Paulin AVRIL, Guillaume BAUDIN DE LA VALETTE, Yaniv BENICHOU, Joao Lucas SANTOS

Compte rendu du fil rouge AAP
Equipe Béhémet
13/01/20

Introduction

Ce travail constitue un travail de groupe constitué de quatre étudiants de 1ère année Guillaume BAUDIN DE LA VALETTE, Paulin AVRIL, Yaniv BENICHOU et Joao LUCAS , dans le cadre de l'implémentation de la méthode de Dijkstra sur des graphes orientés valués, à poids strictement positif et **sans circuit** strictement négatif.

Durant ce compte rendu, nous montrerons et expliquerons toutes les parties impliquées, en particulier l'algorithme de Dijkstra que nous avons effectué en complexité optimale, mais aussi des extensions réalisées tout en évoquant notre gestion du projet.

Attention : il est important d'exécuter les différents programmes depuis leurs répertoires (exemple : programme1.exe doit être exécuté depuis le répertoire /programme1.)

I] Gestion de projet

Très tôt dans le projet, nous avons décidé de collaborer sur Github, qui est une plateforme de versionning de projet, permettant d'avoir un historique de travail ce qui permet de garantir l'intégrité du projet, ainsi que plusieurs outils utiles comme la déclaration de nouvelles branches pour séparer le travail sans impacter celui des autres, y reporter des éventuels bugs ou suggestions d'améliorations.

Pour la répartition du travail, nous pouvons l'identifier comme telle :

- **Paulin :**
  - Implémentation de l'algorithme de Dijkstra avec sa file de priorité
  - Amélioration générale de la clarté et des bugs du Programme 1
- **Guillaume :**
  - Implémentation des exemples 3 et 4 du programme 2 :
    - Génération des fichiers .adj à partir des fichiers .la
    - Génération des fichiers .la à partir des fichiers .adj
- **Yaniv :**
  - Implémentation des exemples 1 et 2 du programme 2 :
    - Création des images PNG à partir des fichiers .adj
    - Création des images PNG à partir des fichiers .la
  - Génération d'une vidéo compilant toutes les images successives produites par le parcours de l'algorithme
  - Amélioration générale de la clarté et des bugs du Programme 1 et du Programme 2
  - Makefile
- **Joao :**
  - Fonctions de traitement de fichiers (lecture ,écriture, affichage)
  - Généralisation des tests :
    - Tests unitaires sur des graphes de taille et contenu aléatoire
    - Mesure expérimentale de la complexité de Dijkstra

II] Programme 1

Idée de l'algorithme, complexité et correction:

On considère le problème de la recherche de plus courts chemins (PCC) dans un graphe orientée valué G= (S,A,w) avec S l'ensemble des sommets,A l'ensemble des arcs et w:A->$\mathbb{R}$+ la fonction poids qui associe à tout arc dans A , une valeur réelle.
Soit s et u deux sommets de S, on note δ(s,u) la distance d'un PCC allant de s à u.
Dans ce cadre,G ne contient donc pas de circuit strictement négatif. C'est pourquoi , le problème de PCC à origine unique qui nous est posé peut être résolu par l'algorithme de DIjkstra.

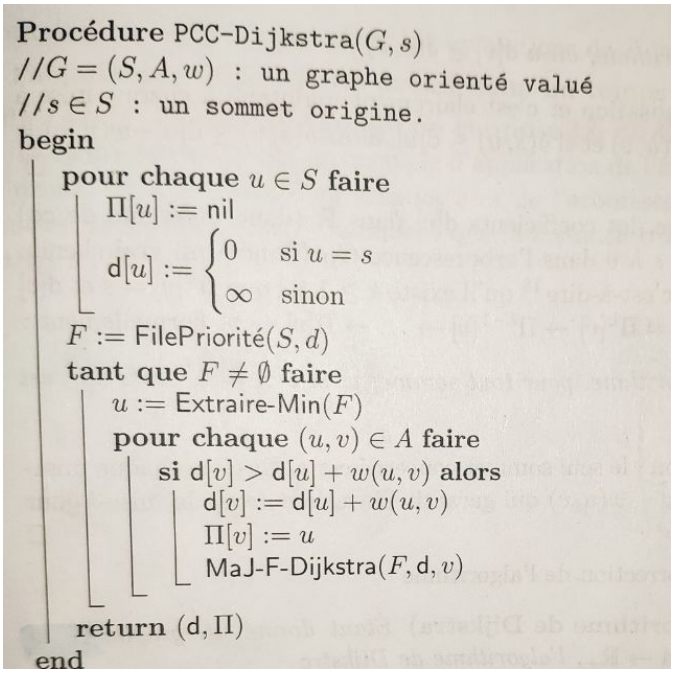
Nous avons voulu implémenté sa version optimale, soit celle avec une complexité temporelle en **O((|A|+|S|)*log(|S|)),**c'est à dire en **O(|A|*log(|S|))** lorsqu'on suppose |S|□ |A|
Pour cela, cet algorithme utilise une file de priorité étendue F avec une fonction de mise à jour de la file lorsque l'on diminue la priorité d'un élément. C'est un algorithme glouton (semblable à l'algorithme de Prim).
Pour implémenter une file de priorité étendue avec une fonction de mise à jour de la clé d'un élément , il suffit de compléter une file de priorité classique(implémentée avec un tableau F d'indice 1...n) avec un tableau IndiceDansF qui associe à chaque élément de F son indice dans F. Il faut bien sûr gérer ce tableau lors de la construction, ajout , et extraction d'un élément : toute réorganisation de F doit être répercutée dans le tableau IndiceDansF.

L'opération de MAJ de la file de priorité F est classique et prend un temps en O(log(|F|)).

```
Procédure MaJ-F-Dijkstra(F, d, v,)
begin
    //F[i] désigne le sommet situé à la position i dans F.
    i := IndiceDansF[v] //on récupère la position de v grâce au tableau.
    tant que (i/2 ≥ 1) ∧ (d[F[i/2]] > d[F[i]]) faire
        F[i] ↔ F[i/2]
        IndiceDansF[F[i]] := i
        IndiceDansF[F[i/2]] := i/2
        i := i/2;
end
```

La file de priorité contient les sommets de S pour lesquels on ne connaît pas encore la distance exacte d'un PCC depuis src, ils sont rangés dans F selon leur distance à s. A chaque fois qu'un sommet u est extrait, c'est que son coefficient d[u] correspond à δ(s,u), on met alors à jour la distance des successeurs de u et on modifie F en conséquence…

Ainsi,le principe de l'algorithme général repose sur la construction de cette file de priorité étendue F, puis on boucle sur F tant que la file n'est pas vide, on en extrait le minimum puis pour tout arc appartenant à A , on vérifie si la distance du voisin v est supérieure à celle de U + le poids entre u et v, condition essentielle pour savoir si l'on doit passer plutôt par v et ainsi mettre à jour nos tableau d de distance, Pi des prédécesseurs ainsi que la MAJ de F.



Procédure PCC-Dijkstra(G, s)
//$G = (S, A, w)$: un graphe orienté valué
//$s \in S$: un sommet origine.
begin
  pour chaque $u \in S$ **faire**
    $\Pi[u] :=$ nil
    $\mathsf{d}[u] := \begin{cases} 0 & \text{si } u = s \\ \infty & \text{sinon} \end{cases}$
  $F :=$ FilePriorité(S, d)
  tant que $F \neq \emptyset$ **faire**
    $u :=$ Extraire-Min(F)
    pour chaque $(u, v) \in A$ **faire**
      si $\mathsf{d}[v] > \mathsf{d}[u] + w(u, v)$ **alors**
        $\mathsf{d}[v] := \mathsf{d}[u] + w(u, v)$
        $\Pi[v] := u$
        MaJ-F-Dijkstra(F, d, v)
  return (d, Π)
end

Pseudo-code de l'algorithme de Dijkstra

Complexité:

Comme nous utilisons une structure de données classique , nous savons que certaines opérations ont un certain coût. Ainsi, la création de la file de priorité des éléments de S en fonction de la priorité distance d , a un coût en O(|S|) , l'extraction du minimum est en O (log [S|) ,

Ensuite, puisqu'on extrait à chaque tour de boucle un élément de F tant que F n'est pas vide, on effectue donc exactement |S| extractions.

De plus, puisqu'on vérifie chaque arc (u,v) dans A , on effectue donc |A| fois cette boucle.

On retient que pour chaque sommet u, on ne visite pas d'autres arêtes que deg(u).

On regarde chaque arc une fois et chaque sommet une fois , donc on a bien du |A|+|S|, (logique semblable à un parcours en profondeur).

En conclusion, on effectue |A|+|S| fois une opération de mise-à-jour de la file F qui est en O(log(|S|), on peut le montrer assez facilement (principe d'une dichotomie).

On a donc une complexité globale en **O((|A| + |S|)*log(|S|).**

Correction:

Tout d'abord, il est évident que l'algorithme termine puisqu'on effectue exactement |S| tours de boucle, correspondant à chaque extraction.

Ensuite, montrons qu'à la fin de l'algorithme, D[u] contient, pour tout sommet u, la plus courte distance entre src et u. Il suffit de montrer que D[u] contient la plus courte distance au moment de l'extraction de u. On le montre par récurrence sur l'itération i.

- i = 1 : le premier sommet à être extrait est src, et D[src] vaut 0.
- i + 1 : soit u le sommet extrait. On suppose par l'absurde que la propriété n'est pas vérifiée. Donc il existe un plus court chemin entre src et u. Soit C un de ces chemins. Soit y le premier sommet de C qui n'a pas encore été extrait de F. Soit x le

prédécesseur de y dans C : x a été extrait donc vérifie l'hypothèse de récurrence et D[x] contient la distance du plus court chemin entre src et x. Donc D[y] a été mis à jour lors de l'examen de l'arête (x,y). Donc D[y] = d(x) + g[x][y] = d(y) où, pour tout v, d(v) est la distance du plus court chemin entre src et v. Or d(y) <= d(u) car les poids sont positifs ou nuls, donc D[y] < D[v], et ce n'était pas possible avant d'extraire v à l'itération i+1 car on aurait d'abord extrait y.

Le fait que si D[u] < MAX_INT alors il existe un plus court chemin de src à u dont la dernière arête parcourue est (Pred[u],u) découle de la démonstration précédente.

L'objectif de ce code est de trouver le plus court chemin entre deux sommets donnés d'un graphe, à partir d'un fichier texte contenant une représentation de la matrice d'adjacence.

La première étape consiste à lire le fichier texte et de créer la matrice associée. On obtient ainsi une matrice d'adjacence sous l'implémentation définie dans le fichier */include/treatmentFiles.h.* La fonction readFile crée cette matrice. On a choisi une **implémentation dynamique** pour pouvoir modifier les tableaux en les faisant passer dans les fonctions via des pointeurs.

```
190     //Dans cette section, le code passe par tous les caractères du fichier,
191     //et à chaque numéro trouvé, il est placé dans la structure, chaque espacement
192     //avance d'un indice de la colonne, et à chaque changement de ligne, avance d'un indice de la ligne.
193     //Si elle trouve i, la matrice de la structure est remplie par la valeur maximale de l'entier
194     while (fscanf(fileMatrix, "%c", &val) != EOF) {
195       if (val == '\t') {
196         number[k] = '\0';
197         if (number[0] != 'i')
198           g->mat[i][j] = atoi(number);
199 
200         k = 0;
201         j++;
202       }
203       else if (val == '\n') {
204         number[k] = '\0';
205         if (number[0] != 'i')
206           g->mat[i][j] = atoi(number);
207 
208         k = 0;
209         j = 0;
210         i++;
211       }
212       else if ((val >= '0' && val <= '9') || (val >= 'a' && val <= 'z')) {
213         number[k] = val;
214         k++;
215       }
216     }
```

Dans la fonction readFile, la partie qui vous intéresse le plus est celle de l'image précédente. La boucle est basée sur la lecture caractère par caractère du fichier, et à chaque espacement ou ligne échangé, le nombre obtenu est placé dans la matrice de la structure g. De cette façon, il est possible de remplir toute la structure g.

```c
void dijkstra(T_graphMD *g, int *D, int *T, int *Pred, int sr, char *filename)
{
  int u;
  int v;
  int i = 1;
  int k = 0;
  int F[g->nbVertices];
  int indiceDansF[g->nbVertices];

  for (u = 0; u < g->nbVertices; u++)
  {
    Pred[u] = -1;
    if (u == sr)
    {
      D[u] = 0;
      F[0] = u;
      indiceDansF[u] = 0;
    }
    else
    {
      D[u] = INT_MAX;
      F[i] = u;
      indiceDansF[u] = i;
      i++;
    }
  }
```

```c
  while (k < g->nbVertices)
  {
    u = F[k];
    if (D[u] != INT_MAX)
    {
      T[u] = 1;
      createPNG(filename, g, u, T, k);
    }
    k++;
    if (D[u] != INT_MAX)
    {
      for (v = 0; v < g->nbVertices; v++)
      {
        if ((v != u) && (g->mat[u][v] != INT_MAX) && (D[v] > D[u] + g->mat[u][v]))
        {

          D[v] = D[u] + g->mat[u][v];
          Pred[v] = u;
          majDijkstra(F, D, v, k, indiceDansF);
        }
      }
    }
  }

  createPNG(filename, g, -1, T, k);
```

Après une initialisation des tableaux D et Pred, qui contiennent la distance par rapport à la source et le prédécesseur du sommet, la boucle principale va, pour chaque sommet, explorer les voisins pour voir si un chemin plus rapide que le précédent a été trouvé. Le sommet suivant à traiter est obtenu grâce à la file de priorité, dont le premier élément est l'élément plus proche de la source qui n'a pas encore été traité.

A chaque passage dans la boucle, la fonction createPNG permet de construire le fichier .dot ainsi que l'image PNG associée. Ces images permettent de représenter le

parcours du graphe au cours de l'algorithme. Pour connaître les sommets déjà explorés, on utilise le tableau T qui est tel que T[i] vaut 1 si le sommet i a été exploré et 0 sinon.

```
void majDijkstra(int *F, int *D, int v, int k, int *indiceDansF)
{
  int i = indiceDansF[v] - k;
  int c;
  while (((i / 2) >= 1) && (D[F[(i / 2) + k]] > D[F[i + k]]))
  {

    c = F[i + k];
    F[i + k] = F[(i / 2) + k];
    F[(i / 2) + k] = c;

    indiceDansF[F[i + k]] = i + k;
    indiceDansF[F[(i / 2) + k]] = (i / 2) + k;

    i = i / 2;
  }
  if (D[F[(i / 2) + k]] > D[F[i + k]])
  {

    c = F[i + k];
    F[i + k] = F[(i / 2) + k];
    F[(i / 2) + k] = c;

    indiceDansF[F[i + k]] = i + k;
    indiceDansF[F[(i / 2) + k]] = (i / 2) + k;
  }
}
```

La fonction majDijkstra permet d'assurer que la file de priorité soit à jour : après avoir récupéré l'indice du sommet dont la distance à la source a été modifiée, on change sa position en comparant sa distance à celle du sommet situé à l'indice i/2. Si la distance du second sommet est plus grande, on échange les deux sommets dans la file. On s'arrête lorsque l'échange ne se fait pas, ou lorsqu'on atteint le début de la file.

Une grosse difficulté rencontrée a été l'implémentation de la file de priorité. Dans un premier temps, il semblait naturel d'utiliser une pile, étant donné que l'objectif est d'avoir en permanence accès au premier élément, qui est le prochain à traiter. Néanmoins, le besoin de remettre à jour cette file nécessitait alors de dépiler jusqu'à trouver l'élément à replacer, puis rempiler en le replaçant à la bonne place. La complexité importante de cette opération faisait perdre le gain d'une file de priorité pour l'algorithme. Les tentatives avec un tableau simple n'étaient pas plus concluantes, puisque le nombre d'éléments est changeant et qu'il faut pouvoir supprimer les éléments déjà traités. Il a donc été décidé de coupler un tableau avec un indice donnant le nombre d'éléments déjà traités, cette information étant de toute façon nécessaire pour la création des fichiers PNG, et de décaler l'indice lors des appels du tableau grâce à ce nombre de sommets traités.

```c
//pour chaque ligne de la matrice adjacente
for (i = 0; i < g->nbVertices; i++) {

  //pour chaque colonne de la matrice adjacente
  for (j = 0; j < g->nbVertices; j++) {

    //est vérifiée s'il y a une valeur et si elle est différente de 0 ou infinie,
    //elle est donc placée dans le .dot
    if (g->mat[i][j] != INT_MAX && i != j) {
      sprintf(number, "\t%d -> %d [label = \"%d\"];\n", i, j, g->mat[i][j]);
      fputs(number, filePNG);
    }
  }
}
```

```c
//dans cette boucle, l'idée est de passer par chaque sommet est de mettre sa couleur
//cette couleur est d'abord définie par le sommet qui a été choisi par la fonction Dijkstra (vert),
//alors si le sommet a été choisi auparavant (rouge)
//sinon c'est bleu

for (i = 0; i < g->nbVertices; i++) {
  number[0] = '\0';
  sprintf(number, "%d", i);

  fputs(number, filePNG);

  if (i == node) {
    putColorNode(filePNG, 'g');
  }
  else if (T[i] == 1) {
    putColorNode(filePNG, 'r');
  }
  else{
    putColorNode(filePNG, 'b');
  }
}
```

Voici les principales parties de la fonction qui permettra de créer le fichier .dot à partir de la structure matricielle adjacente, des sommets déjà traités et du sommet actuel.

La première partie est basée uniquement sur les distances de chaque sommet par rapport à la matrice, en plaçant les indices pour chaque sommet, ainsi que les distances.

Dans l'ordre, l'étape suivante consiste à colorer les sommets à partir du vecteur T et du sommet actuel. Dans ce processus, elle est appelée une fonction qui va essentiellement écrire dans le fichier de paramètres, les lignes du fichier .dot se référant à la façon dont le sommet est peint, en fonction de l'autre paramètre, un caractère qui indique la couleur à peindre.

Une fois la matrice des distances D et des prédécesseurs Pred sont complétées, la fonction showResult permet d'afficher le chemin le plus court entre le départ et la destination, si un chemin existe.

Dans le dossier tests, le fichier main.c permet de tester la fonction créée sur un grand nombre de matrices de grande taille générées aléatoirement.

Exemple2

Cette partie n'est pas dissociable du premier exemple, mais plutôt complémentaire. En effet, en définissant la variable ***DOT_PATH*** ,nous allons générer une série d'images PNG représentant le parcours du graphe par l'algorithme avec un code couleur précis :
rouge -> Déja Visité ; Bleu -> Pas encore visité ; Vert -> Noeud courant;

Enfin, nous avons ajouté la possibilité ici, de passer ***-all*** en argument produisant l'affichage de tous les PCC du graphe à partir d'une origine unique. Les fichiers PNG ne sont pas affectés par ce changement, on produira une unique série d'images du parcours.

III] Programme 2

L'objectif de ce code est de produire la représentation graphique au format PNG à partir de fichiers texte contenant la représentation d'un graphe au format de matrice ou de listes d'adjacences. De plus, ce programme permet de passer d'une représentation de graphe vers l'autre et inversement.

Nous avons donc, en options en ligne de commande :
Entrées :
-a <fichier adj>
-l <fichier la>

Sorties :
-dot : sortie au format dot
-adj : sortie au format matrice d'adjacence
-la : sortie au format listes d'adjacences

Convention pour les listes d'adjacences :

Chaque ligne est composé de la sorte :
`"src  dest1_weight1 dest2_weight2 dest3_weight3 … "`

En effet, sur cette ligne on doit comprendre qu'il existe **un arc (src, dest1)** avec un poids **weight1** , puis un autre arc (src,dest2) avec un poids weight2 etc… jusqu'à la fin de la ligne où on passera à la liste d'adjacence d'un nouveau sommet, soit d'un nouveau nœud src. La liste d'adjacence d'un sommet src est rangée par **ordre croissant des sommets dest**. De même, L'ordre des listes de chaque sommet src est rangé par ordre croissant sur les sommets (on affiche la liste d'adjacence du sommet 0 puis celle de 1 etc ..).

```
1  0  1_12  5_15  6_20
2  1  2_21
3  2  4_3   7_19
4  3  2_7   7_7
5  4  3_13  7_14
6  5  2_17  4_28  6_4
7  6  4_18  7_45
```

exemple de la représentation d'un graphe par listes d'adjacences

Exemple 1 : Création des fichiers PNG à partir d'un fichier .adj

Pour réaliser cette fonction, nous avons procédé en 2 étapes, une première qui consiste à récupérer le T_graphLA correspondant au fichier .adj , puis une seconde qui à partir de ce T_graphLA va pouvoir écrire le fichier .dot au format souhaité afin d'obtenir , à l'aide d'une unique commande Graphviz l'image PNG de ce graphe. et correspond à la liste d'adjacence de src

Ainsi , la première étape est réalisé par :

```
T_graphLA *adjToGraph(const char *filename)
{
  FILE *fp;
  checkExtension("adj", filename);
  CHECK_IF(fp = fopen(filename, "r"), NULL, "L'argument saisi ne semble mené vers aucun fichier ...");

  int ligne = 0, column = 0, size = 0;
  const char *separators = "\t\n";
  char line[100];

  // Parcours le file pour connaitre nb_lignes = graph size
  FILE *fp_copy;
  CHECK_IF(fp_copy = fopen(filename, "r"), NULL, "L'argument saisi ne semble mené vers aucun fichier ...");

  while (fgets(line, 100, fp_copy) != NULL)
  {
    size++;
  }
  fclose(fp_copy);

  T_graphLA *g = newGraphLA(size);

  while (fgets(line, 100, fp) != NULL)
  {
    char *token = strtok(line, separators);
    while (token != NULL)
    {
      if (isNumber(token) == 1 && atoi(token) != 0)
      {
        addEdge(g, ligne, column, atoi(token));
      }
      token = strtok(NULL, separators);
      column++;
    }
    ligne++;
    column = 0;
  }
  fclose(fp);
  return g;
}
```

*fonction T_graphLA *adjToGraph(const char*);*

Après avoir vérifié que l'extension du fichier était bien *".adj"*, cette fonction a pour but tout d'abord d'initialiser un graphe à la bonne taille (nécessitant l'ouverture d'une copie du

fichier), puis parcours ligne par ligne le fichier en stockant la ligne courante dans une variable line, ensuite, on utilisera la fonction *strtok* pour récupérer les tokens à chaque délimiteurs, soit ici , “\t” ou “\n”.Enfin, après vérification que les token récupérés soient bien des entiers , on ajoute l’arc lu au graphe.

Ensuite la seconde étape de notre exemple est effectué par :

```c
void showGraphPNG(const char *filename, T_graphLA *g)
{
  char file_name[50];
  snprintf(file_name, sizeof(file_name), "%s.dot", filename);
  writeDotGraph(file_name, g);
  char command[200];

  mkdir("output/png", 0777);

  snprintf(command, sizeof(command), "dot output/dot/%s.dot -T png -o output/png/%s.png", filename, filename);
  system(command);
  printf("Création du fichier : output/png/%s.png\n", filename);
}
```

*fonction void showGraphPNG(const char *filename, T_graphLA *g);*

En réalité cette fonction ne fait qu'appeler une autre fonction ***writeDotGraph(const char*,T_graphLa*);*** après avoir généré dynamiquement le nom de fichier en .dot et avant d'exécuter une commande Graphviz permettant de générer le PNG à partir du fichier dot généré. Il serait donc plus intéressant de s’intéresser au code de cette fonction qui écrit le .dot.

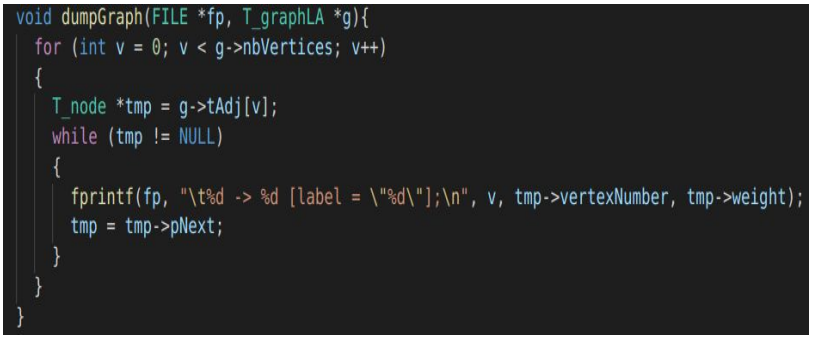
```c
void writeDotGraph(const char *filename, T_graphLA *g)
{
  // Ecrit le format dot complet (entête+structure) dans un fichier
  mkdir("output", 0777);
  mkdir("output/dot", 0777);

  char file_name[50];
  snprintf(file_name, sizeof(file_name), "output/dot/%s", filename);
  FILE *fp;
  CHECK_IF(fp = fopen(file_name, "w"), NULL, "fopen");

  fprintf(fp, "digraph graphe {\n");
  fprintf(fp, "rankdir = LR;\n");
  fprintf(fp, "node [fontname=\"Arial\", shape = circle, color=lightblue, style=filled];\n");
  fprintf(fp, "edge [color=red]\n");
  dumpGraph(fp, g);
  fprintf(fp, "}\n");
  fclose(fp);

  //Affichage sortie Standard
  char command[200];
  snprintf(command, sizeof(command), "cat %s", file_name);
  system(command);
}
```

fonction ***writeDotGraph(const char*,T_graphLa*);***

On y retrouve les procédés habituelles pour vérifier la bonne ouverture du fichier, la création des dossiers où seront stockés les fichiers générés ainsi que l’écriture progressive

de l'en tête du fichier .dot qui ne dépend pas du graphe, mais qui donne des informations sur l'affichage à produire (couleur etc ..) .

Pour l'écriture progressive dans le fichier des éléments du Graphe (Sommets et poids), on appelle une fonction auxiliaire, ***dumpGraph(const char*, T_GraphLA*);***, qui va parcourir notre graphe et écrire dans le fichier les valeurs dont on a besoin. C'est une fonction qui sert surtout à ne pas trop alourdir le code, et avoir quelque chose de modulaire.



```
void dumpGraph(FILE *fp, T_graphLA *g){
  for (int v = 0; v < g->nbVertices; v++)
  {
    T_node *tmp = g->tAdj[v];
    while (tmp != NULL)
    {
      fprintf(fp, "\t%d -> %d [label = \"%d\"];\n", v, tmp->vertexNumber, tmp->weight);
      tmp = tmp->pNext;
    }
  }
}
```

fonction dumpGraph(const char*,T_graphLa*);

Enfin, on exécute la commande ***cat*** sur notre fichier pour avoir sur la sortie standard le contenu de notre fichier .dot.
Tous les fichiers générés, .dot et .png se retrouveront dans le dossier output/dot (respectivement output/png).

Pour tester ce cas, il faut entrer par exemple la commande : "./programme2.exe -dot -a ./input/adj/graph1.adj " après avoir compilé à l'aide de la commande make dans programme2/

Exemple 2 : Création des fichiers PNG à partir d'un fichier .la

Ici, on nous demande de faire un travail qui est similaire à l'exemple précédent, soit de générer sur la sortie standard le contenu du fichier .dot (ce qui revient à la création de l'image PNG) du graphe représenté, non pas par une matrice, mais par des listes d'adjacences, des fichiers avec une extension *".la".*

Nous avons procédé de manière très similaire à l'exemple précédent, en deux étapes, où ici la première étape sera légèrement modifiée (***fonction laToGraph(const char*); au lieu de adjToGraph(const char*))*** puisque l'on devra récupérer les informations dans un fichier avec un format différent (voir convention choisie pour la représentation du graphe au format de listes d'adjacences). La 2e étape, qui consiste à écrire les fichiers .dot et .png à partir du T_graphLA , est exactement la même, puisque le T_graphLA généré ne dépend donc plus de la représentation choisie en fichier.

Pour la première étape, je dirais que la seule différence notable est l'utilisation de la fonction strtok_r qui est la version ré-entrante de strtok , et qui permet de stocker des valeurs récupérés par les délimitations dans des variables plutôt que de devoir le faire nous même avec des variables globales par exemple. Cela nous a été utile puisque avec la convention de format choisie, au sein d'une même ligne, et d'un même token récupéré soit de la forme : ***j_weight*** , il nous fallait une fois supplémentaire, séparé ce token par le délimiteur "_".

```
char *dest = strtok_r(token, "_", &token); //strtok_r est ré-entrant
```

utilisation de strtok_r dans la fonction laToGraph(const char*);

Pour tester ce cas , il faut entrer par exemple la commande : "./programme2.exe -dot -l ./input/la/graph1.la " après avoir compilé à l'aide de la commande make dans programme2/

Exemple 3 : Création d'un fichier .la à partir d'un fichier .adj

Le programme convertit une matrice d'adjacence en une liste d'adjacence avec la fonction:

```
void adjToLa(const char *fichierdepart);
```

Elle prend en argument le fichier contenant la matrice d'adjacence. Il vérifie que tout d'abord que le fichier sélectionné est bien un matrice d'adjacence, c'est-à-dire un fichier avec une extension *".adj".* Le programme vérifie aussi l'ouverture du fichier pour lire dedans, est faite sans problème avec :`CHECK_IF`.
Il va ensuite créer le fichier de la future liste d'adjacence, c'est-à-dire un fichier avec une extension *".la"* et vérifier qu'il est bien ouvert afin de pouvoir écrire dedans.

Puis le programme parcourt les lignes de la matrice en stockant la ligne courante dans une variable line.

Pour chaque ligne, le programme découpe la ligne en morceaux grâce à la fonction *strtok* pour récupérer les tokens à chaque délimiteur: le retour chariot \n" et la tabulation "\t". Il va alors recueillir tant que le token existe, le poids entre le sommet i correspondant à ligne de la matrice et le sommet j correspondant au numéro de la colonne de la matrice.
Ensuite il inscrit dans le fichier à la ligne i:" j_poids" , selon la convention choisie.Une fois toutes les lignes parcourues, les fichiers sont fermés.

Pour tester ce cas, il faut entrer par exemple la commande : "./programme2.exe -la -a ./input/adj/graph1.adj " après avoir compilé à l'aide de la commande make dans programme2/

```c
void adjToLa(const char *fichierdepart)
{
  mkdir("output", 0777);  //creation des répertoires
  mkdir("output/la/", 0777);

  //ouverture des fichiers
  FILE *fp, *dep;
  checkExtension("adj", fichierdepart);
  CHECK_IF(dep = fopen(fichierdepart, "r"), NULL, "L'argument saisi ne semble mené vers aucun fichier ...");

  char cp[50], filename[50];
  strcpy(cp, fichierdepart);
  snprintf(filename, sizeof(filename), "output/la/%s.la", strtok(cp + 10, "."));
  CHECK_IF(fp = fopen(filename, "w"), NULL, "L'argument saisi ne semble pas bien être formulé ...");
  printf("création du fichier : %s \n", filename);

  int ligne = 0, column = 0;
  const char *separators = "\t\n_";
  char line[100];

  while (fgets(line, 100, dep) != NULL)//parcourt les lignes de la matrice
  {
    fprintf(fp, "%d\t", ligne); //écrit le numéro de la ligne
    char *token = strtok(line, separators); //recupère 1er elements de la logne
    while (token != NULL)
    {
      if (isNumber(token) == 1 && atoi(token) != 0 && token != 0)//si c'est le poids d'un arc
      {
        fprintf(fp, "%d_%s\t", column, token);//écrit avec le format choisi
      }
      token = strtok(NULL, separators);//prend le 2eme éléments
      column++;
    }
    fprintf(fp, "\n");
    ligne++;
    column = 0;
  }
  fclose(fp);
  fclose(dep);
}
```

fonction void adjToLa(const char *fichierdepart);

Exemple 4 : Création d'un fichier .adj à partir d'un fichier .la

Le programme convertit une liste d'adjacence en une matrice d'adjacence avec la fonction:

`void laToAdj(const char *fichierdepart)`

Elle prend en argument le fichier contenant la liste d'adjacence. Il vérifie que tout d'abord que le fichier sélectionné est bien une liste d'adjacence, c'est-à-dire un fichier avec une extension *".la".* Le programme vérifie aussi l'ouverture du fichier pour lire dedans, est faite sans problème avec :`CHECK_IF`.

Il va ensuite créer le fichier de la future matrice d'adjacence, c'est-à-dire un fichier avec une extension *“.adj”* et vérifier qu’il est bien ouvert afin de pouvoir écrire dedans.

Puis le programme parcourt les lignes de la liste pour déterminer le nombre de sommets “size” et donc la dimension de la matrice.

Pour chaque ligne, le programme découpe la ligne en token contenus entre les séparateurs qui sont le retour chariot “\t”, le tiret “ _ “ et la tabulation “\n”.

Le programme parcourt de nouveau les lignes de la liste comme dans l’exemple précédent. Il récupère le numéro du 1er sommet j.

Il ajoute:

- soit le poids entre le sommet de la ligne i et j,
- soit “0” si on est sur la diagonale
- soit “i” si le chemin n’existe pas.

Les numéros étant par hypothèse triés dans l'ordre croissant sur chaque ligne, le programme va créer la matrice d'adjacence lier à la liste d'adjacence.

Une fois toutes les lignes parcourues, les fichiers sont fermés.

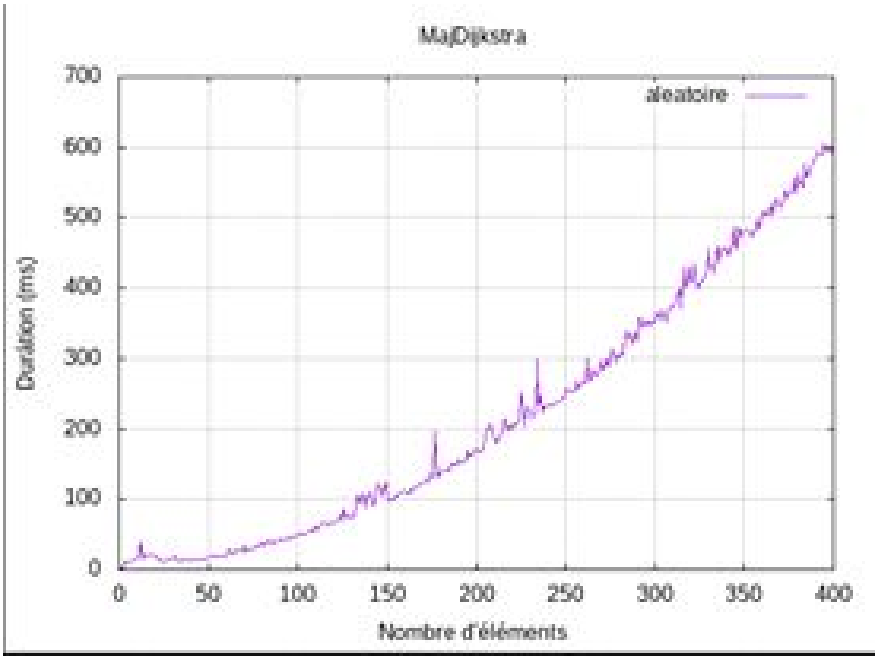
Pour effectuer cet exemple, il faut entrer la commande : “./programme2.exe -adj -l ./input/la/graph1.la “

```c
void laToAdj(const char *fichierdepart)
{
  mkdir("output", 0777);   //creation des répertoires
  mkdir("output/adj/", 0777);

   //ouverture des fichiers
  FILE *fp, *dep;
  checkExtension("la", fichierdepart);
  CHECK_IF(dep = fopen(fichierdepart, "r"), NULL, "L'argument saisi ne semble mené vers aucun fichier ...");

  char cp[50], filename[50];
  strcpy(cp, fichierdepart);
  snprintf(filename, sizeof(filename), "output/adj/%s.adj", strtok(cp + 9, "."));
  CHECK_IF(fp = fopen(filename, "w"), NULL, "L'argument saisi ne semble pas bien être formulé ...");
  printf("création du fichier : %s \n", filename);

  int ligne = 0, column = 0;
  int size = 0;
  const char *separators = "\t\n_";//définition des séparateurs
  char line[100];

  FILE *fp_copy;
  CHECK_IF(fp_copy = fopen(fichierdepart, "r"), NULL, "fopen");
  while (fgets(line, 100, fp_copy) != NULL)//taille de la matrice
  {
    size++;
  }
  fclose(fp_copy);

  while (fgets(line, 100, dep) != NULL)//parcourt des lignes
  {
    char *token = strtok(line, separators);//récupère le 1ER élément qui est n° de la ligne
    token = strtok(NULL, separators);// récupère le 2eme élément qui indique la présence d'un arc

    for (column = 0; column < size; column++) //création des colonnes
    {
      if (token != NULL && atoi(token) == column)//si un arc existe
      {
        token = strtok(NULL, separators); //prendre le poids associé au sommet
        fprintf(fp, "%d\t", atoi(token));
        token = strtok(NULL, separators);
      }

      else if (column == ligne)//si on est en diagonale
      {
        fprintf(fp, "0\t");
      }
      else//si aucun arc existe
      {
        fprintf(fp, "i\t");
      }
    }
    fprintf(fp, "\n");
    ligne++;
  }
  fclose(fp);
  fclose(dep);
```

fonction void laToAdj(const char *fichierdepart)

IV) Extensions

Nous avons rajouté 3 éléments par rapport au travail demandé : une fonction permettant de faire des tests sur des matrices aléatoires de taille variable, la création du vidéo présentant le parcours du graphe dans l'algorithme de Dijkstra et une possibilité dans le programme1 : -all. Ce programme, situé dans le dossier tests, nous permet de mettre en évidence la complexité quasi-linéaire de notre algorithme. Pour cela, on commence par générer des matrices aléatoirement, puis on effectue Dijkstra entre tous les sommets en mesurant le temps de calcul. Cela nous a également permis de vérifier que toutes nos fonctions fonctionnaient correctement.



Le deuxième ajout consiste à créer une vidéo à partir des images générées dans l'algorithme de Dijkstra. Cette vidéo se construit simplement en concaténant les images grâce au package ffmpeg, via une commande shell.

Enfin, il est possible d'utiliser ***-all*** à la place de la destination dans la recherche du plus court chemin pour avoir l'intégralité des plus courts chemins. ./programme1.exe "adresse du graphe" k -all avec k un sommet du graphe renvoie donc tous les chemins entre k et i pour i prenant comme valeurs tous les sommets du graphe (à condition que les chemins existent).

Conclusion

Ce travail permet donc d'une part de résoudre le problème du plus court chemin de manière optimale via l'implémentation avec file de priorité étendue, et d'autre part de traiter différents types de fichiers représentants un graphe en entrée pour les convertir en un autre format de représentation. Nous avons également ajouté au travail des tests à grande échelle ainsi qu'une concaténation des vidéo pour obtenir une meilleure vision du déroulé de l'algorithme.

Notre ressenti sur le travail accompli est très positif, nous savons que nous avons beaucoup travaillé sur ce fil rouge mais nous sommes fiers des résultats obtenus. Nous avons aussi beaucoup appris les uns des autres et aussi chacun individuellement sur les parties traitées.