Commencer facilement avec Boost Graph

Guillaume Belz

Version 1

13 décembre 2011

1 Sommaire

2 Que propose Boost Graph ?

Boost Graph (BGL) propose une interface standard pour manipuler des graphes. Cette interface est similaire aux conteneurs de la bibliothèque standard et permet l'accès aux différents éléments à l'aide d'itérateurs. Il est donc possible d'utiliser les algorithmes de la bibliothèque standard pour travailler sur les graphes ou d'utiliser les algorithmes spécifiques pour les graphes fournis par Boost Graph. On peut aussi manipuler les éléments du graphe, accessibles via la classe de traits : les sommets (ou nœuds, *vertex* dans BGL), les arcs (pour un graphe orienté ou arêtes pour un graphe non orienté, *edge* dans BGL) et le graphe lui-même (*graph* dans BGL)

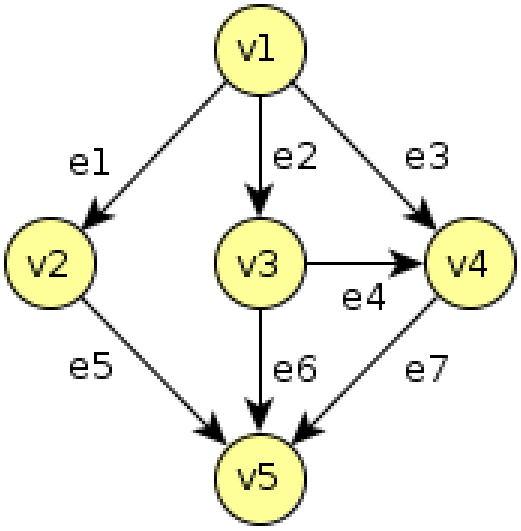

Figure 1: Exemple de graphe avec 5 sommets (v1 à v5) et 7 arcs (e1 à e7) qui sera utilisé tout au long de ce tutoriel

3 Comment débuter avec Boost Graph ?

3.1 Créer un graphe générique

Boost Graph propose de nombreux outils qui nécessite un certain temps pour tout appréhender. Cependant, il est possible d'utiliser la classe *adjacency_list* comme boîte à outils généraliste.

```
#include <boost/graph/adjacency_list.hpp>
```

Les informations relatives à chaque élément d'un graphe sont enregistrées dans des structures :

```
struct VertexProperties { ... };
struct EdgeProperties { ... };
struct GraphProperties { ... };
```

On peut alors créer un graphe générique basé sur *adjacent_list*. Pour des raisons pratique, il est préférable de créer un *typedef* sur cette structure puisqu'elle sera réutilisée plusieurs fois :

```
// définition du graphe
typedef boost::adjacency_list<
   boost::vecS, boost::vecS, boost::bidirectionalS,
   boost::property<boost::vertex_bundle_t, VertexProperties>,
   boost::property<boost::edge_bundle_t, EdgeProperties>,
   boost::property<boost::graph_bundle_t, GraphProperties>,
> Graph;
```

```
// création du graphe
Graph  g;
```

3.2 Comment manipuler les sommets ?

Le type correspondant à la description d'un sommet est accessible via la classe de traits *graph_traits* :

```
typedef boost::graph_traits<Graph>::vertex_descriptor vertex_t;
```

La fonction *add_vertex* permet d'ajouter un sommet dans un graphe. Les informations attachées au sommet peuvent être spécifiées lors de la création :

```
struct VertexProperties
{
   std::string name;
   unsigned id;
   VertexProperties() : name(""), id(0) {}
   VertexProperties(std::string const& n, unsigned i) : name(n), id(i) {}
};
```

```
// appel du constructeur par défaut
vertex_t v1 = boost::add_vertex(g);
```

```
// appel du constructeur avec paramètres
vertex_t v2 = boost::add_vertex(VertexProperties("toto", 12), g);
```

L'opérateur [] permet de récupérer les informations d'un sommet. Il retourne une référence, ce qui permet de pouvoir modifier les informations :

```
// référence constante
VertexProperties const& vertexProperties = g[v1];
std::cout << "Vertex name : " <<  vertexProperties.name << std::endl;
```

```
// référence non constante
VertexProperties& vertexProperties = g[v2];
v2.id = 17;
```

La fonction *num_vertices* permet de connaître le nombre de sommet dans un graphe :

```
// Nombre de sommets
boost::graph_traits<Graph>::vertices_size_type s = num_vertices(g);
```

3.3 Comment manipuler les arcs ?

Le type correspondant à la description d'un arc est accessible via la classe de traits *graph_traits* :

```
typedef boost::graph_traits<Graph>::edge_descriptor edge_t;
```

La fonction *add_edge* permet de connecter deux sommets. Les informations attachées à l'arc peuvent être spécifiées lors de la création :

```
struct EdgeProperties
{
   float weight;
   float distance;
   EdgeProperties() : weight(0.0), distance(0.0) {}
   EdgeProperties(float w, float d) : weight(w), distance(d) {}
};
```

```
// appel du constructeur par défaut
std::pair<Graph::edge_descriptor, bool> e1 = boost::add_edge(v1, v2, g);
```

```
// appel du constructeur avec paramètres
std::pair<Graph::edge_descriptor, bool> e2 =
   boost::add_edge(v1, v2, EdgeProperties(1.0, 50.0), g);
```

L'opérateur [] permet de récupérer les informations d'un arc. Il retourne une référence, ce qui permet de pouvoir modifier les informations :

```
// référence constante
EdgeProperties const& edgeProperties = g[e1];
std::cout << "Edge weight : " <<  edgeProperties. weight << std::endl;
```

```
// référence non constante
EdgeProperties& edgeProperties = g[e2];
e2. distance = 103.8;
```

La fonction *num_edges* permet de connaître le nombre de sommet dans un graphe :

```
// Nombre d'arcs
boost::graph_traits<Graph>::edges_size_type s = num_edges(g);
```

Il est possible de récupérer les descripteurs des sommets associés à un arc à l'aide des fonctions *source()* et *target()* :

```
Graph::edge_descriptor v1 = boost::target(e1, g);
Graph::edge_descriptor v2 = boost::source(e5, g);
if (v1 == v2)
   std::cout << "Same vertex" << std::endl;
```

3.4 Comment supprimer des sommets et des arcs ?

Boost Graph fournit plusieurs fonctions pour supprimer des éléments d'un graphe. Attention, ces fonctions invalident les itérateurs existants.

```
// suppression de tous les sommets et arcs d'un graphe
clear(g);
```

```
// suppression des arcs partant ou arrivant à un sommet
clear_in_edges(v1, g);
clear_out_edges(v2, g);
```

```
// supprimer tous les arcs partant et arrivant à un sommet
clear_vertex(v1, g);
```

```
// suppression d'un arc
remove_edge(v1, v2, g);
remove_edge(e3, g);
```

```
// suppression d'un sommet
// il est nécessaire de supprimer les arcs liés à un sommet...
clear_vertex(v1, g);
// ... avant de le supprimer
remove_vertex(v1, g);
```

3.5 Comment utiliser des algorithmes ?

Boost Graph fournit des fonctions pour récupérer des itérateurs sur les sommets ou les arcs d'un graphe. Il est ensuite possible de les utiliser directement dans les algorithmes de la bibliothèque standard.

```
// Récupérer tous les sommets
std::pair<vertex_iterator_t, vertex_iterator_t> it = boost::vertices(g);
```

```
// Récupérer tous les arcs
std::pair<edge_iterator_t, edge_iterator_t> it = boost::edges(g);
```

```
// Récupérer les arcs partant d'un sommet
std::pair<out_edge_iterator_t, out_edge_iterator_t> it = boost::out_edges(v1, g);

// Récupérer les arcs arrivant sur un sommet
std::pair<in_edge_iterator_t, in_edge_iterator_t> it = boost::in_edges(v1, g);
```

```
// Récupérer les sommets adjacent à un sommet
std::pair<adjacency_iterator_t, adjacency_iterator_t> it =
    boost::adjacent_vertices(v1, g);
```

On peut alors parcourir tous les éléments sélectionnés. Par exemple, pour afficher le nom de tous les sommets :

```
std::pair<vertex_iterator_t, vertex_iterator_t> it = boost::vertices(g);
for( ; it.first != it.second; ++it.first)
   std::cout << get(boost::vertex_bundle, g)[*it.first].name << std::endl;
```

3.6 Quels sont les algorithmes proposés par Boost Graph ?

Breadth First Search	Connected Components
Depth First Search	Strongly Connected Components
Uniform Cost Search	Dynamic Connected Components
Dijkstra's Shortest Paths	Topological Sort
Bellman-Ford Shortest Paths	Transpose
Johnson's All-Pairs Shortest Paths	Reverse Cuthill Mckee Ordering
Kruskal's Minimum Spanning Tree	Smallest Last Vertex Ordering
Prim's Minimum Spanning Tree	Sequential Vertex Coloring

Voir la documentation de Boost Graph pour l'utilisation de ces algorithmes.

4 Aller un peu plus loin avec Boost Graph

4.1 Utiliser la fonction *tie*

La fonction *tie* de *boost/tuple/tuple.hpp* permet de récupérer une s*td::pair* directement dans deux variables :

```
Graph::edge_descriptor e1;
bool succes;
tie(e1, succes) = boost::add_edge(v1, v2, g);
if (succes)
   // using edge e1
```

4.2 Que signifie les paramètres passés à *adjacent_list* ?

adjacency_list est une classe template acceptant plusieurs paramètres permettant de spécifier le comportement de cette classe. Ces paramètres permettent de choisir les types de conteneurs utilisés en interne (*OutEdgeList*, *VertexList* et *EdgeList*), les propriétés associées aux éléments du graphe (*VertexProperties*, *EdgeProperties*, *GraphProperties* et *Directed*).

Voici la liste des paramètres et les valeurs par défaut de *adjacency_list* :

```
adjacency_list<
   OutEdgeList            = vecS,
   VertexList             = vecS,
   Directed               = directedS,
   VertexProperties       = no_property,
   EdgeProperties         = no_property,
   GraphProperties        = no_property,
   EdgeList               = listS
>
```

4.3 Les types des conteneurs

Les types de conteneurs (*OutEdgeList*, *VertexList* et *EdgeList*) peuvent être choisir en utilisant les tags précisés dans la liste suivante :

```
vecS         = std::vector
listS        = std::list
slistS       = std::slist
setS         = std::set
multisetS    = std::multiset
hash_setS    = std::hash_set
```

Le choix du type de conteneur aura une influence sur les performances des algorithmes utilisés. La complexité des fonctions de Boost Graph en fonction du type de conteneur est indiqué dans la page suivante :

http://www.boost.org/doc/libs/1_48_0/libs/graph/doc/using_adjacency_list.html#sec:choosing-graph-type

4.4 L'orientation des arcs

Le paramètre *Directed* permet de préciser si le graphe est orienté ou non et si les arcs seront unidirectionnels ou bidirectionnels.

```
undirectedS         = graphe non orienté
directedS           = graphe orienté avec des arcs unidirectionnels
bidirectionalS      = graphe orienté avec des arcs bidirectionnels
```

En fonction du type de graphe, certaines fonctions ne seront pas disponibles. Par exemple, avec un graphe orienté unidirectionnel, la fonction *in_edges*, permettant de récupérer la liste des arcs entrant, n'est pas utilisable. De même, certains algorithmes ne seront possible quand pour certains types de graphes.

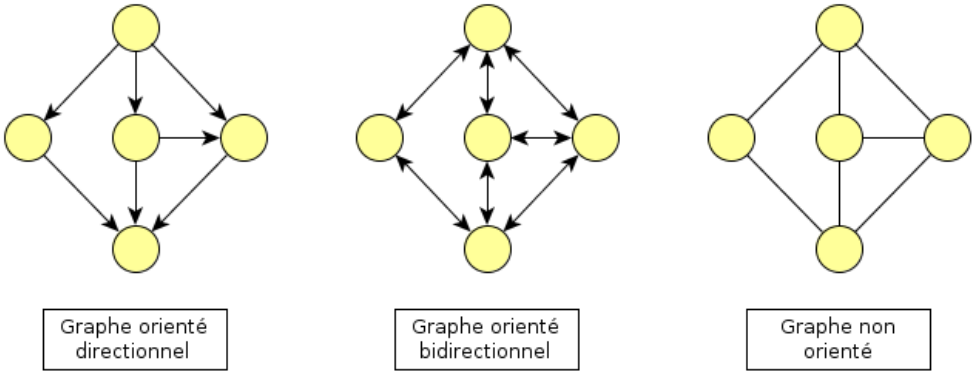

Figure 2: Différents types de graphes

directedS ne permet pas de récupérer les arcs entrants dans un sommet (*in_edges*) d'où l'existence de *bidirectionalS*. Tous deux sont des graphes orientés au niveau conceptuel. Par défaut, on choisit entre *undirectedS* ou *birectionalS* en fonction de la nature du graphe (non orienté / orienté). *directedS* sera une optimisation possible pour consommer moins de mémoire.

4.5 Les propriétés associées

Pour les propriétés associées aux éléments d'un graphe, on peut soit utiliser le mot clé *no_property* pour ne pas associer d'informations, soit utiliser la méthode décrite dans les chapitre 3.2 pour les sommets et 3.3 pour les arcs, soit utiliser une des méthodes décrites ensuite.

Il existe deux approches possible pour enregistrer les informations dans un graphe : utiliser des conteneurs externes (« external property storage ») ou directement dans un graphe (« internal properties »).

4.5.1 Enregistrer les informations dans des conteneurs externes

La première approche utilise Boost *property_map*. Cette bibliothèque fournit une interface générique pour l'accès aux conteneurs associatifs. Les différentes informations sur la *property_map* sont accessible via des classes de traits et des tags. La documentation est accessible en suivant ce lien : http://www.boost.org/doc/libs/1_48_0/libs/property_map/doc/property_map.html.

```
#include <boost/property_map/property_map.hpp>
```

4.5.1.1 Créer une *property_map*

Il existe plusieurs classes dans *property_map* mais nous allons décrire uniquement *associative_property_map* ici. Une *property_map* prend en paramètre un conteneur associatif. Nous

utilisons pour Boost Graph un descripteur de sommet ou d'arcs comme clé :

```
typedef std::map<vertex_t, std::string> names_property_t;
names_property_t names;
boost::associative_property_map<names_property_t> names_map(names);
```

```
typedef std::map<edge_t, float> weights_property_t;
weights_property_t weights;
boost::associative_property_map<weights_property_t> weights_map(weights);
```

4.5.1.2 Accéder aux informations

Pour accéder aux données, on utilise les fonctions *put*, *get* et l'opérateur [] :

```
// avec les fonctions put et get
std::string v1_name = get(names_map, v1) ;
put(names_map, v2, "toto");

// plus directement, avec l'opérateur []
std::string v1_name = names_map[v1];
names_map[v2] = "toto";
```

4.5.2 Enregistrer les informations directement dans le graphe

Il est possible d'utiliser les paramètres template *VertexProperties*, *EdgeProperties* et *GraphProperties* de *adjacent_list* pour enregistrer des informations directement dans le graphe. Chaque information est identifié par un tag, auquel on spécifie un type de données et une valeur.

Par exemple, on peut associer un nom à chaque sommet :

```
// typedef
typedef boost::property<boost::vertex_name_t, std::string> VertexProperties;

// création d'un sommet
vertex_t v1 = boost::add_vertex("toto", g);

// récupérer le nom
std::cout << "Vertex name : " <<  g[v1] << std::endl;

// modifier le nom
v1 = "titi";
```

4.5.2.1 Les tags pré-définis

Il existe un certain nombre de tags pré-définis :

vertex_index_t	edge_residual_capacity_t	vertex_isomorphism_t
vertex_index1_t	edge_reverse_t	vertex_invariant_t
vertex_index2_t	vertex_distance_t	vertex_invariant1_t
edge_index_t	vertex_root_t	vertex_invariant2_t
graph_name_t	vertex_all_t	vertex_degree_t
vertex_name_t	edge_all_t	vertex_out_degree_t
edge_name_t	graph_all_t	vertex_in_degree_t
edge_weight_t	vertex_color_t	vertex_discover_time_t
edge_weight2_t	vertex_rank_t	vertex_finish_time_t
edge_capacity_t	vertex_predecessor_t	

4.5.2.2 Créer ses propres tags

On peut également créer ses propres tags avec le code suivant :

```
namespace boost {
    enum vertex_data_t { vertex_data };
    BOOST_INSTALL_PROPERTY(vertex, data);
}
typedef property<data_t, int> VertexProperties;
```

4.5.2.3 Enregistrer plusieurs informations par élément du graphe

Boost property prend un troisième paramètre template, permettant de chaîner plusieurs propriétés :

```
boost::property <boost::vertex_name_t, std::string,              // nom
    boost::property<boost::vertex_index1_t, unsigned,            // id
        boost::property<boost::vertex_distance_t, float> > >     // distance
```

Dans ce cas, il est nécessaire d'utiliser les fonctions put et get en indiquant le tag utilisé pour récupérer la *property_map* correspondant au tag :

```
// modifier le nom
get(boost::vertex_name_t, g)[v1] = "titi";

// afficher le nom
std::cout << "Vertex name : " << get(boost::vertex_name_t, g)[v1] << std::endl;
```

4.6 Représentation interne et *adjacent_matrix*

La classe *adjacent_list* utilise en interne une liste de sommets aux quels sont associé une liste d'arcs. Il existe une autre structure, *adjacent_matrix*, qui utilise un tableau 2D pour représenter chaque arc, ce qui permet un accès en O(1) aux arcs.

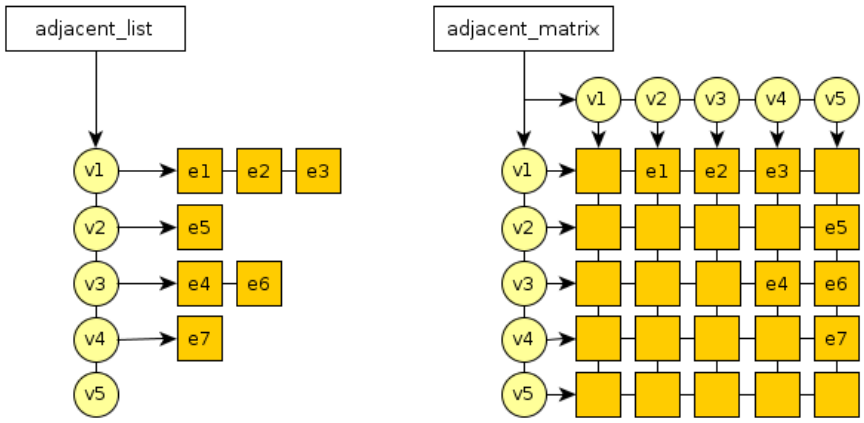

Figure 3: Représentation interne de adjacent_list et adjacent_matrix

5 Références

- http://matthieu-brucher.developpez.com/tutoriels/cpp/boost/graph/implementation/
- http://www.boost.org/doc/libs/1_48_0/libs/graph/doc/index.html
- Boost Graph Library, The: User Guide and Reference Manual de Jeremy G. Siek, Lie-Quan Lee et Andrew Lumsdaine