SWO3	Übung zu Softwareentwicklung mit klassischen Sprachen und Bibliotheken 3	WS 2018/19, Angabe 7

☐ Gruppe 1 (J. Heinzelreiter)

☐ Gruppe 2 (M. Hava) Name: ____Niklas Vest_____ Aufwand [h]: ___7_____

☒ Gruppe 3 (P. Kulczycki) Übungsleiter/Tutor: _____ Punkte: _____

Beispiel	Lösungsidee (max. 100%)	Implement. (max. 100%)	Testen (max. 100%)
1 (30 P)	100%	100%	100%
2 (50 P + 20 P)	100%	100%	100%

Beispiel 1: Autobau (`src/wmb`)

Sie werden von einem großen Münchner Autobauer beauftragt, die Verwaltung von Autobestandteilen softwareseitig zu unterstützen. Für ein Auto werden die folgenden Daten verwaltet: Typ, Farbe, Seriennummer, Produktionsdatum, Produktionsort, Getriebeart, Antriebsart, Höchstgeschwindigkeit und Gewicht. Für einen Motor werden folgende Daten benötigt: Motornummer, Treibstoffart, Leistung, Normverbrauch und Produktionsdatum. Für die Räder eines Autos werden Felgendurchmesser, Produktionsjahr, Geschwindigkeitsindex und Hersteller verwaltet.

Modellieren Sie mindestens die Klassen `auto`, `motor` und `rad` und implementieren Sie diese als C++-Klassen. „Bauen" Sie Autos mit unterschiedlichen Konfigurationen zu Testzwecken zusammen. Es muss auch eine einfache Ausgabe von Fahrzeugdaten mittels `operator<<` möglich sein. Testen Sie ihre Implementierung ausführlich, lesen Sie alle benötigten Testdaten mittels `operator>>` ein.

Beispiel 2: ADT „Graph" (`src/adt`)

(a) Der von Ihnen in Übung 5 implementierte ADT für die Darstellung von ungewichteten gerichteten Graphen durch eine Adjazenzmatrix soll als Ausgangspunkt für eine objektorientierte Implementierung dienen, die dann allerdings gewichtete gerichtete Graphen repräsentieren kann. Gute Kandidaten für die zu implementierenden Klassen sind `vertex_t` und `graph_t`. Instanzen der Klasse `vertex_t` sind benannte Knoten, wobei die Namen Zeichenketten (`std::string`) sind. Die Klasse `graph_t` muss mindestens die folgende Funktionalität aufweisen:

```
handle_t       add_vertex (vertex_t vertex);   // moves 'vertex' into graph
void           add_edge   (handle_t const from, handle_t const to, int const weight);
std::ostream & print      (std::ostream & out) const;
```

Fügen Sie Ihren Klassen weitere notwendige Methoden und Datenkomponenten hinzu und testen Sie Ihre Implementierung ausführlich. Verwenden Sie dafür Streams zusammen mit `operator<<` und `operator>>`.
Anmerkung: Die Klasse `handle_t` identifiziert einen Vertex in einem Graphen eindeutig.

(b) Es gibt einen interessanten gierigen (*greedy*) Algorithmus zur Berechnung des kürzesten Wegs zwischen zwei Knoten in einem Graphen. Recherchieren Sie in der Algorithmenliteratur, suchen Sie nach dem Dijkstra-Algorithmus (*single-source shortest path*). Implementieren Sie in Ihrer Klasse `graph_t` eine Methode

```
int shortest_path (handle_t const from, handle_t const to) const;
```

die die Länge des kürzesten Wegs zwischen den beiden Knoten `from` und `to` nach dem Verfahren von Dijkstra ermittelt.

Ausarbeitung 07

Niklas Vest

December 2, 2018

1 Autobau

1.1 Lösungsidee

Die lösungsidee für diese Aufgabe wurde im Endeffekt in der Angabe beschrieben. Die Klassen sind nach der Methode von Abbott entstanden. Ich stelle neben einem **default**-Konstruktor auch eine Überladung zur Verfügung, die ein Objekt "aus einem Inputstream" erstellt. Dieser Konstruktor lest den stream so lange bis er alle benötigten Daten hat. Dieser Konstruktor wird auch für die Überladung des bitwise right shift operators (») verwendet, um der Funktion keinen Zugang zu privaten Feldern geben zu müssen. Ähnlich stellt jede Klasse eine *print* Methode zur Verfügung, welche dann im overload des bitwise shift left operators («) verwendet wird. In meiner Klassen-"Hierarchie" bietet sich eine Generalisierung auf eine abstrakte "Serializable" (o. ä.) Klasse an, welche die Beschriebene Konstruktor-Überladung und die print-Methode bereit stellt. Ich habe mir den Aufwand aber nicht gemacht, weil es nicht explizit mit der Übung zusammenhängt.

1.2 Implementierung

Listing 1: main.cpp

```
1  #include <iostream>
2  #include <fstream>
3  #include "car.hpp"
4
5  using std::cout;
6  using std::endl;
7  using std::ifstream;
8
9  int main()
10 {
11     ifstream cars_file("./cars.txt");
12
13     car_t car;
14     while (cars_file.good()) {
15         cars_file >> car;
16         cout << car << endl;
```

```
17      }
18
19      return 0;
20  }
```

Listing 2: car.hpp

```
1   #pragma once
2   #include <string>
3   #include <chrono>
4   #include <iostream>
5   #include "motor.hpp"
6   #include "wheel.hpp"
7
8   /**
9    * Random types of cars.
10   */
11  enum class car_type_t
12  {
13      VAN, CONVERTIBLE, SUV, JEEP
14  };
15
16  /**
17   * Kinds of transmission.
18   */
19  enum class transmission_t
20  {
21      AUTOMATIC, MANUAL
22  };
23
24  class car_t
25  {
26  public: // typedefs
27
28      /**
29       * Kilometers per second.
30       */
31      using kmps = float;
32
33  public: // methods
34
35      car_t() = default;
36
37      /**
38       * Creates a car by reading the supplied input stream.
39       * @param is The stream containing car details.
40       */
41      explicit car_t(std::istream &is);
42
43      /**
44       * Prints a few car details to the supplied output stream.
45       * @param os The output stream to write the details to.
46       * @return The output stream (to allow for chaining).
47       */
48      std::ostream & print(std::ostream &os) const;
49
50  private: // members (should be self explanatory due to descriptive naming)
51      car_type_t      _type                   {car_type_t::SUV};
```

```
52      std::string     _color;
53      unsigned        _serial                 {};
54      time_t          _production_date        {};
55      std::string     _production_location;
56      transmission_t  _transmission           {transmission_t::MANUAL};
57      kmps            _max_speed              {};
58      motor_t         _motor;
59      wheel_t         _wheels;
60  };
61
62  std::ostream & operator<<(std::ostream &os, const car_t &car);
63  std::istream & operator>>(std::istream &is, car_t &car);
```

Listing 3: car.cpp

```
1   #include "car.hpp"
2   #include "motor.hpp"
3   #include "wheel.hpp"
4
5   using std::ostream;
6   using std::istream;
7   using std::endl;
8
9   car_t::car_t(std::istream &is)
10  {
11      int type;
12      is >> type;
13      _type = static_cast<car_type_t>(type);
14
15      is >> _color
16         >> _serial
17         >> _production_date
18         >> _production_location;
19
20      int transm;
21      is >> transm;
22      _transmission = static_cast<transmission_t>(transm);
23
24      is >> _max_speed
25         >> _motor
26         >> _wheels;
27  }
28
29  std::ostream &car_t::print(std::ostream &os) const
30  {
31      return os << "==== Car ====" << endl
32                << "Serial: " << _serial << endl
33                << "Color: " << _color << endl
34                << "Produced in: " << _production_location << endl
35                << "Max Speed: " << _max_speed << "km/s" << endl
36                << '\t' << _motor << endl
37                << '\t' << _wheels;
38  }
39
40  ostream & operator<<(ostream &os, const car_t &car)
41  {
42      return car.print(os);
43  }
```

```
44
45
46 istream & operator>>(istream &is, car_t &car)
47 {
48     car = car_t(is);
49     return is;
50 }
```

Listing 4: motor.hpp

```
1  #pragma once
2  #include <string>
3  #include <chrono>
4  #include <iostream>
5
6  /**
7   * All types of fuels (that are supported by this
8   * totally legitimate business software).
9   */
10 enum class fuel_t
11 {
12     DIESEL, GASOLINE
13 };
14
15 class motor_t
16 {
17 public: // typedefs
18
19     /**
20      * Horse power.
21      */
22     using hp = float;
23
24 public: // methods
25
26     motor_t() = default;
27
28     /**
29      * Creates a motor by reading the supplied input stream.
30      * @param is The stream containing motor details.
31      */
32     explicit motor_t(std::istream &is);
33
34     /**
35      * Prints a few motor details to the supplied output stream.
36      * @param os The output stream to write the details to.
37      * @return The output stream (to allow for chaining).
38      */
39     std::ostream & print(std::ostream &os) const;
40
41 private: // members
42     unsigned  _serial           {};
43     // "MY BLOOD IS GASOLINEEE" (R&M Reference)
44     fuel_t    _fuel             {fuel_t::GASOLINE};
45     hp        _hp               {};
46     float     _avg_consumption  {};
47     time_t    _production_date  {};
48 };
```

```
49
50 std::ostream & operator<<(std::ostream &os, const motor_t &motor);
51 std::istream & operator>>(std::istream &is, motor_t &motor);
```

Listing 5: motor.cpp

```
1  #include "motor.hpp"
2
3  using std::ostream;
4  using std::istream;
5  using std::endl;
6
7  motor_t::motor_t(std::istream &is)
8  {
9      is >> _serial;
10
11     int fuel;
12     is >> fuel;
13     _fuel = static_cast<fuel_t>(fuel);
14
15     is >> _hp
16        >> _avg_consumption
17        >> _production_date;
18 }
19
20 std::ostream &motor_t::print(std::ostream &os) const
21 {
22     return os << "==== Motor ====" << endl
23               << "Serial: " << _serial << endl
24               << "Torque: " << _hp << "hp" << endl
25               << "Avg Consumption: " << _avg_consumption << "l/100km";
26 }
27
28 ostream & operator<<(ostream &os, const motor_t &motor)
29 {
30     return motor.print(os);
31 }
32
33 istream & operator>>(istream &is, motor_t &motor)
34 {
35     motor = motor_t(is);
36     return is;
37 }
```

Listing 6: wheel.hpp

```
1  #pragma once
2  #include <string>
3  #include <chrono>
4  #include <iostream>
5
6  class wheel_t
7  {
8  public: // methods
9
10     wheel_t() = default;
11
12     /**
```

```
13      * Creates a wheel by reading the supplied input stream.
14      * @param is The stream containing wheel details.
15      */
16     explicit wheel_t(std::istream &is);
17
18     /**
19      * Prints a few wheel details to the supplied output stream.
20      * @param os The output stream to write the details to.
21      * @return The output stream (to allow for chaining).
22      */
23     std::ostream & print(std::ostream &os) const;
24
25 private: // members
26     float        _diameter          {};
27     time_t       _production_date   {};
28     char         _velocity_index    {};
29     std::string  _brand;
30 };
31
32 std::ostream & operator<<(std::ostream &, const wheel_t &);
33 std::istream & operator>>(std::istream &, wheel_t &);
```

Listing 7: wheel.cpp

```
1  #include "wheel.hpp"
2
3  using std::ostream;
4  using std::istream;
5  using std::endl;
6
7  wheel_t::wheel_t(std::istream &is)
8  {
9      is >> _diameter
10        >> _production_date
11        >> _velocity_index
12        >> _brand;
13 }
14
15 std::ostream &wheel_t::print(std::ostream &os) const
16 {
17     return os << "==== Wheels ====" << endl
18               << "Brand: " << _brand << endl
19               << "Diameter: " << _diameter << endl
20               << "Vel. Index: " << _velocity_index;
21 }
22
23 ostream & operator<<(ostream &os, const wheel_t &wheel)
24 {
25     return wheel.print(os);
26 }
27
28 istream & operator>>(istream &is, wheel_t &wheel)
29 {
30     wheel = wheel_t(is);
31     return is;
32 }
```

1.3 Tests

Listing 8: Test-Datei cars.txt

```
1 0 #fff 76549 0 NYC 1 100 4345 1 120 7.5 724 20.0 3765283 S BBS
2 2 #f00 40574 549805 LINZ 0 230 8534 0 200 9.1 892347 18.5 843743 V Enkei
```

```
=== Car ===
Serial: 76549
Color: #fff
Produced in: NYC
Max Speed: 100km/s
    === Motor ===
Serial: 4345
Torque: 120hp
Avg Consumption: 7.5l/100km
    === Wheels ===
Brand: BBS
Diameter: 20
Vel. Index: S
=== Car ===
Serial: 40574
Color: #f00
Produced in: LINZ
Max Speed: 230km/s
    === Motor ===
Serial: 8534
Torque: 200hp
Avg Consumption: 9.1l/100km
    === Wheels ===
Brand: Enkei
Diameter: 18.5
Vel. Index: V
```

Figure 1: Resultat von *wmb/main.cpp*

2 Graph

2.1 Lösungsidee (a)

Ich verwende in dieser Version des Graph-ADTs diverse STL Container. Dazu möchte ich anmerken, dass ich dazu im Stande wäre, jede von mir verwendete Funktionalität nachzubilden und ich bitte darum, keine Punkte für die Abkürzung abzuziehen. Ich habe dafür eine kleine Extra-Aufgabe gemacht und den Graph selbst zu einem Template-Container gemacht.
Die Knoten werden in einem assoziativen Feld (Map) gespeichert, deren Schlüssel vom Typ *handle_t* (ein schnöder *int*) und die assoziierten Werte vom Typ *vertex_t* sind. Somit kann ein *handle* einen Knoten im Graphen identifizieren. Die Adjazenzmatrix ist ein Vektor von Vektoren vom Typ *weight_t* (ebenfall ein *int*). Der Grund für diese Wahl ist die gemütliche Container-Vergrößerung von *std* :: *vector* und die intuitive Schreibweise $vec[x][y]$ die man aufgrund der Verschachtelung der Vektoren verwenden kann, um an das Gewicht jener Kante zu Kommen, welche vom Knoten x ausgeht und auf dem Knoten y landet.

2.2 Lösungsidee (b)

Ich habe den Algorithmus von Dijkstra aus dem Wikipedia-Artikel übernommen, möchte aber Anmerken, dass ich mich mehr mit dem Algorithmus beschäftigt habe, als bloß den Pseudo-Code abzutippen und zu übersetzen. Zusätzlich habe ich dem Graphen die Methode *get_shortest_path_between* gegeben, welche nicht nur die Länge, sondern den Ganzen Pfad vom Start- zum Zielknoten liefert.

2.3 Implementierung

Listing 9: main.cpp

```
1  #include "graph.hpp"
2
3  using handle_t = graph_t<char>::handle_t;
4  using std::string;
5  using std::cout;
6  using std::endl;
7
8  static void test_basic()
9  {
10     graph_t<char> graph;
11
12     handle_t a_handle = graph.add_vertex(vertex_t<char>('w'));
13     handle_t b_handle = graph.add_vertex(vertex_t<char>('a'));
14     handle_t c_handle = graph.add_vertex(vertex_t<char>('t'));
15
16     graph.add_edge(a_handle, a_handle, 1);
17     graph.add_edge(a_handle, b_handle, 2);
18     graph.add_edge(b_handle, c_handle, 10);
19     graph.add_edge(c_handle, b_handle, 5);
20
```

```
21      cout << graph << endl;
22  }
23
24  static void test_shortest_path()
25  {
26      graph_t<string> graph;
27
28      // add vertices
29      handle_t dis_handle          = graph.add_vertex(vertex_t<string>("This"))
         ;
30      handle_t is_handle           = graph.add_vertex(vertex_t<string>("is"));
31      handle_t a_handle            = graph.add_vertex(vertex_t<string>("a"));
32      handle_t sentence_handle     = graph.add_vertex(vertex_t<string>("
         sentence"));
33      handle_t distraction1_handle = graph.add_vertex(vertex_t<string>("memes")
         );
34      handle_t distraction2_handle = graph.add_vertex(vertex_t<string>("cookies
         "));
35
36      // add edges
37      graph.add_edge(dis_handle, is_handle, 1);
38
39          // distraction 1, those damn memes
40      graph.add_edge(is_handle, distraction1_handle, 1);
41      graph.add_edge(is_handle, a_handle, 1);
42      graph.add_edge(distraction1_handle, a_handle, 2);
43
44          // distraction 2
45      graph.add_edge(a_handle, distraction2_handle, 2);
46      graph.add_edge(a_handle, sentence_handle, 1);
47      graph.add_edge(distraction2_handle, sentence_handle, 1);
48
49      cout << graph << endl;
50
51      cout << "Shortest path from \""
52           << graph.get_vertex(dis_handle)
53           << "\" to \""
54           << graph.get_vertex(sentence_handle)
55           << "\": " << endl;
56      auto path = graph.get_shortest_path_between(dis_handle, sentence_handle);
57      for (const auto &vertex : path) {
58          cout << graph.get_vertex(vertex) << " ";
59      }
60      cout << "(length: " << graph.shortest_path(dis_handle, sentence_handle)
         << ")" << endl;
61  }
62
63  int main()
64  {
65      test_basic();
66      test_shortest_path();
67      return 0;
68  }
```

Listing 10: graph.hpp

```
1  #pragma once
2  #include <map>
```

```
3  #include <vector>
4  #include <iostream>
5  #include <algorithm>
6  #include <list>
7  #include "vertex.hpp"
8
9  template <typename T>
10 class graph_t
11 {
12 public: // typedefs
13
14     using handle_t = unsigned;
15     using weight_t = int;
16
17 public: // methods
18
19     /**
20      * Adds the supplied vertex to the graph.
21      * @param vertex The vertex to add.
22      * @return A handle to identify the added node.
23      */
24     graph_t::handle_t add_vertex(vertex_t<T> vertex);
25
26     /**
27      * @param handle The handle of the vertex to retrieve
28      * @return The vertex associated with a certain handle.
29      */
30     vertex_t<T> get_vertex(handle_t handle) const;
31
32     /**
33      * Adds an edge from the source node denoted by _from_
34      * to the destination node denoted by _to_ and applies
35      * the specified _weight_ to the edge.
36      * @param from The handle for the source node of the edge.
37      * @param to The handle for the destination node of the edge.
38      * @param weight The weight of the edge.
39      */
40     void add_edge(handle_t from, handle_t to, weight_t weight);
41
42     /**
43      * Prints the graph to the supplied output stream.
44      * @param out The output stream to write to.
45      * @return The used output stream.
46      */
47     std::ostream & print(std::ostream &os) const;
48
49     /**
50      * @param from The node at which to start searching for paths to _to_.
51      * @param to The node denoting the path destination.
52      * @return The computed shorted path between the vertices
53      *           associated with the handles _from_ and _to_.
54      */
55     std::vector<handle_t> get_shortest_path_between(handle_t from, handle_t
       to) const;
56
57     /**
58      * @param from The node at which to start searching for paths to _to_.
```

```
59      * @param to The node denoting the path destination.
60      * @return The length of the shortest path from _from_ to _to_.
61      * @note This method does not follow the naming scheme since I suppose
62      *       I am required to coincide with the exercises interface.
63      */
64     int shortest_path(handle_t from, handle_t to) const;
65
66     /**
67      * @param vertex The vertex of which to retrieve all neighbours.
68      * @return All neighbours of _vertex_.
69      */
70     std::vector<handle_t> get_neighbours_of(handle_t vertex) const;
71
72     /**
73      * @param from The source node of the edge.
74      * @param to The destination node of the edge.
75      * @return The weight of the edge going from _from_ to _to_.
76      */
77     weight_t get_edge_weight(handle_t from, handle_t to) const;
78
79 private: // methods
80
81     /**
82      * Asserts that the supplied handle is associated
83      * with a node within the graph.
84      * @param handle The handle to check.
85      * @param msg The message to display if the assertion fails.
86      * @throws invalid_argument if the handle does
87      *         not denote a node.
88      */
89     void _assert_valid_handle(handle_t handle, const std::string &msg) const;
90
91 private: // members
92
93     /**
94      * The adjacency matrix of the graph.
95      */
96     std::vector<std::vector<weight_t>> _adjacencies;
97
98     /**
99      * The nodes with their respective id.
100     */
101    std::map<handle_t, vertex_t<T>> _vertices;
102 };
103
104 template <typename T>
105 std::ostream & operator<<(std::ostream &os, const graph_t<T> & graph);
106
107 /* IMPLEMENTATIONS */
108
109 template <typename T>
110 auto graph_t<T>::add_vertex(const vertex_t<T> vertex) -> handle_t
111 {
112     handle_t new_handle = 0;
113     // find first free id
114     while (_vertices.find(new_handle) != std::cend(_vertices)) {
115         ++new_handle;
```

```
116     }
117
118     // if the new id is outside tha matrix' bounds
119     if (new_handle >= _adjacencies.size()) {
120         // add a new row
121         _adjacencies.push_back(std::vector<weight_t>(new_handle+1, 0));
122
123         // add a new column
124         for (std::size_t i = 0; i < new_handle + 1; ++i) {
125             _adjacencies[i].resize(new_handle+1, 0);
126         }
127     }
128
129     // insert vertex
130     _vertices.insert(std::make_pair(new_handle, vertex));
131     return new_handle;
132 }
133
134 template <typename T>
135 void graph_t<T>::add_edge(const handle_t from, const handle_t to, const
       weight_t weight)
136 {
137     _assert_valid_handle(from, "Invalid source vertex for a new edge.");
138     _assert_valid_handle(to, "Invalid destination vertex for a new edge.");
139     _adjacencies[from][to] = weight;
140 }
141
142 template <typename T>
143 std::ostream & graph_t<T>::print(std::ostream &os) const
144 {
145     // print nodes
146     os << "[   Vertices   ]" << std::endl;
147     auto it = std::cbegin(_vertices);
148     while (it != std::cend(_vertices)) {
149         os << (it->second);
150         ++it;
151         if (it != std::cend(_vertices)) {
152             os << ", ";
153         }
154     }
155     os << ";" << std::endl;
156
157     // print edges
158     os << "[   Edges   ]" << std::endl;
159     for (const auto &from : _vertices) {
160         for (const auto &to : _vertices) {
161             const auto & curr_weight = get_edge_weight(from.first, to.first);
162             if (curr_weight != 0) {
163                 os << from.second << " -> (" << curr_weight << ") -> " << to.
        second << std::endl;
164             }
165         }
166     }
167     return os;
168 }
169
170 template<typename T>
```

```
171 auto graph_t<T>::get_shortest_path_between(const handle_t from, const
        handle_t to) const -> std::vector<handle_t>
172 {
173     using std::cbegin;
174     using std::cend;
175 
176     std::list<handle_t> path;
177 
178     _assert_valid_handle(from, "Invalid start vertex");
179     _assert_valid_handle(to, "Invalid end vertex");
180 
181     // unvisited nodes
182     std::vector<handle_t> queue;
183     // accumulated (min) distances to a particular node
184     std::map<handle_t, weight_t> distances;
185     // path - construction helper
186     std::map<handle_t, handle_t> previous;
187 
188     for (const auto &vertex : _vertices) {
189         // add current vertex
190         queue.push_back(vertex.first);
191         // init distances with "infinity"
192         distances.insert(std::make_pair(vertex.first, std::numeric_limits<
        weight_t>::max()));
193         // init previous with garbage values
194         previous.insert(std::make_pair(vertex.first, -1)); // I use a signed
        literal to init a signed value *dabs*
195     }
196     distances[from] = 0;
197 
198     while (!queue.empty()) {
199         // find vertex with min distance
200         handle_t u = *std::min_element(cbegin(queue), cend(queue), [&
        distances](const auto &p1, const auto &p2) {
201             return distances[p1] < distances[p2];
202         });
203 
204         if (u == to) {
205             break;
206         }
207 
208         // remove u from Q
209         queue.erase(std::find(cbegin(queue), cend(queue), u));
210 
211         // find lowest-cost neighbour of u
212         for (handle_t vertex : get_neighbours_of(u)) {
213             weight_t alt_costs = distances[u] + get_edge_weight(u, vertex);
214             if (alt_costs < distances[vertex]) {
215                 distances[vertex] = alt_costs;
216                 previous[vertex] = u;
217             }
218         }
219     }
220 
221     // fill path list by reverse-iteration
222     handle_t u = to;
223     auto it = std::front_insert_iterator<std::list<handle_t>>(path);
```

```
224     if (previous.find(u) != cend(previous) || u == from) {
225         while (previous.find(u) != cend(previous)) {
226             *it = u;
227             u = previous[u];
228         }
229     }
230 
231     return std::vector<handle_t>(cbegin(path), cend(path));
232 }
233 
234 template<typename T>
235 int graph_t<T>::shortest_path(const handle_t from, const handle_t to) const
236 {
237     return static_cast<int>(get_shortest_path_between(from, to).size());
238 }
239 
240 template<typename T>
241 vertex_t<T> graph_t<T>::get_vertex(handle_t handle) const
242 {
243     auto it = _vertices.find(handle);
244     if (it == std::cend(_vertices)) {
245         throw std::invalid_argument("Invalid vertex handle");
246     }
247     return it->second; // safe do deref iterator here!
248 }
249 
250 template<typename T>
251 auto graph_t<T>::get_neighbours_of(handle_t vertex) const -> std::vector<
        handle_t>
252 {
253     std::vector<handle_t> neighbours{};
254     for (const auto &to : _vertices) {
255         if (get_edge_weight(vertex, to.first) != 0) {
256             neighbours.push_back(to.first);
257         }
258     }
259     return neighbours;
260 }
261 
262 template<typename T>
263 auto graph_t<T>::get_edge_weight(const handle_t from, const handle_t to)
        const -> weight_t
264 {
265     _assert_valid_handle(from, "Invalid start vertex");
266     _assert_valid_handle(to, "Invalid end vertex");
267     return _adjacencies[from][to];
268 }
269 
270 template<typename T>
271 void graph_t<T>::_assert_valid_handle(const handle_t handle, const std::
        string &msg) const
272 {
273     if (_vertices.find(handle) == std::cend(_vertices)) {
274         throw std::invalid_argument(msg);
275     }
276 }
277 
```

```
278 template <typename T>
279 std::ostream & operator<<(std::ostream &os, const graph_t<T> & graph)
280 {
281     return graph.print(os);
282 }
```

Listing 11: vertex.hpp

```
 1 #pragma once
 2 #include <iostream>
 3 #include <memory>
 4
 5 template <typename T>
 6 class vertex_t;
 7
 8 template <typename T>
 9 std::ostream & operator<<(std::ostream &os, const vertex_t<T> &vertex);
10
11 template <typename T>
12 class vertex_t
13 {
14     template <typename _T>
15     friend std::ostream & operator<<(std::ostream &os, const vertex_t<_T> &
       vertex);
16
17 public: // methods
18
19     /**
20      * Avoids unnecessary copies by forwarding
21      * construction to make_shared.
22      * @tparam _Constr The types of the arguments used by
23      *                 one of the constructors for T.
24      * @param args The T-Constructors arguments.
25      */
26     template <typename ..._Constr>
27     explicit vertex_t(_Constr &&...args)
28     : _value {std::make_shared<T>(std::forward<_Constr>(args)...)} { }
29
30 private: // members
31
32     /**
33      * The information held by a vertex.
34      */
35     std::shared_ptr<T> _value;
36 };
37
38 template <typename T>
39 std::ostream & operator<<(std::ostream &os, const vertex_t<T> &vertex)
40 {
41     return os << *vertex._value;
42 }
```

2.4 Tests

```
[   Vertices   ]
w, a, t;
[   Edges   ]
w -> (1) -> w
w -> (2) -> a
a -> (10) -> t
t -> (5) -> a

[   Vertices   ]
This, is, a, sentence, memes, cookies;
[   Edges   ]
This -> (1) -> is
is -> (1) -> a
is -> (1) -> memes
a -> (1) -> sentence
a -> (2) -> cookies
memes -> (2) -> a
cookies -> (1) -> sentence

Shortest path from "This" to "sentence":
This is a sentence (length: 4)
```

Figure 2: Resultat von *graph/main.cpp*