TP1 - Arbre couvrant de poids minimum

Graphes et programmation dynamique

M1 Informatique - semestre d'automne 2022-2023

Université Côte d'Azur

Christophe Crespelle
christophe.crespelle@univ-cotedazur.fr

Le but de ce TP est d'implementer, en language C, les deux algorithmes vu en cours pour le probleme de l'arbre couvrant de poids minimum : Kruskal et Prim. Plusieurs fichiers sont fournis : le fichier contenant le programme principal `main.c` et trois bibliotheques `wgraph.h`, `wnodelist.h` et `utility.h`, avec les `.c` correspondant. Le fichier `main.c` contient une trame de l'architecture generale du programme, a completer, avec des sections predefinies en commentaire qui vous indiquent ou placer certaines des parties de code que vous allez developpez. La bibliotheque `wgraph.h` contient des fonctions de manipulation des graphes ponderes et `wnodelist.h` contient des fonctions de manipulation de listes de noeuds ponderes.
Vous coderez certaines des fonctions necessaires aux deux algorithmes dans d'autres bibliotheques que vous creerez et que vous inclurez de la meme maniere que les autres au debut du `main.c`.
Le programme se compile grace a la commande `make` tapee dans un repertoire contenant le fichier `Makefile` fourni ainsi que tous les fichiers du programme. Il s'execute en tapant `./mst.out`. Tapez `./mst.out -h` pour afficher l'aide. Pensez que lorsque vous inclurez les nouvelles bibliotheques que vous coderez, il vous faudra modifier le fichier `Makefile`.

1 Prise en main du programme

Question 1. Compilez le programme. Combien d'options a-t-il ? A quoi servent-elles ? Quelles sont les valeurs par defaut ?

Question 2. Quelles fonctions contient la bibliotheque `wgraph.h` ? A quoi sert la fonction `wgraph_from_file` ?
Dans quelle structure est stocke un graphe ? De quel type de graphe s'agit-il ?

Question 3. Quelles fonctions contient la bibliotheque `wnodelist.h` ? Quelle structure stocke une liste d'adjacence ? De quel type de liste s'agit-il ?

Pour l'instant, le programme se contente d'ouvrir les fichiers d'entree et de sortie specifies lors de l'appel au programme, respectivement en lecture seule et ecriture seule, et de les fermer a la fin du programme.

Question 4. En utilisant les fonctions de la bibilotheque `wgraph.h`, completez le `main` pour que le programme :

— charge un graphe pondere G en memoire a partir du fichier specifie en entree (ou de l'entree standard),
— ecrive sur la sortie d'erreur standard (`stderr`) le nombre de sommets et le nombre d'aretes du graphe charge,
— libere la place prise par le graphe en memoire avant de quitter.

Vous fermerez le fichier d'entree immediatement apres avoir charge le graphe qu'il contient, a l'aide de la fonction `fclose`, deja utilisee a la fin du main.

> **Indication.** Vous trouverez de nombreux exemples d'ecriture dans un fichier ou sur les sorties standard dans les fichiers qui vous sont fournis. L'annexe B du sujet contient egalement un petit rappel sur l'utilisation de la fonction `fprintf`.

Plusieurs fichiers qui contiennent chacun un graphe pondere, et dont le nom est de la forme `wgraphEL_*` ou `graphEL_*`, vous sont donnes dans le dossier `/data` accessible depuis la page web des TP (voir descriptif succint de leur provenance dans la section 4). Le format qu'ils utilisent pour encoder le graphe G est le suivant :

— la premiere ligne du fichier contient le nombre n de sommets dans le graphe,
— toutes les autres lignes du fichier contiennent une arete du graphe pondere sous la forme de deux entiers u v et d'un reel w separes par un espace, ou u et v sont les identifiants des deux sommets (distincts) qui sont les extremites de l'arete et w est le poids de l'arete uv,
— l'identifiant d'un sommet est un entier compris entre 0 et $n-1$.

Question 5. Essayez votre programme sur les graphes `toygraph` et `toygraph2`. La sortie de votre programme est-elle correcte ?

2 Algorithme de Kruskal

2.1 Trier les aretes par poids croissant

La premiere etape de l'algorithme de Kruskal consiste a trier les aretes du graphe par poids croissant. Pour cela, vous utiliserez la procedure `qsort` de la librairie standard `stdlib.h` de C. La procedure `qsort` prend 4 parametres :

1. l'adresse (de type `void *`) du tableau a trier
2. le nombre d'elements dans le tableau
3. la taille d'un element
4. le nom de la fonction de comparaison a utiliser entre deux elements

et elle trie le tableau passe en parametre, sur place. La fonction de comparaison prend deux parametres de type `void *`, en mode donnee, et retourne un entier qui est selon le resultat de la comparaison :

— strictement negatif, si le premier parametre est strictement inferieur au second,
— 0, si les deux parametres sont egaux,
— strictement positif, si le premier parametre est strictement superieur au second.

Exemple d'utilisation pour trier un tableau de 5 entiers :

```
int tab[] = { 88, 56, 100, 2, 25 };

int compare (const void * a, const void * b) {
   if (*(int*)a < *(int*)b) return -1;
   else if (*(int*)a > *(int*)b) return 1;
   else return 0;
}

int main () {
...
   qsort(tab, 5, sizeof(int), compare);
...
}
```

Question 6. Dans le fichier `main.c` definissez une structure `wedge` representant une arete ponderee.

Question 7. Definissez aussi une fonction de comparaison `comp_wedge` qui compare deux aretes ponderees en fonction de leur poids.

Question 8. Enfin, declarez et initialisez un tableau de `wedge` contenant toutes les aretes ponderees du graphe et triez le a l'aide de la fonction `qsort`.

2.2 Union-Find

Au cours de l'algorithme de Kruskal, lorsqu'une arete est ajoutee a l'arbre couvrant, il faut unifier les deux composantes connexes qu'elle relie. Pour cela, vous allez definir et implementer un module `unionfind.h`.
Cette implementation attribuera a chaque sommet l'identifiant du groupe auquel il appartient, qui sera toujours choisi comme l'identifiant d'un sommet du groupe. Initialement chaque sommet est dans un groupe different (dont l'identifiant est donc l'identifiant du sommet lui meme) et lors d'une union de deux composantes, les sommets de la plus petite des deux composantes se voient attribuer l'identifiant de la plus grande des deux. A la fin de l'algorithme, lorsqu'un arbre couvrant est obtenu, il ne reste qu'une composante et tous les sommets du graphe ont donc le meme identifiant de groupe.
En terme de structure de donnee, on representera les groupes a l'aide de trois structures :

- un tableau de taille n qui a chaque sommet associe l'identifiant de son groupe et
- un tableau de taille n qui a chaque identifiant de groupe associe la liste des sommets qu'il contient et
- un tableau de taille n qui a chaque identifiant de groupe associe le nombre de sommets dans le groupe.

Question 9. Dans le fichier `unionfind.h`, declarez une structure `partition` qui contient toute la structure de donnee necessaire pour faire de l'*Union-Find*. En plus des trois tableaux decrits ci-dessus, elle contiendra le nombre d'elements dans la structure et le nombre courant de parties dans la partition.

Question 10.

Declarez (dans le fichier `unionfind.h`) et implementez (dans le fichier `unionfind.c`) une procedure `init_partition` qui prend deux parametres, un pointeur vers la `partition` a initialiser et le nombre d'elements qu'elle contient, et qui l'initialise en mettant chaque element seul dans son groupe.

Indication. Avant de vous lancer, lisez (ou relisez, si besoin) l'annexe A sur la gestion de la memoire en C.

Question 11. Declarez et implementez la fonction `find` qui prend en parametre une `partition` et l'identifiant d'un de ses elements et qui retourne l'identifiant du groupe de la partition auquel cet element appartient.

Question 12. Declarez et implementez la procedure `make_union` qui prend en parametre un pointeur vers une `partition` et les identifiants de deux de ses groupes et qui actualise la `partition` en realisant l'union de ces deux groupes.

Question 13. Enfin, declarez et implementez la fonction `free_partition` qui prend en parametre un pointeur vers une `partition` et libere proprement l'espace qu'elle occupe.

2.3 Implementation de Kruskal

Pour rappel, l'algorithme de Kruskal consiste a considerer les aretes par ordre croissant de leur poids en ajoutant dans l'arbre toutes celles qui unissent deux composantes connexes de la foret F courante et en abandonant celles qui ont leur deux extremites dans le meme arbre de F.

Question 14. En utilisant ce qui precede, implementez l'algorithme de Kruskal dans la partie *COMPUTE MST* du `main`.

En fin de programme, on veut ecrire dans un fichier resultat l'arbre couvrant de poids minimum obtenu. On adoptera le format de sortie suivant : chaque ligne du fichier contiendra une arete de l'arbre avec son poids, donnee par deux entiers et un reel separes par un espace.

Question 15. Modifiez votre programme pour que l'algorithme stocke chaque arete placee dans l'arbre couvrant dans un tableau de $n-1$ `wedge`.

Question 16. Ecrivez a la fin du main, dans la section *OUTPUT RESULTS*, le code necessaire pour ecrire le resultat dans le fichier de sortie passe en parametre du programme par l'utilisateur.

Question 17. Testez la correction de votre algorithme sur les graphes `toygraph` et `toygraph2`. Essayez aussi sur d'autres graphes ponderes que vous creerez vous meme, par exemple en modifiant `toygraph` et `toygraph2`.

3 Algorithme de Prim

Un element clef de l'algorithme de Prim est la file de priorite qu'il utilise. Votre premiere tache sera donc d'implementer une telle file, avec l'implementation basique (et peu efficace) proposee ci-dessous.

3.1 Implementation du module file de priorite

Les identifiants des elements de la file de priorite sont les entiers compris entre 0 et $n-1$. Ainsi, vous pourrez representer la file par deux tableaux :

— un tableau de taille n qui a chaque element associe sa valeur (un reel) et
— un tableau de taille n qui a chaque element associe un booleen qui indique s'il est present dans la file ou non.

Question 18. Dans le fichier `priorite.h`, declarez une structure `pqueue` qui contient toute la structure de donnee necessaire pour implementer une file de priorite. En plus des deux tableaux decrits ci-dessus, elle contiendra deux entiers : un qui est la capacite de la file (le nombre maximum d'elements qu'elle peut contenir) et l'autre qui est le nombre d'elements effectivement presents dans la file.

Question 19.
Declarez (dans le fichier `priorite.h`) et implementez (dans le fichier `priorite.c`) une procedure `init_pqueue` qui prend deux parametres, un pointeur vers la `pqueue` (vierge) a initialiser et le nombre d'elements qu'elle contient, et qui l'initialise en mettant chaque element present dans la queue avec une valeur infinie.

> **Indication.** Posez vous au prealable la question de savoir quel choix faire pour representer une valeur reelle infinie ?

Question 20. Declarez et implementez la procedure `update_pqueue` qui prend en parametre un pointeur sur une `pqueue`, un element present dans la `pqueue` et un reel et actualise la valeur associee a cet element avec celle fournie en parametre.

Question 21. Declarez et implementez la fonction `isempty_pqueue` qui prend en parametre une `pqueue` et retourne vrai si, et seulement si, elle est vide.

Question 22. Declarez et implementez la fonction `extractmin_pqueue` qui prend en parametre un pointeur sur une `pqueue` et qui retourne un element ayant une valeur associee minimum parmi ceux presents dans la `pqueue`, puis supprime cet element de la `pqueue`.

> **Indication.** Aucune efficacite particuliere n'est recherchee ici, vous pourrez faire un parcours total des tableaux definissant la `pqueue`.

Question 23. Enfin, declarez et implementez la fonction `free_pqueue` qui prend en parametre un pointeur vers une `pqueue` et libere proprement l'espace qu'elle occupe.

3.2 Implementation de Prim

Dans l'algorithme de Prim, independemment de l'implementation de la file de priorite, on a besoin d'avoir acces en temps constant a certaines informations sur un sommet. A cette fin, on va utiliser, en plus de la file, trois tableaux :

— un tableau de taille n qui a chaque sommet associe un booleen qui dit si le sommet est dans la file (sommet *blanc*) ou non (sommet *noir*).
— un tableau de taille n qui a chaque sommet blanc associe son poids, defini comme le poids minimum d'une arete le reliant a un voisin noir

— un tableau de taille n qui a chaque sommet blanc associe son pere, un de ses voisins noirs qui lui est relie par une arete de poids minimum.

Question 24. Dans la partie *DATA STRUCTURE* du fichier `main.c` declarez et initialisez les trois tableaux decrits ci-dessus.

Question 25. En vous aidant du cours, implementez l'algorithme de Prim a l'aide de la structure de donnee mise en place ci-dessus.

Question 26. Modifiez votre code pour que, comme precedemment, chaque arete placee dans l'arbre couvrant soit stockee dans un tableau de `wedge`. Dans la section *OUTPUT RESULTS*, ecrivez l'arbre couvrant resultat de l'algorithme de Prim dans le fichier de sortie specifie par l'utilisateur.

Question 27. Testez la correction de votre algorithme sur les graphes `toygraph` et `toygraph2`. Essayez aussi sur d'autres graphes ponderes que vous creerez vous meme, par exemple en modifiant `toygraph` et `toygraph2`.

4 Application des algorithmes a des jeux de donnees

Plusieurs jeux de donnees synthetiques ou provenant de contextes reels sont donnes dans le dossier `/data` accessible depuis la page web des TP. L'annexe C en contient une breve description. A part `wgraphEL_toygraph` et `wgraphEL_toygraph2`, ces donnees sont des graphes non ponderes, c'est a dire que les lignes decrivant les aretes comportent seulement deux entiers et pas de poids.

Question 28. Pour chacun des graphes fournis, verifiez le nombre de sommets ecrit dans le fichier et comptez le nombre d'aretes presentes dans le fichier.

> **Indication.** Vous vous aiderez de la commande `head -1 fic` du systeme d'exploitation, qui affiche la premiere ligne du fichier `fic` et de la commande `wc -l fic` qui affiche le nombre de lignes dans le fichier `fic`.

Question 29. En utilisant la fonction `rand()` de tirage aleatoire du langage C, modifiez la procedure `wgraph_from_file` pour que lorsqu'une ligne contient exactement deux entiers et pas de reel, le poids de l'arete soit attribue de maniere aleatoire en choisissant un decimal entre 0.0 et 10.0 avec un chiffre apres la virgule.

Question 30. Pour chacun des graphes fournis, en les prenant par taille croissante, notez quel est le temps d'execution de l'algorithme de Kruskal et de l'algorithme de Prim. Pour quelle taille maximum de graphe peut-on calculer un arbre couvrant de poids minimum avec un des deux algorithmes en un temps inferieur a 2 minutes ? Y a-t-il une des deux implementations que vous avez faites qui est plus rapide que l'autre ?

> **Indication.** Utilisez la commande `time COMMANDE` du systeme d'exploitation qui donne le temps pris par l'execution de la commande `COMMANDE`.

Question 31. En tenant compte de la complexite de vos implementations et en extrapolant les resultats obtenus a la question precedente, determinez pour chacun des graphes fournis une estimation du temps de calcul necessaire a votre programme pour calculer un arbre couvrant de poids minimum avec chacun des deux algorithmes.

Question 32. Effectuez une nouvelle implementation de l'Union-Find et de la file de priorite en utilisant des librairies C existantes qui implementent ces structures de maniere plus performante.

Question 33. Comparez les performances des quatre implementations obtenues (les deux implementations basiques et les deux implementations plus performantes) en tracant la courbe de leur temps d'execution en fonction de la taille du graphe en entree. Commentez.

A Rappel sur la gestion de la memoire en C

En C, on doit gerer soi-meme la memoire. On alloue de l'espace memoire sur le tas grace a la fonction `malloc` qui s'utilise ainsi :

```
TYPE* p;
p = (TYPE*) malloc(sizeof(TYPE));
```

La premiere ligne declare un pointeur `p` sur un type `TYPE`, la deuxieme alloue sur le tas l'espace necessaire pour stocker une variable de type `TYPE`. L'argument de `malloc` est la taille en octet de la zone memoire allouee, le forcage de type `(TYPE*)` devant malloc est necessaire pour la concordance des types avec la variable `p` de type pointeur sur `TYPE`. Si on veut reserver un tableau de type `TYPE` et de taille `TAILLE`, on ecrit :

```
p = (TYPE*)malloc(TAILLE*sizeof(TYPE));
```

On accede alors a la case d'indice `i` du tableau `p`, pour `i` entre `0` et `TAILLE-1`, par l'expression `p[i]`.
Pour eviter les problemes lors du deboggage, on protegera toutes les allocations memoires en faisant quitter le programme avec un message d'erreur a chaque fois qu'une allocation est ratee. Pour cela on utilisera la fonction `report_error` fournie dans la ibliotheque `utility.h`, de la maniere suivante :

```
if ( (p = (TYPE*) malloc(TAILLE*sizeof(TYPE))) == NULL )
   report_error("malloc() error");
```

Vous avez de multiples exemples d'utilisation de `malloc` dans les fichiers qui vous sont fournis. Pour desallouer la zone memoire pointee par le pointeur `p` on utilise la fonction `free` :

```
free(p);
p=NULL;
```

Il n'y a pas de garantie sur ce que vaut `p` apres la desallocation. C'est pour cela que par securite, il est conseille de mettre `p` a `NULL` apres desallocation pour eviter d'avoir des pointeurs vers des zones desallouees. Il n'y a pas non plus de garantie sur ce que contient la zone desallouee apres desallocation : elle peut etre reinitialisee ou laissee intacte (le plus courant) et dans tous les cas elle peut bien sur etre reservee plus tard dans le programme a un autre usage.
Les fuites memoires sont un vrai probleme, elles vous empecheront de traiter de grandes instances en entree de votre programme (ce qui est votre but). Il faut donc leur faire la chasse. Pour les eviter, vous desallouerez systematiquement toute memoire reservee, dans le programme principal (`main`) ou dans un sous-programme (fonction ou procedure), des lors que vous savez qu'elle ne servira plus (et dans tous les cas, au plus tard a la fin du `main`). Vous pouvez verifier que votre programme n'a pas de fuite memoire en scrutant la place qu'il utilise en memoire lors de son execution a l'aide de la fonction `top` du systeme d'exploitaion. Par exemple,

```
top -d 1 -u username -o %MEM
```

vous affiche les statistiques des programmes que vous (avec le nom d'utilisateur `username`) avez lance triees par ordre decroissant d'utilisation de la memoire (`-o %MEM`) et actualisees toutes les secondes (`-d 1`). Plus d'info avec `man top`.

B Rappel sur la fonction `fprintf`

La fonction `fprintf` permet d'ecrire des donnees selon differents formats sur un flux de sortie. Un exemple d'utilisation simple est

```
fprintf(FLUX,"Texte quelconque");
```

qui ecrit la chaine de caractere `"Texte quelconque"` sur le flux de sortie `FLUX`, qui peut etre un fichier (type `FILE*`) ouvert en ecriture ou un flux de sortie standard du systeme d'exploitation, comme `stdout` (sortie standard) ou `stderr` (sortie d'erreur standard). Le flux est le premier parametre de `fprintf` et la chaine a ecrire le deuxieme parametre. Il peut y avoir plus de parametres dans l'appel a `fprintf`, places apres le flux et la chaine, qui servent a incorporer d'autres donnees dans la chaine de caractere en specifiant le format auquel elles doivent etre ecrites sur le flux a l'aide du caractere `%`. Un exemple de syntaxe est le suivant :

```
fprintf(stderr,"Il y a %d sommets dans le graphe et %d aretes",n,15);
```

Si `n` est une variable de type `int` qui vaut 8, cela ecrit sur la sortie d'erreur standard la chaine de caracteres

```
Il y a 8 sommets dans le graphe et 15 aretes
```

Le format d'ecriture des entiers `n` et `15` est specifie par le `%d` qui est le format d'ecriture decimale des `int`. Les parametres de `fprintf` qui suivent la chaine de caractere, ici `n` et `15`, sont mis en correspondance avec les indications de format contenues dans la chaine en utilisant l'ordre dans lequel apparaissent les parametres supplementaires passes a la fonction `fprintf` et l'ordre dans lequel apparaissent les indications de format dans la chaine de caractere passee en parametre a `fprintf`. Le nombre de parametres supplementaires doit donc etre identique au nombre d'indications de format contenues dans la chaine et il doit y avoir concordance entre les indications de format et les types des parametres supplementaires. Par exemple, `%u` est le format d'ecriture decimale des `unsigned int`, `%lu` celui des `long unsigned int` et `%x` est le format d'ecriture hexadecimale des `unsigned int`. On aurait par exemple pu appeler `fprintf` ainsi

```
fprintf(stderr,"Il y a %d sommets dans le graphe et %x aretes",n,15);
```

ce qui ecrit la chaine de caracteres

```
Il y a 8 sommets dans le graphe et f aretes
```

Les retours a la ligne dans la chaine de caractere sont codes par `\n`. Exemple :

```
fprintf(stderr,"Il y a %d sommets dans le graphe\net %d aretes",8,15);
```

produit l'ecriture

```
Il y a 8 sommets dans le graphe
et 15 aretes
```

Pour optimiser les performances du systeme, l'ecriture ne se fait pas directement sur le flux specifie en parametre de `fprintf` mais passe au prealable par un cache dont le contenu n'est effectivement ecrit sur le flux que lorsque le cache est suffisemment rempli. On peut forcer l'ecriture sur le flux et le vidage du cache a l'aide de la fonction `fflush` :

```
fflush(FLUX);
```

Afin de faciliter la phase de deboggage, il est conseille de proceder au vidage force du cache regulierement, apres chaque etape importante d'ecriture. En effet, lorsque le programme quitte subrepticement, a cause d'un bug par exemple, le cache des flux ouverts en ecriture n'est pas vide. Ainsi, tout ce qui avait ete ecrit sur le flux mais qui etait encore dans le cache est perdu et n'apparait pas sur le flux. Si l'on se fie a l'affichage sur le flux, cela trompe sur le moment de l'execution auquel le bug s'est produit.

C Jeux de donnees

Dans le dossier `/data` accessible depuis la page web des TP, vous trouverez plusieurs graphes, ponderes ou non, au format decrit au debut du sujet. Certains de ces graphes sont synthetiques, c'est a dire qu'ils ont ete generes soit a la main soit par des methodes de generation automatiques, et d'autres proviennent d'operations de mesure realisees dans des contextes concrets : Internet, publications scientifiques, biologie, environnement, reseaux pair-a-pair, reseaux routiers et reseaux sociaux en ligne. Une breve description de ces graphes est donnee ci-dessous.

`wgraphEL_toygraph`, 7 sommets, graphe pondere jouet construit a la main,

`wgraphEL_toygraph2`, 8 sommets, graphe pondere jouet construit a la main,

`graphEL_rand_100_8`, 100 sommets, graphe aleatoire (Erdos-Renyi G_{nm}) de degre moyen 8,

`graphEL_rand_500_16`, 500 sommets, graphe aleatoire (Erdos-Renyi G_{nm}) de degre moyen 16,

`graphEL_rand_1000_16`, 1000 sommets, graphe aleatoire (Erdos-Renyi G_{nm}) de degre moyen 16,

`graphEL_as2000`, 6474 sommets, graphe de connections entre les systemes autonomes d'Internet en 2000,

`graphEL_ca-GrQc`, 4158 sommets, graphe de coauteurs dans le domaine de la relativite generale et de la cosmologie quantique,

`graphEL_figeys`, 2217 sommets, graphe d'interactions chimiques entre proteines,

`graphEL_foodweb`, 183 sommets, graphe de predation entre especes (chaine alimentaire),

`graphEL_p2p-Gnutella`, 62561 sommets, graphe de connections entre pairs dans le reseau P2P Gnutella,

`graphEL_roadNet-TX`, 1351137 sommets, graphe du reseau routier du Texas,

`graphEL_youtube`, 1134890 sommets, une partie du reseau social de youtube.