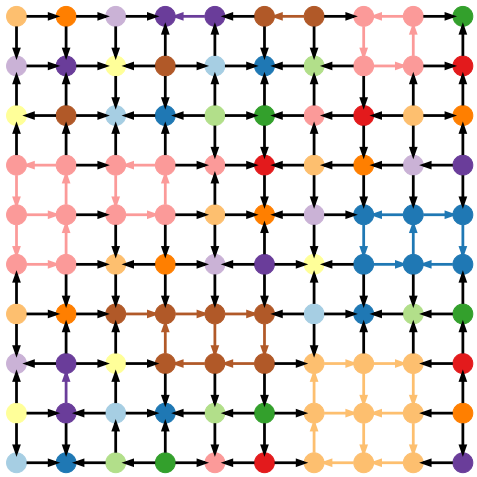
TP 4 : Algorithmes des graphes



Le but de ce TP est de mettre en œuvre l'algorithme de détection des composantes fortement connexes. L'algorithme sera testé sur des graphes générés aléatoirement, composés d'une grille de points reliés à leurs voisins directs, comme ci-contre.
Ce TP est l'occasion de travailler les points suivants en C:

- Représentation de graphe sous forme de matrice d'adjacence.
- Gestion de la mémoire dynamique et des pointeurs
- Parcours en profondeur de graphe, avec temps de passage
- Calcul de transposé d'un graphe
- Tri rapide
- Calcul de composantes fortement connexes d'un graphe orienté
- Écriture dans un fichier

Environnement

Ce TP sera écrit en C. On rappelle qu'un fichier `source.c` se compile en un exécutable `exec` avec la commande `gcc -o exec source.c`.

Remarque Les fonctions devront être protégées contre une mauvaise utilisation: accès hors des bornes, taille invalide.

I Mise en place - Création et affichage de graphe orienté pondéré

Pour faciliter l'opération de transposition d'un graphe, on choisit ici une implémentation de graphe sous forme de matrice d'adjacence.

1. Créer un type `struct node` qui représentera les sommets du graphe. Il devra comporter les champs `debut`, `fin`, `parent` et `status` qui seront utilisés par le parcours en profondeur avec temps de passage.
2. Définir des constantes (distinctes) `VU`, `EN_EXPOLRATION` et `NON_VU` pour les `status`.
3. Définir un type `struct digraph` (*directed graph*) pour représenter les graphes orientés pondérés. On stockera le nombre de sommets `S`, la tableau des sommets de type `node` et la matrice d'adjacence indiquant l'existence ou non d'un arc. La matrice d'adjacence sera stockée sous forme d'un tableau de booléens à une dimension.
4. Écrire une fonction `void set_edge(struct digraph *graph, unsigned int s, unsigned int t, bool is)` (resp. `bool has_edge(struct digraph *graph, unsigned int s, unsigned int t)`) qui définit (resp. retourne) l'existence ou non de l'arc entre les sommets s et t.
5. Écrire une fonction `void init_graph(struct digraph *graph, unsigned int size)` qui prend en argument un graphe non-initialisé et la taille de la grille. Le graphe comportera donc `size` $\times$ `size` sommets. La fonction doit compléter les champs nécessaires dans la structure `digraph`, en particulier elle est responsable de l'allocation de la mémoire. Elle crée aléatoirement des arcs entre ses sommets: pour chaque sommet, pour son voisin du dessus et celui de droite, on choisi aléatoirement le fait de créer un arc du sommet vers le voisin ou l'inverse. (Voir les Annexes pour la gestion de l'aléatoire.)
6. Écrire une fonction `void release_graph(struct digraph *graph)` qui libère la mémoire allouée dynamiquement au graphe par la fonction précédente.

Graphviz permet d'obtenir une représentation graphique d'un graphe à partir d'une description textuelle de celui-ci. Voilà à quoi ressemble le début de la définition du graphe orienté (`digraph`) de l'introduction. Le sommet en position $(0, 0)$ est celui en bas à gauche.

```
digraph G {
  node [shape = point
        colorscheme=paired12
        width=0.4]
  edge [colorscheme=paired12 penwidth=4]
  0[pos="0,0!" color="1"];
  edge[color=0]
  0 -> 1;
  1[pos="1,0!" color="2"];
  2[pos="2,0!" color="3"];
  edge[color=0]
  2 -> 1;
  edge[color=0]
  2 -> 12;
  ...
}
```

Analysons ligne à ligne:

1. Spécifie le type de graphe: un `digraph`.
2. Définit le style des nœuds: ils seront représentés sous forme de points d'épaisseur 0.4 en utilisant la palette de couleur `paired12`.
3. Définit le style des arcs qui seront sur la même palette de couleurs mais avec un trait d'épaisseur 4.
4. Déclare un sommet 0, à la position $(0, 0)$ et utilisant la première couleur de la palette (un bleu clair).
5. Change la couleur des prochains arcs déclarés pour la couleur 0 (en réalité elle n'est pas dans la palette et sera donc en noir).
6. Déclare un arc du sommet 0 au sommet 1 (Graphviz découvre alors qu'il existe un sommet 1 mais on peut le redéfinir après).
7. Déclare le sommet 1, à la position $(1, 0)$ avec la deuxième couleur de la palette (un bleu foncé)

Graphviz a plusieurs moteurs pour organiser les sommets du graphe, on utilisera ici le moteur neato qui permet de placer les sommets manuellement.

7. Écrire une fonction `void to_neato(const struct digraph graph, const char* filename)`[1] qui écrit dans le fichier `filename` le graphe au format graphviz. Pour l'instant on ne spécifie pas la couleur des nœuds ni des arcs.
8. Dans le main, tester votre fonction sur un petit graphe (taille 5 par exemple).
9. Dans le main, récupérer l'entrée de l'exécutable tel que le premier argument est la taille du graphe et le deuxième le nom du fichier ou enregistrer le graphe. Si un des deux arguments manque, le programme affiche un message d'erreur et termine.

```
> ./graphe
Usage ./graphe TAILLE FICHIER.dot
> ./graphe -1 graphe.dot
La taille doit être positive.
> ./graphe 10 graphe.dot
Création d'un graphe de taille 10.
Enregistrement du graphe dans graphe.dot.
```

10. Éxecuter le programme et utiliser ensuite la commande `neato -Tpdf graphe.dot > graphe.pdf` pour obtenir une représentation graphique dans un pdf. (Ou copier le fichier .dot dans `https://graphviz.christine.website` et sélectionner le moteur neato pour le rendu).

II Fonctions auxiliaires

Le calcul de composantes fortement connexes a besoin de trois opérations: la parcours en profondeur avec temps de passage, le calcul de transposé de graph et enfin un tri de nœuds selon les temps de fin[2].

II.1 Parcours en profondeur

On implémente un parcours en profondeur qui modifie les champs `debut`, `fin` et `parent` des nœuds du graphe lors de son passage. La variable `temps` peut être globale ou être passée en argument à la fonction de visite. Le parcours en profondeur a un ordre comme argument afin de pouvoir réaliser le deuxième parcours en profondeur de l'algorithme de recherche des composantes fortement connexes.

[1] Le mot clé `const` indique que l'argument n'est pas modifié par la fonction.

[2] Le tri n'est pas nécessaire ici, on pourrait simplement stocker les noeuds dans une pile lorsqu'on termine les visiter et dépiler pour le deuxième parcours ; mais les besoin de retravailler les tris, nous allons utiliser un tri

11. Implémenter une fonction `void depth_search(struct digraph *graph, unsigned int *order)` qui implémente le parcours en profondeur vu en cours. Dans la boucle principale les sommets seront visités dans l'ordre donné par le tableau `order` vu comme une application $[\![0; |S| - 1]\!] \to S$, à la i-ème itération on visite le sommet `order[i]`.

Pour vérifier le résultat, vous pouvez enlever le style `shape = point` des nœuds dans la sortie Graphviz. À la place précédez chaque déclaration d'un nœud par `node[label="d/f"]` en remplaçant `d` et `f` respectivement par le temps de début et de fin de visite du nœud.

12. Quelle est la complexité de cette recherche en profondeur, comparé à l'algorithme vu en cours ?

II.2 Transposition

13. Écrire une fonction `struct digraph *transpose(struct digraph *graph)` qui renvoie le transposé du graphe d'entrée. La fonction est donc responsable de l'allocation mémoire ainsi que de l'initialisation des différents champs.

II.3 Tri rapide

Grâce au premier parcours en profondeur, chaque sommet du graphe a un temps de fin de parcours. On souhaite trier les sommets dans l'ordre *décroissant* selon ce temps de fin. Le tri le plus efficace et le plus simple à implémenter en C est le tri rapide. Il a en effet l'avantage d'être en place (contrairement au tri fusion) ce qui évite d'allouer dynamiquement de la mémoire.

14. Écrire une fonction de tri rapide `void sort(struct digraph *graph, unsigned int *order, int from, int to)` qui trie le tableau de sommets `order` de l'indice `from` à `to`. Ce tri doit être décroissant selon les temps de fin des sommets dans le graphe `graph`.

$$\forall i, j \in [\![\texttt{to}, \texttt{from}]\!], \quad i < j \implies \texttt{graph->nodes[i].fin} > \texttt{graph->nodes[j].fin}$$

III Calcul de composante fortement connexes

15. Écrire une fonction `void cfc(struct digraph *graph)` qui calcule les composantes fortements connexes du graph d'entrée. La fonction doit:
    - Initialiser un tableau `ordre` pour le premier parcours (par exemple correspondant à la fonction identité $i \mapsto i$).
    - Effectuer le parcours en profondeur.
    - Trier `ordre` dans l'ordre décroissant des temps de fin du parcours précédent.
    - Calculer le transposé du graphe.
    - Effectuer le parcours en profondeur du transposé selon l'ordre obtenu.
    - Appliquer les parents calculés sur le transposé au graphe originel.

    La fonction ne renvoie rien, il faut consulter les attributs `parent` des sommets du graphe pour connaître les composantes fortements connexes.
16. Pour obtenir un représentant unique de la composante, écrire une fonction `unsigned int get_root(struct digraph *graph, unsigned int i)` qui remonte l'arbre des parents depuis le sommet `i` jusqu'à arriver à la racine qui est retournée.
17. Modifier `to_neato` pour que les sommets soient coloriés. On utilisera l'indice de la racine (le représentant de la composante) comme couleur (modulo 12 pour rester dans la palette).
18. Modifier `to_neato` pour colorier les arêtes: soit c la couleur du sommet i et soit (i, j) une arête du graphe. L'arête est coloriée avec la couleur c ssi i et j appartiennent à la même composante (donc on a la même racine).

Annexes

Manipulation de fichier

- Ouverture d'un ficheir en écriture: `FILE* fichier = fopen("file.dot", "w")`. Si le fichier n'a pas pu être ouvert, `fichier` vaut `NULL`, il faut alors arrêter le programme en affichant l'erreur.
- Écriture dans un fichier: `fprintf(fichier, "...", ...)`. Le premier argument est un pointeur `FILE *`, le deuxième une chaîne de formattage. Les arguments suivants dépendent de la chaîne de formattage, comme pour un `printf`.
- Fermeture d'un fichier: `fclose(fichier)`.

Arguments du programme

On rappelle que les arguments de la fonction `int main(int argc, char* argv[])` correspondent aux nombres `argc` d'arguments donnés au programme et que `argv[i]` est la chaine de charactères donnée en i-ème argument: l'argument 0 est en réalité le nom du programme.

La fonction `int atoi(const char *str)` permet de convertir une chaîne de caractères en entier. Elle renvoie 0 si la chaîne de caractères ne représente pas un entier.

Aléatoire

La fonction `rand`, qui permet de tirer des nombres aléatoires, appartient à `<stdlib.h>`. On incluera aussi l'en-tête `<time.h>` pour avoir accès à la fonction `time` et ainsi initialiser la graine du tirage aléatoire.

```
srand(time(NULL)); // Initialise l'aléatoire, à n'appeler qu'une fois dans le programme
int var = rand(); // Renvoie une valeur aléatoire entre 0 et RAND_MAX
```