);` qui effectue l'initialisation du vecteur `V[0]` comme le demande l'algorithme de Bellman (dans le cas où l'on recherche des chemins à partir du sommet 0). Penser à indiquer que les sommets dérivent de 0 lors de cette initialisation.

```
1  void EtapeDirecte_V0(Graphe G, Elnt_V ** V, int n){
2    Elnt *cour;
3    V[0][0].val=0;
4    V[0][0].deriv_ind=-1;
5
6
7    cour=G.Lsucc[0];
8    while (cour!=NULL){
9       V[0][cour->extr].val=cour->lg;
10      V[0][cour->extr].deriv_ind=0;
11      cour=cour->suiv;
12   }
13 }
```

Q 3.6 (2 points) Ecrivez la fonction `void EtapeDirecte(Graphe G, Elnt_V ** V, int n);` qui implémente l'étape Directe de l'algorithme de Bellman pour rechercher des chemin à partir du sommet 0.

```
1  void EtapeDirecte(Graphe G, Elnt_V ** V, int n){
2    int i,k,j,minj;
3    Elnt_liste_arc *cour;
4
5
6     EtapeDirecte_V0(G,V,n);
7
8     for(k=1;k<n;k++){
9        for(i=0;i<n;i++){
10    V[k][i].val=V[k-1][i].val;
11    V[k][i].deriv_ind=V[k-1][i].deriv_ind;
12    cour=G.Lprec[i];
13    while (cour!=NULL){
14      if (V[k-1][cour->extr].val+cour->lg<V[k][i].val){
15        V[k][i].val=V[k-1][cour->extr].val+cour->lg;
16        V[k][i].deriv_ind=cour->extr;
17      }
18      cour=cour->suiv;
19    }
20        }
```

```
    }

}
```

2 pt si cela marche en autorisant même les solutions complexes
1,5 pt s'il y a quelques erreurs légères
1 pt s'il y a beaucoup d'erreurs
0,5 pt s'il y a quelques idées dans le sens de l'énoncé
0 pt s'il y a vraiment trop de fautes

Q 3.7 (1 point) Ecrivez la fonction void `void EtapeRetour(Elnt_V ** V, int n, int i);` qui implémente l'étape Retour de l'algorithme de Bellman. Cette fonction ne fait que simplement afficher (dans un sens ou dans l'autre) la liste des sommets du plus court chemin de 0 à i.

```
void EtapeRetour(Elnt_V ** V, int n, int i){
  int ind;

  ind=i;
  printf("%d derive de : ",i);
  while (ind!=0){
    ind=V[n-1][ind].deriv_ind;
    printf("%d ", ind);
  }
  printf("\n");
}
```

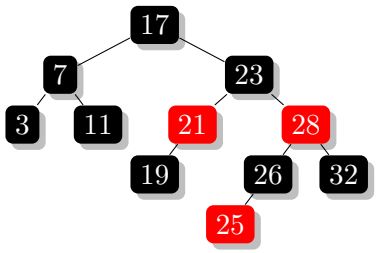
Exercice 4 (8 points) – Arbre Rouge-Noir

Les Arbres Binaires de Recherche (ABR) dans leur version équilibrée (AVL) ont des propriétés très intéressantes mais gardent certains désavantages : par exemple, la suppression d'un élément peut demander un nombre important de rotations. D'autres structures de données existent donc, qui essayent de pallier à ces inconvénients. Par exemple, les arbres rouge-noir. Un arbre rouge-noir (ARN) est un arbre dont les nœuds sont colorés en noir ou rouge et vérifie :

1. l'arbre est un arbre binaire de recherche ;
2. la racine d'un ARN est noire ;
3. les enfants d'un nœud rouge (s'ils existent) sont noirs ;
4. le nombre de nœuds noirs sur tous les chemins de la racine à une feuille est constant.

Dans cet exercice, on appellera simplement "chemin" dans un arbre, un chemin de la racine à une feuille de l'arbre.

La figure ci-dessous représente un ARN (le gris correspondant à la couleur rouge : ici les sommets rouges sont les sommets 21, 25 et 28).



Q 4.1 (2 points) Montrez que dans un ARN, on ne peut avoir deux nœuds rouges successifs le long d'un chemin. En quoi cette propriété aide-t-elle à interdire des ARN trop déséquilibrés ? Plus précisément, il s'agit de prouver que, dans un ARN, la longueur du plus grand chemin ne peut pas être plus de 2 fois plus grands que celui du plus petit chemin.

• Un nœud rouge ne peut pas avoir de fils rouge. ■
• Tout chemin d'un ARN a le même nombre de nœuds noirs. Donc, le plus petit chemin possible ne contient que des nœuds noirs (hn) alors que le plus grand chemin possible possède au plus hn nœuds noirs ains que hn nœuds rouges également (en alternance). Il est donc de taille $2 \cdot hn$. ■

Q 4.2 (1 point) Modifiez la structure du type `btree`, vue en cours, afin d'obtenir un type `ARNtree` représentant un nœud d'ARN.

```
1 typedef enum {rouge,noir} Couleurs;
2
3 typedef struct s_ARNtree {
4   int content;
```

```
  Couleurs co;

  struct s_ARNtree* fd;
  struct s_ARNtree* fg;
} ARNtree;
```

Q 4.3 (2 points) Écrivez une fonction C (`int nbr_rouges(ARNtree* t)`) de calcul du nombre de nœuds rouges dans un ARN.

```
int nbr\_rouges(ARNtree* t) {
  if (t==NULL) return 0;

  int tmp=nbr\_rouges(t->fg)+nbr\_rouges(t->fd);
  if (t->co==rouge) tmp++;
  return tmp;
}
```

Q 4.4 (3 points) Écrivez une ou des fonctions C permettant de vérifier qu'un `ARNtree` vérifie bien les propriétés d'un arbre rouge-noir. Attention à la condition (4) ! Ne pas oublier la condition (1) !

```
bool checkARN(ARNtree *t) {
  if (t==NULL) return true;
  if (t->co==rouge) return false; // condition 2

  int hn=-1;
  checkARNrec(t,0,&hn)
}

bool checkARNrec(ARNtree *t,int current,int* hn) {
  if (t==NULL) { //feuille
    if (*hn==-1)
      *hn=current; // premier chemin fini : on initialise *hn
      return true;
    } else {
      return (current==*hn); // condition 4
    }
  }

  if (t->fg!=NULL && t->fg->content>t->content) return false; //condition 1
  if (t->fd!=NULL && t->fd->content<t->content) return false; // condition 1

  if (t->co==rouge) {
    if (t->fg!=NULL && t->fg->co==rouge) return false; // condition 3
    if (t->fd!=NULL && t->fd->co==rouge) return false; // condition 3

    return checkARNrec(t->fg,current,hn) && checkARNrec(t->fd,current,hn);
  } else {
    return checkARNrec(t->fg,current+1,hn) && checkARNrec(t->fd,current+1,hn);
  }
}
```