gcore: A small, simple, generic and robust graph library

Anton Nefedenkov

1. Introduction

The gcore is a very small graph library that provides two implementations: adjacency list and adjacency matrix. An important idea behind gcore is separation between implementation and idea, thus the algorithms provided by gcore are agnostic to the implementation of the graph they are run on. Generality of gcore starts from the two implementations that are provided by the library, but it by no means ends there. The user is allowed to provide their own implementation for a graph, and as long as this implementation has the required functions, all the algorithms will behave as expected. This way, the user is given not only the flexibility of custom and safe optimization, but also semantic flexibility. One could for example provide a probabilistic implementation for a graph to achieve probabilistic behavior of the algorithms. More specifically, say a node in the graph has probabilistic edges. All what the user needs to do is to override the "neighbors" function and voilà!

The library is written in modern C++, and it was my intention for the user to be encouraged to use the newer features of C++ (more on how this is encouraged in the tutorial). The library makes use of concepts, which is a feature available in GCC6 to save the user from the pain of template error messages.

Before delving into the details of design, I would like to present the vision that I have for graphs. This vision has heavily influenced my design decisions.

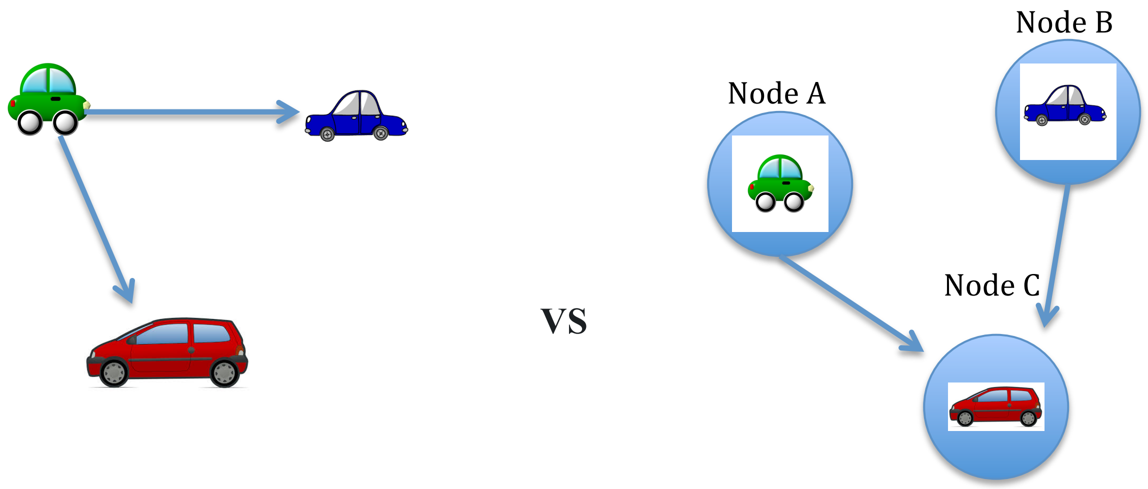
2. What is a Graph? What is a Node? What is an Edge?

Lets discuss the theory a little. A graph $G = (V, E)$ where V is the set of nodes, and E is the set of edges. So however we decide to implement a graph, that implementation should have a routine to *add_node* and *add_edge*. Now consider the situation where we have $V' \subset V$ and we have another graph $G' = (V', E')$. We say that $G \; and \; G'$ share nodes V'. In the world of implementation, the idea of sharing a node translates to not having separate copies i.e $G \; and \; G'$ have access to some shared memory. This means, that when *add_node* adds a node to a graph, it should not make a copy of that node. Not only does this save us memory, but it also has correct semantics. At the same time, if one removes a node from G', this should not impact the structure of G.

In other words, nodes are just points on a plane, that should be able to exist without being part of any graph, just like it makes sense to talk about the set V without any mention of $G \; or \; G'$. Graphs in turn, should be defined on top of these nodes, namely by providing information on connectedness between the nodes that are part of a given graph. Notice, that given $u \notin V \; and \; v \in V$ it is completely senseless to ask if $u \; and \; v$ are connected for two reasons. To ask this question we have to be in the context of a graph, because that is what stores the information about connections. Lets say we have a graph $G = (V, E)$ like before, and we ask G this question. Asking this question is still senseless, because for what G knows, node u does not exist in its universe. To sum up, question of connectedness of $u \; and \; v$ in graph $G = (V, E)$ can only be asked if and only if $u \in V \; and \; v \in V$. These properties, again, give very strong suggestions on the design of the library.

We now have a good level of understanding of the expected semantics for the future graph and node objects. What about edges? Intuitively, for an edge to be an edge it has to possess three pieces of information: its source node, its weight and its destination node. Just like in the case of nodes, talking about a set of edges E independent of any graph should also make sense. Elements of such set describe a relationship between two nodes. We should be able to get hold of an edge e and add that edge to graphs $G, G' and \; G''$ for instance. Crucial observation is that *add_edge(G, e)* only makes sense if both source node and destination node of *e* are already elements of $V\ with\ G = (V, E)$. Trivially, we also observe that there is nothing wrong with a graph $G^* = (V, \emptyset)$, a graph that is only made of nodes. While all the points discussed above have little controversy in them, the next and final observation is my personal leap of faith.

When looking at popular graph libraries I noticed a both intuitive and counterintuitive feature. It is often the case that that the routine of node addition has the following form: *add_node(G, "A")*. And here is my leap of faith: I do not believe that this is semantically sound. I have looked for a good argument, but all I have is a gut feeling. The idea of graphs being made of nodes and edges is so entrenched in my head, that having a graph made of *characters* and edges, *numbers* and edges or *cars* and edges just does not sit right with me.



Graphs are made of nodes, not of cars!

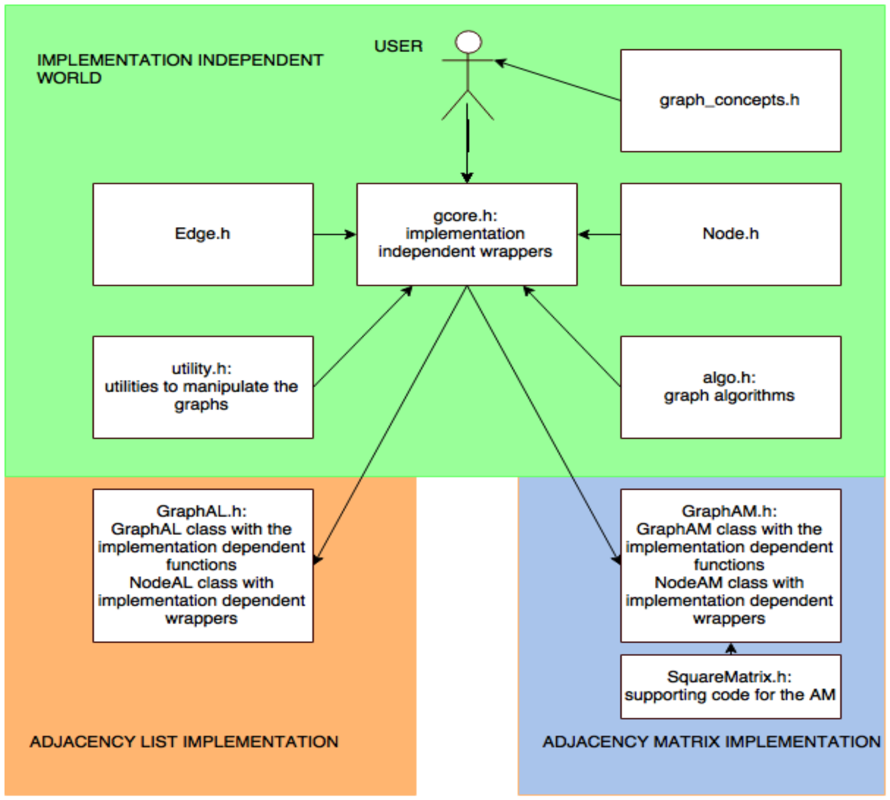
3. Library Architecture Overview

The discussion above has directly impacted the design of the library. There are only two user facing classes: Node and Edge. Both of these can only be created through a call to their respective creation routines. The creation routines make an allocation of the object on the heap, and return a smart pointer to the object. The idea is that there should be no way for the user to get her hands on Node or Edge without going through the creation routine.

There is a third creation routine: the *create_graph*. Same thing as for Node creation – pointer to a heap allocated instance. What is interesting is that one of the template arguments to this routine will specify the implementation that is to be used with this graph. So if the user intends for the new graph to be complete, the adjacency matrix implementation represented by GraphAM class is the way to go. If the graph is going to be sparse, using adjacency list GraphAL class is a better call. At the same time, if the user wants some custom optimized implementation, she can provides it to the *create_graph* just like she would GraphAL or GraphAM.

There is a set of routines that form basis for graph operations. GraphAL and GraphAM of course provide all of these as member functions. The implementions in fact provide more, simply because when some operations are made implementation dependent, even though they do not strictly have to be, a performance boost is guaranteed. One example is *get_edges* that can easily be implemented in terms of basis functions, but very inefficiently. That said, each implementation provides a grand total of 14 member functions that are used by all the algorithms of gcore. So it seems like a nice trade-off between performance and comfort. The only minus, is that if user wants to use their own implementation, they will have to provide all 14 implementation dependent functions.

4. Library Architecture Details


IMPLEMENTATION INDEPENDENT WORLD
USER
graph_concepts.h
Edge.h
gcore.h: implementation independent wrappers
Node.h
utility.h: utilities to manipulate the graphs
algo.h: graph algorithms
GraphAL.h: GraphAL class with the implementation dependent functions NodeAL class with implementation dependent wrappers
GraphAM.h: GraphAM class with the implementation dependent functions NodeAM class with implementation dependent wrappers
SquareMatrix.h: supporting code for the AM
ADJACENCY LIST IMPLEMENTATION
ADJACENCY MATRIX IMPLEMENTATION

gcore.h:

This is the main user-facing file. As mentioned previously it provides the three creation routines:

```
shared_ptr<Node<IdType, DataType>> create_node(IdType id, DataType* data);
shared_ptr<Edge<IdType, WeightType, DataType>> create_edge(
	const shared_ptr<Node<IdType, DataType>> src,
	const WeightType w,
	const shared_ptr<Node<IdType, DataType>> dst);
shared_ptr<GraphType<I, W, D>> create_graph();
```

If the source code is examined, we see how the last function does nothing but calls an implementation specific creation function. This is a pattern that holds true for the rest of the file. Each of the 14 implementation dependent functions have an implementation independent wrapper located here. Finally, some useful operators on smart pointers are defined here, since all of the algorithms provided by the library ultimately manipulate the smart pointers only. All of these wrappers operate on the smart pointers so the semantics discussed in the previous section are reflected in the implementation.

algo.h and utility.h:

One contains the simple algorithms that can be called by the user, while the other provides useful utilities such as *make_undirected_from* routine that takes a directed graph and makes it undirected by adding a duplicate edge when only single directed edge exists. In situations when two directed edges exist, it makes a decision on the weight of the undirected edge based on the passed in *combine* function.

graph_concepts.h:
This file contains few concepts to ensure safe operation of the library, as well as to protect the user from the template errors.

Node.h and Edge.h:
These files contain the respective classes. As a reminder, Node is a class that the user is forced to use to be able to gain access to the library. User must put his objects into Node's first, and then the graph is built on top of these instances. Class Edge gives user the ability to express the idea of connection between two nodes. Edge has three private member variables:

```
shared_ptr<Node<IdType, DataType>> src;
shared_ptr<Node<IdType, DataType>> dst;
WeightType w;
```

This class is protected by a concept that requires WeighType to be Numeric and the IdType to be Comparable. The restriction on the WeightType is not strictly necessary, but since the weight of the edge is passed by value, it seemed beneficial to impose limitations on what a weight can be.

More detail on how to use the interface of the library is provided in the tutorial.

GraphAM.h and GraphAL.h:
These files contain the two implementations provided by the library. I will discuss the specifics of the implementations in the next sections, but what seems to be important to mention, that the member functions in these files are exactly the ones called by the implementation independent wrappers of gcore.h.

5. Adjacency List Implementation: GraphAL class

The semantics described in Section 2 turned out to be quite difficult to implement. Since we want to be able to build multiple graphs on the same set of Node objects, the graph object must be able to keep track of which Node objects are part of the graph without owning the Node objects at the same time. This setup required introduction of a supporting wrapper class NodeAL; instances of these class are going to be managed by a given graph object.

Whenever a Node is added to the graph, a NodeAL is created. The wrapper stores the smart pointer to the underlying node and some auxiliary information to assist performance. Pointer to the NodeAL wrapper is what is ultimately added to the adjacency list. To differentiate between different NodeAL objects of the graph, every NodeAL is assigned a unique integer id at creation. To avoid running out of ids, id recycling is provided by the GraphAL. When a Node is removed from the graph, its NodeAL wrapper is deleted and the id is recycled. Recycling is implemented by keeping around a vector of returned ids.

It is clear that at this point there is a necessity to be able to go from NodeAL to Node and from Node back to NodeAL. The former is easy because NodeAL stores a smart pointer to the underlying Node. The latter requires maintenance of a mapping between Node id of type IdType to the integer ids of the wrapping NodeAL. To achieve this I have used the following map:

```
std::map<IdType, NodeAL<IdType, WeightType, DataType>*> id_map;
```

Let us now take a look at how the adjacency list data structure is actually implemented. In its classic form, and adjacency list is a list of lists of nodes, where each list corresponds to the neighbors of the given node. I have decided to implement the data structure differently to benefit from the existence of NodeAL wrapper objects. Here is the member variable:

```
vector<NodeAL<IdType, WeightType, DataType>*> adjacency_list;
```

I am using a simple vector of pointer to the wrappers. When new Node is added, a new NodeAL is created and pushed back into the vector. The internal id a.k.a the integer id of NodeAL object is the index into this vector. But how does one extracts the neighbors of a certain Node? The information about the neighbors of a Node is actually stored directly in its wrapper. Here is the member variable of NodeAL responsible for this:

```
vector<pair<NodeAL<IdType, WeightType, DataType>*, WeightType>> neighbours;
```

Notice how the vector stores a pair instead of storing the actual Edge objects. This saves us memory because we already know the source node, so we only need to store the neighbor Node and the weight of the connecting edge. As an aside, the use of a naked pointer might not seem too prudent. The truth is that NodeAL objects are exclusively created as part of *add_node* routine, and deleted in only two places: the destructor and the *remove_node* routine. So it seemed beneficial for the performance to avoid the usage of smart pointers for this particular object.

6. Adjacency Matrix Implementation: GraphAM class

The adjacency matrix implementation proved to be trickier than the adjacency list implementation. One important factor that jumps immediately to our attention is the management of the actual matrix. Implementing it as a vector of pointers to vectors did not seem like a prudent idea. Whenever an element is added to the matrix, the matrix has to be grown in both dimensions. I have considered few matrix manipulation libraries, but they did not seem to provide exactly what I needed. So I ended up writing a supporting class SquareMatrix.

SquareMatrix is a simple class that supports only few operations. The square matrix is represented as follows:

```
EntryType * entry;
```

The constructor of the square matrix allocates space for a certain small number of elements. When matrix needs to be grown, bigger space is allocated, old array copied and the deleted. Notice how this is a one-dimensional array. That is of course not a problem.

Just like the adjacency list implementation, GraphAM class has a NodeAM class that serves as an implementation dependent wrapper and plays a very similar role. For example, indexing into adjacency matrix is done on the basis of the integer id of the wrapper. Just like in GraphAL, GraphAM has a map connecting the Node id to the wrapper. The only difference that stands out is the necessity of a second map, from internal id's of wrappers to the actual NodeAM objects:

```
map<int, NodeAM<IdType, WeightType, DataType>*> wrapper_map;
```

It becomes apparent why this map is necessary if we consider what happens when we call neighbors routine on a Node. The function takes a Node as an input, so we proceed to finding the wrapper using the id_map described previously. This way we get our hands on the wrapper, and therefore on its internal id, and therefore on the row where the Node is represented in the graph. Now we need to walk through each entry in that row, and if the entry is not zero, we know that our input is adjacent to the Node represented by the given column. This is where we need the map to go from the column index to the NodeAL object, so we can in turn get the underlying Node object.

7. Tutorial

Lets create few nodes to get ourselves started:

```
auto n1 = create_node<string, int>("A", nullptr);
auto n2 = create_node<string, int>("B", nullptr);
auto n3 = create_node<string, int>("C", nullptr);
auto n4 = create_node<string, int>("D", nullptr);
```

Notice how the **auto** keyword is used through out the tutorial. The types of the objects returned by the interface functions are fairy complicated, so not only does the keyword alleviates one's frustration, but also hides from the eye what is better not seen or reasoned about. Lets **create a graph** now and **add our nodes** to it:

```
auto g1 = create_graph<string, int, int, GraphAL>();
// Add nodes to the graph
add_node(g1, n1);
add_node(g1, n2);
add_node(g1, n3);
add_node(g1, n4);
g1->print_graph();
```

Note how we passed in GraphAL as a template parameter to the graph creation. The graph is now implemented as an adjacency list. Also not the print_graph() member call: this printer is there solely as a sanity check. Lets **add few edges** to our graph.

```
// Add edges
add_edge(g1, n1, 1, n3);
add_edge(g1, n1, 1, n2);
add_edge(g1, n4, 1, n3);
add_edge(g1, n3, 1, n4);
```

Easy. Now lets **remove a node**. Notice how at node removal, all incoming edges and outgoing edges from the removed node are also deleted automatically.

```
remove_node(g1, n3);
g1->print_graph();
// Add it back
add_node(g1, n3);
```

Similarly, we can **remove the edge** between two nodes specifically:

```
remove_edge(g1, n1, n2);
```

But we can also **manipulate Edge objects**. Let us try that out by creating an Edge object first and then adding it to our graph.

```
auto e1 = create_edge<string, int, int>(n2, 7, n3);
add_edge(g1, e1);
g1->print_graph();
```

We can always check **adjacency** of two nodes:

```
cout << adjacent(g1, n2, n3) << endl; // Yes it is
cout << adjacent(g1, n3, n2) << endl; // No it is not, why?
```

Very often we want to get our hands on the **neighbors** of a Node:

```
auto v = neighbours(g1, n2);
print_nodes(v);
```

Notice another useful printer function for sanity checks. This one prints all nodes in a vector. At other times, it is useful to **get all the nodes** that are part of a graph.

```
auto pariticipants = get_nodes(g1);
```

We can do exactly the same thing to **get all edges** of a graph:

```
auto edges = get_edges(g1);
print_edges(edges);
```

Lets take a look at **manipulating the Node object** now. Even though the Id of a Node is immutable, the data pointer is not. So we can reassign it like we want.

```
int a = 30;
n1->set_data(&a);
a = 15;
assert(a == *n1->get_data());
```

Since graphs are only aware of the Node and not of the underlying user owned object, mutations of the object do not influence the structure of the graph. At the same time, if a user defined algorithm somehow mutates the user owned object, all graphs have immediately the updated information.

Often we want to query the graph if a Node or an Edge is part of the given graph:

```
cout << has_node(g1, n4) << endl; // Yes
remove_node(g1, n4);
cout << has_node(g1, n4) << endl; // No
cout << "Lets check if there is an edge (A, 1, C)?\n";
auto e2 = create_edge<string, int, int>(n1, 1, n3);
cout << has_edge(g1, e2) << endl;
cout << "Maybe (C, 1, D)? Remember we removed D\n";
auto e3 = create_edge<string, int, int>(n3, 1, n4);
cout << has_edge(g1, e3) << endl;
```

Just like we get the neighbors of a node, we can also get the edges of a node:

```
auto edges_of = edges_of_node(g1, n2);
print_edges(edges_of);
```

Lets try **an algorithm**. Lets ask gcore to generate the dfs trees for some nodes:

```
// Run DFS
cout << "Print DFS rooted at A\n";
auto dfsn1 = dfs(g1, n1);
dfsn1->print_graph();

cout << "Print DFS rooted at B\n";
auto dfsn2 = dfs(g1, n2);
dfsn2->print_graph();
```

Often times we don't really want to add node after node after node. It is nice to **add an entire vector of nodes** directly to the graph.

```
// Build a graph from multiple edges and nodes (make sure to add nodes first`)
cout << "Lets build a graph from edges and nodes of g1\n";
auto node_v = get_nodes(g1);
auto edges_v = get_edges(g1);
auto g2 = create_graph<string, int, int, GraphAL>();
add_nodes(g2, node_v);
add_edges(g2, edges_v);
g2->print_graph();
```

As you can see, the library provides sufficient flexibility to satisfy the needs of the user while maintaining very robust state.

8. Running the code

To compile the code requires **GCC6**.
The following compilation flags are a must: **-fconcepts -std=c++1z**

9. Future Work

In future, there is a lot of work that has to be done to make gcore usable. Most importantly, it is quintessential to add a range of utility functions that would allow importing graph data from a file. Without this, nothing realistic can be done using gcore. Next, it is necessary to significantly expand the algo.h file: populate it with an array of basic algorithms that would allow effortless construction of more complicated routines. From the newly gained knowledge, it is then important to optimize the two implementations provided by the library for speed; some of the code can be rewritten to better adhere to the C++ Core Guidelines in the process.

10. References

The small cars:
http://www.clker.com/clipart-blue-car-very-small.html
http://www.inautonews.com/top-10-green-cars-to-watch-in-2010
https://pixabay.com/en/red-car-vehicle-automobile-small-34832/