Algorithme de Dijkstra

Louise Gassot & Vivien Cabannes

Automne 2014

Table des matières

1 Implémentation et Lecture d'un Graphe

1.1 Donnée d'entrée

Les graphes sont décrits dans un fichier texte sous forme canonique :
<nombre de sommets> <nombres d'arrêtes>
<sommet origine> <sommet arrivé> <poid reliant ces sommets>
<···> <···> <···>

Exemples : Voici un premier fichier "graphe1.txt" :

```
5 10
0 1 3
0 2 5
1 2 2
1 3 6
2 1 1
2 3 4
2 4 6
3 4 2
4 1 3
4 3 7
```

Suivi d'un deuxième fichier "graphe2.txt" :

```
5 10
0 1 10
0 2 5
1 2 2
1 3 1
2 1 3
2 3 9
2 4 2
3 4 4
4 1 6
4 3 7
```

1.2 Header

```c
#define INFINI -1
//Correspond a un chemin inexistant

typedef struct Graphe{
    int nombre_sommets;
    struct Sommet {
        int degre_sortant;
        int longueur_liste_arcs_sortants; //NB : Pour ne pas
            changer tout le temps la longueur de la liste
            arcs_sortants construite arete par arete.
        struct Arete{
            int destination;
            int poid;
        } arcs_sortants[1]; //Liste des arcs sortants du
            sommet
    } *sommets[1]; //Liste des sommets du graphe
}*grp;

grp nouveauGraphe(int nombreSommets);
int nombreSommets(grp G);
void libererEspaceAlloueGraphe(grp G);
void ajouterAreteOrienteePonderee(grp G, int depart, int
    arrivee, int poid);
void ajouterArete(grp G, int sommet1, int sommet2);

grp lireGraphe(const char *nomFichier);
```

1.3 Source

```c
#include "graphe.h"

/*-------------------------------------------------
   Creation du type Graphe
-------------------------------------------------*/

int nombreSommets(grp G){
    return G->nombre_sommets;
}

grp nouveauGraphe(int nombreSommets){
    grp G;
```

```
13      G = malloc(sizeof(struct Graphe) + sizeof(struct Sommet
            *) * (nombreSommets-1));
14      if (G == NULL){
15          exit(EXIT_FAILURE); //Verification du succes de
                l'allocation pour le graphe
16      }
17      G->nombre_sommets = nombreSommets;
18
19      int i; //Numero du sommet
20      for(i = 0; i < nombreSommets; i+=1) {
21          G->sommets[i] = malloc(sizeof(struct Sommet));
22          if (G->sommets[i] == NULL){
23              exit(EXIT_FAILURE); //Verification de
                    l'allocation pour le sommet i
24          }
25          G->sommets[i]->degre_sortant = 0;
26          G->sommets[i]->longueur_liste_arcs_sortants = 1;
27      }
28      return G;
29  }
30
31  void libererEspaceAllolueGraphe(grp G){
32      int i;
33      for(i = 0; i < G->nombre_sommets; i+=1){
34              free(G->sommets[i]); //Suppression de l'espace
                    alloue pour le sommet i
35      }
36      free(G); //Suppression de l'espace alloue pour le graphe
37  }
38
39  void ajouterAreteOrienteePonderee(grp G, int depart, int
        arrivee, int poid){
40      if (depart < 0 || depart >= G->nombre_sommets || arrivee
            < 0 || arrivee >= G->nombre_sommets){
41          printf("Erreur dans Ajouter_Arete : numerotation
                incoherente\n");
42          exit(EXIT_FAILURE);
43      }
44
45      //Allongement si besoin la liste des arcs sortants
46      if(G->sommets[depart]->degre_sortant >=
            G->sommets[depart]->longueur_liste_arcs_sortants){
47          G->sommets[depart]->longueur_liste_arcs_sortants *=
                2;
48          G->sommets[depart] = realloc(G->sommets[depart],
```

```
            sizeof(struct Sommet) + sizeof(struct Arete) *
            (G->sommets[depart]->longueur_liste_arcs_sortants
            - 1));
49      }
50
51      //Ajout de la nouvelle arete
52      G->sommets[depart]->arcs_sortants[G->sommets[depart]->
53        degre_sortant].destination = arrivee;
54      G->sommets[depart]->arcs_sortants[G->sommets[depart]->
55        degre_sortant].poid = poid;
56      G->sommets[depart]->degre_sortant +=1;
57  }
58
59  void ajouterArete(grp G, int sommet1, int sommet2){
60      ajouterAreteOrienteePonderee(G, sommet1, sommet2, 1);
61      ajouterAreteOrienteePonderee(G, sommet2, sommet1, 1);
62  }
63
64  /*-------------------------------------------------
65     Lecture d'un graphe pondere sous forme canonique
66  -------------------------------------------------*/
67
68  grp lireGraphe(const char *nomFichier){
69      grp G;
70
71      /* Ouvrir le fichier en mode lecture */
72      FILE* fichier = NULL;
73      fichier = fopen(nomFichier, "r");
74
75      /* Si succes : */
76      if (fichier != NULL){
77          int nombreSommet, curseurLecture, depart, arrivee,
                poid, i;
78          char nombreLu[64];
79
80          /* Lire le premier nombre */
81          i = 0;
82          do{
83              curseurLecture = fgetc(fichier);
84              /*... quand on lit un espace, le mot est fini */
85              if (curseurLecture == 32){
86                  nombreLu[i] = '\0';
87              }
88              /*... sinon, noter le chiffre lu */
89              else{
```

```
90                  nombreLu[i] = curseurLecture;
91              }
92              i+=1;
93          }while(nombreLu[i-1]!='\0');
94          /*... convertir le nombre lu en nombre */
95          nombreSommet = atoi(nombreLu);
96          /* C'est le nombre de sommet du graphe */
97          G = nouveauGraphe(nombreSommet);
98
99          /* Aller a la ligne */
100         do{
101             curseurLecture = fgetc(fichier);
102         }while (curseurLecture != '\n' && curseurLecture !=
                EOF);
103
104         do{
105             /* Si on n'est pas a la fin du fichier */
106             if (curseurLecture != EOF){
107                 /* Lire l'arete : */
108                 /*... lire le sommet d'origine */
109                 i = 0;
110                 do{
111                     curseurLecture = fgetc(fichier);
112                     if (curseurLecture == 32){
113                         nombreLu[i] = '\0';
114                     }
115                     else{
116                         nombreLu[i] = curseurLecture;
117                     }
118                     i+=1;
119                 }while(nombreLu[i-1]!='\0');
120                 depart = atoi(nombreLu);
121
122                 /*... lire le sommet d'arrive */
123                 i = 0;
124                 do{
125                     curseurLecture = fgetc(fichier);
126                     if (curseurLecture == 32){
127                         nombreLu[i] = '\0';
128                     }
129                     else{
130                         nombreLu[i] = curseurLecture;
131                     }
132                     i+=1;
133                 }while(nombreLu[i-1]!='\0');
```

```
134                 arrivee = atoi(nombreLu);
135
136                 /*... lire le poid */
137                 i = 0;
138                 do{
139                     curseurLecture = fgetc(fichier);
140                     /* Quand on lit un saut de ligne, le mot
                           est fini */
141                     if (curseurLecture == 10 ||
                           curseurLecture == EOF){
142                         nombreLu[i] = '\0';
143                     }
144                     else{
145                         nombreLu[i] = curseurLecture;
146                     }
147                     i+=1;
148                 }while(nombreLu[i-1]!='\0');
149                 poid = atoi(nombreLu);
150
151                 /* Ajouter l'arete */
152                 ajouterAreteOrienteePonderee(G, depart,
                       arrivee, poid);
153             }
154
155         /* Tant qu'on n'a pas atteint la fin du fichier */
156         } while (curseurLecture != EOF);
157
158         /* Fermer le fichier */
159         fclose(fichier);
160     }
161
162     else{
163         printf("Impossible d'ouvrir le fichier %s",
               nomFichier);
164     }
165
166     /* On renvoit le graphe cree */
167     return G;
168 }
```

2 Test de la connexité d'un graphe

2.1 Source

```
#include "graphe.h"

/*-------------------------------------------------
   Declarations des fonctions
-------------------------------------------------*/

typedef struct Element elt;
typedef struct Pile pile;
pile *initialisation();
void empiler(pile *pileCourante, int ajout);
int depiler(pile *pileCourante);

/*-------------------------------------------------
   Environnement global
-------------------------------------------------*/

void essaiQuestion1(){
    grp G;

    G = nouveauGraphe(1);
    if(estConnexe(G)){
        printf("Le graphe est connexe");
    }
    else{
        printf("Le graphe n'est pas connexe");
    }
    printf("\n");
    libererEspaceAlloueGraphe(G);

    G = nouveauGraphe(3);
    if(estConnexe(G)){
        printf("Le graphe est connexe");
    }
    else{
        printf("Le graphe n'est pas connexe");
    }
    printf("\n");
    libererEspaceAlloueGraphe(G);

    G = nouveauGraphe(3);
    ajouterArete(G, 0, 1);
```

```
42      ajouterArete(G, 1, 2);
43      if(estConnexe(G)){
44          printf("Le graphe est connexe");
45      }
46      else{
47          printf("Le graphe n'est pas connexe");
48      }
49      printf("\n");
50      libererEspaceAlloueGraphe(G);
51      G = lireGraphe("graphe1.txt");
52      if(estConnexe(G)){
53          printf("Le graphe est connexe");
54      }
55      else{
56          printf("Le graphe n'est pas connexe");
57      }
58      printf("\n");
59      libererEspaceAlloueGraphe(G);
60
61      G = lireGraphe("graphe2.txt");
62          if(estConnexe(G)){
63          printf("Le graphe est connexe");
64      }
65      else{
66          printf("Le graphe n'est pas connexe");
67      }
68      printf("\n");
69      libererEspaceAlloueGraphe(G);
70
71      printf("\n");
72  }
73
74  /*-------------------------------------------------
75     Creation d'un type pile pour stocker les sommets a visiter
76  -------------------------------------------------*/
77
78  struct Element{
79      int numero;
80      elt *suivant;
81  };
82
83  struct Pile{
84      elt *tete;
85  };
86
```

```c
pile *initialisation(){
    pile *pileCree = malloc(sizeof(*pileCree));
    pileCree->tete = NULL;
    return pileCree;
}

void empiler(pile *pileCourante, int ajout){
    elt *nouveau = malloc(sizeof(*nouveau));
    if (pileCourante == NULL || nouveau == NULL){
        exit(EXIT_FAILURE);
    }
    nouveau->numero = ajout;
    nouveau->suivant = pileCourante->tete;
    pileCourante->tete = nouveau;
}

int depiler(pile *pileCourante){
    if (pileCourante == NULL){
        exit(EXIT_FAILURE);
    }
    int nombreDepile = -1; //Renvoie -1 si la pile est vide
    elt *elementDepile = pileCourante->tete;
    if (pileCourante->tete != NULL){
        nombreDepile = elementDepile->numero;
        pileCourante->tete = elementDepile->suivant;
        free(elementDepile);
    }
    return nombreDepile;
}

/*-----------------------------------------------------
   1. Implementer une fonction qui teste si un graphe est
      connexe.
-----------------------------------------------------*/

int estConnexe(grp G){
    int i; //Curseur de parcours

    /* Créer un tableau booleens d'appartenance a la
       composante connexe du premier sommet */
    int *sommetsVisites;
    sommetsVisites = malloc(G->nombre_sommets * sizeof(int));
    if (sommetsVisites == NULL){
        exit(EXIT_FAILURE); //Verification de l'allocation
    }
```

```
130     for (i=0; i < G->nombre_sommets ; i+=1){
131         sommetsVisites[i] = 0;
132     }
133 
134     int sommetConsidere = 0; //numero du sommet sur lequel
           on travail
135     sommetsVisites[sommetConsidere] = 1;
136 
137     /*Creer un sac de sommets a visiter */
138     pile *sacSommets;
139     sacSommets = initialisation();
140 
141     /*Mettre le premier sommet dans le sac. */
142     empiler(sacSommets, sommetConsidere);
143 
144     /*Tant que le sac n'est pas vide : */
145     while(sommetConsidere != -1){
146         /*Sortir le premier sommet du sac, ... */
147         sommetConsidere = depiler(sacSommets);
148 
149         if (sommetConsidere!=-1){ //Assertion sac non vide
150             /*... le marquer. */
151             sommetsVisites[sommetConsidere]=1;
152 
153             /*Pour chacun de ses voisins, ... */
154             for(i=0; i < G->sommets[sommetConsidere]->
155               degre_sortant; i+=1){
156                 /*... s'il n'est pas marque,... */
157                 if (sommetsVisites[G->sommets[sommet
158                   Considere]->arcs_sortants[i].destination]
                         == 0){
159                     /*... le mettre dans le sac. */
160                     empiler(sacSommets, G->sommets[sommetCon
161                       sidere]->arcs_sortants[i].destination);
162                 }
163             }
164         }
165     }
166 
167     free(sacSommets); //Liberation de la memoire
168 
169     /*Parcourir les sommets, ... */
170     for (i=0; i < G->nombre_sommets; i+=1){
171         /*Si un sommet n'est pas marque, ... */
172         if (sommetsVisites[i]==0){
```

```
173                 /*... le graphe n'est pas connexe. */
174                 return 0;
175             }
176         }
177
178         /*Sinon, le graphe est connexe.*/
179         return 1;
180 }
```

2.2 Résultats

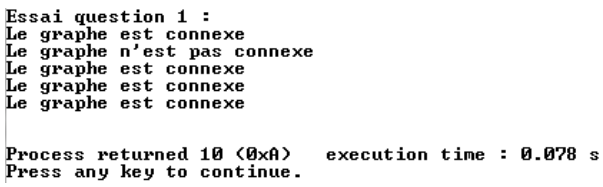

FIGURE 1 – Résultat console, question 1

3 Dijkstra naïf

3.1 Header

```
typedef struct Arc arc;
typedef struct Pile2 pile2;
pile2 *initialisation2();
void empiler(pile2 *pileCourante, int sommetOrigine, int
    sommetArrive, int poid);
int *extraireMin(pile2 *pileCourante);
void afficherPile(pile2 *pileCourante);
int *dijkstraNaif(grp G, int s);
void essaiQuestion2();
```

3.2 Source

```
#include "graphe.h"
#include "question2.h"

/*-------------------------------------------------
   Environnement global
-------------------------------------------------*/

void essaiQuestion2(){
    grp G;

    G = lireGraphe("graphe1.txt");
    int *Tableau = dijkstraNaif(G, 0);
    int i;
    for(i=0; i<nombreSommets(G); i+=1){
        printf("sommet %d est a distance %d de la
            source.\n", i, Tableau[i]);
    }
    printf("\n");
    libererEspaceAlloueGraphe(G);

    G = lireGraphe("graphe2.txt");
    Tableau = dijkstraNaif(G, 0);
    for(i=0; i<nombreSommets(G); i+=1){
        printf("sommet %d est a distance %d de la
            source.\n", i, Tableau[i]);
    }
    printf("\n");
```

```
26      libererEspaceAlloueGraphe(G);
27  }
28
29  /*-------------------------------------------------
30     Implementation d'une file de priorite naive a partir d'un
          type pile
31  -------------------------------------------------*/
32
33  struct Arc{
34      int sommet_entrant;
35      int sommet_sortant;
36      int poid;
37      arc *suivant;
38  };
39
40  struct Pile2{
41      arc *tete;
42  };
43
44  pile2 *initialisation2(){
45      pile2 *pileCree = malloc(sizeof(*pileCree));
46      pileCree->tete = NULL;
47      return pileCree;
48  }
49
50  void empiler2(pile2 *pileCourante, int sommetOrigine, int
        sommetArrive, int poid){
51      arc *nouveau = malloc(sizeof(*nouveau));
52      if (pileCourante == NULL || nouveau == NULL){
53          exit(EXIT_FAILURE);
54      }
55      nouveau->sommet_entrant = sommetOrigine;
56      nouveau->sommet_sortant = sommetArrive;
57      nouveau->poid = poid;
58      nouveau->suivant = pileCourante->tete;
59      pileCourante->tete = nouveau;
60  }
61
62  int *extraireMin(pile2 *pileCourante){
63      /* Tableau de resultats sous la forme
            {indicateurPileVide, u, v, poid}*/
64      int *resultat;
65      resultat = malloc(4 * sizeof(int));
66      if (resultat == NULL){
67          exit(EXIT_FAILURE); //Verification de l'allocation
```

```
68      }
69
70      resultat[0] = 0;
71
72      /* Si la pile est non vide : */
73      if (pileCourante->tete != NULL){
74          resultat[0] = 1;
75
76          /* Rechercher le poid minimum et l'element precedent
                l'element de poid minimum : */
77          int poidMin = pileCourante->tete->poid;
78          arc *elementMinPrecedent = pileCourante->tete;
79          arc *elementParcours = pileCourante->tete;
80
81          /* Distinguer le cas ou l'element se trouve en tete
                de pile a l'aide d'un booleen. */
82          int elementMinSeTrouveAuDebut = 1;
83
84          /* Parcourir la liste */
85          while (elementParcours->suivant != NULL){
86              /* Pour un element de poid plus petit que le
                    poid minimum jusqu'a la obtenu, ... */
87              if (elementParcours->suivant->poid < poidMin){
88                  /* ...actualiser les donnees recherchees */
89                  elementMinSeTrouveAuDebut = 0;
90                  elementMinPrecedent = elementParcours;
91                  poidMin = elementParcours->suivant->poid;
92              }
93              elementParcours = elementParcours->suivant;
94          }
95
96          /* En deduire l'element minimum, et le supprimer de
                la pile */
97          arc *elementMin;
98          if (elementMinSeTrouveAuDebut){
99              elementMin = elementMinPrecedent;
100             pileCourante->tete = elementMin->suivant;
101         }
102         else{
103             elementMin = elementMinPrecedent->suivant;
104             elementMinPrecedent->suivant =
                    elementMin->suivant;
105         }
106
107         /* Extraire les informations voulues */
```

```
108         resultat[1] = elementMin->sommet_entrant;
109         resultat[2] = elementMin->sommet_sortant;
110         resultat[3] = poidMin;
111 
112         /*Liberer la memoire utilisee */
113         free(elementMin);
114     }
115     return resultat;
116 }
117 
118 void afficherPile(pile2 *pileCourante){
119     if (pileCourante == NULL)
120     {
121         exit(EXIT_FAILURE);
122     }
123     arc *actuel = pileCourante->tete;
124     while (actuel != NULL)
125     {
126         printf("Sommet %d -> Sommet %d de poid %d\n",
              actuel->sommet_entrant, actuel->sommet_sortant,
              actuel->poid);
127         actuel = actuel->suivant;
128     }
129     printf("\n");
130 }
131 
132 
133 /*-------------------------------------------------
134    2. Implementer l'algorithme de Dijkstra avec une file de
       priorite naive.
135 -------------------------------------------------*/
136 
137 int *dijkstraNaif(grp G, int s){
138     int i, longueur, *resultat, estNonVide; //Variables
          internes
139 
140     /* Soit P une file de priorite vide. */
141     pile2 *P = initialisation2();
142 
143     /* Soit d_s un tableau de taille |S| initialise a "non
          defini", qui va stocker les plus courts chemins. */
144     int *d_s;
145     d_s = malloc(G->nombre_sommets * sizeof(int));
146     if (d_s == NULL){
147         exit(EXIT_FAILURE); //Verification de l'allocation
```

```
148     }
149     for (i=0; i < G->nombre_sommets ; i+=1){
150         d_s[i] = INFINI;
151     }
152 
153     /* Soit Vus un ensemble vide de sommets. */
154     int *Vus;
155     Vus = malloc(G->nombre_sommets * sizeof(int));
156     if (Vus == NULL){
157         exit(EXIT_FAILURE); //Verification de l'allocation
158     }
159     for (i=0; i < G->nombre_sommets; i+=1){
160         Vus[i] = 0;
161     }
162 
163     /* Considerer le sommet source, puis...*/
164     int v = s, poid = 0;
165 
166     /*...repeter : */
167     do{
168         /* Si v n'a pas ete vu : */
169         if (Vus[v] == 0){
170             /* d_s(v) <- poid */
171             d_s[v] = poid;
172 
173             /* Ajouter v a Vus */
174             Vus[v] = 1;
175 
176             /* Pour tout arc a sortant de v : */
177 //            afficherPile(P);
178             for (i=0; i < G->sommets[v]->degre_sortant;
                    i+=1){
179                 /* longueur <- d_s[u] + w_a */
180                 longueur = d_s[v] +
                        G->sommets[v]->arcs_sortants[i].poid;
181                 /* INSERER(P, a, longueur) */
182                 empiler2(P, v, G->sommets[v]->arcs_sor
183                   tants[i].destination, longueur);
184 //                afficherPile(P);
185                 }
186             }
187         /* (u, v), poid <- EXRTRAIRE-MIN(P)*/
188         resultat = extraireMin(P);
189         estNonVide = resultat[0];
190         v = resultat[2];
```

```
191             poid = resultat[3];
192         /* Tant que P st non-vide : */
193         }while (estNonVide);
194
195         free(P); //Liberation memoire
196
197         return d_s;
198     }
```

3.3 Résultats

```
Essai question 2 :
sommet 0 est a distance 0 de la source.
sommet 1 est a distance 3 de la source.
sommet 2 est a distance 5 de la source.
sommet 3 est a distance 9 de la source.
sommet 4 est a distance 11 de la source.

sommet 0 est a distance 0 de la source.
sommet 1 est a distance 8 de la source.
sommet 2 est a distance 5 de la source.
sommet 3 est a distance 9 de la source.
sommet 4 est a distance 7 de la source.


Process returned 1 (0x1)   execution time : 0.078 s
Press any key to continue.
```

FIGURE 2 – Résultat console, question 2

4 Implémentation Tas de Fibonacci

4.1 Header

```
typedef struct TasFibonacci *fib;
typedef struct Noeud nd;
fib nouveauTas();
void inserer(fib F, int sommetOrigine, int sommetArrive, int
    poid);
void extraireMin3(fib F, int* resultat);
void restructurer(fib F);
```

4.2 Source

```
#include <math.h>
#include "graphe.h"
#include "question3.h"

/*-------------------------------------------------
3. Implementer une file de priorite utilisant les tas de
    Fibonacci
-------------------------------------------------*/

struct TasFibonacci{
    nd *minimum;
    int nombre_noeuds; //Utile pour borner le nombre maximum
       de fils d'un sommet
};

struct Noeud{
    /* Pointeur vers d'autres elements de la structure */
    nd *fils;
    nd *frere_gauche;
    nd *frere_droit;
    int nombre_fils; //Utile pour restructurer

    /* Information relative au noeud */
    int sommet_entrant;
    int sommet_sortant;
    int poid;
};

fib nouveauTas(){
```

```
28      fib F;
29      F = malloc(sizeof(struct TasFibonacci));
30      if (F == NULL){
31          exit(EXIT_FAILURE);
32      }
33      F->minimum = NULL;
34      F->nombre_noeuds = 0;
35      return F;
36  }
37
38  void inserer(fib F, int sommetOrigine, int sommetArrive, int
        poid){
39      nd *nouveau_noeud = malloc(sizeof(*nouveau_noeud));
40      if (F == NULL || nouveau_noeud == NULL){
41          exit(EXIT_FAILURE);
42      }
43      nouveau_noeud->sommet_entrant = sommetOrigine;
44      nouveau_noeud->sommet_sortant = sommetArrive;
45      nouveau_noeud->poid = poid;
46      nouveau_noeud->nombre_fils = 0;
47      nouveau_noeud->fils = NULL;
48      if (F->minimum == NULL){
49          F->minimum = nouveau_noeud;
50          nouveau_noeud->frere_droit = nouveau_noeud;
51          nouveau_noeud->frere_gauche = nouveau_noeud;
52      }
53      else{
54          /* Inserer le nouveau noeud a la liste des sommets
               racines de F, a gauche de F->minimum */
55          nouveau_noeud->frere_gauche =
               F->minimum->frere_gauche;
56          nouveau_noeud->frere_gauche->frere_droit =
               nouveau_noeud;
57          nouveau_noeud->frere_droit = F->minimum;
58          F->minimum->frere_gauche = nouveau_noeud;
59
60          /* On actualise F->minimum */
61          if (nouveau_noeud->poid < F->minimum->poid){
62              F->minimum = nouveau_noeud;
63          }
64      }
65      F->nombre_noeuds += 1;
66  }
67
68  void extraireMin3(fib F, int* resultat){
```

```
69      /* Tableau de resultats sous la forme
           {indicateurPileVide, u, v, poid}
70      NB : l'extraction de u est inutile */
71      if (resultat == NULL){
72          exit(EXIT_FAILURE);
73      }
74      resultat[0]=0;
75      if (F->minimum != NULL){
76          /* Prelevement des informations voulues */
77          resultat[0] = 1;
78          resultat[1] = F->minimum->sommet_entrant;
79          resultat[2] = F->minimum->sommet_sortant;
80          resultat[3] = F->minimum->poid;
81
82          /* Inserer la liste des fils de F->minimum a gauche
               de F->minimum */
83          if(F->minimum->fils != NULL){
84              F->minimum->fils->frere_droit->frere_gauche =
                   F->minimum->frere_gauche;
85              F->minimum->frere_gauche->frere_droit =
                   F->minimum->fils->frere_droit;
86              F->minimum->frere_gauche = F->minimum->fils;
87              F->minimum->fils->frere_droit = F->minimum;
88          }
89
90          /* Enlever F->minimum de la liste des sommets
               racines de F */
91          F->minimum->frere_gauche->frere_droit =
               F->minimum->frere_droit;
92          F->minimum->frere_droit->frere_gauche =
               F->minimum->frere_gauche;
93
94          F->nombre_noeuds-=1;
95          if (F->minimum == F->minimum->frere_droit){ //Le tas
               ne contenait qu'un seul element
96              free(F->minimum); //Liberation de l'espace alloue
97              F->minimum = NULL;
98          }
99          else{
100             nd *pointeur = F->minimum->frere_droit;
101             free(F->minimum); //Liberation de l'espace alloue
102             F->minimum = pointeur;
103             restructurer(F);
104         }
105     }
```

```
106 }
107
108 void restructurer(fib F){
109     /* Resultat souhaite : pour chaque d, on veut d'au plus
        un sommet racine ayant d fils */
110
111     /* Liste se souvenant du sommet vu de degre d */
112     int maxNombreFils =  2 * (2 +
        log(F->nombre_noeuds)/log((1+sqrt(5))/2));
113 /*Normalement Ã§a devrait fonctionner avec :
114     int maxNombreFils =  (1 +
        abs(log(F->nombre_noeuds)/log((1+sqrt(5))/2)));
115 */
116     nd **sommetsRacinesDegre;
117     sommetsRacinesDegre = malloc(maxNombreFils *
        sizeof(nd*));
118     if (sommetsRacinesDegre == NULL){
119         exit(EXIT_FAILURE);
120     }
121     int d;
122     for (d=0; d < maxNombreFils; d+=1){
123         sommetsRacinesDegre[d] = NULL;
124     }
125
126     /* Pour chaque noeud racine de F */
127     nd *sommet = F->minimum, *autreSommet, *temp;
128     do{
129         d = sommet->nombre_fils;
130
131         /* Si on a deja un sommet ayant d fils */
132         while(sommetsRacinesDegre[d]!=NULL){
133             autreSommet = sommetsRacinesDegre[d];
134
135             /* Quite a echanger les deux sommets, supposons
                autreSommet->poid > sommet->poid*/
136             if (sommet->poid > autreSommet->poid){
137                 /* Echanger leur place (utile pour faire
                    l'enumeration des sommets racines) */
138                 temp = sommet->frere_gauche;
139                 sommet->frere_gauche =
                    autreSommet->frere_gauche;
140                 autreSommet->frere_gauche = temp;
141                 sommet->frere_gauche->frere_droit = sommet;
142                 autreSommet->frere_gauche->frere_droit =
                    autreSommet;
```

```
143                 temp = sommet->frere_droit;
144                 sommet->frere_droit =
                        autreSommet->frere_droit;
145                 autreSommet->frere_droit = temp;
146                 sommet->frere_droit->frere_gauche = sommet;
147                 autreSommet->frere_droit->frere_gauche =
                        autreSommet;
148
149                 /* Echanger leur nom */
150                 temp = sommet;
151                 sommet = autreSommet;
152                 autreSommet = temp;
153             }
154
155             /* Affilier l'autre sommet au premier sommet */
156             /*... Enlever l'autre sommet de la liste des
                    sommets racines */
157             autreSommet->frere_gauche->frere_droit =
                    autreSommet->frere_droit;
158             autreSommet->frere_droit->frere_gauche =
                    autreSommet->frere_gauche;
159
160             /*... L'ajouter a la liste des fils de x */
161             if (sommet->fils == NULL){
162                 sommet->fils = autreSommet;
163                 autreSommet->frere_gauche = autreSommet;
164                 autreSommet->frere_droit = autreSommet;
165             }
166             else{
167                 /* Ajout a gauche du fils existant*/
168                 autreSommet->frere_droit = sommet->fils;
169                 autreSommet->frere_gauche =
                        sommet->fils->frere_gauche;
170                 sommet->fils->frere_gauche = autreSommet;
171                 autreSommet->frere_gauche->frere_droit =
                        autreSommet;
172             }
173
174             /*... Augmenter le degre du premier sommet */
175             sommet->nombre_fils += 1;
176
177             /* Il n'y a plus de sommet de degre d parmis
                    ceux deja visite */
178             sommetsRacinesDegre[d] = NULL;
179             /* Le sommet considere a desormais degre d+1 */
```

```
180             d += 1;
181         }
182
183         /* Sinon, il n'y a pas de sommets racines de degre d
               autre que le sommet considere */
184         sommetsRacinesDegre[d] = sommet;
185
186         /* On prend le sommet suivant, tant qu'on ne les a
               pas tous deja vus */
187         sommet = sommet->frere_droit;
188     }while(sommet!=F->minimum);
189
190     /* Recherche du minimum */
191     /* Tous les sommets racines se trouvent sur la liste
           sommetsRacinesDegre */
192     for(d=0; d < maxNombreFils; d+= 1){
193         if (sommetsRacinesDegre[d] !=NULL){
194             if (sommetsRacinesDegre[d]->poid <
                   F->minimum->poid){
195                 F->minimum = sommetsRacinesDegre[d];
196             }
197         }
198     }
199     free(sommetsRacinesDegre); //Liberation memoire
200 }
```

5 Dijkstra final

5.1 Header

```
int *dijkstra(grp G, int s);
void essaiQuestion4();
```

5.2 Source

```
#include <math.h>
#include "graphe.h"
#include "question3.h"
#include "question4.h"

/*-------------------------------------------------
   Environnement global
-------------------------------------------------*/

void essaiQuestion4(){
    grp G;

    G = lireGraphe("graphe1.txt");
    int *Tableau = dijkstra(G, 0);
    int i;
    for(i=0; i<nombreSommets(G); i+=1){
        printf("sommet %d est a distance %d de la
            source.\n", i, Tableau[i]);
    }
    printf("\n");
    libererEspaceAllogueGraphe(G);

    G = lireGraphe("graphe2.txt");
    Tableau = dijkstra(G, 0);
    for(i=0; i<nombreSommets(G); i+=1){
        printf("sommet %d est a distance %d de la
            source.\n", i, Tableau[i]);
    }
    printf("\n");
    libererEspaceAllogueGraphe(G);
}

/*-------------------------------------------------
```

```
32      4. Implementer l'algorithme de Dijkstra avec une file de
           priorite representee par des tas de Fibonacci.
33  ------------------------------------------------*/
34
35  int *dijkstra(grp G, int s){
36      int i, longueur, resultat[4];
37      fib F = nouveauTas();
38      int *d_s, *Vus, estNonVide;
39      d_s = malloc(G->nombre_sommets * sizeof(int));
40      if (d_s == NULL){
41          exit(EXIT_FAILURE);
42      }
43      Vus = malloc(G->nombre_sommets * sizeof(int));
44      if (Vus == NULL){
45          exit(EXIT_FAILURE);
46      }
47      for (i=0; i < G->nombre_sommets; i+=1){
48          d_s[i] = INFINI;
49          Vus[i] = 0;
50      }
51      int v = s, poid = 0;
52      do{
53          if (Vus[v] == 0){
54              d_s[v] = poid;
55              Vus[v] = 1;
56              for (i=0; i < G->sommets[v]->degre_sortant;
                    i+=1){
57                  longueur = d_s[v] +
                        G->sommets[v]->arcs_sortants[i].poid;
58                  inserer(F, v, G->sommets[v]->arcs_sor
59                    tants[i].destination, longueur);
60              }
61          }
62          extraireMin3(F, resultat);
63          estNonVide = resultat[0];
64          v = resultat[2];
65          poid = resultat[3];
66      }while (estNonVide);
67      free(F);
68      free(Vus);
69      return d_s;
70  }
```

5.3 Résultats

```
Essai question 4 :
sommet 0 est a distance 0 de la source.
sommet 1 est a distance 3 de la source.
sommet 2 est a distance 5 de la source.
sommet 3 est a distance 9 de la source.
sommet 4 est a distance 11 de la source.

sommet 0 est a distance 0 de la source.
sommet 1 est a distance 8 de la source.
sommet 2 est a distance 5 de la source.
sommet 3 est a distance 9 de la source.
sommet 4 est a distance 7 de la source.


Process returned 1 (0x1)   execution time : 0.078 s
Press any key to continue.
```

FIGURE 3 – Résultat console, question 4

6 Bonus

6.1 Header

```
typedef struct Liste_hash list_hash;
typedef list_hash** Hashmap;
Hashmap init_hash_map();
int get_hash(Hashmap hash_map, int key);
void add(Hashmap hash_map, int key, int v);
void free_hash_map(Hashmap hash_map);
  int hash(int x);
  void free_list_hash(list_hash* l);

typedef struct Liste_prochain list_prochain;
struct Liste_prochain{
  int index_prochain;
  int duree;
  list_prochain* next;
};
typedef struct Arret arret;
struct Arret{
  int id;
  list_prochain* prochains;
};
void add_prochain(arret* a, int index_prochain, int duree);
void free_arret(arret* ar);
  void free_list_prochain(list_prochain* l);

int lecture_fichier_arrets(Hashmap id_to_index);
  char /*bool*/ next_line(FILE* f);
  char /*bool*/ read_line(FILE* f, Hashmap id_to_index, int
     v);

arret** creation_arrets(Hashmap id_to_index, int nb_arrets);
void free_arrets(arret** arrets, int nb_arrets);

char /*bool*/ lecture_fichier_prochain(arret** arrets, int
    nb_arrets, Hashmap id_to_index);
  typedef struct Temps_arret temps_arret;
  char /*bool*/ prochaine_virgule(FILE* f);
  inline int minus_date(int a, int b);
  inline int calcul_date(int heure, int minute, int seconde);
  char /*bool*/ next_line2(FILE* f);
  char /*bool*/ read_time(FILE* f, temps_arret* trip,
     Hashmap id_to_index);
```

```
39   int read_trip(FILE* f, temps_arret* trip, int trip_id,
        Hashmap id_to_index, arret** arrets);
40
41 char /*bool*/ lecture_fichier_transfers(arret** arrets, int
     nb_arrets, Hashmap id_to_index);
42   char /*bool*/ read_transfers(FILE* f, Hashmap id_to_index,
        arret** arrets);
43
44 void remplit_graphe(grp g, arret** arrets, int nb_arrets);
```

6.2 Source

```
1 #include "graphe.h"
2 #include "bonus.h"
3
4 /*Dans cette question, on s'inspirera du bonus pour créer un
     graphe mais on se servira des données de la RATP (charger
     les fichiers stops.txt stop_times.txt et transfers.txt)
     pour calculer à partir de l'algorithme de dijkstra le
     plus court chemin d'une station de métro à une autre.
5 Sont pris en compte : le temps pris par le transport en
     commun pour aller d'une station à l'autre, et le temps de
     correspondance. Les stations sont représentées par leur
     numéro, les correspondances avec les stations réelles
     sont notées dans stops.txt
6 Les sommets du graphe (orienté) sont les arrets, les arêtes
     les trajets directs effectués par les différents
     transports en commun d'un arret à l'autre. */
7
8 #define HASH_SIZE   30000
9 #define MAX_SIZE_TRIP 200
10
11 /*------------------------------------------------
12 Hash_map(int, int) / Creation d'une table de hachage (les
     numeros des stations sont trop grands)
13 ------------------------------------------------*/
14
15 struct Liste_hash{
16   int v;
17   int key;
18   list_hash *next;
19 };
20
21 Hashmap init_hash_map(){
```

```
22    Hashmap hash_map;
23    int i;
24    hash_map=malloc(sizeof(list_hash*)*HASH_SIZE);
25
26    for(i=0; i<HASH_SIZE; i++){
27      hash_map[i]=NULL;
28    }
29    return hash_map;
30  }
31
32  int hash(int x){
33    return x%HASH_SIZE;
34  }
35
36  int get_hash(Hashmap hash_map, int key){
37    list_hash* l;
38
39    for(l=hash_map[hash(key)]; l!=NULL; l=l->next)
40      if(l->key==key)
41        return l->v;
42    return -1;
43  }
44
45  void add(Hashmap hash_map, int key, int v){
46    list_hash* l, *hd;
47    int index=hash(key);
48
49    l=hash_map[index];
50    hd = malloc(sizeof(list_hash));
51    if(hd == NULL)
52      exit(EXIT_FAILURE);
53    hd->v=v;
54    hd->key=key;
55    hd->next=l;
56    hash_map[index]=hd;
57  }
58
59  void free_list_hash(list_hash* l){
60    if(l!=NULL){
61      free_list_hash(l->next);
62      free(l);
63    }
64  }
65
66  void free_hash_map(Hashmap hash_map){
```

```
67    int i;
68
69    for(i=0; i<HASH_SIZE; i++){
70      free_list_hash(hash_map[i]);
71    }
72    free(hash_map);
73  }
74
75  /*-----------------------------------------------------
76  Structure pour les arrets
77  -----------------------------------------------------*/
78
79  // Vérifie si prochain n'existe pas déjà.
80  void add_prochain(arret* a, int index_prochain, int duree){
81    list_prochain* l;
82
83    for(l=a->prochains; l!=NULL; l=l->next)
84      if(l->index_prochain==index_prochain)
85        break;
86    if(l==NULL){
87      l = malloc(sizeof(list_prochain));
88      if(l == NULL)
89        exit(EXIT_FAILURE);
90      l->index_prochain=index_prochain;
91      l->duree=duree;
92      l->next=a->prochains;
93      a->prochains=l;
94    }
95  }
96
97  void free_list_prochain(list_prochain* l){
98    if(l!=NULL){
99      free_list_prochain(l->next);
100     free(l);
101   }
102 }
103
104 void free_arret(arret* ar) {
105   free_list_prochain(ar->prochains);
106   free(ar);
107 }
108
109 /*-----------------------------------------------------
110 Lecture du fichier stops.txt
111 -----------------------------------------------------*/
```

```
112
113 //next_line retourne 0 si c'est la fin du fichier et 1 sinon
114 char /*bool*/ next_line(FILE* f){
115   char c;
116   while((c=fgetc(f))!='\n')
117     if(c==EOF)
118       return 0;
119   return 1;
120 }
121
122 //read_line met le numéro de l'arrêt dans la table de
      hachage et retourne 0 ssi fin fichier
123 char /*bool*/ read_line(FILE* f, Hashmap id_to_index, int v){
124   int i;
125
126   if(fscanf(f, "%d", &i)!=1)
127     return 0;
128   add(id_to_index,i, v);
129   return next_line(f);
130 }
131
132 // modifie id_to_index (qui effectue une bijection entre les
      arrets (leur id) et [0;n-1]) et renvoie le nombre
      d'arrêts n
133 int lecture_fichier_arrets(Hashmap id_to_index){
134   FILE* farrets = NULL;
135   int nb_arrets;
136
137   farrets = fopen("stops.txt", "r");
138   if (farrets == NULL)
139     return -1;
140
141   if(!next_line(farrets))
142     return -1;
143   for(nb_arrets=0; read_line(farrets, id_to_index,
        nb_arrets); nb_arrets++);
144
145   fclose(farrets);
146
147   return nb_arrets;
148 }
149
150 /*-------------------------------------------------
151 Creation des arrets
152 -------------------------------------------------*/
```

```
153
154 // renvoie un tableau contenant les arrets (dans un ordre
      quelconque)
155 arret** creation_arrets(Hashmap id_to_index, int nb_arrets){
156   arret** arrets;
157   list_hash* l;
158   arret* ar;
159   int i;
160
161   arrets=malloc(sizeof(arret*)*nb_arrets);
162   if(arrets==NULL)
163     return NULL;
164
165   for(i=0; i<HASH_SIZE; i++)
166     for(l=id_to_index[i]; l!=NULL; l=l->next) {
167       if((ar=malloc(sizeof(arret)))==NULL)
168         return NULL;
169       ar->id=l->key;
170       ar->prochains=NULL;
171       arrets[l->v]=ar;
172     }
173
174   return arrets;
175 }
176
177 void free_arrets(arret** arrets, int nb_arrets){
178   int i;
179
180   for(i=0; i<nb_arrets; i++)
181     free_arret(arrets[i]);
182   free(arrets);
183 }
184
185 /*-------------------------------------------------
186  Lecture du fichier stop_times.txt et création des prochains
187 -------------------------------------------------*/
188
189 struct Temps_arret{
190   int date;
191   int index;
192 };
193
194 char /*bool*/ prochaine_virgule(FILE* f){
195   char c;
196   while((c=fgetc(f))!=',')
```

```
197     if(c==EOF)
198       return 0;
199   return 1;
200 }
201
202 inline int minus_date(a,b){
203   return a-b + (b>a?3600*24:0);
204 }
205
206 inline int calcul_date(int heure, int minute, int seconde){
207   return seconde+60*(minute+60*heure);
208 }
209
210 char /*bool*/ read_time(FILE* f, temps_arret
      trip[MAX_SIZE_TRIP], Hashmap id_to_index){
211   int heure,minute,seconde,stop_id,depart;
212   int date, index;
213
214   if(!prochaine_virgule(f))
215     return 0;
216   if(fscanf(f, "%d:%d:%d,%d,%d", &heure, &minute, &seconde,
        &stop_id, &depart)!=5)
217     return 0;
218   date=calcul_date(heure,minute,seconde);
219   index=get_hash(id_to_index,stop_id);
220   trip[depart-1].date=date;
221   trip[depart-1].index=index;
222   //if(depart!=1)
223   //  add_prochain(arrets[dernier->index], arrets[index],
        minus_date(date, dernier->date));
224
225   return next_line(f);
226 }
227
228 int read_trip(FILE* f, temps_arret trip[MAX_SIZE_TRIP], int
      trip_id, Hashmap id_to_index, arret** arrets){
229   int nb_stops,i;
230   int trip_courant=trip_id;
231
232   for(nb_stops=0; trip_courant==trip_id; nb_stops++) {
233     if(!read_time(f, trip, id_to_index))
234       return 0;
235
236     if(fscanf(f, "%d,", &trip_courant)!=1)
237       return 0;
```

```
238   }
239
240   for(i=1; i<nb_stops; i++){
241     add_prochain(arrets[trip[i-1].index], trip[i].index,
          minus_date(trip[i].date, trip[i-1].date));
242   }
243   return trip_courant;
244 }
245
246 char /*bool*/ lecture_fichier_prochain(arret** arrets, int
     nb_arrets, Hashmap id_to_index){
247   FILE* fprochains = NULL;
248   int trip_courant;
249   temps_arret espace_trip[MAX_SIZE_TRIP]; // Alloue une
        unique fois la place nécessaire pour read_trip
250
251   fprochains = fopen("stop_times.txt", "r");
252   if (fprochains == NULL)
253     return 1;
254
255   if(!next_line(fprochains))
256     return 1;
257
258   if(fscanf(fprochains, "%d,", &trip_courant)!=1)
259     return 1;
260   do{
261     trip_courant=read_trip(fprochains,
          espace_trip,trip_courant, id_to_index, arrets);
262   } while(trip_courant);
263
264   fclose(fprochains);
265   return 0;
266 }
267
268 /*--------------------------------------------------
269  Lecture du fichier transfers.txt (temps des correspondances)
270 --------------------------------------------------*/
271
272 char /*bool*/ read_transfers(FILE* f, Hashmap id_to_index,
     arret** arrets){
273   int stop_id,next_stop_id, transfer_time;
274   int index, next_index;
275
276   if(fscanf(f, "%d,%d,", &stop_id, &next_stop_id)!=2)
277     return 0;
```

```
278   if(!prochaine_virgule(f))
279     return 0;
280   if(fscanf(f, "%d", &transfer_time)!=1)
281     return 0;
282   index=get_hash(id_to_index,stop_id);
283   next_index=get_hash(id_to_index,next_stop_id);
284   if(index!=-1 && next_index!=-1)
285     add_prochain(arrets[index], next_index, transfer_time);
286 
287   return next_line(f);
288 }
289 
290 char /*bool*/ lecture_fichier_transfers(arret** arrets, int
      nb_arrets, Hashmap id_to_index){
291   FILE* ftransfers = NULL;
292   int transfer_courant;
293 
294   ftransfers = fopen("transfers.txt", "r");
295   if (ftransfers == NULL)
296     return 1;
297 
298   if(!next_line(ftransfers))
299     return 1;
300   do{
301     transfer_courant=read_transfers(ftransfers, id_to_index,
         arrets);
302   } while(transfer_courant);
303 
304   fclose(ftransfers);
305   return 0;
306 }
307 
308 /*-------------------------------------------------
309  Remplissage du graphe
310 -------------------------------------------------*/
311 
312 void remplit_graphe(grp g, arret** arrets, int nb_arrets){
313   int i;
314   list_prochain* l;
315 
316   for(i=0; i<nb_arrets; i++)
317     for(l=arrets[i]->prochains; l!=NULL; l=l->next)
318       ajouterAreteOrienteePonderee(g, i, l->index_prochain,
          l->duree);
```

```
319 }
```

6.3 Résultats

```
Essai bonus :
1-12506 : 2519(120)
2505 : 2506(120)
2400 : 2412(120)
2236 :
2373 : 2209(60)
1734 : 1790(60)
2546 : 2327(60)
1925 : 1639(120)
2232 : 2491(60)
2134 : 2373(60)
1860 : 2085(120)
1966 : 1805(60)
2139 : 2529(60)
2448 : 2093(60)
2486 : 2448(120)
2345 : 2384(120)
2347 : 2272(120)
2384 : 2435(60)
1676 : 1860(60)
2344 : 2221(60)
2055 : 1744(120)
2209 : 2546(120)
1774 : 1806(60)
1871 :
1639 : 1696(60)
2104 : 2199(120)
2199 : 2236(120)
1876 : 1808(120) 1925(120)
2435 : 2314(120)
1971 : 1847(120)
2301 : 2400(120)
2314 : 2226(60)
1854 : 1648(60)
1685 : 1876(60)
2422 : 2104(120)
2316 : 2134(60)
2085 : 1954(60)
1907 : 1649(60)
1648 : 1685(60)
2528 : 2321(120)
2250 : 2272(120)
1681 : 1907(120)
1744 : 1985(120)
1649 : 2020(60)
2529 : 2232(120)
1740 : 1734(60)
2015 : 1849(60)
2093 : 2175(60)
1944 : 2055(120)
2221 : 2345(60)
1696 : 1695(120)
2418 : 2347(60)
2327 : 2486(60)
1985 : 2080(60)
2412 : 2418(60)
1800 : 2015(60)
```

FIGURE 4 – Résultat console, bonus, première partie

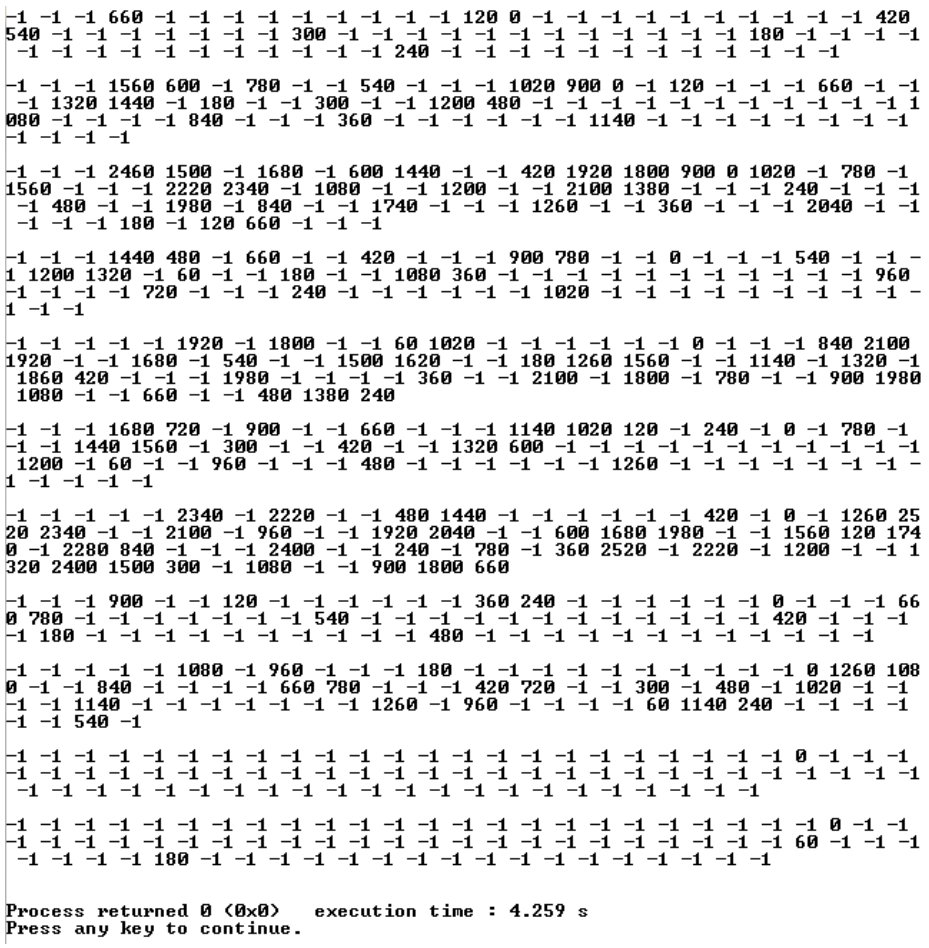

FIGURE 5 – Résultat console, bonus, deuxième partie

7 main

7.1 Source

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include "graphe.h"
4  #include "question2.h"
5  #include "question4.h"
6  #include "bonus.h"
7
8  int main(){
9      printf("Essai question 1 :\n");
10     essaiQuestion1();
11
12     printf("Essai question 2 :\n");
13     essaiQuestion2();
14
15     printf("Essai question 4 :\n");
16     essaiQuestion4();
17
18     printf("Essai bonus :\n");
19   Hashmap id_to_index=NULL;
20   arret** arrets=NULL;
21   list_prochain* a;
22   int nb_arrets=0, i, j;
23   int *t;
24   grp g;
25
26   id_to_index=init_hash_map();
27   if(id_to_index == NULL)
28     return 1;
29   nb_arrets=lecture_fichier_arrets(id_to_index);
30   if(nb_arrets==-1)
31     return 1;
32   arrets=creation_arrets(id_to_index, nb_arrets);
33   if(arrets==NULL)
34     return 1;
35   if(lecture_fichier_transfers(arrets, nb_arrets,
       id_to_index))
36     return 1;
37   if(lecture_fichier_prochain(arrets, nb_arrets,
       id_to_index))
38     return 1;
39   j=get_hash(id_to_index, 3765304);
```

```
40    free_hash_map(id_to_index);
41
42    g=nouveauGraphe(nb_arrets);
43
44    remplit_graphe(g, arrets, nb_arrets);
45
46    printf("%d", g->sommets[0]->degre_sortant);
47    printf("%d", j);
48    for(j=0;j<nb_arrets;j++) {
49      printf("%d : ", arrets[j]->id);
50      for(a=arrets[j]->prochains;a;a=a->next)
51        printf("%d(%d) ", arrets[a->index_prochain]->id,
              a->duree);
52      printf("\n");
53    }
54
55      /* Prendre la borne superieur inferieur a 76. */
56      int borne = 76;
57    for (j=0; j<borne; j++){
58          t=dijkstra(g, j);
59          for(i=0;i<76;i++)
60              printf("%d ", t[i]);
61          printf("\n\n");
62    }
63    free(t);
64    free_arrets(arrets, nb_arrets);
65    libererEspaceAlloueGraphe(g);
66    return 0;
67  }
```