Engineer Thesis

Graph implementation and its impact on algorithm performance

Krzysztof Watras

Computer Science

Politechnika Warszawska

Introduction

Graph theory is crucial to many fields of life. [...]

Index of all names I will use in this project:

Vertex	also known as a node, state or point.
Edge	also known as a path, route or link.
Graph	comprised of Vertices and Edges.
Something else	[...]

1 Key considerations

Although graphs in general have many things in common, they are also significantly different from each other. This makes it incredibly difficult to make a proper, general graph that is efficient for all cases that have to be covered. This is especially true when we consider all possible graph types and their combination. In particular the hard part is to implement three types of graph as one class:

1. unweighted graph
2. weighted graph
3. directed graph
4. undirected graph
5. transition graph

Traditionally, graphs are represented via Adjacency Matrix, Adjacency List or Transition Table in case of transition graphs. However, implementing all five types of graph with either one of those representations is non-trivial and is subject to much consideration. Not to mention, it is possible that some graph could be more of one type. For example, weighted directed graph is both type 2 and 3.

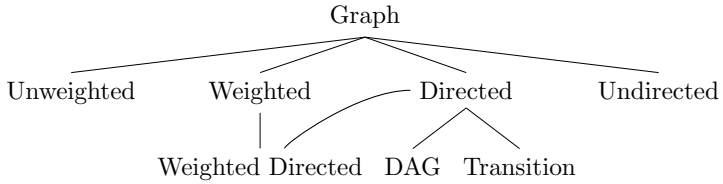
One must also decide whether to implement graphs as an abstract interface that can inherit base behaviour from other graph definitions or should the graph be strongly coupled with its representation. While the former solution is better in terms of general extend-ability with extra features for some special kind of graph that we did non think about before, the latter provides much better performance because as developers of both interface and underlying memory layout we can do assumptions that will make our code run faster and use less memory. On the other hand, in case where the graph we want to represent with code is significantly different from one expected to be general case we may find that decision of making a general interface of methods instead of enforcing specific data layout may allow us to create special graph just for this very specific application. For example, if we know that the graph we work on is a dense graph or even complete graph then it is beneficial to be able to define our own data representation and only use the general api for specific roles.

2 Chosen representation

In my graph representation I choose to create an **abstract interface** for graphs and a **specific implementation** that will inherit the algorithms to test the performance difference between different physical implementations.

3 Requirements

In graph theory we have different graph types, each having its specific application, but also, extra set of features that enable us to perform new operations on it. Below is the tree of graph types that I want to cover in this paper, as well as their mutual dependence.



As we may soon find out, there is not that many functionalities added on each step compared to base Graph class. For example when we use *Undirected Graph* instead of *Graph* we may find that we gained absolutely no additional functionality. However, this approach may give us way to optimise performance of our code because now we can do optimisations to both algorithms and data structures, thanks to assumptions we just made. That being said, it would be wise not to implement them as another abstract class but as a concrete class. This way, we reduce depth of dependencies and make naming our classes far easier.

Our graph will be implemented in c++. Thus, it makes sense to add iterator pattern to make the graph easily extendible with standard library functions like *std::find()*.

3.1 Allowing for abstract graphs

One way of generalising graph to any type of graph is by introducing *Relation* abstraction. This *Relation* class would abstract away all differences between graph types, and allows us to specify the concrete relations between Vertexes when concrete graph needs to be implemented. More concretely, if we have a simple Edge from p to q, then we can create following *Relation* to represent that.

```
class Relation
    int from
    int to
```

If we were to have another graph that has weights on each node then we can simply add *weight* as an attribute of the relation.

```
class Relation
    int from
    int to
    int weight
```

In our code we may want to sort relations to enable faster lookup times. Thus, we may say that

$$\begin{aligned} &\text{Relation } r_1 < \text{Relation } r_2 \\ &\text{iff} \\ &(r_1.from < r_2.from) \vee ((r_1.from = r_2.from) \wedge (r_1.to < r_2.to)) \end{aligned} \tag{1}$$

3.2 Methods to be implemented

After we know what type of project we deal with, it is time to develop the interface. I will start with defining the base *Graph* class methods and move on to derived classes. In particular, I will specify what additional methods I will add and which ones I will modify.

Graph constructor should allow to build graph from:

- empty input
- number of vertexes
- list of vertexes and list of relations

Graph destructor should safely dealocate all memory used by the graph and all children of the graph. Graph copy constructor as well as assigment operator should perfom a deep copy of all graph members and all children of those graph members.

3.2.1 Graph

- constructor
- destructor
- copy constructor
- assigment operator
- import_from(File) → Graph
- save_to(File)
- add_Vertex(Vertex)
- remove_Vertex(Vertex)
- insert_relation(Relation) → int
- remove_relation(Relation) → int
- get_vertex_count() → int
- get_edge_count() → int
- get_neighbour_count(Vertex) → int
- get_neighbours() → List[Relation]
- is_in_relation(Vertex, Vertex) → bool
- find_shortest_path(Vertex, Vertex, Strategy) → List[Relation]
- color_graph() → ?
- get_max_flow(Vertex, Vertex) → int
- get_max_flow_path(Vertex, Vertex) → Tuple[int, List[Relation]]
- find_eulerian_path() → List[Tuple[Vertex]]
- find_eulerian_circuit() → List[Tuple[Vertex]]
- find_bridges() → List[Relation]
- find_articulation_points() → List[Vertex]
- calculate_cartesian_product() → Graph
- calculate_tensor_product() → Graph

3.2.2 Directed Graph

- find_topological_order() → List[Tuple[Vertex]]
- find_cycles() → List[Tuple[Vertex]]
- find_strongly_connected_components() → List[Tuple[Vertex]]
- traveling_salesman() → List[Relation]

3.2.3 Weighted Graph

- find_minimum_spanning_tree() → ?

3.2.4 Weighted Directed Graph

- find_negative_cycles() → List[Relation]

4 Implementation

5 Tests

6 Results

7 Conclusions