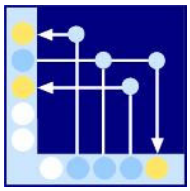

Algorithmen und Datenstrukturen 2

Vorlesung im Wintersemester 2024/2025
Prof. Dr. habil. Christian Heinlein

3. Praktikumsaufgabe (19. Dezember 2024 – 27. Januar 2025)

Aufgabe 3: Graphalgorithmen

Implementieren Sie folgende Graphalgorithmen in C++, indem Sie den Code, der in der Datei `graph.h` auf der Vorlesungswebseite vorgegeben ist, entsprechend ergänzen:

- Breitensuche
- Tiefensuche einschließlich topologischer Sortierung
- Bestimmung starker Zusammenhangskomponenten
- Bestimmung minimaler Gerüste nach Prim
- Bestimmung kürzester Wege nach Bellman-Ford und Dijkstra

Ein Graph kann hierbei ein Objekt `g` eines prinzipiell beliebigen Typs `G` sein, der mindestens folgende Elementfunktionen besitzen muss:

- `g.vertices()` liefert ein Objekt eines prinzipiell beliebigen Containertyps (z. B. `std::list`), das die Knoten des Graphen `g` (in irgendeiner Reihenfolge) enthält.
  Ein solcher Containertyp muss zumindest eine parameterlose Elementfunktion `size` besitzen, die die Anzahl der Elemente des Containers liefert (sodass `g.vertices().size()` die Anzahl der Knoten des Graphen `g` liefert).
  Außerdem muss es möglich sein, die Elemente eines Containers mittels „range-based for loops“ zu durchlaufen (sodass die Variable `v` in der Anweisung `for (V v : g.vertices()) ......` alle Knoten des Graphen `g` durchläuft, wenn diese Typ `V` besitzen).
  Alle Containertypen der C++-Standardbibliothek erfüllen diese Anforderungen.

- Für einen Knoten `u` des Graphen `g` liefert `g.successors(u)` wiederum ein Objekt eines derartigen Containertyps, das die Nachfolger des Knotens `u` (in irgendeiner Reihenfolge) enthält. (Wenn `u` kein Knoten des Graphen `g` ist, darf das Verhalten von `g.successors(u)` undefiniert sein.)

- `g.transpose()` liefert den transponierten Graphen von `g` als neues, von `g` unabhängiges Objekt (d. h. der Graph `g` wird dabei nicht verändert).
  Der Typ des transponierten Graphen kann sich vom Typ `G` des Graphen `g` unterscheiden, muss aber ebenfalls die gerade beschriebenen Elementfunktionen besitzen.

Ein gewichteter Graph kann ebenfalls ein Objekt eines prinzipiell beliebigen Typs sein, der neben den oben genannten Elementfunktionen eine weitere Elementfunktion `weight` besitzt, sodass `g.weight(u, v)` für einen Knoten `u` des Graphen `g` und einen Nachfolger `v` von `u` das Gewicht der Kante von `u` nach `v` als `double`-Wert liefert. (Wenn `u` kein Knoten des Graphen `g` ist oder `v` kein Nachfolger von `u` ist, darf das Verhalten von `g.weight(u, v)` undefiniert sein.)

Die vorgegebenen Klassen `Graph` und `WeightedGraph` erfüllen die oben genannten Anforderungen und können wie folgt zur Erzeugung von Testgraphen verwendet werden:

- Für einen prinzipiell beliebigen Knotentyp `V` kann ein Objekt des Typs `Graph<V>` mit der Adjazenzlistendarstellung eines Graphen als sogenannte Initialisiererliste in geschweiften Klammern initialisiert werden: Jedes Element dieser Initialisiererliste ist dabei ein Paar (ebenfalls in geschweiften Klammern), das aus einem Knoten des Typs `V` und einer weiteren Initialisiererliste mit den Nachfolgern dieses Knotens besteht, zum Beispiel:

  ```
  Graph<string> g({               // Graph g mit Knoten des Typs string.
      { "A", { "B", "C" } },      // Knoten A hat Nachfolger B und C.
      { "B", { } },               // Knoten B hat keine Nachfolger.
      { "C", { "C" } }            // Knoten C hat sich selbst als Nachfolger.
  });
  ```

  Für diesen Graphen `g` liefert `g.vertices()` einen Container mit den Elementen `"A"`, `"B"` und `"C"`, `g.successors("A")` einen Container mit den Elementen `"B"` und `"C"`, `g.successors("B")` einen leeren Container und `g.successors("C")` einen Container mit dem Element `"C"`.

- Analog kann ein Objekt des Typs `WeightedGraph<V>` mit der um Kantengewichte erweiterten Adjazenzlistendarstellung eines Graphen initialisiert werden, in der die „inneren“ Initialisiererlisten anstelle von Nachfolgern wiederum Paare in geschweiften Klammern enthalten, die jeweils aus einem Nachfolger und dem zugehörigen Kantengewicht mit Typ `double` bestehen, zum Beispiel:

  ```
  WeightedGraph<string> wg({
      { "A", { { "B", 2 }, { "C", 3 } } },
      { "B", { } },
      { "C", { { "C", 4 } } }
  });
  ```

  Für diesen Graphen `wg` liefern die Aufrufe von `vertices` und `successors` dieselben Ergebnisse wie für den obigen Graphen `g`; `wg.weight("A", "B")` liefert den `double`-Wert 2, `wg.weight("A", "C")` den Wert 3 und `wg.weight("C", "C")` den Wert 4.

Alle Algorithmen müssen aber auch für andere Typen `G` funktionieren, sofern sie die oben genannten Anforderungen erfüllen.

Jeder Algorithmus wird durch eine Funktionsschablone (function template) mit Typparametern `V` (Typ der Knoten) und `G` (Typ des Graphen) implementiert, die als ersten Parameter einen Graphen `g` mit Typ `G` und als zweiten Parameter eventuell einen Startknoten `s` mit Typ `V` oder eine Liste `vs` von Knoten erhält. Der letzte Parameter, der per Referenz übergeben wird, ist immer eine geeignete Datenstruktur, in der die Ergebnisse des Algorithmus – meist in einer oder mehreren Tabellen (maps) – gespeichert werden.

Eine typische Verwendung sieht zum Beispiel wie folgt aus:

```
// Testgraph mit Knoten des Typs string.
Graph<string> g({
    { "A", { "B", "C" } },      // Knoten A hat Nachfolger B und C.
    { "B", { } },               // Knoten B hat keine Nachfolger.
    { "C", { "C" } }            // Knoten C hat sich selbst als Nachfolger.
});

// Tiefensuche auf g ausführen und das Ergebnis in res speichern.
DFS<string> res;
dfs(g, res);
```

```
// Die Knoten v des Graphen nach aufsteigenden Abschlusszeiten
// durchlaufen und für jeden Knoten seine Entdeckungs- und
// Abschlusszeit ausgeben.
for (string v : res.seq) {
    cout << v << " " << res.det[v] << " " << res.fin[v] << endl;
}
```

Für Algorithmen, die eine Vorrangwarteschlange benötigen, verwenden Sie die auf der Vorlesungswebseite in der Datei `prioqueue.h` bereitgestellte Klasse `PrioQueue` mit derselben Funktionalität wie `BinHeap` aus Aufgabe 2! (Die Bibliotheksklasse `std::priority_queue` ist nicht geeignet, weil sie keine Möglichkeit bietet, die Priorität eines Eintrags nachträglich zu verändern.)

Testen Sie Ihre Implementierung mit unterschiedlichen Graphen und ggf. unterschiedlichen Startknoten sorgfältig und ausführlich!

Auf der Vorlesungswebseite steht hierfür ein Testprogramm `graphtest.cxx` zur Verfügung, das abhängig von den übergebenen Kommandozeilenargumenten einen bestimmten Algorithmus auf einem bestimmten Graphen ggf. mit einem bestimmten Startknoten ausführt und die vom Algorithmus ermittelte Information auf der Standardausgabe ausgibt. Die Liste der Testgraphen kann nach Belieben erweitert werden.

Beim Algorithmus von Prim ist darauf zu achten, dass der jeweilige Testgraph ungerichtet sein muss, d. h. er muss zu jeder Kante auch die entgegengesetzte Kante mit dem gleichen Gewicht enthalten.

Die Datei `container.cxx` auf der Vorlesungswebseite enthält Beispiele zur Verwendung der Containertypen `std::list` und `std::map` der C++-Standardbibliothek.

Um automatisierte Tests der Implementierungen zu ermöglichen, dürfen die vorgegebenen Klassennamen und die Signaturen von Funktionen nicht verändert werden, und es dürfen keine Diagnoseausgaben produziert werden.

Abgesehen von den in `graph.h` bereits verwendeten Bibliotheksklassen (`numeric_limits`, `list`, `map` und `pair`) dürfen keine Bestandteile der C++-Standardbibliothek oder anderer Bibliotheken verwendet werden.

Abzugeben ist die durch Ihren Code erweiterte Datei `graph.h`.
Die E-Mail mit der Abgabe muss als Betreff `Algo2 Gruppe NN` mit zweistelliger Gruppennummer `NN` (z. B. `05` oder `12`) und als Anhang die abzugebende Datei haben. Der Nachrichtentext muss für jedes Gruppenmitglied aus einer Zeile der Art `Vorname,Nachname,Matrikelnummer` bestehen.