Université Paris 8 - Vincennes à Saint-Denis

Licence informatique & vidéoludisme

Projet Graph 8 : Mat3 et Brin

Vincent Bernier

Table des matières

Introduction

Le projet présentait à pour objectif de remplir un grand graphes (plus de 10000 noeuds) aléatoirement, puis de mettre en place les algorithmes de recherche du plus court chemin et d'identification des composantes connexes.

Les structures utilisés pour faire nos graphes sont :
— matrice compacte par triplets Mat3
— tableau de brins Brin

Les Algorithmes utilisés sont :
— Algorithme dijkstra de recherche du plus courts chemins
— Algorithme Union-Find pour composantes connexes

Dans une première partie, nous analyserons la réalisation des graphes effectues avec la matrice compacte par triplets et ses résultats en temps et en mémoire utilisés. Dans un second temps, nous présenterons les graphes construits à l'aide d'un tableau de brins, en détaillant également leurs performances. Dans une troisième partie, nous expliquerons comment sont conçu nos algorithmes de recherche. Pour finir, nous discuterons des résultats obtenue par les 2 structures en temps et en mémoire utilisé pour les comparais.

Graphe Mat3

Le premier programme implémente un graphe réalisé avec une structure de matrice compacte par triplets, que nous appellerons Mat3.

Nous verrons dans ce chapitre la réalisation du graphe, puis nous observerons ses résultats.

1.1 Les structures du graphe

Le graphe est créé avec deux structures, "gramaco" pour la structure général du graphe et "triple_t" pour la structure des arêtes.

```
typedef struct triplet {
    int i;
    int j;
    float poids;
} triple_t;

typedef struct gramaco {
    int nbs;
    int nba;
    triple_t *ares;
} gramaco;
```

Code source 1.1 – Structure

Le graphe est construit par une boucle qui parcourt tous les duo de sommets possibles, est y ajoute aléatoirement selon le "taux" definie, si les deux sommets sont relier par une arête.

```
for (i = 0; i < nbs; i++){
  for (j = 0; j < nbs; j++){
      v = (float)rand() / RAND_MAX;
      if (v < taux){
        if (nba >= max){
          max *= 2;
          g.ares = (triple_t *)realloc(g.ares, max * sizeof(triple_t));
        }
        g.ares[nba].i = i;
        g.ares[nba].j = j;
        g.ares[nba].poids = 1;
        nba++;
```

Code source 1.2 – CreeGraphe mat3

Ici si la conditions "if (v < taux)" est remplie, l'arête est ajoutée entre le sommet i et j. Le poids des arêtes est fixé a 1. La boucle contient aussi une réallocation de mémoire si la mémoire déjà allouée ne suffit plus, cela permet au programme d'allouer de la mémoire au fur est a mesure de ses besoins. En effet, la mémoire est allouée de cette façon :"g.ares = (triple_t *)malloc(max * sizeof(triple_t));" avec max qui est égale au nombre de sommets du graphe. Donc la valeur de base de mémoire allouer ne suffira pas dans de nombre cas. Avec notre taux actuel de 25 et taille de notre graphe de 10000 sommets, nous pouvons a l'avance prévoir que notre graphe contiendra un grand nombre d'arêtes.

1.2 Test et résultat

Les tests sont effectués sur un échantillon de boucle de 10 tests de création de graphe, de l'algorithme de dijkstra de recherche de chemin le plus court et de la recherche de composantes connexes. Nous verrons donc la rapidité moyenne du programme et sa consommation moyenne de mémoire utilisée.

Densité de 25

```
10 tours de 10000 sommets pour mat3
Mémoire utilisé par graphe mat3 : 3 456 080 octet
Mémoire moyenne utilisé : 3 656 160 octet
Temps moyen : 7.31s par graphe

10 tours de 20000 sommets pour mat3
Mémoire utilisé par graphe mat3 : 6 912 080 octet
Mémoire moyenne utilisé : 7 312 160 octet
Temps moyen : 22.07s par graphe
```

Densité de 50

```
10 tours de 10000 sommets pour mat3
Mémoire utilisé par graphe mat3 : 6 912 080 octet
Mémoire moyenne utilisé : 7 112 160 octet
Temps moyen : 15.62s par graphe

10 tours de 20000 sommets pour mat3
Mémoire utilisé par graphe mat3 : 13 824 080 octet
Mémoire moyenne utilisé : 14 224 160 octet
Temps moyen : 38.95s par graphe
```

Graphe par brin

La second programme implémente un graphes réalisé avec une structure de brin. Chaque brin contient dans sa structure un sommet cible et l'indice du brin cible suivant.

Nous verrons dans ce chapitre la réalisation du graphe, puis nous observerons ses résultats.

2.3 Les structure du graphe

Le graphe par brin est créé avec 2 structures, "strandgraph" qui structure le graphe et "strand" pour la structure des brin.

```
struct strand {
  Shu node;
  int next;
};

typedef struct strand strand;

struct strandgraph {
  Shu nbs;
  int nbstr;
  int *node;
  strand *nxt;
};
```

Code source 2.1 – Strcture brin

La fonction "creegraphe_brin" génère le graphe aléatoire à brins en connectant des sommets selon notre taux. Elle est composé d'une double boucle qui parcourt tous les couples possibles de sommets, est y généré des brins contenant les informations nécessaires pour parcourir les voisins des sommets. Ici, le brin stocke le sommet cible et l'indice du brin suivant.

```
for (i = 0; i < nbs; i++) {
  for (j = 0; j < nbs; j++) {
    v = (float)rand() / RAND_MAX;
    if (v < taux) {
      g.nxt[g.nbstr].node = j;
      g.nxt[g.nbstr].next = g.node[i];
      g.node[i] = g.nbstr++;

      g.nxt[g.nbstr].node = i;
      g.nxt[g.nbstr].next = g.node[j];
      g.node[j] = g.nbstr++;
    }
```

Code source 2.2 – CreeGraphe brin

Nous avons pu voir que notre code que les brins stocke les pointeurs(next) dans leurs structures. Ici, contrairement a mat3 la mémoire est allouée a l'avance. Elle est déterminée par la formule "g.nxt = (strand *)malloc(max_arc * sizeof(strand));" ou max_arc correspond au nombre de sommets multiplié par lui même. Cela nous allouera largement assez de mémoire pour la création du graphe.

2.4 Test et résultat

Les tests sont a nouveau fait sur un échantillon d'une boucle de 10 tests de création de graphe, de l'algorithme de dijkstra de recherche de chemin le plus court et de la recherche de composantes connexes. Nous verrons donc la rapidité moyenne du programme et sa consommation moyenne de mémoire utilisée.

Densité de 25

```
10 tours de 10000 sommets pour brin
Mémoire utilisé par graphe brin : 40 081 octet
Mémoire moyenne utilisé : 240 161 octet
Temps moyen : 1.898s par graphe

10 tours de 20000 sommets pour brin
Mémoire utilisé par graphe brin : 80 081 octet
Mémoire moyenne utilisé : 480 161 octet
Temps moyen : 7.188s par graphe
```

Densité de 50

```
10 tours de 10000 sommets pour brin
Mémoire utilisé par graphe brin : 40 081 octet
Mémoire moyenne utilisé : 240 161 octet
Temps moyen : 2.087s par graphe

10 tours de 20000 sommets pour brin
Mémoire utilisé par graphe brin : 80 081 octet
Mémoire moyenne utilisé : 480 161 octet
Temps moyen : 7.824s par graphe
```

Les algorithmes de recherche

Dans cette partie, nous verrons les implémentations des algorithmes de recherche dijkstra pour le chemin le plus court et de recherche de composantes connexes de nos graphes mat3 et brin. Les principes seront les mêmes et seront légèrement adaptés pour le graphe concerné.

3.5 Algorithme de recherche de chemin le plus court, dijkstra

L'algorithme de Dijkstra permet de calculer les plus courts chemins depuis un sommet source dans un graphe. Il fonctionne en explorant progressivement tous les sommets accessibles à partir du sommet source.

À chaque itération, l'algorithme sélectionne un sommet non visité ayant la plus petite distance actuelle par rapport au sommet source.

```
for (i = 0; i < nbs; i++){
  if (!vu[i] && (u == -1 || distance[i] < distance[u])){
    u = i;
  }
 }
```

Code source 3.1 – Sélection du sommet non visité

Ensuite, il examine toutes les arêtes sortantes de ce sommet pour mettre à jour les distances des sommets adjacents.Si un chemin plus court est découvert, les tableaux contenant les distances et les prédécesseurs sont mis à jour avec lui.

```
//Pour Mat3

for (x = 0; x < g.nba; x++) {
  if (g.ares[x].i == u) {
    v = g.ares[x].j;
    poid = g.ares[x].poids;
    if (!vu[v] && distance[u] + poid < distance[v]) {
      distance[v] = distance[u] + poid;
      predecesseur[v] = u;


//Pour Brin

for (arc = g.node[u]; arc != -1; arc = g.nxt[arc].next){
  v = g.nxt[arc].node;
  poid = 1.0;
  if (!vu[v] && distance[u] + poid < distance[v]){
    distance[v] = distance[u] + poid;
    predecesseur[v] = u;
```

Code source 3.2 – Mise à jour des distances

Une fois qu'un sommet est traité, il est marqué comme visité, et l'algorithme passe au sommet suivant.

3.6 Algorithme de recherche de composante connexe

La fonction composantes_connexes détermine le nombre de composantes connexes dans un graphe, c'est a dire chaque sommet accessible depuis tout autre sommet du sous-ensemble et donc inaccessible par tous les sommets d'une autre sous ensemble. Nous utilisons une structure d'union-find pour comptabiliser les sous-ensembles.

Notre algorithme parcourt toutes les arêtes du graphe pour associer les sommets qu'elles relient. Cette association est réalisée par une fusion des composantes des sommets via notre structure union-find.

```
  racine_x = find(parent, x);
  racine_y = find(parent, y);

  if (racine_x != racine_y){
    parent[racine_y] = racine_x;
  }
}
```

Code source 3.1 – Union

Ici lorsqu'une arête connecte deux sommets, les sommets sont fusionnées, ce qui signifie qu'ils appartiennent maintenant à la même composante. Enfin,

notre fonction identifie et compte les composantes unique en recherchant les racines des sommets. le nombre total de composantes connexes est obtenu en comptant le nombre de racines unique obtenu.

Résultats et comparaisons

Au cours de ce rapport, nous avons pu examiner les différents résultats obtenus avec chacun de nos programmes, qui implémentent les graphes de manière différente. Nous allons maintenant comparer ces résultats afin de déterminer quelles structures sont les plus performantes, que ce soit en termes de temps d'exécution ou de mémoire utilisée.

Avec la structure mat3, la mémoire utilisée augmente considérablement avec le nombre de sommets et la densité du graphe, ce qui la rend très peu efficace en comparaison avec la structure en Brin. Cela s'explique par le fait que mat3 utilise une matrice dense pour représenter les arêtes, où chaque couple de sommets occupe un espace mémoire. Concernant le temps d'exécution, on observe également un retard important par rapport à la structure en brin, avec un temps moyen largement supérieur. La structure mat3 montre rapidement ses limites lorsque les valeurs augmentent pour nos grands graphes. L'augmentation du nombre de sommets et de la densité entraîne une forte croissance de la mémoire utilisée et du temps de calcul, de manière presque linéaire par rapport à ces paramètres.

Avec la structure en brin, la mémoire utilisée pour le graphe reste bien inférieure. Cette structure est beaucoup moins gourmande, car elle ne stocke que les arêtes présentes, sous forme de listes, au lieu de représenter toutes les connexions possibles dans une matrice. Même pour des graphes denses, l'utilisation de la mémoire reste très raisonnable en comparaison avec Mat3. De plus, on observe une meilleure efficacité de la structure en Brin en termes de temps d'exécution. La différence avec la structure Mat3 est significative. Prenons, par exemple, nos résultats :

```
10 tours de 20000 sommets pour brin, densité 25
Temps moyen : 7.188s par graphe
10 tours de 20000 sommets pour mat3, densité 25
Temps moyen : 22.07s par graphe
```

Nos test montre un temps d'exécution au minimum 3 fois plus long et pouvant aller jusqu'a 5 fois plus long avec une densité de 50.//

Les résultats montrent clairement que brin est meilleur avec notre graphes de grande taille, grâce à sa faible consommation de mémoire et à ses temps d'exécution réduits. En revanche, mat3, bien que simple à faire et manipuler, devient rapidement inefficace pour les graphes denses ou volumineux. À son crédit, de la structure mat3, on peut noté que cette structure reste largement suffisante pour de graphes plus petits. L'on peut trouver des cas de graphe de petite taille avec mat3 montrer de meilleur résultats en mémoire utiliser total.