Final submission

Andrew Grant
amg2215@columbia.edu

Somya Vasudevan
sv2500@columbia.edu

4/28/2017

Contents

1 Introduction

We've provide you with links to a bunch of the required files, as well as added most documents into this pdf. If any of the links do not work for some reason, please reach out to us via email. Nonetheless, all files will be in the repository which is located at: `https://github.com/andyg7/Graph-Library` which should almost assuredly work and so you can look there too to find some of the documents.

2 Development Environment

- GCC 6.2
- OS: Ubuntu 16.10
- C++ standard libary used: c++1z
- Compiler options: -fconcepts

Compile times are pretty slow due to use using concepts.

3 Links to all requires files

- Repository of project: `https://github.com/andyg7/Graph-Library`
- Source code of graph library: `https://github.com/andyg7/Graph-Library/tree/master/src`
- Tests: `https://github.com/andyg7/Graph-Library/tree/master/tests`
- Examples of using library:
  - `https://github.com/andyg7/Graph-Library/tree/master/cities_examples`
  - `https://github.com/andyg7/Graph-Library/tree/master/examples`
  - `https://github.com/andyg7/Graph-Library/tree/master/expander_examples`
- Tutorial: `https://github.com/andyg7/Graph-Library/blob/master/docs/Tutorial.pdf`

- Design Document: `https://github.com/andyg7/Graph-Library/blob/master/docs/DesignDocument.pdf`
- Third Party code - we used some concepts from `https://github.com/CaseyCarter/cmcstl2`
- Commit history: `https://github.com/andyg7/Graph-Library/commits/master`
- Real usage of library - as solved the 8 puzzle game using our library. The code to do this is here: `https://github.com/andyg7/Graph-Library/tree/master/expander_examples`

4 How to run our code (and our application)

- Go to `lib_examples` directory (`https://github.com/andyg7/Graph-Library/tree/master/lib_examples`)
- There are three subdirectories (`cities_examples, expander_examples, simple_examples`) that contain examples of code using our library
- run "make" to compile. "make clean" will clean the dir first. The binary is called "main".
- `expander_examples` contains the application of our library. The design document explains the application in more detail.

5 Testing Code

Our unit testing suite is in Graph-Library/tests. Just run "make". The binary is called "main".

6 Group Review

Andrew wrote the core of the graph library, path searching code and some of the concepts. Somya wrote the unit tests, designed and implemented some of the concepts, came up with the application usage, handled the documentation and did the measurements.

7 Design Document

7.1 Introduction

We've built a graph and path searching library, built in C++. The library is designed to make it really easy for users to use their own vertex and edge data structures and get going using graphs and running algorithms on them right away. Currently the library supports the following graphs: DG (directed graph), DAG (directed acyclic graph), DT (directed tree), Matrix(undirected). The idea is that users define their own vertex and edge data types, and then the library handles everything else under the hood. We also provide very generic path searching code that can be used in many domains.

7.2 Motivation

Conceptually, a graph is made up of a bunch of vertices and edges. At a minimum there must be some way to distinguish between vertices, distinguish between edges, and define edges as made up of two vertices. Nonetheless, users often want to embed extra information in these ADTs. For example, a user may want a graph representing cities and the highways between them. The user may have a City class; cities must of course have some unique id (e.g. city name), but they may also have extra information such as population, GDP, etc. The same goes for edges; maybe a Road class is used, and the class also has miles, age of road etc.

We wanted to design a library that made it super easy for users to user their *own* vertex and edge data types. The philosophy of the library was that it should be really easy for users to be able to use their own

vertex and edge data structures, and that the the library would handle everything else under the hood. In order to make it easy to use the library, we designed the library with minimal requirements when it came to the user defined vertex and edge types.

7.3 General Graph Design

7.3.1 Graph Data Types

We provide the following graph types: DG (adjacency list), DAG (adjacency list), DT (adjacency list), and Matrix (undirected). All of these types are represented by simple structs. Each struct holds certain information about the graph it is representing. For example, the adjacency list structs will point to lists of vertices, while the matrix struct will point to some sort of matrix. An important member type of each struct is `graph_type`. This member type is set to one of: `DAG, DG, DT, Matrix_graph`. Concepts (described below) use this member type to determine what graph type is being used, and thus be able to determine, at compile type, which function to call. This is important as fundamentally there is no structural difference between a DT and DG; by using this member type we can have a function that works on a DT (and checks to make sure the tree structure invariant is maintained) and another function that works on a DG (and doesn't have to check that tree structure is maintained).

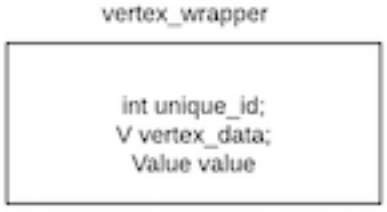
7.3.2 User Defined Data Structures

The library was designed so that users are able to design their own vertex and edge data structures. All that is required of the users is the following:

- Vertex and Edge must be comparable (that is they must implement the == operator for their types)
- Vertex and Edge must be hashable
- Edge must have two fields `v_1` and `v_2`, that are of the same type as the Vertex that they defined.
- To use the path algorithms, Edge must have a cost field that is numeric. If the user does not define a cost, the library will automatically given an edge a cost of 1.
- Vertex must have a `to_string` function that returns a string representation of the vertex.

7.3.3 Wrappers

Because minimal requirements are imposed on the user when it comes to his/her vertex and edge type, we needed to wrap the users data types in our own struct for certain bookkeeping reasons. Thus, under the hood the graph library stores each vertex in a `vertex_wrapper` and each edge in a `edge_wrapper`. Both wrappers also store a field of type `Value` which is a pair consisting of a string and an int. This lets the user set some sort of value for each vertex and/or edge, without their own data type knowing about such a value. Lastly, the wrapper also stores a unique id (of type int) that the user is again completely oblivious to. This id is generated by the graph library for internal bookkeeping (for example, it's used as the index when a matrix graph representation is used). The following image illustrates this.



The idea is the same for the `edge_wrapper`

`vertex_data` points to the user provided vertex data. This `vertex_wrapper` is the fundamental data type of our graph library. Whether a DAG, DG, DT or matrix graph representation is used, each vertex is stored in this struct.

7.3.4 Graph Interface

Because all graphs are represented by structs, pretty much every graph library function requires such a graph struct as a formal argument. The remaining arguments are almost always one of the user defined vertex or edge. Thus, users do not need to know about wrappers at all. The wrappers are handled by the library. Furthermore, `shared_ptr<>s` are required when using the API. That is, the library requires every formal argument to be of type `shared_ptr<>`

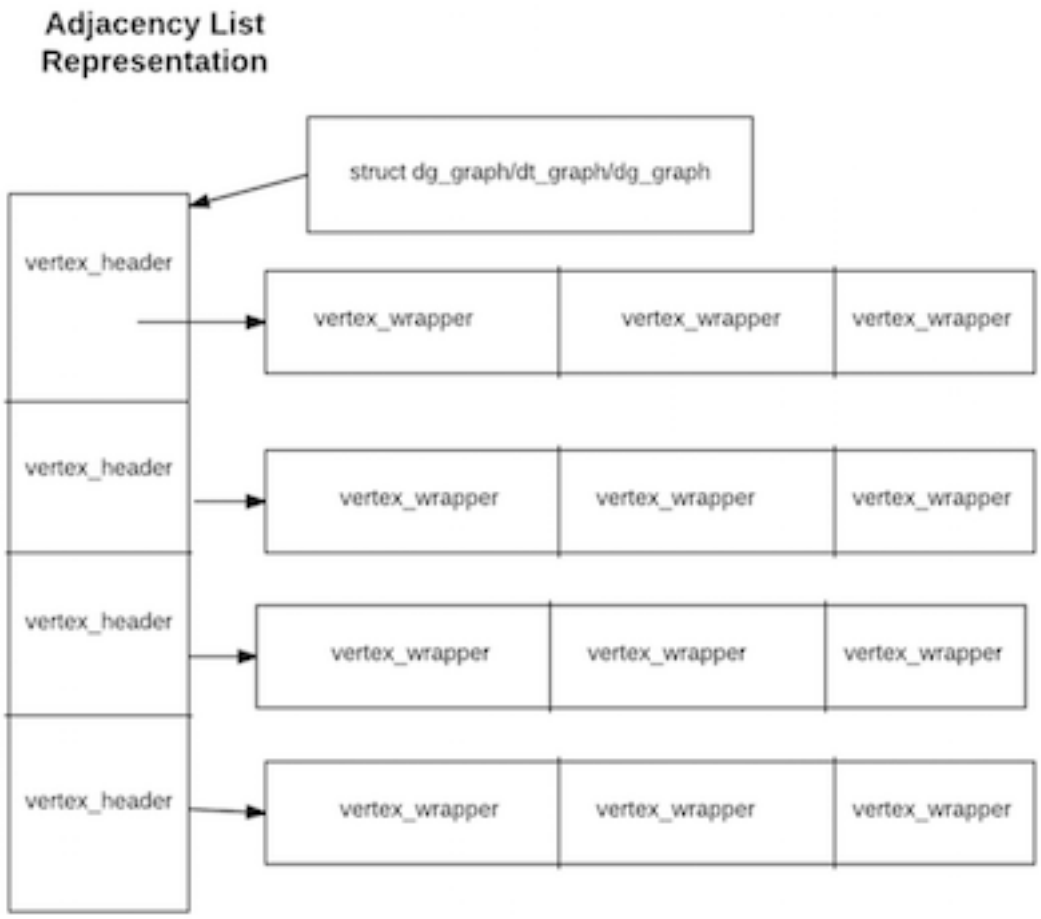
Concepts allow us to use the same function name for different graph types. For example, there is a different function `add_edge(G g, E e)` for each graph type.

```
template<typename G, typename E>
requires Graph<G> && Edge_ptr<E>
bool add_edge(G g, E e)
{
        bool added_edge = add_edge_blindly_w_edge(g, e);
        return added_edge;
}

template<typename G, typename E>
requires DAG_Graph<G> && Edge_ptr<E>
bool add_edge(G g, E e)
{
        typedef typename G::element_type::vertex_type vertex_type;
        shared_ptr<vertex_type> tmp_v1 = make_shared<vertex_type>(e->v1);
        shared_ptr<vertex_type> tmp_v2 = make_shared<vertex_type>(e->v2);
        bool path_ex = path_exists(g, tmp_v2, tmp_v1);
        if (path_ex == true) {
                return false;
        }
        bool added_edge = add_edge_blindly_w_edge(g, e);
        return added_edge;
}

template<typename G, typename E>
requires DT_Graph<G> && Edge_ptr<E>
bool add_edge(G g, E e)
{
        typedef typename G::element_type::vertex_type vertex_type;
        shared_ptr<vertex_type> tmp_v1 = make_shared<vertex_type>(e->v1);
        shared_ptr<vertex_type> tmp_v2 = make_shared<vertex_type>(e->v2);
        bool path_ex = path_exists(g, tmp_v2, tmp_v1);
        if (path_ex == true) {
                return false;
        }
        bool has_par = has_parent(g, tmp_v2);
        if (has_par == true) {
                return false;
        }
        bool added_edge = add_edge_blindly_w_edge(g, e);
        return added_edge;
}
```

7.4 Adjacency List Design

7.4.1

DAG, DG and DT are are represented using adjacency lists. In order to implement an adjacency list, we actually created another data type to wrap `vertex_wrapper`, which we call `vertex_header`. The `vertex_header` is the head of a vertex's adjacency list. Thus, the difference between a `vertex_header` and `vertex_wrapper`, is that the `vertex_header` is the conceptual head of a vertex's adjacency list and points to a list of `vertex_wrapper`.



The image above shows the `vertex_header` pointing to a list of `vertex_wrapper`. `vertex_header` actually also points to a list of `edge_wrapper` too.

7.4.2 Data Structures

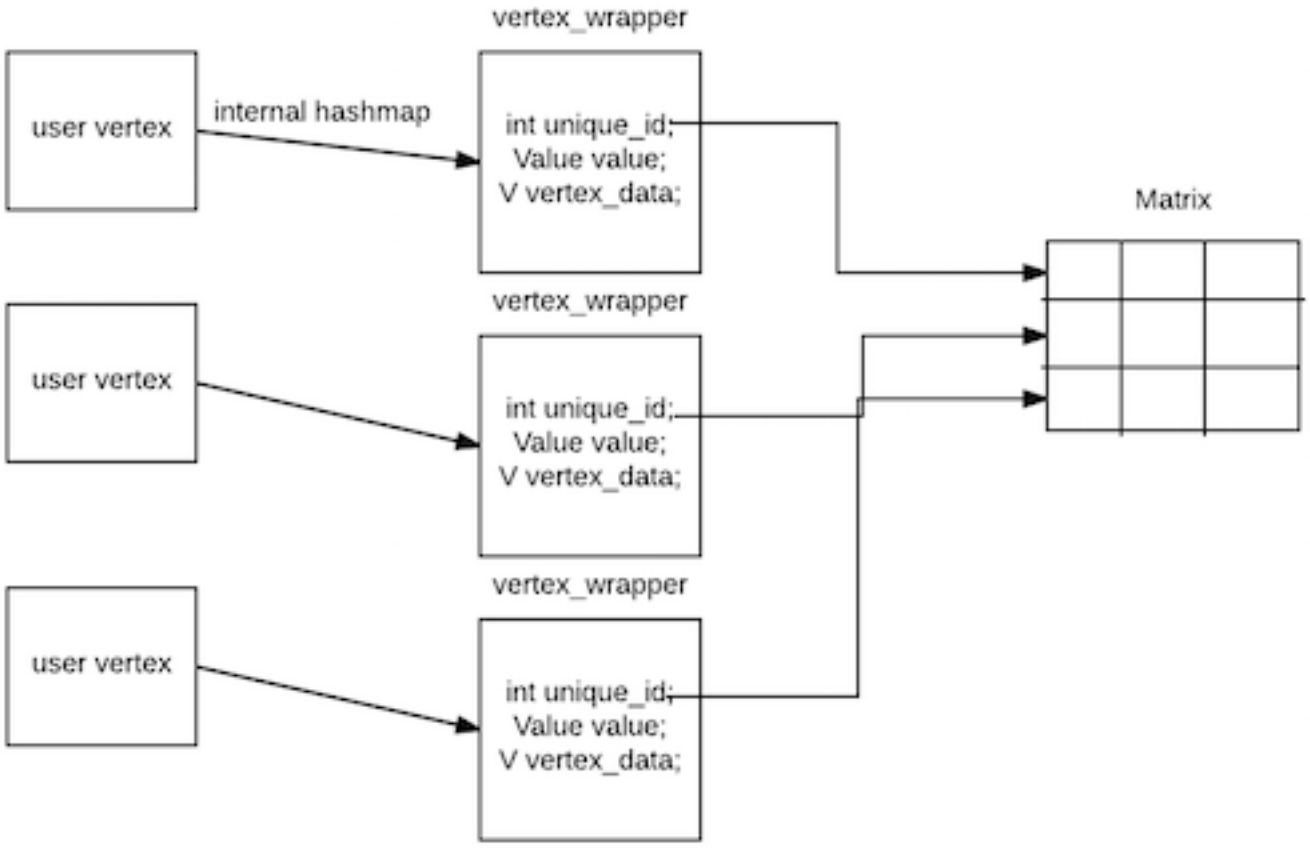
The list of vertex headers is stored in a vector. The list of neighbors (aka `vertex_wrapper` is also stored in a vector. The list of edges (aka `edge_wrapper` is also stored in a vector).

7.5 Matrix Design

7.5.1

As with the adjacency list representation, `vertex_wrapper` is a fundamental data type to our matrix implementation. As emphasized, we wanted the user to be able to use his/her own vertex and edge types. In order to be as flexible as possible we did *not* require the user to assign each vertex a unique integer id.

Nonetheless, in order to use matrix graph representations we need some way to index each vertex into the matrix. We decided to use the internal id embedded in each `vertex_wrapper` for this purpose. However, using this unique id wasn't enough. The whole point of using a matrix representation is the access efficiency; we needed some way to immediately map an incoming vertex to its unique id without iterating over *all* vertices. Thus, to make the matrix representation work we maintain a hashmap pointing from a vertex to its `vertex_wrapper`, which contains it's unique id.



7.5.2 Data Structures

We use a vector of vectors to represent the matrix. Since the internal id is an int, the matrix is thus a vector of vector of ints. We also maintain a hashmap pointing from a vertex to its wrapper.

7.6 Path Searching algorithms

7.6.1 Algorithms Supported

We support the following path searching algorithms:

- BFS
- DFS
- UCS
- A*

7.6.2 Implementation

User Defined State We designed the path algorithms such that they only require a "start" state and "goal" state, where "state" is an abstract type. The user simply defines his/her own start and goal state, and the code handles the rest. Of course the semantics of path finding require that any given state must adhere to some requirements; we use concepts to enforce these:

- a state must be comparable
- a state must be hashable
- a state must have a `to_string()` member function
- a state must have a field called parent that is of type `shared_ptr<state>`
- a state must have a field called cost that is numeric
- a state must have a member function called "expand" that returns a vector of states
- if A* is used, a member function called `heuristic_func()` is required that returns a numeric type

As long as the user provides a type that adheres to these properties, any of our algorithms can be run

Path Code We designed the path searching part of the library such that *all* algorithms actually run on the same while loop. The function that eventually gets called accepts a few parameters that determine which algorithm gets run. For example, if we want to run DFS we send the path code a stack to be used as the frontier set. If we want to run BFS we send it a queue for the frontier set. Thus, all the different algorithms actually run on the same code; it's the inputs that decide which algorithm gets run.

Obviously we wouldn't want the user to know or worry about such details. Thus, the user simply calls `find_path_ast(s, g)`, and under the hood the library creates the appropriate frontier set and calls the generic code from there.

Path Interface As mentioned above, the path searching algorithms requires only a start state and goal state. Both states must be wrapped in a `shared_ptr<>`

7.6.3 Graph Library Integration

The core graph library doesn't know anything about a "start" state and a "goal" state. A vertex is not the same thing as a state; for example, a vertex does not have the concept of a parent pointer. In order to use the path searching code, we therefore has to create state wrappers that integrated with the pre-existing graph data types. This made sure we could still use code described above, while also making sure the user did not have to worry about such transformations.

7.7 Concepts

7.7.1 What are concepts?

Concepts are essentially compile time predicates. That is, they are requirements on the types that are passed into functions. If an argument doesn't satisfy the concept, the compiler will immediately report an error. This makes debugging much easier when using templates. Without concepts, debugging can be very tricky when using templates, as it's often late in the compilation process that the compiler realizes a type is no good. As a result error messages can be extremely long, making dubugging tricky.

7.7.2 Library Usage

In terms of this library, there are two main ways we use concepts.

The first is to make debugging easier; user defined vertices and edges must satisfy certain properties and concepts are used to enforce these and catch bugs earlier in the compilation process. For example, an Edge must point to two vertices; an Edge must be comparable; a Vertex must have a unique identifier.

Perhaps more importantly, we use concepts to support function overloading. The names of the functions that work on different graph types are all the same; thus the user can call the same function whether he

or she is working with a matrix or adjacency list. Concepts determine which function should be called at *compile* time, not runtime.

7.8 Application

For our application we wrote a solver for the 8 puzzle game. `https://en.wikipedia.org/wiki/15_puzzle`. We were able to use the path searching code to easily use the generic code to solve the game using BFS, DFS, UCS and A*. All we needed to go was create a class to represent the state of the board at any given time, and write a function called "expand" which returned a vector of board configurations that could be reached in one stop. At that point we instantiated two objects, the initial board config, and goal board config and the library handled the rest. Check out `https://github.com/andyg7/Graph-Library/tree/master/lib_examples/expander_examples` to see the code to actually solve the puzzle using our library.

7.8.1 References

- Concepts reference: `http://www.stroustrup.com/good_concepts.pdf`
- The complete project is located at `https://github.com/andyg7/Graph-Library`.
- The core graph library code is located at `https://github.com/andyg7/Graph-Library/src`

8 Tutorial

8.1 Introduction

This is a short tutorial on how to get going using the graph library. The big picture idea is for you, the user, to create your own Vertex and Edge data structures; once you provide the library with these two user defined types, you'll be immediately able to start creating graphs and running algorithms on these graphs using our simple API.

8.2 How to Use the Library

8.2.1 User Defined Vertex and Edge

The most important thing is for the user to define his/her vertex and edge data types. The types must adhere to the following requirements:

- Vertex and Edge must be comparable
- Vertex and Edge must be hashable
- Edge must have two fields `v_1` and `v_2`, that are of the same type as the Vertex defined.
- To use the path algorithms, Edge must have a cost field that is numeric. If the user does not define a cost, the library will automatically given an edge a cost of 1.
- Vertex must have a `to_string` function that returns a string representation of the vertex.

8.2.2 Choosing a Graph Type

Then the user should select one of `graph_dg`, `graph_dag`, `graph_dt`, `matrix_graph` and provide the struct with two template parameters that specify the vertex and edge types (the library provides vertex and edge types for user, but most likely the user will to provide his/her own data types) For example:

- `dag_graph<my_vertex_1, my_edge_1> my_graph;`
- `dt_graph<vertex, edge> my_graph;`
- `dg_graph<my_vertex_2, my_edge_2> my_graph;`

At this point the user can easily start using the library provided functions. All functions require the user to provide at least the graph object. Furthermore, all arguments should be `shared_ptrs`; this is to avoid the cost of copying. These savings could be substantial for large graph objects. Note that the same function name is used for all graph types, vertex types and edge types. This is thanks to concepts; that is, concepts are used to make sure the right function is called using overloading. This makes it super easy for the user to write generic programs that work on different graph types.

8.2.3 Path Algorithms

The path searching code is very generic and is designed to work with our graph library as well as our path searching domains. **Path Algorithms Using Our Graph Library** The path searching algorithms are accessible just like regular graph functions. One thing to note is the return type of the path algorithms. The path algorithms actually return a pointer to a struct called `path_data`. This struct points to various interesting information about the path found. It contains the following data: 1) the cost of the path 2) a vector of the vertices along the path 3) a vector of string representations of the path 4) a function `to_string` that returns a string of the path

One important note is how the cost is computed. The cost is determined by the cost of the edge; if the user does not define a cost for an edge a cost of 1 is assumed.

Path Algorithms in Other Domains Our path algorithms are also designed to work in other domains where the user can provide a start state and goal state. The idea is that our code will work will pretty much any user defined "state" as long as it adheres to a few requirements:

- a state must have a parent pointer; this is to retrace the found path once the goal state is found
- a state must have a cost field that is numeric
- a state must be hashable
- a state must be comparable
- a state must have a function called `expand()` that returns a vector of states. This function should return a given states successor states.

As long as the user defined data type adheres to these requirements, any of the path finding algorithms can be used.

8.2.4 Important Notes

The API requires the user to send in pointers, to avoid the cost of copying. Nonetheless, the library copies data into the graph data structures. For example, when a vertex is added to a graph, a copy of a vertex is added to the graph struct; this is exactly how vector from the standard library works. This prevents the user from maintaining pointers into the underlying graph data structure and changing data from under it's feet. We believe this will prevent nasty bugs.

8.2.5 Hashable

As mentioned above vertices and edges should be hashable; this involves creating a struct called "hash", providing it with a type, and implementing the () operator. The following code illustrates how to define a hash function for a user defined class called "city"

```
namespace std
{
        template <>
                struct hash<city>
                {
                        size_t operator()(const city& n) const noexcept
                        {
                                return std::hash<string>()(n.get_name());
```

```
                }
        };
}
```

In this instance we're hashing on the "city" class' "name" field, which is accessible via `get_name`. Here's another example where we has on an int.

```
namespace std
{
        template<>
        struct hash<vertex>
                {
                        size_t operator()(const vertex& n) const noexcept
                        {
                                return std::hash<int>()(n.vertex_id);
                        }
                };
        }
```

In both instances we use the library provide hash function (`std::hash<int>, std::hash<string>`; this is not required (but it is encouraged). As long as some sort of hash of type `size_t` is returned you should be good to go.

8.3 Examples

Here we provide some examples on how to use the library.

8.3.1 Creating a graph

Here we create a matrix graph, where Vertex type is "city", and Edge type is "road". "city" and "road" are both user defined classes. 10 refers to the dimension of the matrix. We highly recommend using the keyword "auto" whenever possible.

```
auto my_graph = make_shared<matrix_graph<city, road>>(10);
```

Here we create an adjacency list DAG graph.

```
auto my_graph = make_shared<dag_graph<city, road>>();
```

8.3.2 Creating a Vertex using helper function

Create a Vertex using helper function. A unique id is automatically assigned to v1. This is a helper function for the user if he/she needs to create a ton of vertices on the fly and doesn't want to manually create them and worry about creating unique ids. Nonetheless, the user can create his/her own vertices manually as well of course.

```
auto v1 = create_vertex(my_graph);
```

8.3.3 Setting a value to a Vertex

Note, the user defined Vertex does not need to know anything about Value

```
set_value(my_graph, v0, Value {"A", 1990});
```

8.3.4 Adding a vertex to a graph

```
add(my_graph, v0);
```

8.3.5 Creating Edge

```
auto e1 = create_edge(my_graph, v0, v2);
```

8.3.6 Checking if vertices are adjacent

```
bool a = adjacent(my_graph, v0, v1)
```

8.3.7 Getting vertices by value

```
vector<V> vertices_by_value = get_vertices_by_value(my_graph);
```

8.3.8 Removing a Vertex

```
remove(my_graph, v11);
```

8.3.9 Finding a path between Vertices

Here we show you how to use the path finding algorithms. Notice how we get a struct from the function. In the second line we get a vector of the vertices along the path. So in our case we would get a vector of "city" objects. In the third line we are able to get a string; this string is a string of the path. Note how this requires the "city" provides a `to_string` function.

```
struct path_data path_v0_v1_data = find_path_ucs(my_graph, v0, v1);
auto vector_of_vertices = path_v0_v1_data->path_v;
string string_of_path = path_v0_v1_data->to_string();
```

8.3.10 Finding a path from start state to goal state (8 puzzle problem)

Here we show how our code can be used to solve the 8 puzzle problem. The user has defined a class that represents the state of the board at any given time. The class has a function called `expand()` that returns a vector of states that can be moved to in one move. The class has a parent pointer. The class has a cost. It is hashable and it is comparable.

```
class Node : public std::enable_shared_from_this<Node>
{
        public:
                int vertex_id;
                using cost_type = int;
                Node(vector<vector<int>>, int, shared_ptr<Node>);
                vector<shared_ptr<Node>> expand();
                int get_element(int, int) const;
                int get_element(tuple<int,int>) const;
                void set_element(int, int, int);
                void set_element(tuple<int,int>, int);
                void switch_elements(tuple<int, int>, tuple<int, int>);
                bool is_goal_node();
                void set_cost(int);
                int get_cost() const;
                void set_parent(shared_ptr<Node> p);
                shared_ptr<Node> get_parent() const;
                bool operator==(const Node& rhs) const;
                bool operator!=(const Node& rhs) const;
                int heuristic_func() const;
```

```
                shared_ptr<Node> parent;
                int cost;
                string to_string() const;

                //int get_estimated_cost_to_end() const;
        private:
                vector<vector<int>> grid;
                tuple<int,int> get_zero_element() const;
                //int estimated_cost_to_end;
};
```

The user is now able to create a start state, and goal state and call our code and get the path in return:

```
shared_ptr<Node> n = make_shared<Node>(v, 0, nullptr);
        n->set_element(0,0,1);
        n->set_element(1,0,7);
        n->set_element(2,0,4);
        n->set_element(0,1,3);
        n->set_element(1,1,5);
        n->set_element(2,1,6);
        n->set_element(0,2,2);
        n->set_element(1,2,8);
        n->set_element(2,2,0);
        shared_ptr<Node> goal_state = make_shared<Node>(v, 0, nullptr);
        goal_state->set_element(0,0,1);
        goal_state->set_element(1,0,2);
        goal_state->set_element(2,0,3);
        goal_state->set_element(0,1,4);
        goal_state->set_element(1,1,5);
        goal_state->set_element(2,1,6);
        goal_state->set_element(0,2,7);
        goal_state->set_element(1,2,8);
        goal_state->set_element(2,2,0);
        if (!path_exists(n, goal_state)) {
                cout << "No path ! \n";
                return 0;
        }
        cout << "calculating ast path...\n";
        shared_ptr<struct path_data<Node, int>> path_d = find_path_ast(n, goal_state);
```

Hopefully these examples have given you an idea of what our library can be used for!