TU Wien
Institut für Logic and Computation
Forschungsbereich Algorithms and Complexity

186.866 Algorithmen und Datenstrukturen VU

Programmieraufgabe P2

PDF erstellt am: 18. März 2023

1 Vorbereitung

Um diese Programmieraufgabe erfolgreich durchführen zu können, müssen folgende Schritte umgesetzt werden:

1. Laden Sie das Framework `P2.zip` aus TUWEL herunter.
2. Entpacken Sie `P2.zip` und öffnen Sie das entstehende Verzeichnis als Projekt in IntelliJ (nicht importieren, sondern öffnen).
3. Öffnen Sie die nachfolgend angeführte Datei im Projekt in IntelliJ. In dieser Datei sind sämtliche Programmiertätigkeiten durchzuführen. Ändern Sie keine anderen Dateien im Framework und fügen Sie auch keine neuen hinzu.
   `src/main/java/exercise/StudentSolutionImplementation.java`
4. Füllen Sie Vorname, Nachname und Matrikelnummer in der Methode `StudentInformation provideStudentInformation()` aus.

2 Hinweise

Einige Hinweise, die Sie während der Umsetzung dieser Aufgabe beachten müssen:

- Lösen Sie die Aufgaben selbst und nutzen Sie keine Bibliotheken, die diese Aufgaben abnehmen.
- Sie dürfen beliebig viele Hilfsmethoden schreiben und benutzen. Beachten Sie aber, dass Sie nur die oben geöffnete Datei abgeben und diese Datei mit dem zur Verfügung gestellten Framework lauffähig sein muss.

3 Übersicht

In dieser Programmieraufgabe wird das Thema Minimum Spanning Trees bearbeitet. Dazu werden die Algorithmen von Prim und Kruskal implementiert sowie dazu nützliche Datenstrukturen. Für Prim wird eine Priority Queue genutzt, die mittels Min Heap implementiert ist und für Kruskal die Union-Find-Datenstruktur.

4 Theorie

Die notwendige Theorie kann in den Vorlesungsfolien „Graphen“ und „Greedy-Algorithmen“ gefunden werden.

5 Implementierung

Zwischen mehreren Standorten sollen Kabel verlegt werden, um diese miteinander zu vernetzen. Allerdings befinden sich nicht an sämtlichen Standorten entsprechende Netzwerkkomponenten, wodurch nur bestimmte Standorte angebunden werden können. Zusätzlich gibt es ein knappes finanzielles Budget, weshalb die gesamte zu verlegende Kabellänge auf das Minimum reduziert werden soll.

Mithilfe eines Minimum Spanning Trees soll das entsprechende Netzwerk entworfen werden. Implementieren Sie dazu zunächst den Algorithmus von Prim und im Anschluss den Algorithmus von Kruskal. Es wurde bereits im Vorhinein abgeklärt, dass es möglich ist einen Baum zu finden, der aus exakt den kompatiblen Standorten besteht. Spezielle Fehlerbehandlung ist daher nicht notwendig.

5.1 Priority Queue - Min Heap

Um die Laufzeit des Algorithmus von Prim gering zu halten, wird auf eine selbst implementierte Priority Queue zurückgegriffen, die mithilfe eines Min Heaps implementiert werden soll.

Vervollständigen Sie die Priority Queue indem Sie die fehlenden Methoden `void heapifyUp(PriorityQueue priorityQueue, int index)` und

`void heapifyDown(PriorityQueue priorityQueue, int index)` implementieren. `PriorityQueue` nutzt ein Array um den Min Heap zu verwalten. Dabei ist zu beachten, dass die erste Position im Heap-Array (wie in den Vorlesungsfolien) den Index 1 hat.

Mittels des ersten Parameters `PriorityQueue priorityQueue` kann auf das Heap-Array zugegriffen werden. Dazu stellt sie drei Methoden zur Verfügung:

- `double getWeight(int index)`: Mittels `double weight = priorityQueue.getWeight(1)` kann z.B. die Gewichtung des Elements auf Index 1 im Heap-Array ausgelesen werden. Wenn ein ungültiger Index übergeben wird, wird -1 zurückgegeben.
- `void swap(int index1, int index2)`: Diese Methode ermöglicht das Vertauschen zweier Elemente im Heap-Array. Um beispielsweise die Elemente an Index 1 und 2 zu vertauschen, muss `priorityQueue.swap(1, 2)` aufgerufen werden. Ist einer der beiden Indizes ungültig, werden keine Elemente vertauscht.
- `int length()`: Durch Aufruf dieser Methode, kann die Länge des Heap-Arrays abgefragt werden. Z.B. kann der Index des letzten Elements im Heap-Array mit `int n = priorityQueue.length()- 1` ermittelt werden.

Der zweite Parameter `int index` gibt den Index des Heap-Arrays an, für welchen die Heapify-Up- bzw. Heapify-Down-Operation durchgeführt werden soll.

Sie selbst müssen die Heapify-Up- und Heapify-Down-Operationen im weiteren Verlauf nicht aufrufen.

5.2 Algorithmus von Prim

Implementieren Sie nun den Algorithmus von Prim in der Methode `double prim(Graph g, PriorityQueue q, int[] predecessors)`.

Der erste Parameter `Graph g` ist der Graph, auf dem der Algorithmus von Prim angewandt werden soll. Jeder Knoten repräsentiert hierbei einen Standort, wobei Verbindungen zwischen Standorten durch die Kanten des Graphs dargestellt werden. Die Gewichte der Kanten stellen die Entfernung

zwischen zwei Standorten dar und entsprechen der Länge, die ein Kabel haben müsste, wenn zwei Standorte entlang dieser Kante verbunden werden würden. Nicht alle Knoten sind relevant, da nur einige Standorte mit entsprechender technischer Ausstattung an das Netzwerk angeschlossen werden können. Bedenken Sie also auch, dass ein Kabel nur zwischen zwei relevanten Standorten verlegt werden kann und somit auch keine indirekte Verbindung über einen nicht relevanten Standort möglich ist.

In Abbildung 1 ist ein Beispielgraph zu sehen. Knoten mit Rahmen sind relevante Standorte. Rote Kanten markieren das berechnete minimale Netzwerk. Beachten Sie, dass Knoten 3 keinen relevanten Standort darstellt und dieser deshalb einerseits nicht angebunden wird und andererseits über diesen auch keine indirekte Verbindung von Knoten 1 und 6 erfolgen kann (auch wenn diese aufgrund der kürzeren Verbindungen von Vorteil wäre).

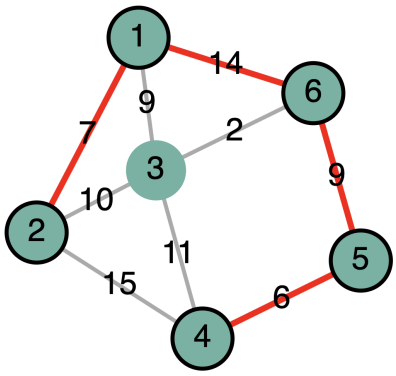

Abbildung 1: Beispielgraph

`Graph g` stellt einige Methoden zur Verfügung, die zur Implementierung des Algorithmus von Prim benötigt werden:

- `int numberOfVertices()`: Mithilfe von `int number = g.numberOfVertices()` können Sie die Anzahl der Knoten im Graph auslesen. Knoten werden mittels aufsteigender IDs gekennzeichnet. Angenommen es gibt insgesamt n Knoten, dann gibt es jeweils Knoten mit Knoten-ID 1, 2 ... $n-1$ und n.
- `boolean isRelevant(int vertexId)`: Diese Methode gibt Ihnen Auskunft darüber, ob ein Knoten mit einer bestimmten Knoten-ID (`int vertexId`) relevant ist oder nicht. Wird eine ungültige Knoten-ID abgefragt, ist der Rückgabewert `false`.

- `double getEdgeWeight(int vertexIdStart, int vertexIdEnd)`: Das Kantengewicht einer Kante kann durch die Angabe von zwei Knoten (mittels Knoten-ID) ausgelesen werden. Gibt es zwischen zwei Knoten keine Kante, erhalten Sie den Wert -1 als Ergebnis.

- `int[] getNeighbors(int vertexId)`: Die Nachbarn eines Knoten können mittels der Knoten-ID (`int vertexId`) ermittelt werden. Die Nachbarn werden als Array von Knoten-IDs repräsentiert. Bei der Übergabe einer ungültigen Knoten-ID, wird `null` zurückgegeben.

Der zweite Parameter `PriorityQueue q` ist die zuvor zu vervollständigende Priority Queue. Folgende Methoden benötigen Sie zur Umsetzung des Algorithmus von Prim:

- `void add(double weight, int vertexId)`: Ein neuer Knoten kann der Priority Queue hinzugefügt werden, indem eine Gewichtung `double weight` und eine Knoten-ID `int vertexId` angegeben wird. Sollte eine Knoten-ID bereits in der Priority Queue vorhanden sein, hat der Aufruf dieser Methode keine Folgen. Sie selbst müssen sich nicht um die Einhaltung der Heap-Bedingung kümmern. Achten Sie allerdings darauf, dass eine mögliche Fehlerquelle Ihrer Implementierung des Algorithmus von Prim auch die zuvor programmierte Heapify-Up-Operation sein kann.

- `boolean isEmpty()`: Gibt Ihnen Auskunft darüber, ob sich keine Knoten mehr in der Priority Queue befinden.

- `int removeFirst()`: Mittels `int idOfFirstVertex = q.removeFirst()` können Sie die Knoten-ID des Knotens mit der niedrigsten Gewichtung aus der Priority Queue auslesen und gleichzeitig aus dieser entfernen. Ist die Priority Queue leer, wird -1 zurückgegeben. Auch hier müssen Sie sich nicht selbst um die Einhaltung der Heap-Bedingung kümmern. Achten Sie allerdings hier ebenfalls darauf, dass eine mögliche Fehlerquelle die zuvor programmierte Heapify-Down-Operation sein kann.

- `void decreaseWeight(double weight, int vertexId)`: Um die Gewichtung eines Knotens innerhalb der Priority Queue zu senken, kann diese Methode aufgerufen werden. Einerseits muss das neue, verringerte Gewicht angegeben werden (`double weight`) und

andererseits die Knoten-ID des gewünschten Knotens (`int vertexId`). Änderungen passieren nur, wenn ein Gewicht angegeben wird, das tatsächlich kleiner ist, als das bisherige, und wenn ein Knoten mit der Knoten-ID in der Priority Queue vorhanden ist. Auch hier gelten dieselben Hinweise wie für `void add(double weight, int vertexId)`.

Der dritte Parameter `int[] predecessors` wird zum Testen und zur späteren Visualisierung benötigt. Dieses ist ein bereits initialisiertes Array, in dem für jeden Knoten ein Wert abgelegt werden kann. Ein Wert für den Knoten mit Knoten-ID n wird an die Stelle mit dem Index n im Array hinterlegt. Hinterlegen Sie hierbei für ...

- den ersten gewählten Knoten die zugehörige Knoten-ID. Hat beispielsweise der erste gewählte Knoten die Knoten-ID 1, hinterlegen Sie in `predecessors[1]` den Wert 1.
- den Knoten mit der Knoten-ID v die Knoten-ID u des Knotens am anderen Ende der Kante $e = (u, v)$, die für den Minimum Spanning Tree gewählt wurde (`predecessors[v] = u`). Dabei ist u die Knoten-ID jenes Knotens, der sich zum Zeitpunkt der Auswahl des Knotens mit Knoten-ID v bereits in der Menge S befunden hat und $e = (u, v)$ jene Kante mit dem geringsten Gewicht die den Knoten mit Knoten-ID v mit S verbindet. Die Menge S ist entsprechend der Vorlesungsfolien zu verstehen.

Im Fall von Abbildung 1 und unter der Annahme, dass der Knoten mit Knoten-ID 1 der erste gewählte war, entspricht das folgendem Array („-“ bedeutet, dass der Wert nicht relevant ist):

Index	0	1	2	3	4	5	6
Wert	-	1	1	-	5	6	1

Als Rückgabewert von `double prim(Graph g, PriorityQueue q, int[] predecessors)` wird das Gesamtgewicht des Minimum Spanning Trees erwartet.

Hinweis

Bei der Implementierung können Sie auch auf eine `ArrayList` zurückgreifen. Im Gegensatz zu einem Array haben Sie hier den Vorteil nicht im Vorhinein die benötigte Länge wissen zu müssen. Eine neue `ArrayList` kann wie folgt erzeugt werden: `ArrayList<Integer> s = new ArrayList<Integer>()`. Mittels `s.add(u)` kann ein `int u` hinzugefügt werden. Nutzen Sie `s.contains(u)`, um abzufragen ob `int u` enthalten ist.

5.3 Union-Find-Datenstruktur

Vervollständigen Sie die Union-Find-Datenstruktur, indem Sie die fehlende Methode `int findset(UnionFindDataStructure unionFindDataStructure, int vertexId)` implementieren.

Der erste Parameter `UnionFindDataStructure unionFindDataStructure` erlaubt den Zugriff auf den Elternknoten eines Knotens.

- `int getParent(int vertexId)`: Mittels `int parentVertexId = unionFindDataStructure.getParent(1)` kann z.B. der Elternknoten des Knotens mit der Knoten-ID 1 abgerufen werden.

Der zweite Parameter `int vertexId` ist die Knoten-ID eines Knotens aus der Menge, für die der Repräsentant gefunden werden soll.

Der Rückgabewert ist der Repräsentant der Menge in Form einer Knoten-ID.

Sie selbst müssen die Find-Set-Operation im weiteren Verlauf nicht aufrufen.

5.4 Algorithmus von Kruskal

Implementieren Sie nun den Algorithmus von Kruskal in der Methode `double kruskal(Graph g, UnionFindDataStructure u, boolean[] chosenEdges)`.

Der erste Parameter `Graph g` ist der Graph, auf dem der Algorithmus von Kruskal angewandt werden soll. Der Graph ist genauso wie bisher zu

interpretieren. Zusätzlich zu den bereits beschriebenen Methoden stellt `Graph g` noch eine weitere zur Verfügung:

- `int[][] getEdgesOrderedByWeight()`: Mithilfe dieser Methode können die Kanten aufsteigend sortiert nach dem Kantengewicht in Form eines zweidimensionalen Arrays ausgelesen werden. Die Kante mit dem geringsten Gewicht befindet sich an Stelle 0. Jede Kante wird durch ein zweistelliges Array repräsentiert, wobei an jeder Stelle jeweils die Knoten-ID einer der beiden inzidenten Knoten hinterlegt ist. Angenommen die sortierten Kanten wurden in `int[][] orderedEdges` hinterlegt, so sind die Knoten-IDs der Kante mit dem geringsten Gewicht `orderedEdges[0][0]` und `orderedEdges[0][1]`.

Der zweite Parameter `UnionFindDataStructure u` ist die zuvor zu vervollständigende Union-Find-Datenstruktur. Sie stellt die aus der Vorlesung bekannten Methoden zur Verfügung:

- `void makeset(int vertexId)`: Erzeugt eine neue Menge mit dem Knoten mit der angegebenen Knoten-ID.
- `void union(int representativeId1, int representativeId2)`: Vereinigt die beiden durch die Repräsentanten angegebenen Mengen.
- `int findset(int vertexId)`: Liefert den entsprechenden Repräsentanten in Form einer Knoten-ID. Achten Sie darauf, dass eine mögliche Fehlerquelle für Ihre Implementierung des Algorithmus von Kruskal die zuvor programmierte Find-Set-Operation sein kann.

Der dritte Parameter `boolean[] chosenEdges` ist ein Array mit einer Stelle für jede Kante im Graphen. Wenn Ihr Algorithmus eine Kante aus `int[][] getEdgesOrderedByWeight()` mit dem Index n für den Minimum Spanning Tree auswählt, setzen Sie `chosenEdges[n]` auf `true`.

Als Rückgabewert von `double kruskal(Graph g, UnionFindDataStructure u, boolean[] chosenEdges)` wird das Gesamtgewicht des Minimum Spanning Trees erwartet.

6 Testen

Führen Sie zunächst die `main`-Methode in der Datei `src/main/java/framework/Exercise.java` aus.

Anschließend wird Ihnen in der Konsole eine Auswahl an Testinstanzen angeboten, darunter befindet sich zumindest `abgabe.csv`:

```
Select an instance set or exit:
[1] abgabe.csv
[0] Exit
```

Durch die Eingabe der entsprechenden Ziffer kann entweder eine Testinstanz ausgewählt werden oder das Programm (mittels der Eingabe von `0`) verlassen werden. Wird eine Testinstanz gewählt, dann wird der von Ihnen implementierte Programmcode ausgeführt. Kommt es dabei zu einem Fehler, wird ein Hinweis in der Konsole ausgegeben.

Relevant für die Abgabe ist das Ausführen der Testinstanz `abgabe.csv`.

Die weiteren Testinstanzen `small-prim.csv`, `medium-prim.csv`, `small-kruskal.csv` und `medium-kruskal.csv` sind nur zum jeweiligen Testen der einzelnen Unteraufgaben gedacht.

7 Evaluierung

Wenn der von Ihnen implementierte Programmcode mit der Testinstanz `abgabe.csv` ohne Fehler ausgeführt werden kann, dann wird nach dem Beenden des Programms im Ordner `results` eine Ergebnis-Datei mit dem Namen `solution-abgabe.csv` erzeugt.

Die Datei `solution-abgabe.csv` beinhaltet Zeitmessungen und Minimum Spanning Trees der Ausführung der Testinstanz `abgabe.csv`, welche in einem Web-Browser visualisiert werden können. (Auch Ergebnis-Dateien anderer Testinstanzen können zu Testzwecken visualisiert werden.) Öffnen Sie dazu die Datei `visualization.html` in Ihrem Web-Browser und klicken Sie rechts oben auf den Knopf *Ergebnis-Datei auswählen*, um `solution-abgabe.csv` auszuwählen.

Beantworten Sie basierend auf der Visualisierung die Fragestellungen aus dem folgenden Abschnitt.

8 Fragestellungen

Öffnen Sie `solution-abgabe.csv` und bearbeiten Sie folgende Aufgaben- und Fragestellungen:

1. Vergleichen Sie die Graphen zu *Small - Sparse - Prim 55* und *Small - Sparse - Kruskal 173* mithilfe entsprechender Auswahl im Dropdown links unter den Plots. Sind in beiden Fällen die gleichen Kanten rot markiert? Überprüfen Sie auch in beiden Fällen die Richtigkeit Ihres Ergebnisses. Sollte es Schwierigkeiten beim Lesen der Kantengewichte geben, können Sie durch Klicken und Ziehen mit dem Mauszeiger Knoten verschieben.

   Erstellen Sie im Anschluss einen Screenshot der beiden Graphen.

2. Vergleichen Sie die Graphen zu *Small - Sparse - Prim 55* und *Small - Dense - Prim 118* mithilfe entsprechender Auswahl im Dropdown. Worin unterscheiden sich dichte und dünne Graphen? Überprüfen Sie auch beim dichten Graphen die Richtigkeit Ihres Ergebnisses. Auch hier können Knoten verschoben werden.

   Erstellen Sie im Anschluss einen Screenshot der beiden Graphen.

3. Durch Klicken auf Gruppennamen in der Legende neben der Plots, lassen sich einzelne Gruppen aus- bzw. einblenden. Blenden Sie alles bis auf *Large - Sparse - Prim* und *Large - Dense - Prim* aus. Vergleichen Sie die Laufzeitverhalten des dichten und des dünnen Graphen mithilfe der Plots miteinander und argumentieren Sie deren Zustandekommen.

   Drücken Sie im Anschluss in der Menüleiste rechts über dem Plot auf den Fotoapparat, um den Plot als Bild zu speichern.

4. Blenden Sie nun alles bis auf *Large - Sparse - Kruskal* und *Large - Dense - Kruskal* aus. Vergleichen Sie die Laufzeitverhalten des dichten und des dünnen Graphen mithilfe der Plots miteinander und argumentieren Sie deren Zustandekommen.

   Erstellen Sie im Anschluss wieder mithilfe der Menüleiste ein Bild des Plots.

5. Blenden Sie nun alles bis auf *Large - Dense - Prim* und *Large - Dense - Kruskal* aus. Sind größere Unterschiede zu sehen? Wenn ja, wie erklären Sie sich diese?

   Erstellen Sie nun ein Bild der beiden Plots.

Falls sich im Zuge der Evaluierung die Darstellung der Plots auf ungewünschte Weise verändert (z.B. durch die Auswahl eines zu kleinen Ausschnitts), können Sie mittels Doppelklick auf den Plot oder Klick auf das Haus in der Menüleiste die Darstellung zurücksetzen.

Fügen Sie Ihre Antworten in einem Bericht gemeinsam mit den drei erstellten Bildern der Visualisierungen sowie den zwei Screenshots der Graphen der Testinstanz `abgabe.csv` zusammen.

9 Abgabe

Laden Sie die Datei `src/main/java/exercise/StudentSolutionImplementation.java` in der TUWEL-Aktivität *Hochladen Source-Code P2* hoch. Fassen Sie diesen Bericht mit den anderen für das zugehörige Abgabegespräch relevanten Berichten in einem PDF zusammen und geben Sie dieses in der TUWEL-Aktivität *Hochladen Bericht Abgabegespräch 1* ab.

10 Nachwort

Durch die Eigenschaft, dass ein minimaler Spannbäume alle Knoten eines zusammenhängenden Graphen auf kostenminimale Weise verbindet, stellt er eine sehr grundlegende Struktur eines Graphen dar. Anwendungen kann man sich natürlich gut im Bereich der Kommunikationstechnik vorstellen, wenn es darum geht Endpunkte auf möglichst kostengünstige Weise zu vernetzen. Natürlich aber spielen in realen Anwendungen im Bereich der Telekommunikation meist noch zahlreiche zusätzliche Aspekte eine Rolle, sodass realistischere Problemformulierungen von Netzwerkdesign-Aufgaben oft deutlich diffiziler sind. Beispielsweise müssen Kapazitätsbeschränkungen von Verbindungen oft berücksichtigt werden, der Grad von Knoten im aufzubauenden Netz kann beschränkt sein, oder es soll die Anzahl der Zwischenknoten auf jedem möglichen Pfad limitiert werden um zu große Latenzen von Übertragungen zu vermeiden. In Bezug auf Ausfallsicherheit geht man häufig auch von einfachen Baumstrukturen ab und fordert, dass zumindest zwischen zentraleren Vermittlungsknoten mehrere kantenunabhängige Pfade existieren. Lösungsalgorithmen zu realen Netzwerkdesign-Aufgaben sind deshalb im Allgemeinen deutlich komplexer, greedy Konstruktionsverfahren wie die Algorithmen von Kruskal bzw. Prim

liefern dann meist keine optimalen Lösungen mehr. Dennoch stellen diese einfachen Algorithmen häufig eine gewisse Ausgangsbasis dar bzw. liefern mit den Kosten ihrer Lösungen manchmal wertvolle untere Schranken.

Die im Algorithmus von Prim verwendete Datenstruktur des Heaps bzw. allgemeiner Prioritätswarteschlangen werden uns im Laufe der Lehrveranstaltung noch mehrfach begegnen. Ihr Einsatz in diversen Algorithmen macht sehr häufig den Unterschied zwischen einer naiven Implementierung und einer deutlich effizienteren aus.