Graph and Search Library Tutorial

Andrew Grant
amg2215@columbia.edu

Anton Igorevich
ain2108@columbia.edu

Somya Vasudevan
sv2500@columbia.edu

4/28/2017

Contents

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.

2 How to Use the Library

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.

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.

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.

2.3.1 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.

2.3.2 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.

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.

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.

3 Examples

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

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>>();
```

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);
```

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});
```

3.4 Adding a vertex to a graph

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

3.5 Creating Edge

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

3.6 Checking if vertices are adjacent

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

3.7 Getting vertices by value

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

3.8 Removing a Vertex

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

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();
```

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!