ENSICAEN – 1 ère année Électronique et physique appliquée

Année 2019-2020 – Semestre 5

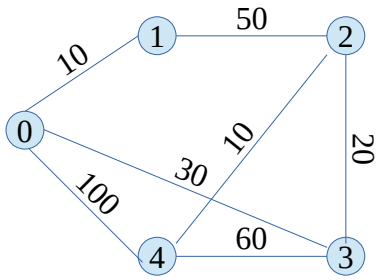
Examen Informatique et langage C

janvier 2020 – 1h30 – P Lefebvre

Tous documents papier autorisés - CORRECTION

Répondre sur ce document

Un graphe est un ensemble de nœuds reliés les uns aux autres par des arcs. Les arcs peuvent être valués pour représenter un coût ou une distance entre chaque nœud. Un réseau routier peut être représenté par un graphe. Ci-dessous un graphe de 5 nœuds. La distance entre le nœud 0 et le nœud 1 est de 10.

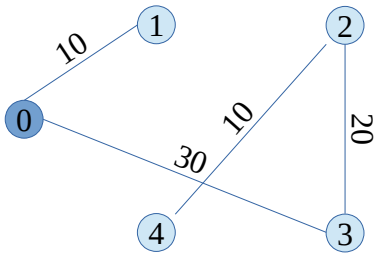


0	10	0	30	100
10	0	50	0	0
0	50	0	20	10
30	0	20	0	60
100	0	10	60	0

matrice d'adjacence

La matrice d'adjacence représente le graphe. En ligne 0, colonne 1 on trouve la valeur 10 qui signifie que la distance entre le nœud 0 et le nœud 1 est de 10. Cette matrice est ici symétrique car c'est la même distance pour aller du nœud 0 au nœud 1 que pour aller du nœud 1 au nœud 0.

Dans certaines applications on cherche la distance minimale (ou coût minimal) entre un nœud de départ et le reste des nœuds. Par exemple la distance minimale pour atteindre le nœud 4 à partir du nœud 0 est 60. En effet le chemin le plus court pour atteindre le nœud 4 est de passer par les nœuds 3 puis 2. Le coût est alors 30 + 20 + 10 = 60. Si on trace le graphe avec seulement les chemins les plus courts on obtient un arbre dont la représentation peut être la suivante :



Cet arbre est appelé l'arbre couvrant à coût minimal.

L'algorithme de Dijkstra permet de trouver un tel arbre à partir de la matrice d'adjacence. Cet algorithme a besoin de tableaux intermédiaires :

– `cost[i][j]` stocke le coût entre 2 nœuds i et j. C'est presque l'image de la matrice d'adjacence sauf que s'il n'y a pas de chemin direct entre 2 nœuds, `cost[i][j]` vaut l'infini.

– `distance[i]` stocke la distance minimale totale (somme des coûts) entre un nœud i et le nœud de départ. Ce tableau est initialisé avec le coût entre le nœud de départ et le nœud i (*cf. exemple ci-après*).

– `pred[i]` stocke le meilleur prédécesseur de chaque nœud i. Ce tableau est initialisé avec l'indice du nœud de départ.

– `visited[i]` qui stocke 1 ou 0 selon que le nœud i a été visité ou non. Ce tableau est initialisé à 0 sauf pour le nœud de départ qui est initialisé à 1.

L'idée de cet algorithme est de partir du nœud de départ, puis de trouver le nœud le plus proche. Dans l'exemple c'est le nœud 1 qui est le plus proche. Ce nœud est alors marqué « visité ». Ensuite, en partant de ce nœud on trouve tous les nœuds atteignables (ici, seulement le nœud 2). On met à jour le tableau distance, si la distance cumulée est plus courte : c'est le cas pour le nœud 2 ; la distance est mise à jour à 60 puisque 2 est atteignable par 1.

i	0	1	2	3	4
distance [i] initialement	0	10	∞	30	100
distance [i], après un tour de boucle	0	10	60	30	100

Puis l'algorithme recommence en trouvant le nœud « non visité » le plus proche du nœud de départ.

Ci-dessous l'implémentation en C de l'algorithme (*inspiré du site www.thecrazyprogrammer.com*) . Il manque quelques fonctions...

```
#define INFINITY 999999
#define G_DIM 10

int lireMatrice(int G[G_DIM][G_DIM],  char *nomFichier) {
	int n, i, j ;
	FILE *fp ;
	fp = fopen(nomFichier, "r") ;
	fscanf(fp, "%d", &n) ; // lecture du nombre de sommets

	//	A TERMINER
}

void dijkstra(int G[G_DIM][G_DIM], int n, int startNode, int distance[], int pred[]) {
	int cost[G_DIM][G_DIM];
	int visited[G_DIM], count, mindistance, nextnode, i, j ;

	for(i = 0 ; i<n ; i++) //initialise cost[]
		for(j = 0 ; j<n ; j++)
			if(G[i][j]==0)
				cost[i][j] = INFINITY ;
			else
				cost[i][j] = G[i][j] ;

	// initialiser pred[],distance[] et visited[]
	// A FAIRE

	distance[startNode] = 0 ;
	visited[startNode] = 1 ;
	count = 1 ;  //count gives the number of nodes seen so far

	while(count < n-1) { // tant que tous les noeud n'ont pas été vus
		mindistance = INFINITY ;

		//cherche nextnode qui est le noeud dont la distance à startNode est minimum
		for(i = 0 ; i<n ; i++)
			if(distance[i]<mindistance && !visited[i]){
				mindistance = distance[i] ;
				nextnode = i ;
			}

			//check if a better path exists through nextnode
			visited[nextnode] = 1 ;
			for(i = 0 ; i<n ; i++)
				if(!visited[i])
					if(mindistance+cost[nextnode][i]<distance[i]){
						distance[i] = mindistance+cost[nextnode][i] ;
						pred[i] = nextnode ;
					}
		count++ ;
	}
}

int main() {
	int distance[G_DIM], pred[G_DIM] ;
	int G[G_DIM][G_DIM], n, u ;
	n = lireMatrice(G, "matAdjacence.txt") ;
	u = 0 ;  // numéro du sommet de départ
	afficheMatrice(G, n) ; // affiche la matrice d'adjacence
	dijkstra(G, n, u, distance, pred) ;
	affiArbre( ...) ; // affiche le résultat
	return 0 ;
}
```

1 Analyse de l'algorithme (2 pts)

Complétez ci-dessous l'évolution du tableau distance[ ] à chaque tour de boucle sur l'exemple proposé. A vous de déterminer quand l'algorithme s'arrête.

i	0	1	2	3	4
distance [i] initialement	0	10	∞	30	100
distance [i], après 1 tour de boucle	0	10	60	30	100
distance [i], après 2 tours de boucle	0	10	50	30	90
distance [i], après 3 tours de boucle	0	10	50	30	60
distance [i], après 4 tours de boucle					
distance [i], après 5 tours de boucle					

2 Initialisation (2 pts)

Dans la fonction dijkstra( ), un commentaire invite à initialiser les tableaux pred[ ],distance[ ] et visited[ ]. Écrire le code C correspondant :

```
for(i = 0 ; i<n ; i++) {
      distance[i] = cost[startNode][i] ;
      visited[i] = 0 ;
      pred[i] = startNode;
}
```

3 Affichage (2 pts)

Dans la fonction main( ), l'appel à la fonction afficheMatrice affiche la matrice d'adjacence. Écrire le code de cette fonction afin que le résultat affiché soit le suivant :

```
0 10 0 30 100
10 0 50 0 0
0 50 0 20 10
30 0 20 0 60
100 0 10 60 0
```

```
void afficheMatrice(int G[G_DIM][G_DIM], int n) {
      int i, j  ;
      for(i = 0 ; i<n ; i++) {
            for(j = 0 ; j<n ; j++) {
                  printf("%d ", G[i][j]) ;
            }
            printf ("\n") ;
      }
}
```

4 Lecture fichier (3 pts)

Dans la fonction lireMatrice( ), un commentaire invite à terminer le code. La fonction retourne le nombre de nœuds du graphe. Le fichier matAdjacence.txt a cette structure :

```
5
0 10 0 30 100
10 0 50 0 0
0 50 0 20 10
30 0 20 0 60
100 0 10 60 0
```

Écrire le code C correspondant :

```
for(i = 0 ; i<n ; i++)
      for(j = 0 ; j<n ; j++)
            fscanf(fp, "%d", &(G[i][j])) ;
fclose (fp) ;
return n ;
```

5 Gestion des erreurs fichier (2 pts)

Dans la fonction lireMatrice( ), il peut y avoir des problèmes au moment de l'appel de la fonction fopen. Comment gérer ce problème ? Écrire le code C à insérer juste après l'appel à fopen( ) :

```
fp = fopen(nomFichier, "r") ;
if (fp == NULL ) {
      perror ("pb fopen") ;
      return -1 ;
}
```

6 Affichage du résultat (3 pts)

Dans la fonction main( ), l'appel à la fonction affiArbre affiche le résultat de l'algorithme de Dijkstra. Écrire le code de cette fonction afin que le résultat affiché soit le suivant :

```
Distance du noeud 1 = 10, chemin = 1<-0
Distance du noeud 2 = 50, chemin = 2<-3<-0
Distance du noeud 3 = 30, chemin = 3<-0
Distance du noeud 4 = 60, chemin = 4<-2<-3<-0
```

```
void affiArbre( int distance[], int pred[], int n, int startNode) {
      int i, j ;
      for(i = 0 ; i<n ; i++)
            if(i != startNode) {
                  printf("Distance du noeud %d = %d", i, distance[i]) ;
                  printf(", chemin = %d", i) ;
                  j = i ;
                  do {
                        j = pred[j] ;
                        printf("<-%d", j) ;
                  } while(j != startNode) ;
                  printf("\n") ;
      }
}
```

7 Affichage du résultat (2 pts)

Dans la fonction main( ), comment utiliser la fonction affiArbre ? Écrire le code correspondant

```
affiArbre(distance, pred, n, 0) ;
```

8 Structures et allocations dynamiques (4 pts)

Dans la fonction lireMatrice, la dimension du graphe G est limité par la macro G_DIM. On se propose d'écrire une alternative à cette fonction. Nous l'appellerons lireMatriceDyn. La matrice d'adjacence du graphe sera allouée de manière dynamique dans la fonction lireMatriceDyn.
lireMatriceDyn retournera une structure de données contenant la matrice d'adjacence ainsi que le nombre de sommets lus.

Écrire le code C de la structure de données :

```
typedef struct sDisj {
      int n ;
      int **G ;
} Graphe ;
```

Écrire le code permettant l'appel de la fonction lireMatriceDyn dans le main( )

```
g = lireMatriceDyn ("matAdjacenceDyn.txt") ;
```

Écrire le code de la fonction lireMatriceDyn sans vous soucier de la gestion des erreurs.

```
Graphe lireMatriceDyn(  char *nomFichier) {
      Graphe g;
      g.n = -1 ;
      int i, j ;
      FILE *fp ; // déclarer fp
      fp = fopen(nomFichier, "r") ;
      fscanf(fp, "%d", &g.n) ;
      g.G = (int**) malloc (g.n*sizeof(int*)) ;
      for (i = 0 ; i < g.n ; i++ )
            g.G[i] = (int *) malloc ( g.n*sizeof(int)) ;

      for(i = 0 ; i<g.n ; i++)
            for(j = 0 ; j<g.n ; j++)
                  fscanf(fp, "%d", &(g.G[i][j])) ;
      fclose (fp) ;
      return g ;
}
```