GRAPHLIB - DESIGN DOCUMENT

INTRODUCTION

Graphs have important applications in areas such as network management, route finding, social network analysis etc. GraphLib is aimed at providing an easy-to-use, maintainable, efficient and elegant C++ library for graph construction and manipulation. The GraphLib is built around a number of unique C++11 features in an attempt to make it a state-of-the-art implementation. The present document discusses the rationale behind GraphLib's underlying design and presents the same elaborately. It also includes a detailed explanation of the library's interfaces.

DESIGN REQUIREMENTS:

The characteristics of a good design are typically listed as (to name a few)

1. Simplicity
2. Efficiency
3. Usability
4. Safety
5. Scalablity
6. Extensibility

GraphLib aims to satisfy as many of the above characteristics as possible. In fact, some of the major design decisions were made by weighing one quality against the other and deciding which is more important and superior in that context.

DESIGN ALTERNATIVE 1:

The first and the simplest design that comes to mind when one thinks of a graph implemented as an adjacency list is using vectors. The idea is to use a vector to store nodes and another to store edges.

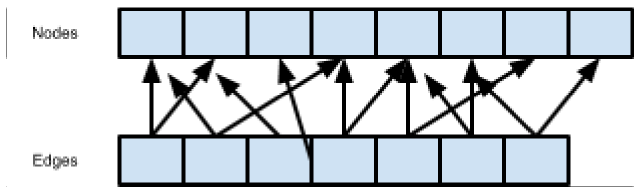


Fig 1: Design Alternative 1 - A vector based approach

The node vector contains the graph nodes as elements and the edge vector contains a pair of pointers to the nodes in the node vector.

PROS:

This is a very simple structure and easy to understand and implement.

CONS:

The simplicity of this design seems to be overwhelmed with other problems. The list below details some of them:

1. There is no easy way to tell if an edge is present or not in the graph. That would require iterating over all edges in the worst case.

2. Accessing adjacent nodes of a given node again requires iterating the entire set of edges which could be really expensive if we are dealing with a dense, huge graph.

3. Deleting an element from a vector is highly inefficient and usually not recommended by the C++ standard. However, deleting a node from a graph is not a very uncommon scenario. If a node which is a vector element is deleted from the vector of nodes, the corresponding edges will suffer from dangling pointers. Any pointer from outside the structure to a node in the graph would be invalidated by the deletion of the node and hence would dangle. This is a very serious problem leading to potential memory leaks and bad memory accesses. This is the biggest drawback of this design.

DESIGN ALTERNATIVE 2:

Since the first design option suffers from obvious and major shortcomings, another alternative is to use a different kind of data structure, like a hashmap, to represent a graph. In fact, a graph could be represented as a hashmap of hashmaps as shown in the figure below:

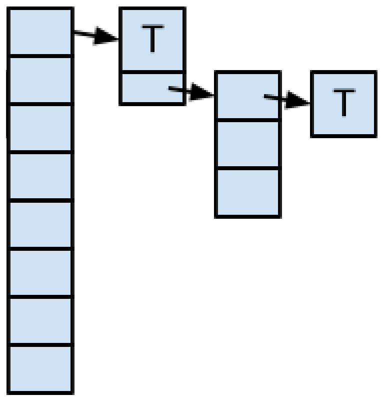


Fig 1: Design Alternative 2 - A map based approach

The first/primary hashmap stores the graph nodes as values based on some key (provided by the user or generated by the library). Every node in the primary hashmap itself contains a secondary hashmap that stores adjacent nodes as values. This way, we do not require a separate data structure to store the edges of the graph.

PROS:

1. Hashmaps have the inherent advantage of extremely fast data access, provided the correct key is known. Adding a node and accessing a node in such a graph takes O(1) time.
2. Finding adjacent nodes to a given node is efficient and requires just two hops - accessing the node and iterating over the hashmap attached to the node.
3. Ascertaining if an edge is present in the graph, which is critical for certain algorithms, is equivalent to finding adjacent nodes of the edge's source node. By the previous argument, this is fast too.

CONS:

1. The biggest drawback of this design is deciding the key of the hash map . It would seem best to use a data type that has built-in hash functions defined, as the key - like int, string etc. While we can assign a unique integer to each successive node that is added to the graph and use that integer as the key, the next question that arises is - "Who supplies that integer to the hashmap - the library or the user? If it is the library that generates an integer automatically for every addition of a node, how would this be communicated to the user? Unless the user knows the key to the map, retrieval cannot be as fast as it could be with a standard hashmap. All the same, asking the user to supply a unique integer with every node is not just inconvenient but highly error-prone.

2. Moreover, when a node is deleted, the whole map has to be searched to find out where that node appears as an adjacent node to another node. This would mean, in the worst case, iterating over the entire hashmap.

3. Hashmaps in general have a linked structure. Every key value pair comprises a link and such links are not stored in contiguous memory locations. This poses serious caching problems especially when the graph is big.

4. Finally, since there is no concrete object representing an edge, it remains a question as to how important edge attributes, like weight would be stored. These attributes are extremely critical to algorithms like shortest path, minimum spanning tree, edge coloring etc.

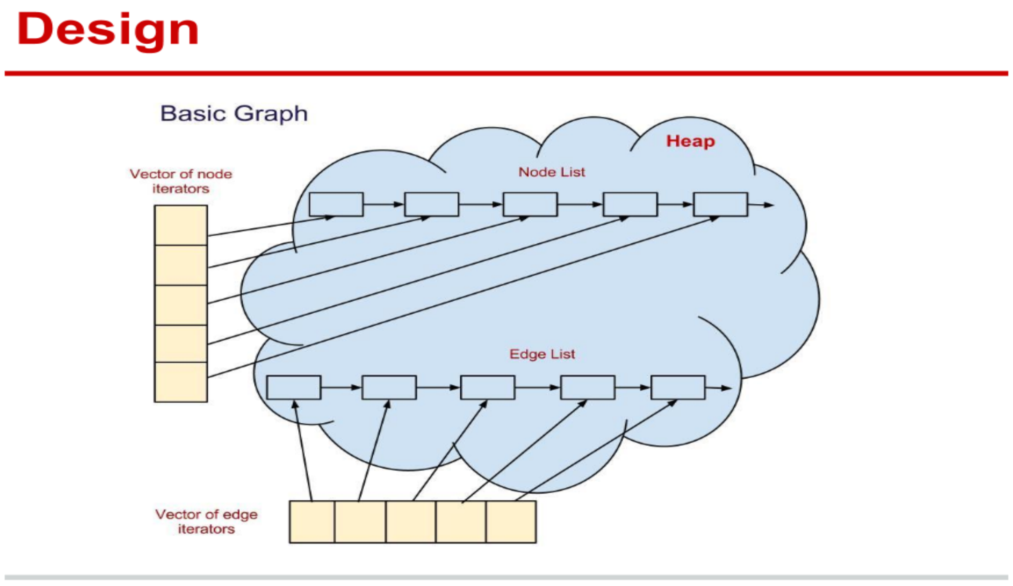
Since the cons seem to outweigh the pros, we abandon this design for a better one, as explained in the next section.

DESIGN OF THE GRAPH LIBRARY

THE GRAPH DATA STRUCTURE

The final design of the graph library is based on a compromise between the simple, cache friendly vector structure and the scalable link structure. In fact, the present design combines the two in a meaningful manner.

Nodes and edges are represented by vectors but the vectors themselves do not contain the actual nodes as in design alternative 1. Instead, the node vector is actually a set of pointers to nodes that are allocated elsewhere, preferably on the heap or even in the cloud. Similarly, the edge vector is a set of pointers to edge objects which themselves contain pointers to the actual node objects in the heap or in the cloud, as shown in the figure below:



By using a hybrid approach, the above design combines the good qualities of the vector based design - simplicity and ease of access; and that of the map based design - scalability via usage of links.

INTERFACE

Graph
template <typename T> class Graph;

Graph is implemented as vector of node ordinals and edge ordinals, with each ordinal storing a pointer to the node or edge information. This allows fast subscript based access and also deals with issues such as vector reallocation.

Constructor

	Graph() Constructs a Graph Object, with no nodes or edges
	Graph(const Graph<T>& g) Copy constructor Throws: invalid_argument{} if arguments not proper.
	Graph(const Graph<T>&& g) Move constructor

	Throws: invalid_argument{} if arguments not proper.

Operations

Graph&	=(const Graph& g2) copy operation
bool	==(const Graph& g2) Checks if the given graphs are equal
bool	==(const Graph& g2) Checks whether given graphs are not equal
Graph&	!() complement of graph

Structure

node_ordinal?	add_node(T n1) add node to the graph. return ordinal
vector <node_ordinal>	add_nodes(vector<T> n) add nodes from vector to the graph. returns vector of ordinals
vector <node_ordinal>	add_nodes(T n1, T n2, ...) add nodes (variadic template) to the graph. returns vector of ordinals
const T&	get_node(node_ordinal) get node object
vector<const T &>	get_nodes() returns vector of node objects.
Graph< T>::node_iterator &	nodes_begin() returns nodes begin iterator.
Graph< T>::node_iterator &	nodes_end() returns nodes end iterator.
void	update_node(node_ordinal, T n) updates node object at the specified ordinal
void	delete_node(node_ordinal) deletes the node.
void	delete_nodes(vector<node_ordinal>) deletes the nodes corresponding to ordinals

edge_ordinal	add_edge(node_ordinal n1, node_ordinal n2) add edge between the 2 nodes of the graph. return edge ordinal
edge_ordinal	add_edge(T n1, T n2) add edge between the 2 nodes of the graph. return edge ordinal
vector <edge_ordinal>	add_edges(vector<pair<node_ordinal, node_ordinal>>) add edges from vector to the graph. returns vector of ordinals
vector <edge_ordinal>	add_edges(pair<node_ordinal, node_ordinal> e1, pair<node_ordinal, node_ordinal> e2, …) add edges (variadic template) to the graph. returns vector of ordinals
void	add_node_attribute(int node_ordinal, string key, string value) adds an attribute to the node, where key denotes the name of the attribute
void	add_node_attribute(int node_ordinal, string key, double value) adds an attribute to the node, where key denotes the name of the attribute
double	get_node_attribute(int node_ordinal, string key) get a node attribute, already added. Raises an exception if key does not exist
string	get_node_string_attribute(int node_ordinal, string key) get a node attribute, already added. Raises an exception if key does not exist
vector<pair<string, double>>	get_node_attributes(int node_ordinal) get all node attributes.
vector<pair<string, string>>	get_node_string_attributes(int node_ordinal) get a node attributes
pair<node_ordinal, node_ordinal>	get_edge(edge_ordinal) get edge object
vector<pair<node_ordinal, node_ordinal>>	get_edges() returns edge iterator
void	delete_edge(edge_ordinal) deletes the edge.
void	delete_edges(vector<node_ordinal>) deletes the nodes corresponding to ordinals
void	delete_node_edges(int node_ordinal) Deletes the edges containing the given node ordinal.

void	add_edge_attribute(int edge_ordinal, string key, string value)

	adds an attribute to the edge, where key denotes the name of the attribute
void	add_edge_attribute(int edge_ordinal, string key, double value) adds an attribute to the edge, where key denotes the name of the attribute
double	get_edge_attribute(int edge_ordinal, string key) get an edge attribute, already added. Raises an exception if key does not exist
string	get_edge_string_attribute(int edge_ordinal, string key) get an edge attribute, already added. Raises an exception if key does not exist
vector<pair<string, double>>	get_edge_attributes(int edge_ordinal) get all edge attributes.
vector<pair<string, string>>	get_edge_string_attributes(int edge_ordinal) get all edge string attributes

Additions in directed graph

size_t	in_degree(node_ordinal) returns in degree of node
size_t	out_degree(node_ordinal) returns out degree of node
vector<node_ordinals>	successors(node_ordinal) returns iterator over successor nodes
vector<node_ordinals>	predecessors(node_ordinal) returns iterator over predecessor nodes

vector<edge_ordinals>	in_edges(node_ordinal)
	return in-edges

vector<edge_ordinals>	out_edges(node_ordinal)
	return out-edges

Additions in Undirected graph

vector<int>	get_edges(int node_ordinal) Returns the ordinals of the edges connected to the given node.
vector<int>	neighbors(int node_ordinal) Returns the ordinals of the neighbouring nodes.
vector<std::pair<int, int> >	neighbors_with_edges(int node_ordinal) Returns the ordinals of the neighbouring nodes along with the connecting edge ordinals.

int	degree(int node_ordinal) Returns the count of the edges that have the given node as one of its end points.

Algorithms

Traversals

vector<node_ordinals>	breadth_first_traversal(graph) returns breadth first traversal of nodes starting from first possible node ordinal.
vector<node_ordinals>	breadth_first_traversal(graph, node_ordinal) returns breadth first traversal of nodes starting from the given node ordinal.
vector<node_ordinals>	depth_first_traversal(graph) returns depth first traversal of nodes starting from first possible node ordinal.
vector<node_ordinals>	depth_first_traversal(graph, node_ordinal) returns depth first traversal of nodes starting from the given node ordinal.

Path

bool	is_path_exists(graph, node_ordinal, node_ordinal) returns true if path exists between the specified nodes, false otherwise
vector<int>	get_simple_path(graph, node_ordinal, node_ordinal) returns the vector of nodes in the simple path between the given nodes
std::vector<int>	get_shortest_path(T g, int node_source, int node_target) returns the vector of nodes in the simple path between the given nodes
vector<Graph<T> >	get_connected_components(graph) returns a vector of graphs each of which is a distinct connected component in the given input graph; best case size of the vector is 1, worst is |V|.

Spanning Tree

std::vector<int>	get_minimum_spanning_tree(Undirected graph) returns vector of edges that form a minimum spanning tree in the given graph, by running prim algorithm (Assumes that the graph be connected).

Topological Sort

std::vector<int>	topological_sort(DirectedGraph<T> g) Returns the topological ordering of the nodes in the given graph. If the order is not possible, exception message is printed and and empty vector is returned.

Bipartite/Matching

bool	is_bipartite(T Graph) returns true if the input directed/undirected graph is bipartite; false, otherwise.
pair<vector<int>, vector<int> >	get_bipartite_nodes(T Graph) returns the bipartite matching as 2 groups of node ordinals.
vector<edge_ordinals>	get_maximum_matching(graph) returns a vector of edges (that are a maximum subset of those in the input graph such that no two edges have the same endpoint.)

Subgraph/Isomorphism

A graph H is a subgraph of a graph G, if G contains H or an isomorphic version of H. So far, so good. The problem however, is with subgraph isomorphism which is an NP complete problem.
So we should discuss more on this.

Drawing

void	print_as_ascii(graph, stream = cout) prints the given graph to the given stream (which by default is cout) in an ascii fashion. (a number of lines where each line contains a node ordinal and a comma separated list of incident edge ordinals).

Export

bool	export_to_csv(graph, filename) returns true is the input graph was successfully saved to the given 'filename' as csv, false otherwise. A csv for a graph would be just a listing of all nodes with the incident edges in a comma separated fashion.

Link Analysis(?)

Random Graphs(?)

Sparse Matrices(?)

V1.2
Max flow

map<edge_ordinal, float>	get_maximum_flow(graph) returns a map of every edge to the final flow on the edge (represented as float). Note that a graph with multiple sources and multiple sinks might need some modifications before running the max flow algorithm.
bool	is_maximum_flow(graph, map<edge_ordinals, float>) returns true if the given flow (mapping of every edge to a rational number) is a maximum flow possible in that graph; false, otherwise

Connected Components

bool	is_strongly_connected(graph, node_ordinal, node_ordinal) returns true if the given nodes are strongly connected in the given graph, false otherwise. Note that one way to implement this is to call is_path_exists(n1, n2) & is_path_exists(n2, n1).
vector<node_ordinals>	get_connected_component_of_node(graph, node_ordinal) returns a vector of nodes that are in the same connected component as the input node in the given graph.
	is_connected.

Spanning tree
Connected Components

bool	is_strongly_connected(graph, node_ordinal, node_ordinal) returns true if the given nodes are strongly connected in the given graph, false otherwise. Note that one way to implement this is to call is_path_exists(n1, n2) & is_path_exists(n2, n1).
vector<node_ordinals>	get_connected_component_of_node(graph, node_ordinal) returns a vector of nodes that are in the same connected component as the input node in the given graph.