Lukas Ruppert
Raphael Braun
Timo Sachsenberg
Wolfgang Fuhl
Dimitrios Koutsogiannis
Frieder Wallner
Martin Röhm

Eberhard Karls
Universität
Tübingen

Programming in C++

Sheet 10

Submission date: 13.09.2023 12:00

10.1 Directed Graph (30 Points)

C++

In this exercise you will implement a simple directed graph using adjacency lists. A graph consists of nodes (sometimes also called vertices) and edges. Each edge represents a connection from one node to another node. Optionally, an edge can also carry an associated value – the weight of the edge. This weight can be used to represent e.g., the distance, traversal cost, or, in our case, the probability for choosing an outgoing edge when randomly traversing the graph.

To represent the graph, the `class AdjacencyListGraph` already exists and takes a `typename TNode` as a template parameter to represent its nodes' values. This might for instance be a 3D point, but in general arbitrary data can be stored here. The `AdjacencyListGraph` itself privately derives from `std::vector<TNode>` to store this node data and thus functions like a `std::vector<TNode>` itself. To make some of that vector's functions usable from outside, they are made `public` with the `using` keyword. We do this to avoid writing some boilerplate code that would be necessary to forward all calls to a member variable `std::vector<TNode> nodes;` in an alternative implementation.
For testing purposes, you can create a graph with `int`s stored in the nodes as following:

```
AdjacencyListGraph<int> graph;
```

you can resize the graph (here we use a wrapper around `resize()` to prevent shrinking) with

```
graph.initialize_nodes(1000); // resize to 1000 default-initialized nodes
```

or add a single node with

```
graph.push_back(42); // add a single node containing the value 42
```

For now, we first deal with its edges. For edges, a non-templated base `class AdjacencyListGraphBase` is used which represents weighted, directed edges.
Internally, we use a vector `edges` to store outgoing edges for each node. Per node, another vector keeps pairs of the target node id and the edge weight per outgoing edge:

```
std::vector<std::vector<std::pair<uint32_t, float>>> edges;
```

For example, the vector of all outgoing edges of the node with index 2 is stored in `edges[2]`. `edges[2][3].first` and `edges[2][3].second` would be target node and weight of an outgoing edge at position 3 in the edge list.
It is your task to complete its implementation in `adjacency_list_graph.cpp`:

a) Implement the `add_edge(uint32_t from, uint32_t to, float weight)` function. Its task is to add an edge to the graph. If the edge already exists, the function should throw an exception of type `std::runtime_error`. Note that you may have to extend the outer vector of `edges` before you can insert the edge. You can check how many nodes the graph currently has using `get_num_nodes()`. When the index `from` is not a valid node index, throw an exception.

b) Implement the `remove_edge(uint32_t from, uint32_t to)` function. If the edge exists, remove it from the graph. Otherwise, throw an exception of type `std::runtime_error`.

c) Implement the `get_edge(uint32_t from, uint32_t to)` function. This function returns a `std::optional<float>`. If the edge exists, the return value should contain the value of the edge's weight. Otherwise, the return value should be empty, i.e. `std::optional<float>{}`.

d) Implement the `get_edges_starting_at(uint32_t node)` function. Its job is to return the vector of weighted edges starting at the given node. If there are none, return an empty vector, but keep in mind that the vector is returned by `const` *reference*. Think about the *lifetime* of objects in C++ and how you can influence it. Do *not* return a temporary (which would be destroyed before access) and do not leak any memory either. Instead, return an empty vector of which you are sure that its lifetime is at least long as the enclosing instance of the `AdjacencyListGraphBase`.

10.2 (De-)Serialization (30 Points)

C++

We want to be able to serialize and deserialize these graphs as they may become quite large and we do not want to generate them from scratch each time. A function template for serializion already exists:

```
std::enable_if_t<std::is_trivially_copyable_v<TNode>, void>
  serialize(const std::string& filename) const
```

a) In `adjacency_list_graph.h`, implement the matching deserialization function to import a graph from its serialized form that returns a new instance:

```
static std::enable_if_t<std::is_trivially_copyable_v<TNode>, DerivedType>
  deserialize(const std::string& filename)
```

In general, you want to follow the same steps as the serialization function, but think of the tasks that need to be done additionally during deserialization.

Here, we use the curiously recurring template pattern (CRTP) to determine the derived type of the `AdjacencyListGraph`, e.g. `RandomWalkGraph`. So, you can initialize the result directly as an object of the `DerivedType` and return it.

b) Using your deserialization function, import the serialized graph in the file `"../secret.graph"` as a `RandomWalkGraph`.

10.3 Random Walks (25+15+0 = 40 Points)

C++

Now, we want to figure out the probabilities of visiting each node in the graph when taking random walks. We do this by simulating thousands of random walks through the graph, starting at random locations.

a) Implement the random walk for a given number of steps in `random_walk_graph.cpp` in

```
void simulate_random_walk(uint32_t num_steps)
```

- For each walk, uniformly choose one of the nodes as a starting point using std::uniform_int_distribution.
- At each step, fetch the outgoing edges of the current node and choose an edge proportionally to its weight:
  - First, pick a random number between 0 and the sum of edge weights per node using std::uniform_real_distribution. (In the provided example, all edge weights sum up to 1 per node, but do not assume that in your final submission.)
  - Loop over the edges and add the weight of each edge to another sum starting at 0.
  - If that sum is larger than your random number, pick the current edge.
  - If there is no outgoing edge, stay at the current node.
- After each step, increment the node's value which represents its visited counter.

b) Store the histogram image in the simple Portable Gray Map format (Netpbm mode P5). Fill in

```
RandomWalkGraph::write_histogram_pgm(const std::string& filename,
  uint32_t width, uint32_t height) const
```

It takes a filename, a width and a height. Verify that the number of nodes in the graph matches the number of pixels for the requested resolution. If there is a mismatch, throw an exception.

Use `std::ofstream` to open the file for writing and store the following contents:

First, write the header for a binary-mode greyscale image (insert the values for width and height):

```
P5
<width> <height>
255
```

Then, write the binary data in the form of `unsigned char`, i.e. one byte per value.

The pixel values can be computed using the provided normalization function template. Use the specialization for `unsigned char` to get pixel values between 0 and 255, occupying one byte each.

```
std::vector<T> compute_normalized_histogram(T max_value) const
```

You can simply write the entire pixel vector to the file in binary form using std::ostream::write. To access the vector's data, use std::vector::data and cast the pointer to a `const char*` using `reinterpret_cast`.

c) Simulate 1000 random walks with 100000 steps each in your main function. (For testing, using fewer steps will work but the result will be quite noisy.) Write the result image to `"histogram.pgm"`. Its resolution is 270x354. You can inspect the result image with the image viewer of your choice. (While the format is simple, it is not very common, so not every program will work. When in doubt, GIMP will be able to open a correctly written file.)