Ausgewählte Algorithmen auf Graphen

Einführung

In diesem Praktikum ermitteln Sie kürzeste Wege in Graphen: verpflichtend mit dem Dijkstra-Algorithmus, zusätzlich, freiwillig mit dem Floyd-Warshall-Algorithmus. Sie verarabeiten weiterhin Eingabedaten von der Standardeingabe und geben diesmal Entfernungen und Vorgängerknoten über die Standardausgabe aus.

Single-Responsibility-Principle

Klassen sollten nach Möglichkeit nur eine Aufgabe haben und diese bestmöglich erledigen. Ihre Graph-Klasse verwaltet momentan Knoten und deren Nachbarschaften. Vermutlich kümmert sie sich zusätzlich auch um die Ausgabe entsprechend der vorherigen Praktikumsaufgaben. Genau genommen hat die Klasse damit schon zu viele Aufgaben. Die Ausgabe ließe sich leicht auslagern (siehe Vertiefung im vorherigen Praktikum) und Ihre Graph-Klasse erfüllt das Single-Responsibility-Principle[1].
Damit Ihre Graph-Klasse nicht noch mehr Aufgaben erhält, sollten Sie die in diesem Praktikum zu entwickelnden Algorithmen jeweils in einer separaten Klasse implementieren. So wird Ihr Code deutlich modularer, lässt sich fokussierter entwickeln und warten und veraussichtlich einfacher wiederverwenden.
Graph und Algorithmus sind dann eigenständige Klassen, deren Interna jeweils vollkommen unabhängig voneinander entwickelt und verändert werden können. Lediglich eine gemeinsam vereinbarte Schnittstelle muss eingehalten werden. In diesem Praktikum benötigen die Algorithmen
die Anzahl der Knoten im Graph sowie eine Information, ob zwei per Index angegebene Knoten verbunden sind. Dies lässt sich über Methoden der Graph-Klasse leicht bewerkstelligen.
Der jeweilige Algorithmus benötigt dann lediglich eine Referenz auf den Graphen und kann die zusätzlich benötigten Datenstrukturen selbst initialisieren und verwalten. Die Initialisierung dieser Daatenstrukturen erfolgt im Konstruktor. Die eigentliche Arbeit kann in einer Methode erfolgen, beispielsweise im `operator()`. Dieser Operator kann die ermittelten Entfernungen und Vorgänger zurückgeben, per Rückgabeparameter oder als `std::pair`[2]. Alternativ sind hier durchaus auch Getter gerechtfertigt.

Magic Numbers

Sowohl beim Dijkstra-Algorithmus als auch beim Floyd-Warshall-Algorithmus benötigen Sie eine Kennzeichnung für *kein Vorgänger*. Im Gegensatz zu Fließkomma-Datentypen, die den Wert *not a number* kennen[3][4], haben Ganzzahl-Datentypen keinen solchen speziellen Wert. Um ein zusätliches Datenfeld für *kein Vorgänger* zu vermeiden, sollten Sie einen Zahlenwert für *kein Vorgänger* selbst festlegen und reservieren, d.h. nicht als Knotenindex verwenden.
Hierfür bietet sich beispielsweise die größte, im gewählten Datentyp darstellbare Ganzzahl an. Bei `std::size_t` ist dies `std::numeric_limits<std::size_t>::max()`[5]. Damit dieser spezielle Zahlenwert im Code unter einem sprechenden Namen verfügbar wird und nicht als *magic number* vom Himmel fällt, legen Sie in Ihrer Algorithmus-Klasse eine statische, öffentliche Konstante oder sogar `constexpr`[6] an, die genau diesen Wert erhält. Auf diese Weise wird Ihr Code deutlich lesbarer und Ihre Intention klarer.

Algorithmen in der Standardbibliothek

Die C++-Standardbibliothek – genauer die Standard-Template-Library (STL) – stellt Ihnen neben Containern auch Algorithmen zur Verfügung, die auf den Containern arbeiten. Um diese Algorithmen nutzen zu können, binden Sie den Header `algorithm` ein.
Je nachdem wie Sie den Dijkstra-Algorithmus implementieren, kann es sein, dass Sie eine Liste aller mittlerweile als erreichbar eingestuften, noch zu verarbeitenden Knoten mitführen. Diese können Sie mit einem `std::vector` implementieren, der die Indizes der noch zu verarbeitenden Knoten aufnimmt. Wenn Sie diesem Container weitere Knoten hinzufügen, kann es sein, das Duplikate entstehen. Um diese wieder zu entfernen, sortieren Sie den Container mit `std::sort`[7], entfernen benachbarte Duplikate mit `std::unique`[8] und entfernen überflüssige

[1] Robert C. Martin: The Single Responsibility Principle. In The Clean Code Blog. 2014. Link
[2] `https://en.cppreference.com/w/cpp/utility/pair`
[3] `https://en.cppreference.com/w/cpp/numeric/math/nan`
[4] `https://en.cppreference.com/w/cpp/numeric/math/NAN`
[5] `https://en.cppreference.com/w/cpp/types/numeric_limits/max`
[6] `https://en.cppreference.com/w/cpp/language/constexpr`
[7] `https://en.cppreference.com/w/cpp/algorithm/sort`
[8] `https://en.cppreference.com/w/cpp/algorithm/unique`

Elemente am Ende mit `std::vector<T>::erase`[9]. Der letzte Schritt ist notwendig, da durch `std::unique` lediglich Elemente nach vorne rücken, die Kapazität aber nicht verrignert wird und am Ende Duplikate der letzten Elemente entstehen (siehe auch *dynamische Arrays* in GIP1).

Lambda-Ausdrücke – für Fortgeschrittene

Lambda-Ausdrücke[10] erzeugen ein aufrufbares Objekt, das Sie beispielsweise in einer Variable ablegen und ähnlich wie einen Funktionspointer aufrufen können. Diese Funktions-Objekte können im Code z.B. einer anderen Funktion oder Methode erstellt werden und die im jeweiligen Code-Block gültigen Objekte und Variablen verwenden.
Bisher haben Sie komplexe Funktionalität in Form von Funktionen oder Methoden gruppiert. Im Sinne objektorientierter Programmierung fügt eine Methode einer Klasse bestimmtes Verhalten hinzu. Wenn Sie allerdings die Gruppierung von Funktionalität benötigen ohne Verhalten erzeugen zu wollen – öffentlich oder privat – bieten sich Lambda-Ausdrücke an.
Dazu ein Beispiel, das allerdings etwas weiter ausholt. Gegeben sei folgender Code:

```
std::vector<int> values{4, 9, 2, 6, 0, 3, 5, 5, 0, 9};
std::vector<std::size_t> to_be_processed{2,3,5,7};
```

`values` enthält Elemente, die einen Wert haben. `to_be_processed` listet die Indizes der noch zu verarbeitenden Elemente aus `valuesauf`. Aufgabe ist nun, das Element aus `values` zu ermitteln, das

- noch zu verarbeiten ist
- *und* den kleinsten Wert hat.

Mit `std::min_element`[11] können Sie direkt einen Iterator auf das Element in `values` bestimmen, das den kleinsten Wert hat. Um das kleinste, noch zu verarbeitende Element zu bestimmen, müssen Sie einen Umweg über `to_be_processed` gehen. Dieser Code

```
const auto compare = [&values](const std::size_t& a,
                               const std::size_t& b) {
  return values.at(a) < values.at(b);
};
```

erzeugt mit dem enthaltenen Lambda-Ausdruck der Form `[...](...){...}` ein aufrufbares Objekt `compare` das zwei `std::size_t`-Referenzen als Indizes in `values` übernimmt und die bezeichneten Elemente miteinander vergleicht. Der Ausdruck in Eckigen Klammern sorgt dafür, dass die lokale Variable `values` als Referenz im Funktionskörper des Lambda Ausdrucks verfügbar wird. Nun können Sie mit

```
std::min_element(to_be_processed.begin(), to_be_processed.end(), compare);
```

einen Iterator auf das Element in `to_be_processed` ermitteln, das den Index des kleinsten, noch zu verarbeitenden Elements in `values` darstellt. Der Vergleich erfolgt dabei intern durch Aufruf des in `compare` abgelegten Funktions-Objekts.
Sie hätten `std::min_element` auch einen Funktions-Zeiger auf eine selbstimplementierte Vergleichsfunktion übergeben können. Dieser kommt in diesem Fall aber deutlich mehr Bedeutung zu als nötig; je nachdem wie Sie diese Funktion anlegten, würde Ihr Code dieses neue Verhalten auch anderen Nutzern (im Sinne von Code-Stellen) zur Verfügung stellen. Durch den Lambda-Ausdruck bleibt diese Funktionalität hingegen absolut lokal.
Für weitere Erklärungen sei auf die Dokumentation zu Lambda-Ausdrücken verwiesen.

Einstieg

1. Minimales Element eines Vektors

Beginnen Sie mit einer Hello-World-Applikation.

Gegeben sei folgender Container:

[9] https://en.cppreference.com/w/cpp/container/vector/erase
[10] `https://en.cppreference.com/w/cpp/language/lambda`
[11] `https://en.cppreference.com/w/cpp/algorithm/min_element`

```
std::vector<int> v{13, 55, 74, 7, 75, 41, 16, 87, 34, 46};
```

(a) Schreiben Sie Code, mit dem Sie den kleinsten Wert in diesem Container bestimmen.

(b) Schreiben Sie Code, der den Index des Elements mit dem kleinsten Wert ermittelt.

2. Index-Array

Beginnen Sie mit einer Hello-World-Applikation.

Gegeben seien folgende Container:

```
std::vector<int> v{13, 55, 74, 7, 75, 41, 16, 87, 34, 46};
std::vector<std::size_t> i{2, 0, 5};
```

(a) Schreiben Sie Code, der den Inhalt des Containers `v` ausgibt. Jedes Element wird in einem zwei Zeichen breiten Feld ausgegeben. Verwenden Sie hierzu `std::setw`. Die Felder sind durch ein einzelnes Leerzeichen getrennt. Am Ende der Zeile steht ein einzelnes Leerzeichen.
Gewünschte Ausgabe:

```
13 55 74  7 75 41 16 87 34 46
```

(b) Schreiben Sie Code, der lediglich die Elemente von `v` ausgibt, die durch die Elemente in `i` indiziert werden.
Gewünschte Ausgane:

```
74 13 41
```

3. 2D-Array mit Containern

Beginnen Sie mit einer Hello-World-Applikation.

(a) Legen Sie mittels verschachtelter Container vom Typ `std::vector` ein 2D-Array von Ganzzahlen an.

(b) Initialisieren Sie alle Einträge mit dem Wert `0`.

(c) Geben Sie das 2D-Array am Bildschirm aus.

(d) Setzen Sie einige selbst gewählte Einträge jeweils auf einen anderen, selbst gewählten Wert.

(e) Prüfen Sie, ob diese Werte an den richtigen Stellen in der Ausgabe stehen.

Aufgabe

4. Dijkstra-Algorithmus

Arbeiten Sie mit der Projektmappe aus dem vorherigen Praktikum weiter.

Implementieren Sie den Dijkstra-Algorithmus. Geben Sie sowohl die kürzesten Entfernungen vom gegebenen Startknoten zu allen anderen Knoten als auch die jeweiligen Vorgängerknoten auf dem kürzesten Weg aus. Der Start-Knoten wird als Kommandozeilenparameter angegeben. Die Kantengewichte betragen 1.

Folgende Eingabe entspricht dem Graphen, der in der Vorlesung verwendet wurde:

```
10↩
L␣v0␣0␣1↩
L␣v1␣1␣0↩
L␣v2␣2␣1↩
L␣v3␣1␣2↩
L␣v4␣2␣0↩
L␣v5␣3␣2↩
L␣v6␣3␣0↩
L␣v7␣4␣2↩
L␣v8␣4␣0↩
L␣v9␣5␣1↩
E↩
0␣1↩
0␣2↩
0␣3↩
1␣4↩
2␣5↩
2␣9↩
3␣5↩
4␣6↩
5␣7↩
5␣9↩
6␣8↩
6␣9↩
7␣9↩
8␣9↩
```

Ihre Implementierung sollte für den Startknoten v_0 folgende Ausgabe erzeugen:

```
␣0␣␣1␣␣1␣␣1␣␣2␣␣2␣␣3␣␣3␣␣3␣␣2␣↩
↩
--␣␣0␣␣0␣␣0␣␣1␣␣2␣␣4␣␣5␣␣9␣␣2␣↩
```

In der ersten Zeile stehen die Entfernungen. Zur besseren Übersicht folgt eine Leerzeile. In der dritten Zeile stehen die jeweiligen Vorgängerknoten. Innerhalb der ersten und dritten Zeile beziehen sich die einzelnen Felder jeweils auf einen Knoten in der Reihenfolge, in der die Knoten in der Eingabe angegeben wurden. Jedes Feld ist genau zwei Zeichen breit. Dies erreichen Sie durch `std::setw`[12]. Die einzelnen Felder sind durch ein einzelnes Leerzeichen voneinander getrennt. Am Zeilenende steht ein einzelnes Leerzeichen.

Für Entfernungen `dn` und Vorgänger `pn` eines Knotens v_n ergibt sich also:

```
d0␣d1␣d2␣d3␣d4␣d5␣d6␣d7␣d8␣d9↩
↩
p0␣p1␣p2␣p3␣p4␣p5␣p6␣p7␣p8␣p9↩
```

Vertiefung

5. Floyd-Warshall-Algorithmus

Arbeiten Sie mit der Projektmappe aus der vorherigen Aufgabe weiter.

Implementieren Sie den Floyd-Warshall-Algorithmus. Geben Sie sowohl die Matrix mit den kürzesten Entfernungen zwischen den Knoten als auch die jeweiligen Vorgängerknoten auf dem kürzesten Weg aus. Die Kantengewichte betragen weiterhin 1.

Ihre Implementierung sollte für die Eingabe aus der voerherigen Aufgabe folgende Ausgabe erzeugen:

[12] `https://en.cppreference.com/w/cpp/io/manip/setw`

```
␣0␣␣1␣␣1␣␣1␣␣2␣␣2␣␣3␣␣3␣␣3␣␣2␣↩
␣1␣␣0␣␣2␣␣2␣␣1␣␣3␣␣2␣␣4␣␣3␣␣3␣↩
␣1␣␣2␣␣0␣␣2␣␣3␣␣1␣␣2␣␣2␣␣2␣␣1␣↩
␣1␣␣2␣␣2␣␣0␣␣3␣␣1␣␣3␣␣2␣␣3␣␣2␣↩
␣2␣␣1␣␣3␣␣3␣␣0␣␣3␣␣1␣␣3␣␣2␣␣2␣↩
␣2␣␣3␣␣1␣␣1␣␣3␣␣0␣␣2␣␣1␣␣2␣␣1␣↩
␣3␣␣2␣␣2␣␣3␣␣1␣␣2␣␣0␣␣2␣␣1␣␣1␣↩
␣3␣␣4␣␣2␣␣2␣␣3␣␣1␣␣2␣␣0␣␣2␣␣1␣↩
␣3␣␣3␣␣2␣␣3␣␣2␣␣2␣␣1␣␣2␣␣0␣␣1␣↩
␣2␣␣3␣␣1␣␣2␣␣2␣␣1␣␣1␣␣1␣␣1␣␣0␣↩
↩
--␣␣0␣␣0␣␣0␣␣1␣␣2␣␣4␣␣5␣␣9␣␣2␣↩
␣1␣--␣␣0␣␣0␣␣1␣␣2␣␣4␣␣5␣␣6␣␣2␣↩
␣2␣␣0␣--␣␣0␣␣1␣␣2␣␣9␣␣5␣␣9␣␣2␣↩
␣3␣␣0␣␣0␣--␣␣1␣␣3␣␣9␣␣5␣␣9␣␣5␣↩
␣1␣␣4␣␣0␣␣0␣--␣␣9␣␣4␣␣9␣␣6␣␣6␣↩
␣2␣␣0␣␣5␣␣5␣␣6␣--␣␣9␣␣5␣␣9␣␣5␣↩
␣1␣␣4␣␣9␣␣5␣␣6␣␣9␣--␣␣9␣␣6␣␣6␣↩
␣2␣␣0␣␣5␣␣5␣␣6␣␣7␣␣9␣--␣␣9␣␣7␣↩
␣2␣␣4␣␣9␣␣5␣␣6␣␣9␣␣8␣␣9␣--␣␣8␣↩
␣2␣␣0␣␣9␣␣5␣␣6␣␣9␣␣9␣␣9␣␣9␣--␣↩
```

Im ersten Block stehen die Entfernungen. Zur besseren Übersicht folgt eine Leerzeile. Im zweiten Block stehen die jeweiligen Vorgängerknoten. Innerhalb jedes Blocks beziehen sich sowohl die einzelnen Zeilen als auch die einzelnen Spalten jeweils auf einen Knoten in der Reihenfolge, in der die Knoten in der Eingabe angegeben wurden. Jedes Feld ist genau zwei Zeichen breit. Dies erreichen Sie durch `std::setw`[13]. Die einzelnen Felder sind durch ein einzelnes Leerzeichen voneinander getrennt. Am Zeilenende steht ein einzelnes Leerzeichen. Die Blöcke geben Entfernungen und Vorgänger jeweils für ein Paar aus Start- (Zeile) und Zielknoten (Spalte) an. Gibt es für ein Paar aus Start- und Zielknoten keinen Vorgänger, steht im jeweiligen Feld ein doppelter Trennstrich (`--`).

In der Vorlesung wurde gezeigt, wie Sie die Entfernungsmatrix initialisieren und aktualisieren. Überlegen Sie selbst, wie Sie die Vorgängermatrix initialisieren können. Für die Aktualisierung der Vorgänger gilt dies:

- Gegeben seien Start- (v_s), Zwischen- (v_z) und Endknoten (v_e).
- $p(v_i, v_k)$ liefert Ihnen den Vorgänger auf dem Weg von v_i zu v_k.
- Der aktualisierte Vorgänger $p'(v_s, v_e)$ lautet $p(v_z, v_e)$.

6. Writer-Klassen für Graph-Algorithmen

Arbeiten Sie mit der Projektmappe aus der vorherigen Aufgabe weiter.

Legen Sie wie in der Vertiefung des vorherigen Praktikums für jeden implementierten Algorithmus Writer-Klassen an, die die Entfernungen und Vorgänger, die durch den jeweiligen Algorithmus ermittelt wurden, im passenden Format in einen Ausgabestream ausgeben.

7. For Each X

Arbeiten Sie mit der Projektmappe aus der vorherigen Aufgabe weiter.

In den Graph-Algorithmen, die Sie implementiert haben, sollten Sie Schleifen erkennen, die sich als

- `for_each_neighbour` oder
- `for_each_edge`

bezeichnen lassen.

(a) Extrahieren Sie diese jeweils als eigenen Algorithmus; legen Sie selbst fest, ob als eigene Klasse oder als freistehende Funktion. Übergeben Sie den Operator – das, was für jeden Nachbarn/jede Kante ausgeführt werden soll – als Funktionsobjekt vom Typ `std::function`[a]. Legen Sie dieses z.B. mittels Lambda-Ausdruck an.

[a] `https://en.cppreference.com/w/cpp/utility/functional/function`

(b) Implementieren Sie Iteratoren für Nachbarn, Kanten etc., die kompatibel zu Range-based-`for`-Schleifen und den STL-Algorithmen sind. Verwenden Sie diese in Ihrem Code entsprechend.

[13] `https://en.cppreference.com/w/cpp/io/manip/setw`