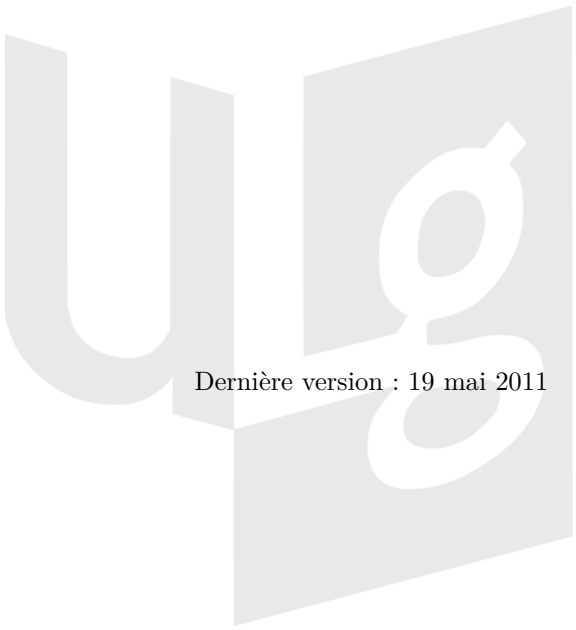
Rapport Projet N°3

Rapport portant sur le cours *Technique de Programmation*

Document rédigé par **Julien Mahin & David Taralla**
2e Bachelier en Sciences Informatiques



Dernière version : 19 mai 2011

Table des matières

1 Informations préliminaires

1.1 Modification de graphe.c

Il s'est avéré que le fichier `graphes.c` contenait une erreur. Cette erreur était présente dans la fonction `ajouterArc`. En effet, lorsque l'arc à ajouter était déjà présent dans le graphe, la fonction mettait à jour le poids de cet arc et incrémentait le nombre d'arcs du graphe, alors qu'aucun nouvel arc n'avait été ajouté.

Pour remédier à ce problème, nous avons simplement décrémenté le nombre d'arcs du graphe dans le cas où l'arc est déjà présent. L'incrémentation a toujours lieu, ce qui nous donne un résultat nul, qui ne modifie au final pas le nombre d'arcs du graphe.

Nous aurions pu opter pour des façons de faire plus élégantes, mais nous voulions modifier le moins possible le fichier `graphes.c` qui nous avait été fourni.

1.2 Graphes orientés/non-orientés

Nous travaillons dans ce projet avec des graphes non-orientés, mais de par la contrainte de l'implémentation des graphes qui nous était fournie, nous avons utilisé la représentation suivante : lorsqu'un sommet a et un sommet b sont liés par un arc, on retrouve dans la liste liée un arc de a vers b ainsi qu'un arc de b vers a.

Donc, par exemple, pour avoir un graphe de n sommets connexe, il faut au minimum $2(n-1)$ arcs vu que chaque arc est représenté dans les deux sens.

2 Fonction de génération de graphes aléatoires

2.1 Code source

```
int genererGraphe(graphe *g, unsigned int nbSommets,
                  unsigned int nbArcs, int connexe) {
  if (nbSommets == 0 && nbArcs > 0)
    return -1;

  if (nbArcs > nbSommets * (nbSommets - 1))
    return - 1;

  if (connexe != 0 && nbArcs  < 2 * (nbSommets - 1))
    return - 1;

  // Ajout des sommets
  for (unsigned int i = 0; i < nbSommets; i++)
    ajouterSommet(g,0);

  srand(time(0));

  if (connexe != 0) { // Le graphe doit etre connexe
    int nbConnectes = 1;
    int nbNonConnectes = nbSommets - 1;
    int connectes[nbSommets];
    int nonConnectes[nbSommets];
    for (unsigned int i = 0; i < nbSommets - 1; i++)
      nonConnectes[i] = i + 2;
    connectes[0] = 1;

    // D'abord, on fait en fait en sorte que le graphe soit connexe
    while(nbNonConnectes > 0) {
       // Un noeud parmi les connectés
      int indiceA = rand() % nbConnectes;
      int a = connectes[indiceA];

      // Un noeud parmi les non-connectés
      int indiceB = rand() % nbNonConnectes;
      int b = nonConnectes[indiceB];

      int poids = rand() % 100 + 1;
      ajouterArc(g, a, b, poids);
      ajouterArc(g, b, a, poids);

      connectes[nbConnectes] = b;
      nbConnectes++;
      nonConnectes[indiceB] = nonConnectes[nbNonConnectes - 1];
      nbNonConnectes--;
    }
  }
```

```
    // On complète le graphe en ajoutant des arcs aléatoires
    while ((unsigned int) (g->nbA) < nbArcs) {
      int a = rand() % (nbSommets) + 1;
      int b = rand() % (nbSommets) + 1;
      if (a != b) {
        int poids = rand() % (100) + 1; // poids des arcs entre 1 et 100
        ajouterArc(g, a, b, poids);
        ajouterArc(g, b, a, poids);
      }
    }
    return 0;
  }
```

2.2 Fonctionnement

La première partie de cette fonction consiste à vérifier les arguments :
- Un graphe de 0 sommets ne peut pas avoir d'arcs.
- Un graphe de n sommets ne peut pas contenir plus de $n(n-1)$ arcs.
- Un graphe connexe de n sommets doit contenir au minimum $2(n-1)$ arcs.

Après ces vérifications, nous ajoutons tous les sommets au graphe. Nous pouvons ensuite passer à l'ajout des arcs.

Dans le cas où le graphe doit être connexe, nous commençons par relier tous les sommets afin d'avoir un arbre. Pour cela, nous utilisons deux tableaux. Le premier contient les sommets qui sont déjà connectés, et le second ceux qui ne le sont pas encore. Ainsi, tant qu'il y a des sommets non connectés, on sélectionne au hasard un sommet dans chacun des tableaux et on les relie par un arc. Au final, tous les sommets seront reliés en une seule composante connexe.

L'étape suivante, la dernière, consiste à compléter le graphe en ajoutant des arcs aléatoires afin d'en obtenir le nombre voulu. Cette étape est identique dans le cas où le graphe est connexe et où il ne l'est pas. En effet, dans le premier cas de figure, on aura déjà rempli le graphe de n sommets avec $2(n-1)$ arcs, et si le graphe n'est pas connexe, il contiendra bien tous les sommets mais 0 arcs. Nous pouvons donc les remplir de la même façon.

Ainsi, nous sélectionnons aléatoirement deux sommets a et b du graphe et nous créons les arcs de a vers b et de b vers a, ayant pour poids un nombre aléatoire choisi entre 1 et 100 (le même pour les deux sens).

2.3 Améliorations possibles

Lors de la création aléatoire des arcs, nous choisissons aléatoirement deux sommets et tentons de les relier. Cela n'est pas optimal car nous pourrions tomber deux fois sur le même sommet, et nous ne pourrions alors pas créer l'arc. Nous pourrions aussi vouloir créer un arc déjà présent dans le graphe. Tout cela nous conduirait à faire beaucoup plus d'itérations que ce qui est réellement nécessaire.

Un autre point posant problème au niveau de l'optimalité vient directement de l'implémentation de `graphes.c`, qui nous contraint à ajouter chaque arc en double (un dans chaque sens).

3 Fonction de détection de cycle dans un graphe

3.1 Code source

```
sommet* getSomByLabel(graphe* g, int label) {
  sommet* currS = g->premierSommet;
  while (currS != NULL) {
    if (currS->label == label)
      return currS;
    else
      currS = currS->suivant;
  }

  return NULL;
}

void dfsCycle(graphe* g, sommet* s, sommet* previous, bool* marked,
              int labelToFind, bool* result) {
  marked[s->label] = true;
  eltadj* currA = s->adj;

  // Si on a trouvé un cycle, plus besoin de continuer le parcours !
  while (currA != NULL && !(&result)) {
    sommet* destS = getSomByLabel(g, currA->dest);

    if (!marked[destS->label])
      dfsCycle(g, destS, s, marked, labelToFind, result);
    else if (labelToFind == destS->label && previous != destS)
      *result = true;

    currA = currA->suivant;
  }
}

int estAcyclique(graphe *g) {
  if (g == NULL || g->nbS < 3) return 0;

  sommet* currS = g->premierSommet;
  while (currS != NULL) {
    bool* marked = (bool*)calloc(g->maxS + 1, sizeof(bool));
    bool result = false;
    dfsCycle(g, currS, NULL, marked, currS->label, &result);
    free(marked);
    if (result)
      return 1;

    currS = currS->suivant;
  }

  return 0;
}
```

3.2 Fonctionnement

L'idée principale est la suivante.

On explore tous les chemins possibles partant d'un sommet s en utilisant l'algorithme *DFS*. Si pendant qu'on réalise ceci, il arrive un moment où l'algorithme regarde un sommet qui est déjà marqué **mais** dont le label est celui du nœud de départ sans que celui-ci soit le nœud précédent du chemin en cours, cela veut dire qu'on a trouvé un cycle !

On applique ceci à chacun des sommets du graphe tant qu'on n'a pas trouvé de cycle, et on s'arrête dès qu'on en a trouvé un.

Tout ceci est orchestré en deux fonctions : `estAcyclique` s'occupe de lancer la recherche de cycle sur chaque sommet, tandis que `dfsCycle` se charge d'explorer tous les chemins possibles partant d'un seul sommet.

La fonction `getSomByLabel`, comme son nom l'indique, permet de récupérer un sommet dans le graphe à partir de son label unique.

dfsCycle Cette fonction implémente en fait une version modifiée de l'algorithme *DFS*. On a simplement ajouté un moyen de détecter les cycles lors de l'exploration des chemins. Finalement, vu que ce qui nous intéresse, c'est la détection de cycle et non l'exploration, une fois que l'on en a trouvé un, on arrête l'algorithme prématurément (au moyen du gardien modifié du `while`).

estAcyclique Cette fonction itère sur chacun des sommets pour voir si l'un d'eux fait partie d'un cycle, en travaillant conjointement avec `dfsCycle`.

3.3 Améliorations possibles

Le principal problème ici, c'est la contrainte de l'implémentation de l'interface fournie. Il serait intéressant de rajouter un champ `visited` à chaque nœud, afin d'optimiser l'espace pris par le programme dans la mémoire de l'ordinateur et le temps d'exécution. En effet, à chaque itération de la fonction `estAcyclique`, on alloue beaucoup trop d'espace comparé au nombre réel de booléens à retenir, et ici c'est le seul moyen pour retenir l'état visité ou non de chaque sommet pendant un parcours. Aussi, la fonction `getSomByLabel` n'est pas optimale (recherche exhaustive) mais est nécessaire, vu qu'un arc ne pointe pas vers l'adresse du sommet destination mais contient le *label* de sa destination... Donc lorsqu'on désire récupérer le sommet destination d'un arc, on est forcé de passer par elle.

De plus, cet algorithme doit être testé sur chaque sommet avant d'en conclure qu'un graphe est acyclique, alors qu'il devrait y avoir un moyen pour détecter un cycle qui ne contienne pas le sommet source.

Enfin, on utilise aucune propriété mathématique dans cet algorithme, alors qu'en retenir certaines lors de la construction du graphe nous permettrait d'utiliser des conditions suffisantes pour caractériser la présence de cycles dans le graphe.

4 Implémentation de l'algorithme de Kruskal

4.1 Code source

```
typedef struct arc_t {
  int from;
  eltadj* to;
} arc;


// Fonction de comparaison utilisée pour trier les arcs
int comp(const void** a,const void** b) {
  arc const *pa = *a;
  arc const *pb = *b;
  if (pa->to->poids == pb->to->poids)
    return 0;
  if (pa->to->poids < pb->to->poids)
    return -1;
  return 1;
}


void algoKruskal(graphe *g, graphe *h) {
  // Remplir un tableau avec les arcs
  arc* arcsTab[g->nbA];
  int i = 0;
  sommet* psommet = g->premierSommet;
  eltadj* padj;
  do {
    if (psommet->adj != NULL) {
      padj = psommet->adj;
      do
      {
        arc* parc = (arc*) malloc(sizeof(arc));
        parc->from = psommet->label;
        parc->to = padj;
        arcsTab[i] = parc;
        i++;
        padj = padj->suivant;
      }
      while (padj != NULL);
    }
    psommet = psommet->suivant;
  } while (psommet != NULL);

  /* Tri des arcs par poids croissants */
  qsort(arcsTab, sizeof(arcsTab) / sizeof(*arcsTab), sizeof(*arcsTab),
        comp);
  printf("\n");
```

```
  // Ajout de tous les sommets à l'arbre
  for (int i = 0; i < g->nbS; i++)
    ajouterSommet(h, 0);

  // Ajout des arcs
  for (int i = 0; i < g->nbA; i++) {
    ajouterArc(h, arcsTab[i]->from, arcsTab[i]->to->dest,
               arcsTab[i]->to->poids);

    // L'arbre ajouté crée-t-il un cycle ? Si oui, on le supprime
    if (estAcyclique(h) != 0)
      supprimerArc(h, arcsTab[i]->from, arcsTab[i]->to->dest);
  }

  // Libération de la mémoire
  for (int i = 0; i < g->nbA; i++)
    free(arcsTab[i]);
}
```

4.2 Fonctionnement

Pour pouvoir mettre en place l'algorithme de Kruskal, nous avons du préalablement créer une nouvelle structure `arc` qui permet de retenir un lien vers un élément adjacent ainsi que le sommet de départ. Nous avons aussi implémenté une fonction de comparaison `comp` qui permet de comparer les poids de deux arcs. Celle-ci sera utilisée pour trier les arcs par ordre de poids croissants.

Une fois ces préparatifs implémentés, l'algorithme de Kruskal est très simple à programmer. Nous commençons par récupérer tous les arcs du graphe en le parcourant et nous les conservons dans un tableau. Nous trions ensuite ces arcs par ordre de poids croissants au moyen de `qsort` et de notre fonction de comparaison. Une fois le tri effectué, nous pouvons nous attaquer à la construction de l'arbre couvrant de poids minimum.

Par définition, cet arbre contient tous les sommets du graphe. C'est pourquoi nous commençons par ajouter chaque sommet au nouveau graphe. Ensuite, nous parcourons le tableau d'arcs et nous ajoutons un à un les arcs qu'il contient, dans l'ordre. Lorsque l'ajout d'un arc crée un cycle, alors il est retiré avant d'ajouter le suivant.

Et c'est ainsi qu'avec l'algorithme de Kruskal nous obtenons l'arbre couvrant de poids minimum.

4.3 Améliorations possibles

Afin d'éviter d'utiliser de l'espace mémoire supplémentaire, nous pourrions faire en sorte de ne pas avoir recours à l'utilisation d'une nouvelle structure pour retenir les arcs, même si cela permet de simplifier le problème.

Ici encore, l'implémentation de `graphes.c` fait que dans notre arbre couvrant minimum, nous avons chaque arc en double (un dans chaque sens). Au final, le meilleur moyen d'améliorer le tout serait de réimplémenter les graphes de façon à ce qu'ils conviennent parfaitement aux tâches qu'ils devront être capables d'implémenter.