Aufgabenblatt 8

Single Source Shortest Paths

Abgabe (bis 26.06.2023 23:59 Uhr)

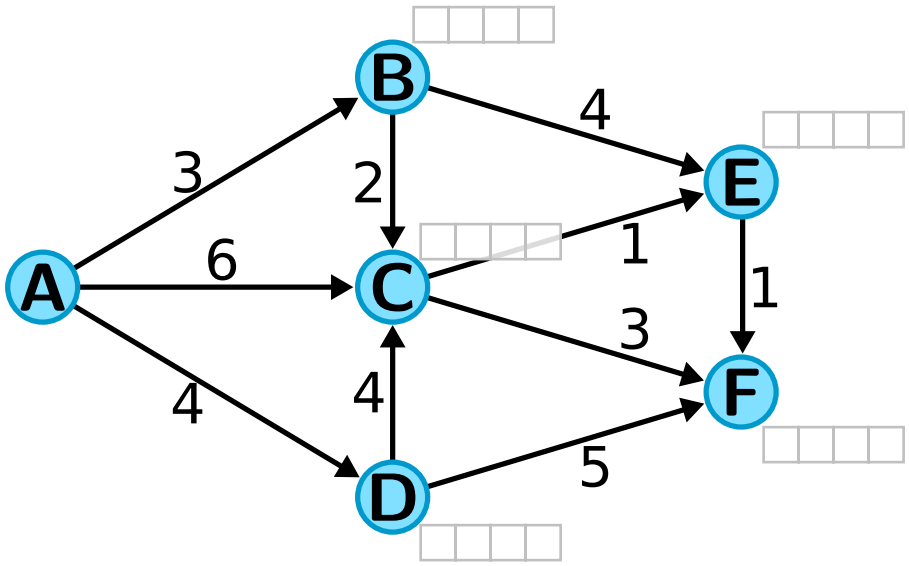
Die folgenden Dateien werden für die Bepunktung berücksichtigt:

`Blatt08/src/MatrixImage.java`	Aufgabe 3.2
`Blatt08/src/ShortestPathsTopological.java`	Aufgabe 3.3
`Blatt08/src/ContentAwareImageResizing.java`	Aufgabe 3.4

Als Abgabe wird jeweils nur die letzte Version im git gewertet.

Aufgabe 1: Algorithmus von Dijkstra (Klausurvorbereitung)

In der folgenden Abbildung ist ein kantengewichteter Digraph dargestellt. Die Zahlen an den Pfeilen sind die Gewichte jeder jeweiligen gerichteten Kanten. Die Knoten sind mit den Buchstaben von **A** bis **F** bezeichnet. Gehen Sie davon aus, dass die Kanten in alphabetischer Reihenfolge in den Adjanzenlisten sind, z. B. **A**→**B** vor **A**→**C** und **B**→**E** vor **D**→**C** (gilt für alle Handsimulationen).



1.1 Führen Sie eine Handsimulation des Dijkstra Algorithmus zur Bestimmung kürzester Wege von Knoten **A** zu allen anderen Knoten aus. Sie können die Distanzen in die grauen Kästchen eintragen (Variable `distTo` in unserer Graphen Implementation). Es sind mehrere Kästchen an den Knoten, damit die Bestimmung der Entfernung während des Durchlaufes aktualisiert werden kann (Relaxierung). Sie werden nicht alle Kästchen benötigen.

1.2 Tragen Sie die Knoten in die unten stehende Tabelle in der Reihenfolge ein, in der sie der Priorityqueue entnommen werden. Tragen Sie auch die Distanzen der kürzesten Wege von **A** zu jedem der anderen Knoten in die Tabelle ein.

A	B	C	D	E	F
0					

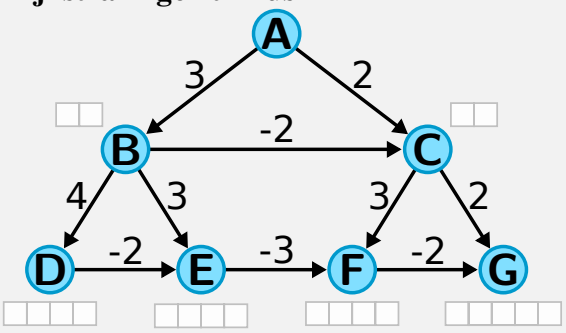
1.3 Ändern Sie möglichst wenige Kantengewichte des gegebenen Graphen ab, so dass im veränderten Graphen der kürzeste Weg von **A** zu Knoten **F** 10 Einheiten lang ist. Notieren Sie Kanten und die zugehörigen Gewichte Ihrer Änderungen hier:

Aufgabe 2: Algorithmus von Bellman-Ford (Klausurvorbereitung)

Zur Bestimmung von kürzesten Wegen in Graphen mit negativen Kantengewichten wird der Algorithmus von Bellman-Ford empfohlen. Der Algorithmus von Dijkstra findet in diesem Fall zwar auch die kürzesten Wege (zumindest in der Variante, die in der Vorlesung vorgestellt wurde), allerdings ist er in diesem Fall nicht mehr '*greedy*': Kanten müssen ggf. mehrfach relaxiert werden und die Effizienz des Algorithmus ist nicht mehr gewährleistet.

Abbildung 2: Vergleich Dijkstra und Bellman-Ford

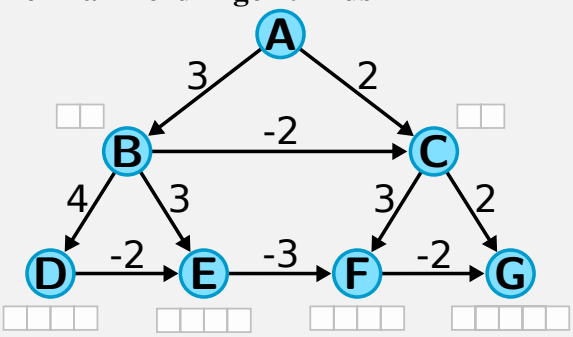
Dijkstra Algorithmus



Anzahl der Relaxierungen:

A→B	A→C	B→C	B→D	B→E
C→F	**C→G**	**D→E**	**E→F**	**F→G**

Bellman-Ford Algorithmus



Anzahl der Relaxierungen:

A→B	A→C	B→C	B→D	B→E
C→F	**C→G**	**D→E**	**E→F**	**F→G**

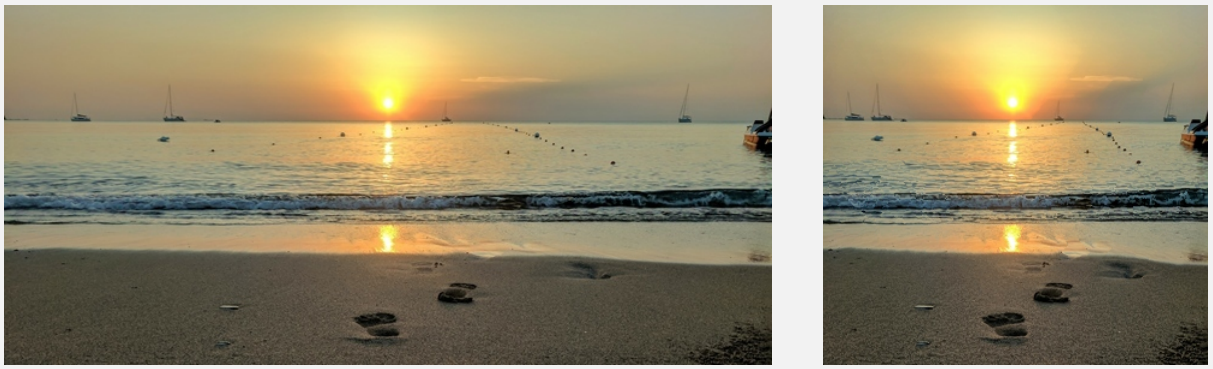
2.1 In Abbildung 2 ist derselbe Graph zweimal abgebildet. Führen Sie auf der linken Seite eine Handsimulation des Dijkstra Algorithmus und auf der rechten Seite eine Handsimulation des Bellman-Ford Algorithmus durch. Notieren Sie in beiden Fällen für jede Kante, wie oft sie relaxiert wurde.

Aufgabe 3: Inhaltsabhängige Bildskalierung (Hausaufgabe)

Bei der inhaltsabhängigen Bildskalierung (*content-aware image resizing* oder *seam carving*) wird eine der beiden Bilddimensionen so verkleinert, dass die wesentlichen Bildelemente möglichst nicht verändert werden, siehe Abbildung 1. Für die horizontale Stauchung werden iterativ Pfade von der

oberen zur unteren Bildkante gesucht und aus dem Bild gelöscht, die das Überschreiten von Kontrastgrenzen möglichst vermeiden.

Abbildung 3: Beispiel für inhaltsabhängige Bildskalierung

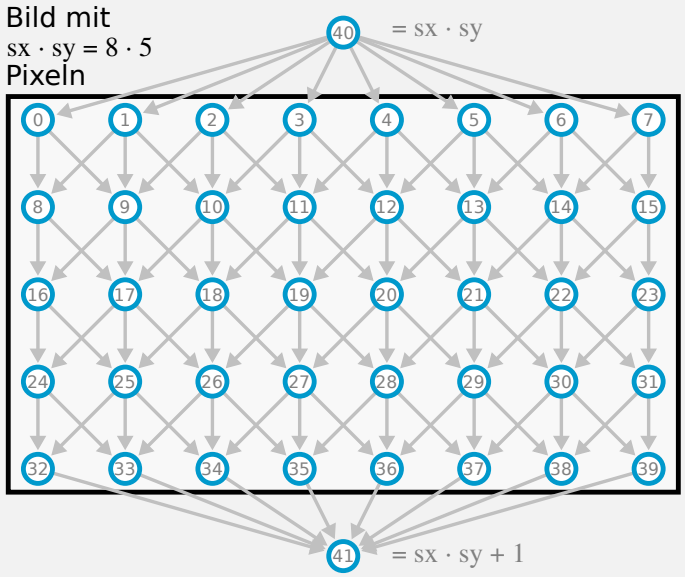


Das Foto von der linken Seite wurde mit inhaltsabhängiger Bildskalierung auf die halbe Breite verkleinert. Bei dem Resultat auf der rechten Seite sind die meisten Bildelemente unverzerrt, aber näher aneinander gerückt.

Die inhaltsabhängige Bildskalierung wird hier mit Hilfe von kürzesten Wegen in einem azyklischen gerichteten Graphen (DAG) realisiert. Dazu wird das Bild durch einen Gridgraphen repräsentiert, bei dem die Knoten Bildpunkte darstellen, siehe Abbildung 4. Für eine horizontale Skalierung werden iterative Pfade aus dem Bild herausgelöscht, die von der oberen zur unteren Bildkante verlaufen und auf ihrem Weg Kontrastüberschreitungen (von einem Pixel zum nächsten) minimieren. Zu diesem Zweck werden die Kantengewichte als Kontrast zwischen den jeweiligen Pixeln definiert. Von jedem Knoten verlaufen Kanten zu den drei Nachbarknoten in der darunterliegenden Zeile.

Da die gesuchten Pfade von einem beliebigen Knoten der obersten Zeile starten können und das Ziel ein beliebiger Knoten der untersten Zeile ist, muss man sich eines kleinen Tricks bedienen. Man fügt einen Startknoten zu dem Gridgraphen hinzu und lässt von ihm aus Kanten zu allen Knoten der obersten Zeile verlaufen. Ebenso fügt man einen Endknoten ein zu dem Kanten von allen Knoten der unteresten Zeile verlaufen. Alle diese zusätzlichen Kanten erhalten das Gewicht 0.

Abbildung 4: Graph zur Bildrepräsentation



Jedes Pixel eines Bildes der Größe $sx \cdot sy$ wird als ein Knoten in einem gerichteten Graphen dargestellt. Die Knoten werden in einem Gitter der Größe $sx \cdot sy$ angeordnet (Gridgraph). Von jedem Knoten gehen drei Kanten zu den unteren Nachbarn (bei Randlage weniger). Die Knoten haben die Indizes von 0 bis $sx \cdot sy - 1$ wie im Bild links gezeigt. Zusätzlich gibt es einen Startknoten mit Index $sx \cdot sy$ und einen Endknoten mit Index $sx \cdot sy + 1$, siehe Erläuterung im Haupttext.

Der Code, der implementiert werden soll, kann auf JPG Dateien angewendet werden, und kann so das in Abbildung 3 gezeigte Ergebnis erzielen. Die Klasse `PictureImage` zum Laden, Verändern und Anzeigen eines Bildes ist vorgegeben. Zum Testen und Debuggen der eigenen Implementation gibt es eine zu vervollständigende Klasse `MatrixImage`, mit deren Hilfe die Bildskalierung auf einfache Matrizen angewendet werden kann. Damit das Verfahren wahlweise für richtige Bilder und für einfache Matrizen ausgeführt werden kann, gibt es das Interface `Image`, das den Zugriff standardisiert.

3.1 Vorhandene Klassen und Schnittstellen untersuchen (0 Punkte)

Schauen Sie sich die vorhandenen Klassen und Schnittstellen genau an, damit Sie die Funktionsweise verstehen. Die Klasse `Bag` (Multimenge) wurde in Vorlesung 2 eingeführt und in Graph Implementationen genutzt. Die Klassen `DirectedEdge` und `WeightedDigraph` wurden in Vorlesung 8 eingeführt. Die Klasse `TopologicalWD` stellt die Funktionalität der Tiefensuche mit Zykluserkennung und Rückgabe der topologischen Sortierung zur Verfügung (Vorlesungen 7 und 9).

Das Interface `Image` definiert die Schnittstelle für Klassen von Bildern, die mit der zu implementierenden inhaltsabhängigen Bildskalierung bearbeitet werden können. Sie benutzt die Klasse `Coordinate` für die Koordinaten von Pixeln.

public interface Image

`int`	`sizeX()`	Größe des Bildes in x-Richtung
`int`	`sizeY()`	Größe des Bildes in y-Richtung
`double`	`contrast(Coordinate p1, Coordinate p2)`	quantifiziert den Kontrast zwischen den Bildpunkten an den Koordinaten `p1` und `p2` (die üblicherweise benachbart sind)
`void`	`removeVPath(int[] path)`	Entfernt einen Pfad aus dem Bild, der von der oberen zur unteren Bildkante verläuft und in jeder Zeile genau einen Punkt beinhaltet. In dem Array `path` ist zu jedem Index `y` die entsprechende x-Koordinate des zu entfernenden Pfades gespeichert. Der Pfad verläuft in dem Bild also entlang der Koordinaten (path[y], y) für y von 0 bis `sizeY()-1`.
`void`	`render()`	visuelle Darstellung des Bildes

Es gibt zwei Klassen, die das Interface `Image` implementieren: Mit der fertig implementier-

ten Klasse `PictureImage` können Bilder im JPG Format bearbeitet werden. In der Klasse `MatrixImage` sind zwei Methoden zu implementieren. Sie ist zum schnellen Testen und Debuggen der inhaltsabhängigen Bildskalierung gedacht. Mit ihr können `int` Matrizen bearbeitet werden.

3.2 Implementieren der Klasse `MatrixImage` (20 Punkte)

Implementieren Sie die Methoden

```
public double contrast(Coordinate p0, Coordinate p1)
public void removeVPath(int[] path)
```

der Klasse `MatrixImage` gemäß der API von `Image`. Als Kontrast in `contrast()` soll der Absolutbetrag der Differenz der Matrixwerte an den gegebenen Koordinaten zurückgegeben werden. `p0` und `p1` können beliebige Koordinaten des Bildes sein. Werfen Sie eine *Exception*, wenn eine der Koordinaten außerhalb des Bildes ist.

Hinweise:

- Bei `removeVPath()` sollten Sie nicht versuchen, die existierende Matrix zu ändern. Definieren Sie eine neue Matrix in der passenden Größe und kopieren Sie den Inhalt passen rüber. Am Ende können Sie die Referenz der Objektvariable `field` auf die neue Matrix setzen.
- Seien Sie nicht irritiert. Die Implementation von `contrast()` ist hier sehr einfach.

3.3 Implementieren der Klasse `ShortestPathsTopological` (50 Punkte)

Implementieren Sie Konstruktor und Methode

```
public ShortestPathsTopological(WeightedDigraph G, int s)
public void relax(DirectedEdge e)
```

der Klasse `ShortestPathsTopological`. In dem Konstruktor sollen kürzeste Wege im Graphen `G` ausgehend von Startknoten `s` zu allen anderen Knoten gesucht und in dem Array `parent[]` gespeichert werden. Mit der vorhandenen (und aus der Vorlesung bekannten) Methode `pathTo()` kann daraus der kürzeste Weg zu einem gegebenen Zielknoten abgerufen werden. Die Bestimmung kürzester Wege soll für alle zyklenfreien gewichteten Digraphen funktionieren (und nicht nur für die speziellen Gridgraphen, die in dieser Aufgabe vorkommen).

Die kürzesten Wege sollen mit der Methode der topologischen Sortierung bestimmt werden, damit eine Laufzeit in $\boldsymbol{O}(V + E)$ erreicht wird.

Bemerkung:

- Die Methode `order()` der Klasse `TopologicalWD` gibt die umgekehrte Hauptordnung der Knoten als `Stack<Integer>` zurück. Damit es beim Umgang mit dieser Reihenfolge zu keinen bösen Überraschungen kommt, muss die folgende Besonderheit beachtet werden.
- Die Klasse `Stack` aus `java.util` implementiert zwar das Interface `Iterable`, aber die Reihenfolge, in der Iterator durch die Elemente des Stacks läuft, ist nicht LIFO sondern FIFO.
- Um die Knoten also tatsächlich in der umgekehrten Hauptordnung zu durchlaufen, muss man bei der Rückgabe von `order()` die Stack Methode `pop()` benutzen.

3.4 Implementieren der Klasse `ContentAwareImageRezising` (30 Punkte)

Implementieren Sie die Methode

```
public int[] leastContrastImageVPath()
```

der Klasse `ContentAwareImageRezising`, die einen 'vertikal verlaufenden' Pfad mit den schwächsten Kontrastüberschreitungen in dem Bild `image` (Objektvariable) zurückgibt. Der Pfad muss in der Form zurückgegeben werden, dass er als Eingabe der Methode `removeVPath` der Schnittstelle `Image` geeignet ist. Um diese Aufgabe zu lösen, müssen die vorhandenen Methoden der Klasse `ContentAwareImageRezising` genau verstanden worden sein.

Bemerkungen:

- Die Implementation ist bei wiederholter Ausführung nicht effizient, da nach dem Löschen eines Pfades der Graph und die Distanzen der kürzesten Wege von null neu bestimmt werden. Mit mehr Implementationsaufwand (und dynamischen Graphen) könnten die redundanten Berechnungen eingespart und das Vefahren deutlich beschleunigt werden.
- Eine effizientere Lösung kann mit dynamischer Programmierung erreicht werden. Dafür definiert man einen Energiewert für jedes Pixel als den Kontrast des Bildpunktes im Vergleich zu seiner Umgebung. Dieses Maß ersetzt den Kontrast zwischen Bildpunkten, den wir hier benutzt haben.

Was Sie nach diesem Blatt wissen sollten:

- Was sind gewichtete Graphen?
- Was ist die Relaxierung (engl. *Relaxation*) und welche Rolle spielt sie in den SSSP Algorithmen?
- Wie kann man mit Topologischer Sortierung kürzeste Wege in einem Graphen finden? Welche Voraussetzungen müssen daür erfüllt sein?
- Wie kann man mit Dijkstra kürzeste Wege in einem Graphen finden? Welche Voraussetzungen müssen daür erfüllt sein?
- Wie kann man mit Bellman-Ford kürzeste Wege in einem Graphen finden? Welche Voraussetzungen müssen dafür erfüllt sein?