TP1

GRAPHES

VEYSSEIRE Daniel

&

FABRE Mickael

Supposition

Le graphe à traiter est sans circuit.

Dans le fichier de description du graphe, les activités sont écrites dans l'ordre.

Listing Complet Du Programme

Seance 1

Le parcours en profondeur est réalisé par la fonction :

```
vector <int> DFS()
```

Elle permet de réaliser un tri topologique et renvoie donc les numéros des sommets triés. Le tri a été implémenté de la même façon que celle vue en cours. (Les programmes on été commentés)

L'affichage des vecteurs d'entiers peut se faire avec la fonction

```
void afficheIntVector(vector<int> v)
```

Avec Test_10_trie.dat (i.e Test_10.dat en ayant trié les sommets dans le fichier)

On obtient :

```
v[0] 0 v[1] 4 v[2] 2 v[3] 1 v[4] 6 v[5] 3 v[6] 8 v[7] 5
v[8] 7 v[9] 9
```

Ce qui signifie que les Activités sont triés dans cet ordre :

0,4,2,1,6,3,8,5,7,9

Ce qui est bien le résultat escompté.

La fonction

```
vector <int> longueurCheminOpti()
```

Renvoie la longueur du chemin optimiste le plus court entre s et chaque sommet.

En utilisant le fichier Test_10.dat et en affichant le résultat avec `afficheIntVector` on obtient :

```
v[0] 0 v[1] 1 v[2] 1 v[3] 8 v[4] 1 v[5] 7 v[6] 7 v[7] 8
v[8] 7 v[9] 10
```

Ce qui signifie que le sommet 0 est à une distance 0 de la source, le sommet 1 à une distance de 1, etc.

Ce qui correspond bien à ce qui est calculé manuellement.

L'algorithme se déroule ainsi :

On initialise toutes les distances des activités au sommet à l'infini.

La distance du sommet est mise à 0.

On fait un tri topologique, puis pour chacun de ses sommets pris dans l'ordre, on regarde ses successeurs.

Si la durée minimal de l'activité + la distance de l'activité est inférieur < distance du successeur

Alors distance du successeur = durée minimal de l'activité + la distance de l'activité

Ainsi grâce au tri topologique et au fait que le graphe n'a pas de circuit, lorsqu'on traite un sommet, on a déjà traité tous ses prédécesseurs donc on ne peut plus diminuer sa distance. Si il n'avait pas de prédécesseur, sa distance reste à l'infini car il est impossible d'aller de s à ce sommet.

De même la fonction

```
vector <int> longueurCheminPessi()
```

Renvoie la longueur du chemin pessimiste le plus court entre s et chaque sommet de la même façon.

Avec Test_10.dat on obtient :

```
v[0] 0 v[1] 2 v[2] 2 v[3] 11 v[4] 2 v[5] 17 v[6] 11 v[7] 14
v[8] 14 v[9] 19
```

Seance 2

La fonction

```
vector< vector<int> > toutChemin()
```

Renvoie la liste des chemins de s à t.

Elle utilise la fonction

```
vector<vector<int>> recChemin(
vector<vector<int>> allPath,
vector<int> pathRunning,
Activity activiteCourante)
```

Qui est appelée récursivement.
La fonction recChemin prend en paramètre le chemin courant, l'ensemble des chemins calculés et l'activité courante et renvoie l'ensemble des chemins calculés concaténé avec les chemins descendant de l'activité activiteCourante dans l'algo de recherche en profondeur.

La remarque faite dans le sujet (en évitant les redondances) est difficilement compréhensible car on est en présence d'un graphe sans circuit donc il ne peut pas y avoir de redondance avec l'algorithme de descente en profondeur.

La fonction

```
void afficheIntVectorVector(vector< vector<int> > v)
```

Permet d'afficher un vecteur de vecteur d'entier.
Le résultat sur l'ensemble des chemins pour Test_10.dat est le suivant :

```
vecteur[0]
v[0] 0 v[1] 1 v[2] 6 v[3] 7 v[4] 9
vecteur[1]
```

```
v[0] 0 v[1] 1 v[2] 3 v[3] 8 v[4] 9
vecteur[2]
v[0] 0 v[1] 1 v[2] 3 v[3] 5 v[4] 7 v[5] 9
vecteur[3]
v[0] 0 v[1] 2 v[2] 8 v[3] 9
vecteur[4]
v[0] 0 v[1] 2 v[2] 6 v[3] 7 v[4] 9
vecteur[5]
v[0] 0 v[1] 2 v[2] 5 v[3] 7 v[4] 9
vecteur[6]
v[0] 0 v[1] 4 v[2] 8 v[3] 9
vecteur[7]
v[0] 0 v[1] 4 v[2] 7 v[3] 9
```

Ce qui signifie qu'il y a 8 chemins de s à t qui sont :

0,1,6,7,9

0,1,3,8,9

0,1,3,5,7,9

0,2,8,9

0,2,6,7,9

0,2,5,7,9

0,4,8,9

0,4,7,9

La fonction

```
int Regrets(vector< vector<int> > allPath,int numChemin)
```

Calcule le regret pour un chemin et prend en paramètre la liste des chemins et le numéro du chemin voulu.

La fonction

```
int afficheRegretMinChem(vector< vector<int> > allPath)
```

Prend en paramètre tous les chemins et retourne l'indice du chemin avec le regret minimal.

La fonction

```
void afficheRegrets(vector< vector<int> > allPath)
```

permet d'afficher tous les regrets.
Pour le fichier Test_10.dat, on obtient:

```
regret 0 14
regret 1 19
regret 2 15
regret 3 16
regret 4 17
regret 5 10
regret 6 14
regret 7 7
```

Ce qui signifie que le chemin 0 (passant donc par les sommets 0,1,6,7,9, cf page precedente) a un regret de 14, etc...

La fonction

```
void chRobusteMinRegret(vector< vector<int> > allPath, int&
chem,int& reg)
```

Utilise le passage par référence. Elle permet de renvoyer le numéro du chemin robuste et la valeur du regret minimal correspondant.

Pour le fichier Test_10.dat, on a :

```
chemin robuste v[0] 0 v[1] 4 v[2] 7 v[3] 9

regret 7
```

On pourrait s'arrêter là car on a répondu à toutes les questions, mais on a pas répondu à la question p2.

Ambiguité sur la notion de regret des chemins passant par une activité i

Il y a deux manières de comprendre ce qui est attendu pour la question p2. Le calcul du chemin robuste et du regret en sera changé.

Possibilité 1 : Le regret d'un chemin p entre s et t passant par i est défini comme étant la différence entre la longueur d'un chemin p passant par i dans le cas ou toutes les activités de p sont exécutées selon leur durée maximale (les autres activités étant réalise selon leur durée minimal) et la longueur du plus court chemin **passant par i** pour le même scénario de durée.

Cette interprétation du regret est parfaitement possible :

On doit nécessairement réaliser l'activité i, on veut maintenant minimiser le regret parmi ces chemins que je suis obligé de prendre.

Possibilité 2: Le regret d'un chemin p entre s et t passant par i est défini comme étant la différence entre la longueur d'un chemin p passant par i dans le cas ou toutes les activités de p sont exécutées selon leur durée maximale (les autres activités étant réalise selon leur durée minimal) et la longueur du plus court chemin **ne passant pas forcément** par i pour le même scénario de durée.

Cette interprétation du regret semble être plus en accord avec la description du regret faite dans le sujet mais paraît étrange à mettre en pratique. En effet quel est l'intérêt de prendre en compte des chemins par lesquels je ne vais pas passer? Difficile de comprendre son utilité... Peut être pour pouvoir mieux comparer le regret si on passe par une activité ou pas. D'où l'ambiguïté.

Possibilité 1

La fonction

```
vector< vector<int> > toutCheminPassantPar(vector< vector<int> >
allPath,int numAct)
```

Permet, à partir de allPath, de ne garder que les chemins contenant l'activité i.

Pour cela on a utilisé la fonction

```
bool belongInt(vector<int> v,int elem)
```
qui permet de savoir si un élément appartient à un vecteur.

On peut ainsi avoir la liste des chemins possibles passant par i et utiliser `chRobusteMinRegret` pour calculer le chemin robuste passant par i et le regret correspondant.

Ce que fait la fonction
```
void chemRobRegPassntPar(vector< vector<int> > allPath)
```

Qui affiche tous les chemins robustes pour chaque activité et les regrets associés en utilisant la définition de regret décrite en possibilité 1.

Resultat pour Test_10.dat :

chemin robuste passant par 0

0 4 7 9

regret associe: 7

chemin robuste passant par 1

0 1 6 7 9

regret associe: 9

chemin robuste passant par 2

0 2 5 7 9

regret associe: 4

chemin robuste passant par 3

0 1 3 5 7 9

regret associe: 4

chemin robuste passant par 4

0 4 7 9

regret associe: 1

chemin robuste passant par 5

0 2 5 7 9

regret associe: 6

chemin robuste passant par 6

0 1 6 7 9

regret associe: 3

chemin robuste passant par 7

0 4 7 9

regret associe: 5

chemin robuste passant par 8

0 2 8 9

regret associe: 6

chemin robuste passant par 9

0 4 7 9

regret associe: 7

Quelques petits calculs permettent de vérifier la cohérence. La robustesse et le regret du chemin passant par 4 est facile à calculer par exemple.

Possibilite 2

La fonction

```
void regretsPassantPar(vector< vector<int> > allPath,int act)
```

Permet d'afficher le chemin robuste et le regret parmi les chemins passant par l'activite act en utilisant la notion de regret définie par possibilité 2.

La fonction

```
void tousRegrets(vector< vector<int> > allPath)
```

Permet d'afficher tous les regrets et tous les chemins robustes passants par chaque activité du graphe d'après la définition du regret donnée par possibilité 2.

On obtient les résultats suivant qui diffèrent de ceux trouvés avec la possibilité 1 :

chemin robuste entre s et t passant par 0
0 4 7 9
regret 7
chemin robuste entre s et t passant par 1
0 1 3 5 7 9
regret 15
chemin robuste entre s et t passant par 2
0 2 5 7 9
regret 10
chemin robuste entre s et t passant par 3
0 1 3 5 7 9
regret 15
chemin robuste entre s et t passant par 4
0 4 7 9
regret 7
chemin robuste entre s et t passant par 5
0 2 5 7 9
regret 10
chemin robuste entre s et t passant par 6
0 2 6 7 9
regret 17
chemin robuste entre s et t passant par 7
0 4 7 9

regret 7

chemin robuste entre s et t passant par 8

0 4 8 9

regret 14

chemin robuste entre s et t passant par 9

0 4 7 9

regret 7

Après des calculs laborieux, j'ai trouvé le même résultat pour le chemin passant par 5.

Remarque

Les durées max étant souvent exprimées en flottant avec une précision de 2 chiffres après la virgule dans les fichiers tests, il ne faut pas oublier de mettre 100 en troisième argument de la fonction pour se ramener à des entiers. Il faudra ensuite diviser le regret par 100. Je ne savais pas qu'il fallait faire cette manip et n'ait pas compris tout de suite pourquoi il y avait des regrets nuls.

Cependant, cela ne suffit pas à empêcher quelques erreurs d'approximation. En effet, de la même manière que

```
int coeff=100;
    cout << (int) (floor((float)3.78 *coeff));
```

Affiche 3.77
il y a des problèmes d'approximation.

On le remarque avec le fichier wPat1.rcp.
Lorsqu'on affiche le graphe, en mettant 100 comme troisième argument, on obtient 377 au lieu de 378 pour la durée dimax de l'activité 1.

(alors meme que
`cout << (int) (floor((float)3.78 *100));`
renvoie bien 3.78).

Ces erreurs d'approximations assez sournoises sont difficiles à comprendre et à éviter. Essayer de passe en double peut peut être arranger ces problèmes.

Fiche résultat

Le sujet ne dis pas explicitement quel algorithme évaluer. Nous considérons donc qu'il s'agit juste de trouver le chemin robuste et le regret associé dans un graphe.

Il s'agit donc d'évaluer les performances de la fonction `chRobusteMinRegret.`

	wPat1.rcp	wPat2.rcp	wPat3.rcp	wPat4.rcp	wPat5.rcp
min(s)	3,01119	4,90071	3,19376	3,58734	3,32528
max(s)	3,01131	5,00174	3,2827	3,66885	3,33313
moy(s)	3,01125	4,951225	3,23823	3,628095	3,329205
chemin	0 11 97 101	0 6 44 91 101	0 21 85 101	0 26 95 101	0 23 85 101
regret	0,42	0,55	0,42	0	0

	wPat6.rcp	wPat7.rcp	wPat8.rcp	wPat9.rcp	wPat10.rcp
min(s)	3,42069	3,4791	2,71715	3,16053	3,1967
max(s)	3,42994	3,48572	2,72155	3,1973	3,20517
moy(s)	3,425315	3,48241	2,71935	3,178915	3,200935
chemin	0 2 99 101	0 5 61 80 101	0 5 94 101	0 15 96 101	0 16 53 79 101
regret	0	0,86	0	0	0,36

Conclusion

Le calcul du chemin robuste de s à t passant par i souffre de deux interprétation possible.

Avec la possibilité 1 on réduit le graphe en ne gardant que les chemins passant par i et on calcule le chemin robuste et le regret à partir de ce nouveau graphe.

Avec la possibilité 2 on calcule tous les regrets à partir du graphe d'origine et on ne garde que les regrets et chemins associés à un chemin contenant i.

Le calcul du chemin robuste et du regret restent inchangé.

Bien que les exemples ne soient pas très gros (à peine 102 sommets pour chaque graphe), la recherche de chemin robuste et de regret associé est lente (de l'ordre de 3 secondes), de plus, nous avons vu plus haut qu'il y avait des erreurs d'approximation du à la récupération des données et aux multiples casts de types.

L'algorithme pourrait sûrement être mieux optimisé, il faudrait avoir des graphes plus gros pour mieux comparer les performances et peut être ne pas utiliser le temps CPU pour qualifier l'algorithme mais le nombre d'instruction et la place mémoire utilisé.

Contact

Si il y a des erreurs, des remarques, des ajouts à faire, etc.
Veuillez en faire part à une de ces adresses :

wedg@hotmail.fr (Veysseire Daniel)
mickaelfabre@free.fr (Fabre Michael)