Assignment 3 - Generic Directed Weighted graph (GDWG)

Change Log

- 17/07: erase(const N& n) changed to DeleteNode(const N& n)
- 17/07: GetNodes() and GetConnected() added
- 18/07: GetConnected() now says sorted by nodes (instead of edge - which made less sense)
- 20/07: References to graph_const_iterator and non-const iterators removed
- 20/07: rbegin and rend shouldn't return const_iterators - they should return const_reverse_iterators
- 23/07: GetConnected clear that is only for outgoing nodes
- 27/07: operator<< schematic output commas removed to match example
- 27/07: rbegin() and rend() more clearly return const_reverse_iterator
- 27/07: fixed operator<< example for outbound edges from node 3
- 28/07: Removed reverse to iterator invalidation in erase function
- 29/07: Spacing between now and open paranthesis clarified in operator<<
- 30/07: GetWeights direction clarified
- 31/07: Clarification: If there are multiple edges between node A and node B, then the result of GetConnected() of A will only have one node B in it's vector array
- 31/07: Clarification: Example data struture updated from set to map
- 31/07: Clarification: GetConnected() on a src node only returns the nodes connecting to it via src's OUTGOING edges
- 01/08: Clarification: crend() now has a correct return type of const_reverse_iterator
- 02/08: Clarification: erase(N, N, E) now states that if the edge isn't present it returns false.
- 02/08: In section 3.5 (Edge Iterator) the type has been updated from **gdwg::Graph<N, E>::const_iterator** to **gdwg::Graph<N, E>::const_iterator**
- 02/08: Phrase "If an exception is not thrown" added to bool Replace function
- 03/08: Find now says "Returns an iterator to the found edge" instead of "Returns an iterator to the found node"
- 04/08: Explanation about how an edge iterator would print out N N E error fixed

1. Overview

Write a "Graph" Class Library in C++, with its interface given in `graph.h` and its implementation in `graph.tpp`.

1.1. Aims

- Smart Pointers
- Exception Handling
- Lambda Functions
- Custom Iterators
- Function and Class templates
- Operator Overloading
- Friends
- Namespaces

1.2. Description

In this assignment, you will write a Generic Directed Weighted graph (GDWG) with value-like semantics in C++. Both the data stored at a node and the weight stored at an edge will be of generic types. Both generic types may be different. For example, here is a graph with nodes storing std::string and edges weighted by int:

```
gdwg::Graph<std::string,int> g;
```

Formally, this directed weighted graph G=(N,E) will consist of a set of nodes N and a set of weighted edges E. Give a node, an edge directed into it is called an *incoming edge* and an edge directed out of it is called an *outgoing edge*. The *in-degree* of a node is the number of its incoming edges. Similarly, the *out-degree* of a node is the number of its outgoing edges. Given a directed edge from src to dst, src → dst, src is the *source* node and dst is known as the *destination* node. G is a multi-edged graph, as there may be two edges from the same source node to the same destination node with two different weights. However, all nodes are distinct, as they contain different data values.

All nodes in the graph are unique.

Between any two nodes (or a node and itself), all edges must be unique

2. Your task

2.1 Constructors & Destructors

Name	Constructor	Description and Hints	Examples
Default Constructor	`gdwg::Graph<N, E>`	The default constructor for a graph.	`gdwg::Graph<N, E> a;`

Constructor	`gdwg::Graph<N, E>(`
	`  std::vector<N>::const_iterator,`
	`  std::vector<N>::const_iterator`
	`)`
		Takes the start and end of a const_iterator to a std:vector<N> and adds those nodes to the graph.	`std::vector<std::string> v{"Hello", "how", "are"`
			`gdwg::Graph<std::string, double> b{v.begin(),v.e`

Constructor	`gdwg::Graph<N, E>(`
	`  std::vector<std::tuple<N, N, E>>::const_iterator,`
	`  std::vector<std::tuple<N, N, E>>::const_iterator`
	`)`
		Iterators over tuples of (source node, destination node, edge weight) and adds them to the graph. You are responsible for creating nodes if they don't exist as you iterate through.	`std::string s1{"Hello"};`
			`std::string s2{"how"};`
			`std::string s3{"are"};`
			`auto e1 = std::make_tuple(s1, s2, 5.4);`
			`auto e2 = std::make_tuple(s2, s3, 7.6);`
			`auto e = std::vector<std::tuple<std::string, std::`
			`gdwg::Graph<std::string, double> b{e.begin(), e.en`

Constructor	`gdwg::Graph<N, E>(std::initializer_list<N>)`	A constructor that takes an initialiser list of Nodes to populate the graph.	`gdwg::Graph<char, std::string> b{'a', 'b', 'x', 'y`
Copy Constructor	`gdwg::Graph<N, E>(const gdwg::Graph<N, E>&)`		`gdwg::Graph<std::string, int>` `gdwg::Graph<std::string, int> aCopy{a};`
Move Constructor	`gdwg::Graph<N, E>(gdwg::Graph<N, E>&&)`		`gdwg::Graph<std::string, int> a;` `gdwg::Graph<std::string, int> aMove{std::move(a)};`
Destructor	`~gdwg::Graph<N, E>()`		N/A

2.2. Operations

Name	Operator	Description	Examples	Exception: Why thrown & what message
Copy Assignment	`gdwg::Graph<N, E>& operator=(const gdwg::Graph<N, E>&)`	A copy assignment operator overload	`a = b;`	N/A
Move Assignment	`gdwg::Graph<N, E>& operator=(gdwg::Graph<N, E>&&)`	A move assignment operator	`a = std::move(b);`	N/A

2.3. Methods

Prototype	Description	Usage	Exception: Why & what message

`bool InsertNode(const N& val)`	Adds a new node with value val to the graph. This function returns true if the node is added to the graph and false if there is already a node containing val in the graph (with the graph unchanged).	`std::string s{"a"};` `g.InsertNode(s);`	N/A
`bool InsertEdge(const N& src, const N& dst, const E& w)`	Adds a new edge src → dst with weight w. This function returns true if the edge is added and false if the edge src → dst with weight w already exists in the graph. Note: Nodes are allowed to be connected to themselves.	`std::string u{"c"};` `g.InsertEdge("a",u,1);`	• **Condition**: src or dst no be found in • **Throw**: std::runtime • **With string** "Cannot call Graph::Inse when either node does n
`bool DeleteNode(const N&)`	Deletes a given node and all its associated incoming and outgoing edges. This function does nothing if the node that is to be deleted does not exist in the graph. Hint: if you are using weak ptrs for edges you may be able to do this quite simply. This function returns an boolean as to whether the item has been removed or not (true if removed).	`g.DeleteNode("b")`	N/A
`bool Replace(const N& oldData, const N& newData)`	Replaces the original data, oldData, stored at this particular node by the replacement data, newData. If an exception is not thrown, this function returns false if a node that contains value newData already exists in the graph (with the graph unchanged) and true otherwise.	`g.Replace("a", "e")`	• **Condition**: node that co value oldDat found • **Throw**: std::runtime • **With string** "Cannot call Graph::Repl node that de exist"
`void MergeReplace(const N& oldData, const N& newData)`	All instances of node oldData in the graph are replaced with instances of newData. After completing, every incoming and outgoing edge of oldData becomes an incoming/ougoing edge of newData, except that duplicate edges must be removed. Examples at the bottom of the table.	`g.MergeReplace("c", "e")`	• **Condition**: node cannot in the graph • **Throw**: std::runtime • **With string** "Cannot call Graph::Merg on old or ne they don't e graph"

`void Clear()`	Remove all nodes and edges from the graph. New nodes or edges can be added to the graph afterwards.	`g.Clear()`	N/A
`bool IsNode(const N& val);`	Returns true if a node with value val exists in the graph and false otherwise.	`g.IsNode("a")`	N/A
`bool IsConnected(const N& src, const N& dst);`	Returns true if the edge src → dst exists in the graph and false otherwise.	`g.IsConnected("e","b")`	• **Condition**: src or dst is the graph • **Throw**: std::runtime • **With string** "Cannot call Graph::IsCo if src or dst don't exist in graph"
`std::vector<N> GetNodes()`	Returns a vector of all nodes in the graph. Sorted by increasing order of node.	`g.GetNodes()`	N/A
`std::vector<N> GetConnected(const N& src)`	Returns a vector of the nodes (found from any immediate outgoing edge) connected to the src node passed in. Sorted by increasing order of node (of those nodes that are connected).	`g.GetConnected("e")`	• **Condition**: not in the gr • **Throw**: std::out_of_ • **With string** "Cannot call Graph::GetC if src doesn' the graph"
`std::vector<E> GetWeights(const N& src, const N& dst)`	Returns a vector of the weights of edges between two nodes. Sorted by increasing order of edge. These edges are all outgoing edges of src toward dst.	`g.GetWeights("e","b")`	• **Condition**: src or dst is the graph • **Throw**: std::out_of_ • **With string** "Cannot call Graph::GetV src or dst no exist in the

`const_iterator find(const N&, const N&, const E&);`	Returns an iterator to the found edge in the graph. If the edge is not found the equivalent value of gdwg::Graph<N, E>::cend() is returned.	`gwdg::Graph<std::string, int> g;`
		`std::string a1{"e"};`
		`std::string a2{"i"};`
		`int e = 8;`
		`auto it = g.find(a1, a2, e)`
			N/A

`bool erase(const N& src, const N& dst, const E& w)`	Deletes an edge from src to dst with weight w, only if the edge exists in the graph. This function returns an boolean as to whether the item has been removed or not (true if removed). If the edge is not in the graph it returns false.	`g.erase("b", "c", 1)`	N/A

`const_iterator erase(const_iterator it);`	This function removes the position at the location the iterator points to. This function returns an iterator to the element AFTER the one that has been removed. If no erase can be made, the equivalent of gdwg::Graph<N, E>::end() is returned.	```
		gwdg::Graph<std::string, int> g;
		auto it = g.find("e", "i", 8);
		if (it != g.end()) {
		  g.erase(it);
		}
		``
		N/A

`const_iterator cbegin();`	Returns a const_iterator pointing to the first in the container. A const_iterator is an iterator that points to const content. This iterator can be increased and decreased (unless it is itself also const), but it cannot be used to modify the contents it points to, even if the content it points to is not itself const.	`auto it = g.cbegin()` N/A
`const_iterator cend();`	Returns a const_iterator pointing to the past-the-end element in the container. A const_iterator is an iterator that points to const content. This iterator can be increased and decreased (unless it is itself also const), but it cannot be used to modify the contents it points to, even if the content it points to is not itself const. If the container is empty, this function returns the same as vector::cbegin. The value returned shall not be dereferenced.	`auto it = g.cend()` N/A
`const_reverse_iterator crbegin();`	Returns a const_reverse_iterator pointing to the last element in the container. A const_reverse_iterator is a reverse iterator that points to const content. This iterator can be increased and decreased (unless it is itself also const), but it cannot be used to modify the contents it points to, even if the content it points to is not itself const.	`auto it = g.crbegin()` N/A

	Returns a const_reverse_iterator pointing to the before-the-first element in the container. A const_reverse_iterator is a reverse iterator that points to const content. This iterator can be increased and decreased (unless it is itself also const), but it cannot be used to modify the contents it points to, even if the content it points to is not itself const. If the container is empty, this function returns the same as vector::crbegin. The value returned shall not be dereferenced.		
`const_reverse_iterator crend();`		`auto it = g.crend()`	N/A
`const_iterator begin();`	Exactly the same as cbegin()	`auto it = g.begin()`	N/A
`const_iterator end();`	Exactly the same as cend()	`auto it = g.end()`	N/A
`const_reverse_iterator rbegin();`	Exactly the same as crbegin()	`auto it = g.rbegin()`	N/A
`const_reverse_iterator rend();`	Exactly the same as crend()	`auto it = g.rend()`	N/A

MergeReplace examples
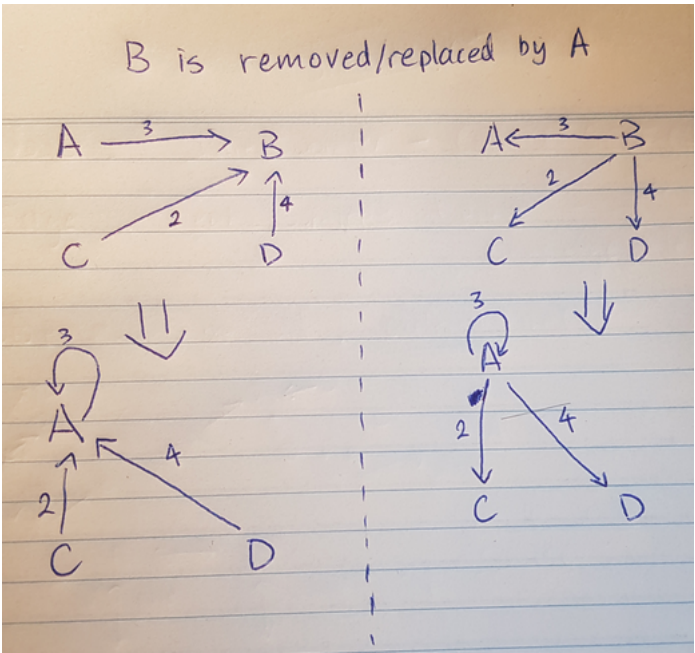
Format is (N_src, N_dst, E)

Example 1 - basic

- Operation: MergeReplace(A, B)
- Graph before: (A, B, 1), (A, C, 2), (A, D, 3)
- Graph after : (B, B, 1), (B, C, 2), (B, D, 3)

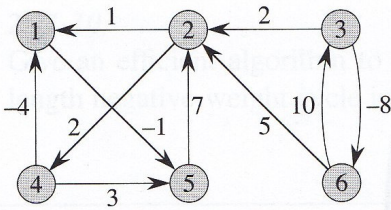Example 2 - duplicate edge removed

- Operation: MergeReplace(A, B)
- Graph before: (A, B, 1), (A, C, 2), (A, D, 3), (B, B, 1)
- Graph after : (B, B, 1), (B, C, 2), (B, D, 3)

Example 3 - Diagramatic

2.4. Friends

In addition to the operations indicated earlier, the following operations should be supported as friend functions. Note that these friend operations don't modify any of the given operands.

Name	Operator	Description
Equal	`bool operator==(const gdwg::Graph<N, E>&, const gdwg::Graph<N, E>&)`	True if the two graphs contain the same nodes and edges.
Not Equal	`bool operator!=(const gdwg::Graph<N, E>&, const gdwg::Graph<N, E>&)`	True if the two graphs do not contain the same nodes or ed

Output Stream	`std::ostream& operator<<(std::ostream&, const gdwg::Graph<N, E>&)`	Prints out the graph in the following format:
		
		``[node1][1 space](
		[2 spaces][node1_connected_node1][1 space][pipe][1 s
		[2 spaces][node1_connected_node2][1 space][pipe][1 s
		[etc etc]
		)
		[node2][1 space](
		[2 spaces][node2_connected_node1][1 space][pipe][1 s
		[2 spaces][node2_connected_node2][1 space][pipe][1 s
		[etc etc]
		)
		[node3][1 space](
		[2 spaces][node3_connected_node1][1 space][pipe][1 s
		[2 spaces][node3_connected_node2][1 space][pipe][1 s
		[etc etc]
		)
		[etc etc]```
		
		For example, for gdwg::Graph<int, int>, an output may be
		
		``1 (
		  5 | -1
		)
		2 (
		  1 | 1
		  4 | 2
		)
		3 (
		  2 | 2
		  6 | -8
		)
		4 (
		  1 | -4
		  5 | 3
		)
		5 (
		  2 | 7
		)
		6 (
		  2 | 5
		  3 | 10
		)```
		
		Note that output is sorted by 1) Src node (increasing order node (increasing order); then 3) Edge weight (increasing o
		
		Note that there is a newline after the last ')'
		
		Note that if a node has no outgoing edges then you should such as:
		Node 7 has no outgoing edges:
		
		``7 (
		)```

3.5. Your "Edge" Custom Iterator

You are required to define an iterator class **gdwg::Graph<N, E>::const_iterator**.

The iterators will come in a const form. The iterator should respond to the *, ++, --, == and != operators

When iterating through your graph the order of iteration should be based on the operator< for the underlying node type N

Your iterators should be appropriately declared as bi-directional. You should implement and test the complete bi-directional iterator functionality. This includes pre/post incre- ment and decrement. The post-increment/decrement operations should return a copy of the value pointed to before the increment/decrement is performed.

We will assume that all iterators are invalidated after any modification of the graph

3.5.1. Iterator Operators & Behaviour

Iterator operators should have the same structure/schema as discussed in week 7 lectures

The **value_type** within your iterator should be

```
std::tuple<N, N, E>
```

The **reference** type within your iterator should be

```
std::tuple<const N&, const N&, const E&>
```

. Note that this means that when returning references, you cannot return a reference to your value type. Instead, you should write

```
return {node1, node2, edge};
```

, where node1, node2, and edge are references to the nodes and the edge. Additionally, **do not** use std::make_tuple, as that will always copy values instead of returning references.

In this way, your iterator will iterator through a graph like the one below producing the following tuple values when deferenced each time:

```
gdwg::Graph<int, int>
```
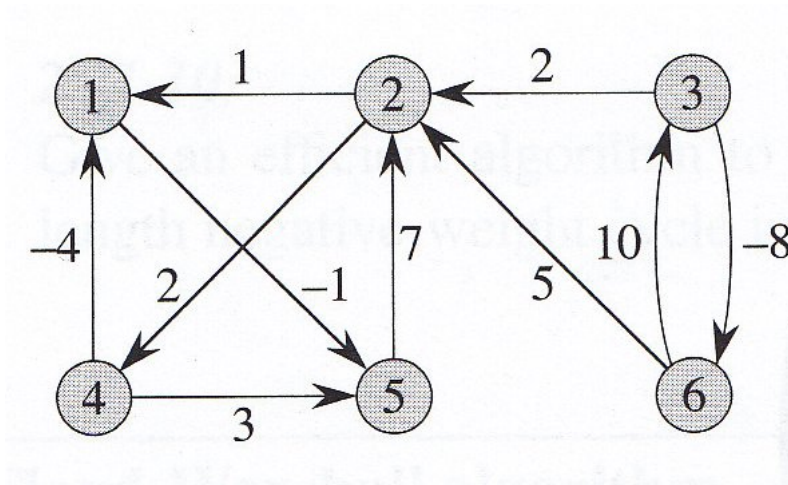

3.5.2. Iterator order

Note that output is sorted by 1) Src node (increasing order); then 2) Dst node (increasing order); then 3) Edge weight (increasing order)

```
(1, 5, -1)
(2, 1, 1)
(2, 4, 2)
(3, 2, 2)
(3, 6, -8)
(4, 1, -4)
(4, 5, 3)
(5, 2, 7)
(6, 2, 5)
(6, 3, 10)
```

3.6. Internal Representation

Your Graph is **required** to own the resources (nodes and edges) that are passed in through the insert functions. This means creating memory on the heap and doing a proper copy of the values from the caller. This is because resources in your graph should outlive the caller's resouce that was passed in in case it goes out of scope. For example, we want the following code to be valid.

```
int main() {
  gdwg::Graph<std::string, int< g;
  {
    std::string s1{"Hello"};
    g.InsertNode(s1);
  }

  // Even though s1 has gone out of scope, g has its own
  //  copied resource that it has stored, so the node
  //  will still be in here.
  std::cout << g.IsNode("Hello") << "\n"; // prints 'true';
}
```

Your Graph is **required** to use smart pointers (however you please) to solve this problem.

For each edge, you are only allowed to have one underlying resource (heap) stored in your graph for it.

For each node, you are only allowed to have one underlying resource (heap) stored in your graph for it.

Hint: In your own implementation you're likely to use some STL containers to store things, and somewhere in those containers you'll use instances of either:

- std::unique_ptr<N>
- std::shared_ptr<N>

depending on your implementation choice.

But why smart pointers

You could feasibly implement the assignment without any smart pointers, through lots of redundant copying. For example, having a massive data structure like:

```
std::map<N, std::vector<std::pair<N, E>>>
```

You can see that in this structure you would have duplicates of nodes when trying to represent this complex structure. This takes up a lot of space. We want you to build a space efficient graph. This means only storing one instance of each node and edge.

3.7. Throwing Exceptions

You are required to throw exceptions in certain cases. These are specified in the tables above.

3.8. Other notes

You must:
- Include a header guard in graph.h
- Use C++17 style and methods
- Leave a moved-from object in a state with 0 nodes
- Mark all appropriate functions *noexcept*

You must not:
- Write to any files that aren't provided in the repo (e.g. storing your graph data in an auxilliary file)

3.9. Const Correctness

You must ensure that each method (2.3) and friend (2.4) appropriately either has:

- A const member function; or
- A non-const member function; or
- Both a const and non-const member function

Please think carefully about this. The function prototypes intentionally do not specify their constness. This has an explicit const and non-const prototype to help you out.

In most cases you will only need a single function, but in a couple of cases you will need both a const and non-const version.

4. Getting Started

If you haven't done so already, clone the repository:

```
$ git clone https://github.com/cs6771/comp6771 comp6771
```

Then navigate to the **assignments/dg** directory

All of the files you need are in this directory. Here is a list of files that and a description of their purpose:

File	Description
client.(cpp|out)	A simple use case of a client using your graph **Note: Do NOT modify this file. We will potentially update it. If you want to modify it, make a local copy first.**
graph.(tpp|h)	Your GDWG interface and implementation.

graph_test.cpp	Your tests for your GDWG file
BUILD	Build file containing build, test, dependency instructions

4.2. Reference Solution & Time Limits

Due to the simple nature of this data type, there will be performance based measurements

Each test will still have a time limit to run (1 second max), but unless you do something completely insane this will be more than enough time for all of your operations to complete.

4.3. Running your assignment

4.3.1. Running a basic use case

You can run your code against a basic (non-testing) use case to get a better sense of the behaviour

From your project directory:

```
$ bazel build //assignments/dg:client
```

```
$ ./bazel-bin/assignments/dg/client | diff ./assignments/dg/client.sampleout -
```

4.3.2. Running your tests

```
$ bazel build //assignments/dg:graph_test
```

```
$ bazel run //assignments/dg:graph_test
```

4.4. Autotests

You can run some basic tests on your code on any CSE machine. In a folder that contains your files, simply run the following command:

```
6771 dgtest
```

Note: graph.tpp and graph.h must be in that directory.

Note: This test will not be available until the at the earliest the final 7 days before submission

4.5. Assessment

This assignment will contribute 20% to your final mark.

The assessment for the assignment recognizes the difficulty of the task, the importance of style, and the importance of appropriate use of programming methods (e.g. using while loops instead of a dozen if statements).

60%	**Correctness** The correctness of your program will be determined automatically by tests that we will run against your program. We will create a series of test files that contain their own main function and are compiled with your GDWG file(s).
20%	**Your tests** You are required to write your own tests to ensure your program works. You will write tests in `graph_test.cpp` . At the top of this file you will also include a block comment to explain the rational and approach you took to writing tests. Please read the Catch2 tutorial (https://github.com/catchorg/Catch2/blob/master/docs/tutorial.md) or review lecture/tutorial content to see how to write tests. Tests will be marked on several factors. These include, but are not limited to • Correctness - an incorrect test is worse than useless. • Coverage - your tests might be great, but if they don't cover the part that ends up failing, they weren't much good to you. • Brittleness - If you change your implementation, will the tests need to be changed (this generally means avoiding calling functions specific to your implementation where possible - ones that would be private if you were doing OOP). • Clarity - If your test case failed, it should be immediately obvious what went wrong (this means splitting it up into appropriately sized sub-tests, amongst other things).
15%	**C++ style** Your adherence to good C++ style. This is **not** saying that if you conform to the style guide you will receive full marks for this section. This 20% is also based on how well you use modern C++ methodologies taught in this course as opposed to using backwards-compatible C methods. Examples include: Not using primitive arrays and not using pointers.

5%	**cpplint and clang-format**
	In your project folder, run the following commands on all C++ files you submit:
	`$ clang-format -i /path/to/file && python cpplint.py /path/to/file`
	If the program outputs the following, you will receive full marks for this section (5%). Otherwise you will receive no marks.

```
Done processing assignments/dg/graph.h
Total errors found: 0
```

The following actions will result in a 0/100 mark for gdwg::Graph<N, E>, and in some cases a 0 for COMP6771:

- Knowingly providing your work to anyone and it is subsequently submitted (by anyone).
- Submitting any other person's work. This includes joint work.
- Submitting another person's work without their consent.

The lecturer may vary the assessment scheme after inspecting the assignment submissions but it will remain broadly similar to the description above.

4.6. Originality of Work

The work you submit must be your own work. Submission of work partially or completely derived from any other person (other than your partner) or jointly written with any other person is not permitted. The penalties for such an offence may include negative marks, automatic failure of the course and possibly other academic discipline. Assignment submissions will be examined both automatically and manually for such submissions.

Relevant scholarship authorities will be informed if students holding scholarships are involved in an incident of plagiarism or other misconduct.

Do not provide or show your assignment work to any other person — apart from the teaching staff of COMP6771. If you knowingly provide or show your assignment work to another person for any reason, and work derived from it is submitted, you may be penalized, even if the work was submitted without your knowledge or consent. This may apply even if your work is submitted by a third party unknown to you.

Note you will not be penalized if your work has the potential to be taken without your consent or knowledge.

4.7. Group Work

For this assignment, you can choose to work in a group (of 2) or work by yourself. We STRONGLY recommend you work in a group of 2. Working by yourself is not grounds for special consideration, or compensation of marks. Working in a team will provide you someone you can share and discuss code with and develop your teamwork skills with.

If working in a group (RECOMMENDED)

You are **required** to use a **private** git repository (e.g. github) to store your assignment code. This repository must be kept after the assignment is due and will be used to resolve any disputes regarding relative contributions between team members.

If you are pair programming (i.e. working off one computer), just write "Pair programming" in the commit message. However, if you pair program and commit with a single username, we will have trouble resolving any disputes.

Groups will be finalised on Friday 19th July at 11:59pm

You must register your group here (https://docs.google.com/forms/d/e/1FAIpQLSe554NdOWiXghFDNCw0YVqPF2g/viewform) before that deadline

If working individually (NOT RECOMMENDED)

You are strongly recommended, but not required to use a repository to store your code. If you do use a repository to store your code, it **must** be a **private** repository

4.8. Submission

This assignment is due **Tuesday 6th of August, 23:59:59**. Submit the assignment using this *give* command:

```
give cs6771 graph graph.h graph.tpp graph_test.cpp
```

4.9. Late Submission Policy

If your assignment is submitted after this date, each hour it is late reduces the maximum mark it can achieve by 1%. For example if an assignment worth 76% was submitted 5 hours late, the late submission would have no effect (as maximum mark would be 95%). If the same assignment was submitted 30 hours late it would be awarded 70%, the maximum mark it can achieve at that time.