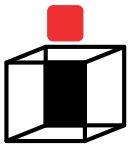

Algorithmique et structures de données 2

Sylvia Chalençon

Janvier 2022

Table des matières

Chapitre 1

Introduction

Ce document est le support du cours « Algorithmique et structures de données 2 », enseigné au second semestre de deuxième année de la licence Informatique & Vidéoludisme. Il fait suite au cours « Algorithmique et structures de données 1 », enseigné au semestre précédent. Dans ce cours, nous allons coder en *C++*, apprendre à manipuler les différentes structures de données offertes par la STL (*Standard Template Library*) et comprendre les algorithmes qu'elle fournit. Le but est de consolider les acquis du cours du semestre précédent, choisir effcicacement des structures de données en fonction de la problématique abordée et appréhender le coût d'un algorithme.

Je ne ferai pas une présentation exhaustive de la syntaxe et des concepts du langage car ceux-ci sont identiques au *C* dont la maîtrise est supposée à ce stade de la formation. J'insisterai au contraire sur les différences avec le *C*. De même, je ne m'attarderai pas sur les notions de programmation objet, vous avez un cours qui développe ces aspects.

En 1979, alors qu'il travaille dans les laboratoires *Bell* avec un certain Dennis Ritchie, inventeur du *C*, Bjarne Stroustrup commence le développement de *C with classes*. Son but est d'améliorer le langage C sans alourdir ses performances, en lui ajoutant des fonctionnalités comme la notion de classes (et de classes dérivées), d'encapsulation de données, les arguments par défaut ou l'extension inline. En 1983, nait le nom de *C++*, en référence à l'opérateur d'incrémentation ++ du *C*, et en 1984, *C with classes* devient le *C++*, présenté donc comme une évolution du *C*.

Par la suite, le langage évolue et permet l'ajout de nombreuses fonctionnalités (références, héritage multiple, classes abstraites, templates, espaces de noms, typage dynamique ...), cela aboutit en 1998 à la standardisation du *C++* (ISO/CEI 14882 :1998) par un comité réunissant l'ANSI et l'ISO. Parallèllement à l'évolution du langage, la bibliothèque standard du C++ fournissant classes et fonctions pour le *C++* s'étoffe. Elle englobe no-

tamment la *Standard Template Library* (STL) qui fournit un ensemble de conteneurs (vecteurs, listes, tableaux associatifs ...), ainsi qu'une abstarction des pointeurs (itérateurs) et des algorithmes génériques pour manipuler ces conteneurs.

Il n'y a pas de documentation officielle, mais https://en.cppreference.com devrait faire partie de vos favoris.

Chapitre 2

Du *C* au *C++*

2.1 Pour commencer

Le premier avantage du *C++* est sa compatibilité avec le *C*. Les deux principaux compilateurs *C++* sont *g++* et *clang++*. Il faut savoir que le *C++* applique des règles de typage et des exigences d'initialisation plus strictes que le *C*.

Une autre différence est ce qu'on appelle la déclaration des variables au plus près en *C++*. Contrairement au *C* qui ne permet la déclaration de variable uniquement en début de bloc, le *C++* autorise les déclarations partout. Ceci permet de repousser la déclaration jusqu'au moment où nous avons besoin d'utiliser la variable et suffisamment d'informations pour l'initialiser.

En *C++*, on évite au maximum les macro-définitions. C'est ainsi que pour déclarer une constante, alors qu'en *C* nous utilisions *#define PI 3.14* par exemple, la syntaxe *C++* serait *const double PI= 3.14*. Dans cette idée, le *C++* a introduit les fonctions *inline*. Au moment où une telle fonction apparait dans le code, le compilateur remplace directement l'appel par le code de la fonction. Le but d'une telle fonction est évident : l'efficacité. Ceci est valable pour une fonction appelée souvent, dont le code est court et dont l'éxecution est rapide. Le service rendu est similaire à celui rendu par les macros en *C* mais plus fiable car le compilateur peut y effectuer des vérifications. Pour déclarer une fonction en ligne, il faut utiliser le mot-clé *inline*.

```
1 inline double perimetre_cercle (double r){
2   return 2*PI*r;
3 }
```

Listing 2.1 – Fonction inline

Un autre apport du *C++* réside dans la possibilité de donner des valeurs par défaut à un ou plusieurs arguments d'une fonction lors de sa déclaration. Les arguments qui ont une valeur par défaut doivent être les derniers dans la liste d'arguments de la fonction. Lors de l'appel de cette fonction, il est alors possible d'omettre les paramètres qui ont une valeur par défaut définie.

```
int somme (int a=0, int b=0){
  return a+b;
}
int main (void){
  int s=somme (2, 3);//s==5
  s=somme (2);//s==2
  s=somme ();//s=0
}
```

Listing 2.2 – Valeurs par défaut

De plus, en *C++*, le nombre et le type des arguments d'une fonction font partie de sa signature (en plus de son nom), ce qui permet d'avoir plusieurs fonctions qui portent le même nom tant que leur signature est différente. C'est ce qu'on appelle la « surcharge ».

```
int puissance (int x=0, int n=0){
  //x*x*x*...*x, n fois
}
double puissance (double a, double b){
  //Pas la même histoire...
}
```

Listing 2.3 – Surcharge de fonction

2.2 Les entrées-sorties

Nous venons de voir qu'il est possible de surcharger des fonctions en *C++*, et il en existe des plus particulières qu'il est également possible de surcharger : les opérateurs. Cela rend la gestion des entrées-sorties beaucoup plus simple qu'en *C*. Pour utiliser les flux d'entrées-sorties, il faut inclure le fichier *iostream.h*. Trois variables représentent le flux d'entrée standard (*cin*) normalement associé au clavier, le flux de sortie standard (*cout*) et le flux de sortie d'erreur (*cerr*) tous deux associés à l'écran. Les opérateeurs d'injection (<<) et d'extraction (>>) permettent d'écrire et de lire sur ces flux.

```
#include<iostream>
int somme (int a=0, int b=0){
  return a+b;
}
int main (void){
  int i,j;
  std::cin>>i>>j;
  int s=somme (i, j);
  std::cout<<"s == "<<s<<std::endl;
}
```

Listing 2.4 – Entrées-sorties

Ces flux sont définis dans l'espace de noms *std*. Pour ne pas avoir à répéter *std* :: à chaque utilisation, il faut déclarer son utilisation auparavant :

```
#include<iostream>
using namespace std;
int somme (int a=0, int b=0){
  return a+b;
}
int main (void){
  int i,j;
  cin>>i>>j;
  int s=somme (i, j);
  cout<<"s == "<<s<<endl;
}
```

Listing 2.5 – Espace de nommage

2.3 Allocation dynamique

Comme en *C*, les fonctions *malloc* et *free* sont disponibles mais il est préférable d'utiliser les opérateurs *new* et *delete* car ils font appel aux constructeur et destructeur des objets manipulés.

```
int *tab=new tab[5];//Alloue un tableau de 5 entiers
...
delete []tab; //restitution de la mémoire
```

Listing 2.6 – Allocation dynamique

L'opérateur *new* renvoie une variable de type *pointeur*. Il est possible que

la dimension d'un tableau ne soit pas connue au moment dela compilation, donc donnée par une variable mais dans ce cas la taille de composante doit être connue. Cela signifie que dans le cas d'un tableau à plusieurs dimension, seule la première dimension peut-être définie par une variable.

```
1 int  n;
2 int (*tab)[5];
3 ...//Calcul de la valeur de n
4 tab=new int[n][5];//Alloue n tableaux de 5 entiers
5 ...
6 //libération de la mémoire
7 delete[] tab;
```

Listing 2.7 – Allocation dynamique - suite

2.4 Ajout de types

En plus des types défini par le programmeur (les classes), le *C++* a introduit de nouveaux types.

2.4.1 Les booléens

En *C++*, le type *bool* permet de déclarer une variable de type booléen qui ne pourra donc prendre que deux valeurs : *true* ou $false$. Les opérandes des opérateurs logiques doivent donc être des booléens et c'est le type de leur valeur de retour. Les conditions sont également censées être de type booléen. Il est donc fortement conseillé de ne pas utiliser des entiers comme des booléens et réciproquement.

2.4.2 Les références

Plus simple à manipuler que les pointeurs, le *C++* dispose d'une notion supplémentaire : les références. Une référence est un moyen d'accéder en lecture et/ou écriture à une donnée via son adresse. Les références sont déclarées en faisant suivre le spécificateur de type par le symbole &.

```
1 int i;
2 int &j = i;
```

Listing 2.8 – Références - déclaration

La variable j est donc une référence sur la variable i, j est un *alias* (synonyme) de i. Une variable de type référence est une adresse mais, une fois son initialisation effectuée, toute opération réalisée sur la référence se fait sur l'objet référencé et non sur l'adresse. C'est pourquoi une référence doit être initialisée à la création. Une référence ne peut donc référencer qu'un seul objet tout au lon de sa vie.

```
1 int i, k=0;
2 int &j = i;
3 j=4;//Cette instruction affecte 4 à i
4 j=k;//Cette instruction ne transforme pas j
5     //en référence sur k,
6     //mais affecte la valeur de k à i
```

Listing 2.9 – Références - illustration

L'utilisation principale des références est le passage de paramètres modifiables aux fonctions sans utiliser les pointeurs.

```
1 void swap (int &a, int &b){
2   int tmp=a;
3   a=b;
4   b=tmp;
5 }
6 int main (void){
7   int x=2, y=3;
8   swap(x,y);//x vaut maintenant 3 et y 2
9 }
```

Listing 2.10 – Passage de paramètres par référence

Cela est également très utile dans le cas d'objets de grandes tailles qu'il serait bien dommage de passer par copie, mais, attention dans ce cas, de ne pas oublier de rendre l'objet référencé constant pour éviter toute modification non souhaitée du paramètre par la fonction. Cela se fait grâce au modificateur *const*.

```
1 void maFonction (const Type & t){
2   //t n'est pas une copie de l'argument effectif
3   //grâce à la référence mais
4   //ne peut être modifié grâce au const
5 }
```

Listing 2.11 – Paramètre non modifiable

Le type de retour d'une fonction peut être une référence, cela permettra à l'appel de cette fonction d'être membre de gauche d'une affectation.

```
#include <string.h>
char *names[5]={"Lynda","Omar","Axel","Lisa","Habib"};
int ages[5]={21, 32, 34, 20, 37};

int & age(char *name) {
  for (int i = 0; i < 5; i++) {
    if (strcmp(name, names[i]) == 0) {
      return ages[i];
    }
  }
}
int main (void){
  age("Lynda")=22;
  age("Habib")++;
}
```

Listing 2.12 – Référence en type de retour d'une fonction

Attention à la confusion, & peut avoir plusieurs significations :

- Dans une déclaration, comme vu précédemment, il sert à définir un type référence.
- Utilisé ailleurs, il sert à obtenir l'adresse de la variable dont il précède le nom.
- Entre deux opérandes, il est l'opérateur *et* bit à bit.

2.4.3 Les classes

C++ est un langage dans lequel on peut choisir de faire de la programmation objet à base de classes. Un objet représente un concept, une idée ou toute entité du monde physique. Les objets ont une structure interne et un comportement.

Une classe est la déclaration d'un type d'objet. On parle d'instance d'une classe lors de la création d'un objet du type de cette classe. Les données internes de la classe appelées attributs. Les traitements sur ces données et/ou d'autres données sont appelées méthodes. Le paradigme de programmmation objet introduit les notions d'objet et d'héritage : un objet contient implicitement les variables et les fonctions de ses ancêtres.

Comme les structures en *C*, les classes permettent de créer de nouveaux type. La syntaxe est similaire mais elle dispense de l'utilisation du *typedef*, en effet, le nom de la classe est directement accessible. L'ordre des membres

n'a aucune importance. Au sein d'une classe, on accède à ses membres simplement par leur nom. On peut accéder à soi-même avec le mot-clé *this* qui est un pointeur sur l'instance de la classe (l'objet) qui a appelé la fonction.

Le *C++* permet de préciser le type d'accès des membres (attributs et méthodes) d'un objet :

— *public* : les membres publics sont en accès libre partout.
— *private* : les membres privés d'une classe ne sont accessibles que par les méthodes membres ou amies de leur classe.
— *protected* : niveau intermédiaire où les menbres sont accessibles par les fonctions membres et amies de la classe proriétaire et aussi par les fonctions membres et amies des classes qui en héritent directement. Par défaut les membres d'une classe sont privés, contrairement aux membres d'une structure en *C* qui eux sont publics.

C'est cette gestion des accès qui permet l'encapsulation. Ainsi, un utilisateur de la classe ne pourra pas accéder à tous les éléments de celles-ci. Il sera obligé d'utiliser les fonctions membres publiques de la classe. L'objet peut ainsi être vu de l'extérieur comme une « boîte noire » possédant certaines propriétés et des comportements spécifiés. C'est le comportement d'un objet qui modifiera son état.

Lorsqu'on programme une classe, il faut toujours penser réutilisation. C'est pourquoi, il est bien d'avoir un peu de discipline lorsqu'on code une classe. En général, il est donc recommandé de placer les définitions des classes dans des fichiers en-tête (fichiers .h, .hpp, .hh, ...), à raison d'une classe (ou exceptionnellement un groupe de classes si le lien entre-elles le permet) par fichier. Les définitions des fonctions membres qui ne sont pas définies à l'intérieur de leurs classes sont écrrites dans des fichiers sources (fichiers .cpp, .cc, ...). Dans ce cas, pour signifier qu'une fonction est membre d'une classe, on utilise l'opérateur de résolution de portée, noté ::.

Pour initialiser un objet, il existe une méthode spéciale : le constructeur. Le constructeur porte le nom de la classe, n'a pas de type de retour et par conséquent n'a pas d'instruction *return*. Une classe peut avoir plusieurs constructeurs, ceux-ci doivent avoir des signatures différentes. Dès qu'un variable est définie avec un type objet, le constructeur est appelé explicitement ou implicitement. En l'absence de constructeur dans la définition de la classe (mais uniquement dans ce cas), le compilateur en crée un par défaut. Quand une classe a des objets membres, ceux-ci doivent être instanciés et initialisés lors de l'instanciation de cette classe. Il existe une syntaxe spéciale pour faire appel aux constructeurs des objets membres avant même d'entrer dans le corps du constructeur. Cette syntaxe est valable également avec les types de base.

```
#ifndef PILE_H
#define PILE_H

class Pile{
  private:
    typedef struct E{
      int valeur;
      struct E *suivant;
    } Element;

    Element * _premier;
  public:
    Pile  ();
    Pile  (int);
    ~Pile ();

    void empiler  (int);
    void depiler  ();
    bool vide     () const;
    int sommet () const;
    void afficher () const;

};

#endif
```

Listing 2.13 – pile.h

```
#include <limits>
#include <iostream>
#include "pile.h"

Pile::Pile ():_premier(nullptr){}

Pile::Pile (int t):_premier(new Element){
  _premier->valeur=t;
  _premier->suivant=nullptr;
}
Pile::~Pile(){
  while (!vide()) depiler();
  return;
}
void Pile::empiler (int t){
  Element *nouveau = new Element;
  nouveau->suivant= _premier;
  _premier = nouveau;
  nouveau->valeur=t;
}
void Pile::depiler  (){
  if (! vide ()){
    Element * tmp = _premier;
    _premier = _premier->suivant;
    delete tmp;
  }
}
bool Pile::vide () const {
  return _premier==nullptr;
}
int Pile::sommet ()const {
  if (!vide())
    return _premier->valeur;
  return INT_MAX;
}
void Pile::afficher () const{
  Element *courant=_premier;
  if (courant==nullptr)
    std::cout<<"La pile est vide  "<<std::endl;
  std::cout<<"La pile : "<<std::endl;
  while (courant!=nullptr){
    std::cout<<courant->valeur<<" ";
    courant=courant->suivant;
  }
  std::cout<<"\n";
}
```

Listing 2.14 – pile.cpp

```
#include <iostream>
#include "pile.h"
int main (){
 Pile p;//appel implicite du constructeur par défaut
 p.empiler(4);
 p.empiler(5);
 p.empiler(6);
 p.afficher();
 p.depiler();
 p.afficher();
 //3 différents appels explicites au constructeur
 //qui conduisent à la créations de 3 objets
 //identiques :
 Pile p2 (3);
 Pile p3 = Pile (3);
 Pile p4 = 3;//possible car il existe
             //un constructeur à 1 paramètre
}
```

Listing 2.15 – main_pile.cpp

Dans cet exemple, on peut remarquer l'existence d'une autre fonction spéciale : le destructeur. Il n'a pas de valeur de retour, ni d'argument et porte le nom de la classe pécédé d'un ~. En l'absence de destructeur dans la classe, le compilateur en génère un par défaut qui ne fait rien pour les membres de type de base et appelle le destructeur des objets membres dans l'ordre inverse de leur création dans le constructeur. Le destructeur d'une classe est appelé lorsqu'un objet de la classe est détruit. Dans notre exemple le destructeur de pile entraîne la libération de l'espace alloué pour son premier élément.

2.5 Les modèles

Les modèles (*templates*) sont une particularité du *C++*, qui permet l'écriture d'un code qui fait abstraction du type des données avec lesquelles il sera finalement utilisé. Les *templates* introduisent le concept de programmation générique. La bibliothèque standard de *C++* fournit de nombreux outils bien pratiques fait avec les *templates*, en particulier dans la *STL* (nous y reviendront très bientôt).

2.5.1 Les modèles de fonction

Les templates, dont l'objectif est la réutilisation d'algorithmes, sont une alternative aux macros (appel au préprocesseur), fortement déconseillées en *C++*. Un exemple fréquent est la fonction qui retourne le minimum de deux valeurs. Cette fonction se code de manière similaire que le type des arguments comparés soit des *int* ou des *double* :

```
1 int min (int a, int b){
2   return a<b ? a : b;
3 }
4 double min (double a, double b){
5   return a<b ? a : b;
6 }
```

Listing 2.16 – Répétition de code

En *C*, la solution serait la définition d'une macro :

```
1 #define min (a,b)  ((a)<(b)?(a):(b))
```

Listing 2.17 – Macro

Cette syntaxe a le défaut de son avantage : ce n'est pas typé. De plus, il n'y a aucun contrôle du compilateur. La solution offerte par le *C++* est donc une fonction générique. Le mot-clé *template* suivi d'une liste de paramètres formels se place devant la définition de la fonction.

```
1 template <class T>
2 T min (T a, T b){
3  return a<b ? a : b;
4 }
```

Listing 2.18 – Modèle de fonction

La directive *template <class T>* indique que la fonction est paramétrée par le type *T*. Ensuite, vous utilisez T comme un type normal dans le corps de la fonction. Il est possible d'avoir une liste plus longue de paramètres :

```
1 template <class T,class U,class V>.
```

Listing 2.19 – Template sur plusieurs types

Ici le mot-clé *class* n'a pas sa signification habituelle mais indique qu'un paramètre de type va suivre. Les identificateurs *T*, *U* et *V* pourront alors être utilisés dans la portée du *template* (fonction ou classe décrite par le

modèle). Chaque identificateur de type ne peut apparaître qu'une fois dans la liste des paramètres formels. À l'utilisation :

```
int main (void) {
  int x = 12, y = -7 ;
  float pi = 3.1415 ;
  char c = 'C' ;
  cout << "Résultats:" << endl;
  cout << min (x,y) << endl;
  cout << min (pi, float (y))<< endl;
  cout << min (c, 'x') << endl;
}
```

Listing 2.20 – Instanciation

Dans cet exemple, il y a instanciation de 3 fonctions min :

```
int min (int a, int b);
float min (float a, float b);
char min (char a, char b);
```

Listing 2.21 – Intsanciation - suite

Le compilateur générera autant de fonctions que d'utilisation de la fonction min avec différents type. Les modèles de fonction permettent donc d'éviter de d'avoir à écrire plusieurs fonctions pour chaque type de donnée traité. Attention par contre il faut savoir que le compilateur ne fait pas les conversions standard. Ainsi, si on reprend les modèles de fonctions créés précédemment, dans notre *main*, on ne peut pas écrire :

```
cout<< min (x, c) <<endl;
```

Listing 2.22 – Instanciation - erreur

C'est à nous soit de convertir explicitement (*cast*) le caractère *c* en un *int*, soit d'écrire un modèle de fonction à deux paramètres pour *min*.

2.5.2 Les modèles de classe

Les modèles de classe se déclarent avec une syntaxe similaire :

```
template<class T,int max_size>
class Tab{
    public:
        Tab();
```

```
5           int getSize();
6           T get(int index);
7       //Méthode d'ajout d'un élément à la table
8       //renvoie true si l'élément a pu être ajouté
9           bool add(T element);
10
11      private:
12    T tab[max_size];
13          int size;
14  };
```

Listing 2.23 – Définition de classe Tab

L'implémentation des méthodes a une syntaxe plus compplexe. En effet, les fonctions membres d'un modèle de classe sont des modèles de fonctions avec les mêmes paramètres, ce qui est implicite quand leur définition se situe dans la définition de la classe. Sinon, il faut rappeler qu'il s'agit d'un modèle de classe avec sa liste d'arguments avant la définition de la fonction. Le nom de la méthode est précédé du nom de la classe suivi des paramètres du patron de classe entre <>, afin que le compilateur détermine de quelle classe il s'agit.

```
1  template<class T,int max_size>
2  Tab<T,max_size>::Tab(): size(0){}
3
4  template<class T,int max_size>
5  int Tab<T,max_size>::getSize(){
6      return size;
7  }
8
9  template<class T,int max_size>
10 T Tab<T,max_size>::get(int index){
11     return tab[index];
12 }
13
14 template<class T,int max_size>
15 bool Tab<T,max_size>::add(T element){
16     if (size>=max_size) return false;
17     tab[size++]=element;
18     return true;
19 }
```

Listing 2.24 – Définition des méthodes

Il faut savoir que dès lors qu'on travaille sur des modèles de classes, il

faut mettre les définitions des méthodes dans les fichiers d'en-tête (*.h*).

```
#include <iostream>
#include "Tab.h"
int main (){
  Tab <int, 100> t;
  t.add(12);
  t.add(200);
  for (int i=0; i<t.getSize (); i++)
    std::cout<<t.get(i)<<"   ";
  std::cout<<std::endl;
}
```

Listing 2.25 – Un main

2.6 Le mot-clé *auto*

Depuis *C++ 11*, il existe un mécanisme dit « d'inférence de type » qui permet de déclarer une variable dont le type est déduit de l'expression d'initialisation dans sa déclaration. Ainsi, avec le mot-clé *auto*, on déclare une variable sans déclarer son type. Il est par contre impératif d'initialiser cette variable.

En cas de présence du mot-clé *auto*, le compilateur utilise l'expression d'initialisation d'une variable, ou d'un paramètre d'expression lambda, pour déduire son type.

```
#include <iostream>
#include <cmath>
#include <typeinfo>

template<class T, class U>
//Le type de retour de add est le  type de retour de l'op
    érateur d'addition operator+(T,U)
auto add(T t, U u) -> decltype(t + u){
  return t + u;
}

//équivalent à  double (*get_fun(int))(double)
auto get_fun(int arg)->double(*)(double){
  switch (arg) {
    case 1: return std::fabs;
    case 2: return std::sin;
```

```
16      default: return std::cos;
17    }
18  }
19
20  int main() {
21    auto a = 1 + 2;
22    std::cout << "type de a: " << typeid(a).name() << '\n';
23    auto b = add(1, 1.2);
24    std::cout << "type de b: " << typeid(b).name() << '\n';
25    auto d = {1, 2};
26    std::cout << "type de d: " << typeid(d).name() << '\n';
27
28    auto my_lambda = [](int x) { return x + 3; };
29    std::cout << "my_lambda: " << my_lambda(5) << '\n';
30
31    auto my_fun = get_fun(2);
32    std::cout << "type de my_fun: " << typeid(my_fun).name
       () << '\n';
33    std::cout << "my_fun: " << my_fun(3) << '\n';
34  }
```

Listing 2.26 – Inférence de type - exemple issu de cppreference.com

Le mot-clé *auto* est un moyen simple de déclarer une variable qui a un type complexe. Par exemple, vous pouvez utiliser *auto* pour déclarer une variable dans laquelle l'expression d'initialisation implique des *templates*, des pointeurs vers des fonctions ou des pointeurs vers des membres...

L'une des raisons les plus intéressantes d'utiliser le mot-clé *auto* est donc la simplicité. En effet, l'inférence de type gagne en intérêt lorsqu'on doit travailler avec des types longs à écrire. Voici un exemple :

```
1  #include<iostream>
2  #include<vector>
3  using namespace std;
4  int main(){
5    vector<double> v = {1, 2, 3, 4, 5, 6};
6    for (auto it = v.begin(); it != v.end(); ++it)
7      cout << *it << endl;
8  }
```

Listing 2.27 – auto - simplicité

Chapitre 3

Les itérateurs

Un conteneur est un objet qui contient d'autres objets. Il fournit des méthodes pour gérer son contenu ainsi que l'accès à son contenu. Souvent cet accès se fait au moyen d'un « itérateur », qui peut être vu comme une abstraction de la notion de pointeur. Ainsi, il offre des services du même type tels que le déréférencement qui permet d'accéder à l'élément courant (actuellement pointé) ou l'incrémentation qui permet de se déplacer pour pointer sur l'élément suivant.

3.1 Opérations et syntaxes

Il faut pouvoir créer un itérateur pointant sur le premier élément et être capable de savoir si on a parcouru l'ensemble des éléments du conteneur. Les itérateurs permettent de parcourir un conteneur, d'accéder séquentiellement à tous ses éléments tout en isolant le développeur de la structure interne du conteneur. Voici donc comment manipuler un itérateur `it` :

- Déréférencement (accès à l'objet pointé) : `*it`,
- Incrémentation (déplacement de l'itérateur afin de désigner l'objet suivant) : `it++` ou `++it`,
- Récupération de l'itérateur de début de conteneur : `conteneur.begin()`,
- Récupération de l'itérateur situé juste après le dernier élément du conteneur : `conteneur.end()`

3.2 Types d'itérateurs

Il existe plusieurs types d'itérateurs. Tous n'offrent pas les même fonctionnalités. Par exemple, un itérateur sur une liste doublement chaînée permettra un parcours séquentiel bidirectionnel de cette liste (on pourra donc

avancer ou reculer dans cette liste). Cet itérateur aura donc les opérateurs d'incrémentation `++` et de décrémentation `-`. Par contre, un itérateur sur une liste simplement chaînée ne permettra un parcours des éléments que vers « l'avant » et n'aura donc à disposition que l'opérateur d'incrémentation.

Il faut donc connaître le type d'itérateur que l'on manipule pour savoir les opérations qu'il est possibles de réaliser avec celui-ci. La liste des itérateurs existant est consultable ici : https://en.cppreference.com/w/cpp/iterator. Il existe, donc, depuis *C++ 17*, six catégories d'itérateurs. En cliquant sur chacune d'entre elles, vous pourez consulter les conditions requises pour implémenter cette catégorie.

Les catégories d'itérateurs ne sont pas définies par des types spécifiques, mais par les opérations qui peuvent être effectuées sur leurs itérateurs. Cette définition signifie que n'importe quel type qui prend en charge les opérations nécessaires peut être utilisé comme un itérateur. Par exemple, un pointeur prend en charge toutes les opérations requises par un `RandomAccessIterator`, donc un pointeur peut être utilisé partout où un `RandomAccessIterator` est attendu.

Chapitre 4

Les conteneurs de la STL

En *C++* la bibliothèque standard (*Standard Template Library*) est mise en œuvre à l'aide des *templates* et fournit :

- Différentes classes de conteneurs telles que les vecteurs, les listes, les tables associatives. Ces types de données sont implémentés de manière générique et efficace.
- Les itérateurs : une abstraction des pointeurs qui permet de parcourir simplement les séquences d'éléments d'un conteneur.
- Des algorithmes génériques tels que les algorithmes de tri, insertion, suppression, recherche ...
- La classe `String` qui permet une gestion efficace et sûre des chaînes de caractères.

.

4.1 La classe *vector*

Un `vector` est un conteneur séquentiel qui encapsule un tableau de taille dynamique. Les éléments sont stockés de manière contigüe, ils sont donc accessible via les itérateurs mais également avec les pointeurs classiques. On peut également accéder aux éléments par leur index avec l'opérateur `[ ]`.

Le stockage du vecteur est géré automatiquement, sa taille est modifiée en fonction des besoins. Les vecteurs occuppent plus d'espace que les tableaux car d'avantage d'espace mémoire est alloué pour anticiper la croissance future. De cette façon, un vecteur n'a pas besoin de réallouer de la mémoire à chaque insertion d'élément mais uniquement lorsque la mémoire supplémentaire est épuisée.

La quantité de mémoire allouée peut être connue par un appel à la méthode `capacity ()`. De la mémoire peut être rendue au système par un ap-

pel à `shrink_to_fit ()`. Les réallocations sont généralement des opérations coûteuses en terme de performance, c'est pourquoi si le nombre d'éléments est connu à l'avance, il est recommandé d'utiliser la méthode `reserve ()` ou de donner le nombre d'éléments à la construction du vecteur.

Voici la complexité des opérations de base réalisées sur les vecteurs :

— Accès aléatoire à un élément : temps constant $\mathcal{O}(1)$
— Insertion/suppresssion d'un élément à la fin : constante amortie $\mathcal{O}(1)$
— Insertion/suppression d'un élément : temps linéaire proportionnel à la distance de l'élément à la fin du vecteur $\mathcal{O}(n)$

Voici la liste des constructeurs de la classe `vector` :

— `vector ()` : Construteur par défaut, construit un vecteur vide.
— `vector (size_type count, const T& value = T(), const Allocator& alloc = Allocator())` : construit un vecteur qui contient `count` copéies d'éléments de valeur `value`.
— `vector (size_type count)` : construit le vecteur avec `count` instances de T construites en-place (sans copie).
— `vector (const vector& other)` : constructeur par copie. Construit un vecteur avec la copie des éléments contenus dans `other`.
— `vector (InputIt first, InputIt last, const Allocator& alloc = Allocator())` : construit un vecteur avec les éléments contenus dans la plage $[first, last)$.

Pour déclarer et utiliser un vecteur il ne faut pas oublier l'instruction `#include <vector>`. Ensuite, pour déclarer un vecteur, la syntaxe sera la suivante : `vector <int> v (6);`. Dans ce cas, nous avons déclaré un vecteur d'entiers `v`. Nous avons choisi que `v` contienne 6 éléments.

Quelques méthodes et opérateurs qui s'appliquent au vecteur :

— `void assign(size_type count, const T& value );` : remplace le contenu du vecteur par `count` copies de l'elément `value`. Complexité : $\mathcal{O}(count)$
— `template <class InputIt> void assign( InputIt first, InputIt last );` : remplace le contenu du vecteur par la copie des éléments de l'intervalle $[first, last)$. Comlexité $\mathcal{O}(last - first)$.
— `reference at(size_type pos);` : renvoie une référence vers l'élément d'indice `pos` si ce dernier est bien dans l'intervalle d'indices du vecteur. Si ce n'est pas le cas, une exception `std::out_of_range` est levée.
— `reference operator[]( size_type pos );` : renvoie une référence vers l'élément d'indice `pos` sans vérification.
— `reference front ();` : retourne une référence sur le premier élément.
— `reference back ();` : retourne une référence sur le dernier élément.
— `void push_back (const T& value);` : Ajoute l'élément `value` à la

fin du vecteur.
— `iterator begin();` : retourne un itérateur sur le premier élément.
— `iterator end ();` : retourne un itérateur après le dernier élément.

La *STL* possède des conteneurs réduisant les possibilités de `Vector` à quelques fonctionnalités : `Stack` (pile), `queue` (file), `priority_queue` (file à priorité).

4.2 La classe *Deque*

La classe `Deque` pour *Double Ended QUEue* (file à deux entrées) s'utilise comme un vecteur à deux différences près :

— `Deque` est optimisé pour au l'ajout ou le retrait d'élements se fasse aux extrémités du conteneurs, soit à la fin grâce aux méthodes `push_back` et `pop_back` ou bien au début avec les méthodes `push_front` et `pop_front`.
— `Deque` ne stocke pas nécéssairement les éléments de façon contigüe. Il est donc plus efficace en cas de réallocation mémoire car il n'est pas nécessaire de recopier les éléments existants.

Voici la complexité des opérations de base réalisées :

— Accès aléatoire à un élément : temps constant $\mathcal{O}(1)$
— Insertion/suppresssion d'un élément à une extrémité : temps constant $\mathcal{O}(1)$
— Insertion/suppression d'un élément : temps linéaire proportionnel à la distance de l'élément à la fin du vecteur $\mathcal{O}(n)$

4.3 La classe *List*

Le conteneur `List` est implémentée comme une liste doublement chaînée. Les listes doublement chaînées permettent de stocker chacun de leurs éléments dans des emplacements de stockage différents et non liés. L'ordre est conservé en interne puisque chaque élément possède un pointeur vers l'élément qui le précède et un pointeur vers l'élément suivant. La liste utilise donc plus d'espace mémoire mais peut être parcourue dans les deux sens. Il est impossible d'accéder à un élément directement, il faut parcourir la liste à l'aide d'un itérateur. Les opérations d'insertion et de suppression se font à temps constant ($\mathcal{O}(1)$) n'importe où dans le conteneur.

Par rapport aux autres conteneurs séquentiels vus précédemment (tableau, vecteur et deque), les listes fonctionnent généralement mieux pour insérer, extraire et déplacer des éléments dans n'importe quelle position du conteneur pour laquelle un itérateur a déjà été obtenu, et donc, par conséquent, dans les algorithmes qui font un usage intensif de ceux-ci, comme les

algorithmes de tri.

Quelques méthodes et opérateurs qui s'appliquent aux listes (mais je vous encourage vivement à aller consulter la page de doc) :

- `list();` : Constructeur par défaut, construit une liste vide. Complexité : $\mathcal{O}(1)$
- `list( const list& other );` : Constructeur par copie. Construit une liste qui contient une copie des éléments de `other`. Comlexité : $\mathcal{O}(n)$ avec n le nombre d'éléments de `other`.
- `list( size_type count, const T& value, const Allocator& alloc = Allocator());` : Construit une liste avec `count` copies de l'élément `value`. Complexité : $\mathcal{O}(count)$.
- `void assign(size_type count, const T& value );` : remplace le contenu de la liste par `count` copies de l'elément `value`. Complexité : $\mathcal{O}(count)$
- `template <class InputIt> void assign( InputIt first, InputIt last );` : remplace le contenu de la liste par la copie des éléments contenus dans l'intervalle défini par les itérateurs $[first, last)$. Comlexité : $\mathcal{O}(last - first)$.
- `reference front ();` : retourne une référence sur le premier élément de la liste. Complexité : $\mathcal{O}(1)$
- `reference back ();` : retourne une référence sur le dernier élément de la liste. Complexité : $\mathcal{O}(1)$
- `void push_back (const T& value);` : Ajoute l'élément `value` à la fin de la liste mais en deux étapes et une copie temporaire de `value` est générée (Le nouvel élément est initialisé en tant que copie de `value`, puis `value` est déplacé dans le nouvel élément). Complexité $\mathcal{O}(1)$. De la même manière la méthode `void push_front (const T& value);` existe.
- `void emplace_back (const T& value);` : Ajoute l'élément `value` à la fin de la liste. L'élément est réalisé in-situ, c'est à dire pas de copie temporaire suivi d'un déplacement. Le constructeur de l'élément est appelé avec exactement les mêmes arguments que ceux fournis à la fonction. Complexité $\mathcal{O}(1)$. De la même manière la méthode `void emplace_front (const T& value);` existe.
- `void pop_back();` : Supprime le dernier élément de la liste. . De la même manière la méthode `void pop_front (const T& value);` existe.
- `iterator begin();` : retourne un itérateur sur le premier élément de la liste.
- `iterator end ();` : retourne un itérateur après le dernier élément de la liste.

Chapitre 5

Les graphes

Le formalisme des graphes permet de décrire des problèmes difficiles à résoudre de manière simple.

5.1 Quelques notions

5.1.1 Graphe orienté

Un graphe orienté peut être représenté par le couple $G = (S, A)$ où :

— S : Ensemble fini d'éléments représentant les *sommets* du graphe.

— A : Ensemble fini de couples de sommets appelés *arcs* ou *arêtes*.

Dans le cas d'un graphe orienté, ce sont bien des couples qui représentent les arêtes et non des paires, cela signifie que l'ordre est important : $(x, y) \neq (y, x)$. S'il existe un couple $(x, y) \in A$, alors on trace une fléche (un arc) allant du sommet x au sommet y. x et y sont appelés extrémités de l'arc.

Dans le cas du graphe 5.1, nous avons :

— $S = \{0, 1, 2, 3, 4, 5, 6, 7\}$

— $A = \{(0, 1), (0, 4), (1, 2), (1, 3), (2, 0), (2, 3), (2, 5), (3, 6), (3, 7), (4, 4), (6, 4), (6, 5), (6, 7)\}$

Si $(x, y) \in A$, on dit que y est un *successeur* de x et x est un *prédécesseur* de y. On dit également que x et y sont *adjacents*. Le *degré entrant* d'un sommet correspond au nombre de prédécesseurs de celui-ci. De même, le nombre de successeurs d'un sommet définit son *degré sortant*. Le *degré* d'un sommet est égal à la somme de son degré entrant et de son degré sortant.

S'il existe plusieurs arcs entre deux sommets, on parle de *multigraphe*. Deux arcs sont adjacents s'ils ont au moins une extrémité en commun. Dans le cas de l'arc (x, x), on parle de *boucle*.

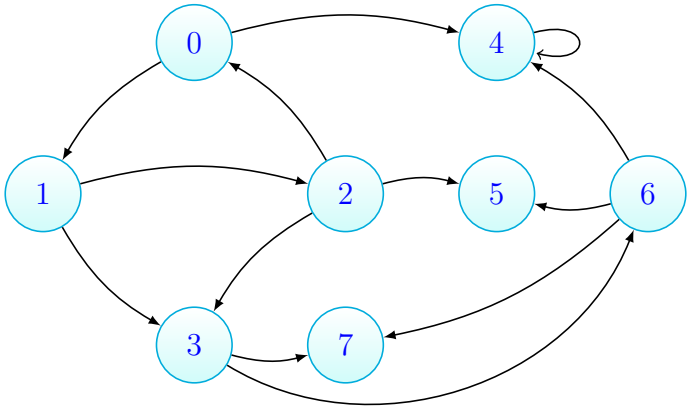

FIGURE 5.1 – Un exemple de graphe orienté

5.1.2 Graphe non-orienté

Les graphes non-orientés sont un cas particulier des graphes orientés. La présence d'une arête entre deux sommets établit entre-eux, un lien symétrique. On parle alors de sommets voisins ou adjacents s'il existe une arête les reliant.

Un graphe non-orienté peut être représenté par le couple $G = (S, A)$ où :
— S : Ensemble fini d'éléments représentant les *sommets* du graphe.
— A : Ensemble fini de paires de sommets appelés *arcs* ou *arêtes*.

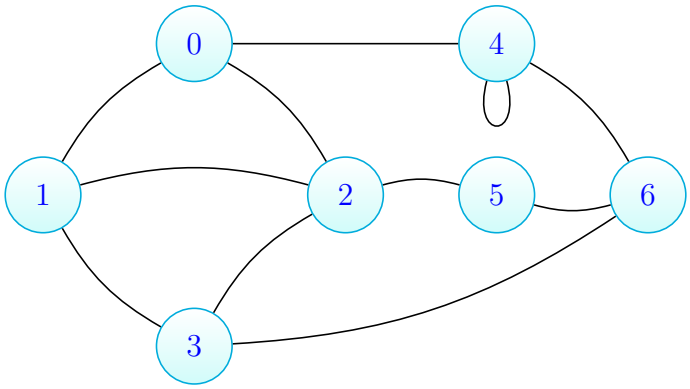

FIGURE 5.2 – Un exemple de graphe non-orienté

Dans le graphe de la figure 5.2, nous avons :
— $S = \{0, 1, 2, 3, 4, 5, 6\}$

— $A = \{\{0,1\},\{0,2\},\{0,4\},\{1,2\},\{1,3\},\{2,3\},\{2,5\},\{3,6\},\{4,4\},\{4,6\},\{5,6\}\}$

Pour tout couple, on a donc : $Si(x,y) \in A, alors(y,x) \in A$. C'est pourquoi, dans la définition d'un graphe non-orienté, nous utiliserons des paires plutôt que des couples pour représenter les arêtes. En effet, la paire $\{x,y\}$ indique la présence d'une arête entre les sommets x et y sans notion d'ordre entre les éléments de la paire. Le *degré* d'un sommet correspond au nombre de voisins de ce sommet.

5.1.3 Graphe pondéré

Un graphe pondéré ou valué est un graphe orienté ou non pour lequel on associe une fonction de coût aux arêtes (cf. fig 5.3).

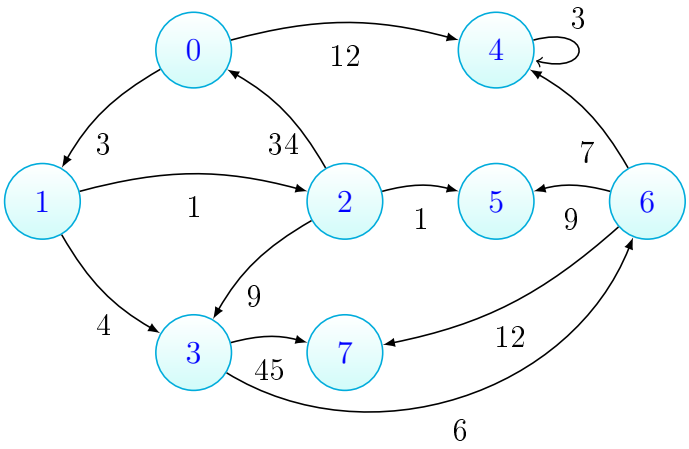

FIGURE 5.3 – Un exemple de graphe pondéré

5.2 Représentation

5.2.1 Matrice d'adjacence

Dans le cas d'une représentation à l'aide d'une matrice d'adjacence le graphe est repprése nté par un tableau 2D indexé sur l'ensemble des sommets. Les éléments du tableau peuvent être de nature différentes selon le type du graphe considéré :

— graphe non pondéré : l'élément d'indice $[i][j]$ vaut vrai si l'arc (S_i, S_j) figure dans le graphe (i.e. il existe un arc reliant S_i et S_j). Dans le

cas d'un graphe non-orienté, si l'élément $[i][j]$ est vrai alors l'élément $[j][i][$ l'est aussi.

— graphe pondéré : les éléments stockés seront des réels correspondant à la fonction de coût. On aura donc :

$$\begin{aligned} M[i][j] &= c(i,j) \text{ si } (i,j) \in G \\ &= -1 \text{ par convention, sinon.} \end{aligned}$$

— multigraphe : les éléments du tableau sont des entiers correspondant au nombre d'arcs reliant deux sommets.

Si nous reprenons les exemples précédents, nous obtenons donc les matrices d'adjacences suivantes :

— graphe 5.1 : Nous utilisons ici des 0 et 1 pour représenter les valeurs bouléennes *false* et *true*

$$M = \begin{pmatrix} 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

— graphe 5.2 :

$$M = \begin{pmatrix} 0 & 1 & 1 & 0 & 1 & 0 & 0 \\ 1 & 0 & 1 & 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 1 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 1 & 1 & 0 \end{pmatrix}$$

— graphe 5.3 :

$$M = \begin{pmatrix} 0 & 3 & 0 & 0 & 12 & 0 & 0 & 0 \\ 0 & 0 & 1 & 4 & 0 & 0 & 0 & 0 \\ 34 & 0 & 0 & 9 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 6 & 45 \\ 0 & 0 & 0 & 0 & 3 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 7 & 9 & 0 & 12 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

Cette représentation des graphes a pour avantages :

— la simplicité de représentation,
— l'accès direct aux arêtes : permet de tester leur présence, d'ajouter ou de supprimer une arête en temps constant,
— l'écriture simplifiée des algorithmes.

Cependant, on peut noter les inconvénients suivants :

— encombrement de mémoire (n^2 éléments),
— coût d'initialisation ($o(n^2)$),
— coût de la recherche des successeurs d'un sommet ($o(n)$),
— ajout d'un sommet ?

5.2.2 Listes d'adjacence

Nous utiliserons cette représentation, puisqu'elle permet d'optimiser l'espace mémoire et offre des algorithmes plus permformants de manipulation du graphe. La graphe est alors représenté par un tableau de N (N étant les nombre de sommets du graphe) listes. Chaque nœud aura donc sa propre liste d'adjacence. Celle-ci ne contient que les successeurs du nœud en question (ou bien des paires (successeur, coût) dans le cas d'un graphe pondéré). Si on reprend l'exemple du graphe 5.1, la figure 5.4 illustre comment celui-ci serait représenté avec des listes d'adjacence.

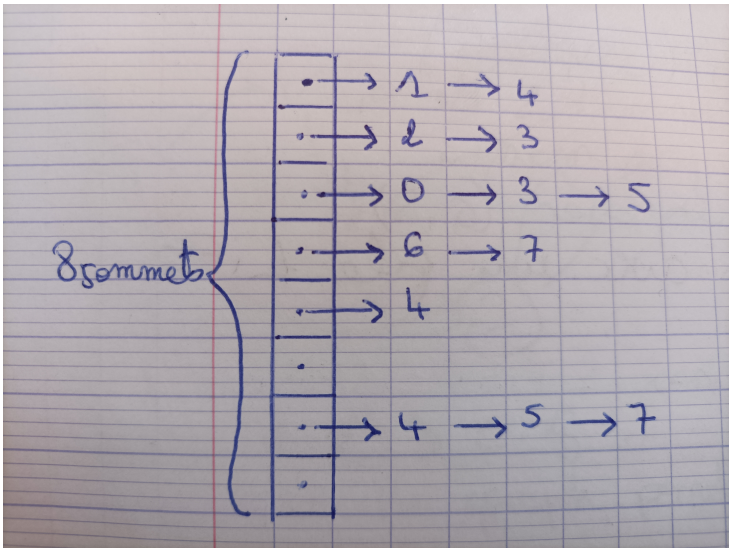

FIGURE 5.4 – Représentation du graphe avec des listes d'adjacence

Voici, le squelette de la classe `Graph` que nous coderons en TP.

```
#ifndef GRAPH_HPP___
#define GRAPH_HPP___

#include <vector>
#include <list>
#include <utility>

class Graph {
  public:
    typedef std::pair<unsigned int, unsigned int> arete;
    typedef std::list<arete> liste_adjacence;
  public:
    Graph (unsigned int nb_sommets =0);
    void ajouter_arete (unsigned int origine, unsigned
   int destination, unsigned int poids);
    void afficher ();
    void explorer_branche (arete a, unsigned int
   destination, liste_adjacence chemin_courant, std::
   vector<std::pair<liste_adjacence, int>> * chemins);
    void chemins_profondeur (unsigned int origine,
   unsigned int destination, std::vector<std::pair<
   liste_adjacence, int>> * chemins);
    void chemins_largeur (unsigned int origine, unsigned
   int destination, std::vector<std::pair<liste_adjacence
   , int>> * chemins);
    int dijkstra (unsigned int origine, unsigned int
   destination, std::vector <arete> * chemins);
    static void charger (Graph *);

    struct comp {
      bool operator() (const arete &a, const arete &b)
   const {return a.second < b.second;}
    };
  private:
    std::vector<liste_adjacence> graph;
};
#endif // GRAPH_HPP___
```

Listing 5.1 – graph.hpp

5.3 Parcours

5.3.1 Parcours en largeur

Le parcours en largeur (ou en anglais BFS : *breadth first search*) consiste à parcourir le graphe de la manière suivante : on commence par explorer un nœud puis ses successeurs, puis les successeurs non visités des successeurs. La liste des sommets à visiter fonctionne comme une file (premier arrivé, premier servi), selon les étapes suivantes :

1. Mettre le noeud dans la file
2. Retirer le premier nœud de la file pour le traiter
3. Mettre tous ses successeurs non visités auparavant dans la file
4. si la file n'est pas vide, reprendre en 2)

Voici l'algorithme en pseudo-code :

Algorithm 1 Parcours en largeur

```
function PARCOURS_LARGEUR(Graph G, Sommet s)
    Chemin=creer_file()
    f=creer_file()
    f.enfiler(s)
    marquer (s)
    while f non vide do
        s=f.defiler()
        Chemin.enfiler(S)
        for all v voisin de s dans G do
            if v non marqué then
                f.enfiler(v)
                marquer(v)
            end if
        end for
    end while
end function
return Chemin
```

5.3.2 Parcours en profondeur

L'algorithme de parcours en profondeur (en anglais DFS : *Depth-First Search*) se décrit naturellement de manière récursive. Classiquement, il sert à déterminer s'il existe un chemin d'un sommet à un autre. Le principe est

le suivant : on part du sommet de départ et on poursuit le chemin jusqu'à arriver à une impasse ou un sommet déjà visité. On revient alors au dernier sommet pour lequel on avait le choix d'un autre chemin. Voici l'algorithme en pseudo-code :

Algorithm 2 Parcours en profondeur

```
function EXPLORER_BRANCHE(Graph G, Sommet s, File chemin)
    Ajouter s au chemin
    marquer (s)
    for all v voisin de s dans G do
        if v non marqué then
            explorer_branche(G, v, chemin)
        end if
    end for
end function
function PARCOURS_LARGEUR(Graph G, Sommet s)
    Chemin = creer_file()
    Chemin.ajouter(s)
    for all v voisin de s dans G do
        if v non marqué then
            explorer_branche (G, v, Chemin)
        end if
    end for
    return Chemin
end function
```

5.4 Plus court chemin : algorithme de Dijkstra

L'algoriothme de Dijkstra permet de déterminer le plus court chemin entre un nœud donné d'un graphe orienté et tous les autres nœuds de ce graphe. À partir d'un sommet source d'un graphe orienté pondéré par des réels positifs, on construit progressivement un sous-graphe dans lequel sont classés les différents sommets par ordre croissant de leur distance minimale au sommet de départ. La distance correspond à la somme des poids des arcs empruntés.

Nous allons nous intersser à un cas particulier : chercher le plus court chemin entre deux sommets du graphe. Voici l'algorithme en pseudo-code, qu'on appliquera à notre modèle de graphe :

Algorithm 3 Dijkstra

```
function DIJKSTA(Sommet origine, Sommet destination, liste_aretes chemin)
    a_suivre = creer_file_priorisee()
    precedents = vecteur stockant les precedents de chaque sommet du graphe associe au poids du chemin
    distances = vecteur des distances a chaque sommet du graphe
    a_suivre.enfiler (creer_paire (origine, 0)) ;
    while a_suivre n'est pas vide do
        suivant = a_suivre.dépiler() ;
        Si suivant est marque, passer au prochain element de a_suivre
        marquer(suivant) ;
        for all les arete a au depart de suivant do
            d = distances [suiv] + poids(a) ;
            if distances[destination(a)] > d then
                distances[destination de a]=d ;
                precedents[destination(a)]=paire (suiv, poids(a)) ;
                a_suivre.inserer(paire (destination(a), d)) ;
            end if
        end for
    end while
    distance = distances[destination] ;
    while destination != origine do
        chemin.enfiler (paire(destination, poids(precedents))) ;
        destination=premier(precedents[destination]) ;
    end while
    chemin.enfiler(make_pair (origine, 0)) ;
    return distance ;
end function
```