Prof. Dr. Ralf Hinze
Sebastian Schweizer, M.Sc.
Markus Heinrich, M.Sc.

TU Kaiserslautern
Fachbereich Informatik
AG Programmiersprachen

Projekt 1: Programmierpraktikum 2020

Ausgabe: 02. Juni 2020
Abgabe: 23. Juni 2020, 15 Uhr

GitLab Team Repositories

Wir verwenden auch für die Projekte wieder die GitLab Repositories. Führen Sie den Befehl `git pull` aus, damit die Vorlagen heruntergeladen werden. Sie sollten nun einen neuen Ordner `P1` sehen, den Sie wie gewohnt in IntelliJ IDEA öffnen können. Es handelt sich wieder um ein Gradle Projekt, sodass Abhängigkeiten automatisch heruntergeladen werden.

Ihre Lösungen müssen Sie in das lokale Repository committen (`git commit`) und dann auf den GitLab Server übertragen (`git push`). Sprechen Sie sich zur Bearbeitung der Aufgaben mit Ihren Abgabepartner*innen ab, damit Sie unnötige Konflikte vermeiden.

Vorlesungsumfrage

Wir sind an Ihrem Feedback interessiert. Bitte nehmen Sie sich die Zeit, die anonyme Vorlesungsumfrage auszufüllen. Studierende mit `@cs.uni-kl.de` Mailadresse haben dazu bereits eine E-Mail erhalten, alle anderen können sich auf https://vlu.cs.uni-kl.de/teilnahme/ z. B. mit ihrer RHRK Mailadresse einen Link anfordern. Ihr Feedback hilft uns, die Vorlesung und die Übungen weiter zu verbessern.

Bearbeitungshinweise

Sie haben für die Bearbeitung des Projekts drei Wochen Zeit, d. h. es ist entsprechend umfangreich. Beginnen Sie also rechtzeitig mit der Bearbeitung und nutzen Sie die Sprechstunden!

Wir erwarten, dass folgende Meilensteine erreicht werden (d. h. Code befindet sich auf GitLab):

1. Bis Dienstag, 09.06.2020 23:59 Uhr, müssen mindestens die Testfälle von Aufgabe 1 a) in sinnvollem Umfang erstellt sein.
2. Bis Dienstag, 16.06.2020 23:59 Uhr, soll Aufgabe 1 weitestgehend gelöst sein.

Einleitung

Ziel dieses Projekts ist es ein Mengenüberdeckungsproblem[1] mit Hilfe eines Greedy-Algorithmus approximativ zu lösen (das Mengenüberdeckungsproblem ist im Allgemeinen NP-schwer).

Zunächst werden wir uns jedoch der Implementierung von Prioritätswarteschlangen zuwenden, mit deren Hilfe sich der Greedy-Algorithmus einfach formulieren lassen wird.

Aufgabe 1 Prioritätswarteschlangen

Prioritätswarteschlangen haben Sie bereits in der Vorlesung *Algorithmen und Datenstrukturen* kennengelernt. Eine Prioritätswarteschlange (Interface `PriorityQueue<E>`) ist ein abstrakter Datentyp, welcher Elemente eines Typs `E` speichert und die folgenden Operationen anbietet:

```
public interface PriorityQueue<E> {
    void insert(E elem);
    void merge(PriorityQueue<E> otherQueue);
    E deleteMax();
    E max();
    boolean isEmpty();
    boolean update(E elem, E updatedElem);
    void map(UnaryOperator<E> f);
}
```

Mit der Methode `insert(E elem)` wird ein Element `elem` in die Prioritätswarteschlange eingefügt. Die Methode `queue.merge(PriorityQueue<E> otherQueue)` fügt alle Elemente von `otherQueue` in die Prioritätswarteschlange `queue` ein. Sie können sich Arbeit sparen, indem Sie `merge` durch `insert` oder umgekehrt `insert` durch `merge` ausdrücken.

`deleteMax` gibt das maximale Element der Prioritätswarteschlange zurück und entfernt es dabei. Entsprechend gibt `max` nur das maximale Element zurück ohne es zu entfernen.

Die Methode `isEmpty` prüft, ob die Prioritätswarteschlange leer ist.

Mit der Methode `update(E elem, E updatedElem)` wird das Element `elem` in der Prioritätswarteschlange durch `updatedElem` aktualisiert. Hier weichen wir von der in der Vorlesung *Algorithmen und Datenstrukturen* diskutierten Operation `increaseKey` etwas ab, da wir nicht nur die Erhöhung der Priorität von Einträgen, sondern auch die Verringerung unterstützen möchten. Eine mögliche Umsetzung könnte so aussehen, dass `elem` zunächst entfernt wird und anschließend `updatedElem` eingefügt wird. Ist das zu aktualisierende Element nicht enthalten, soll **`false`** zurückgegeben werden, bei erfolgreicher Aktualisierung entsprechend **`true`**.

Die Methode `map`, welche eine einstellige Funktion `f` auf jedes Element in der `PriorityQueue` anwendet und das Ergebnis zurückgibt, gehört nicht zur kanonischen Schnittstelle einer Prioritätswarteschlange. Wir haben diese jedoch mit aufgenommen, da sich dadurch die zweite Aufgabe einfacher und effizienter lösen lässt. `UnaryOperator`[2] ist Bestandteil von Javas *Functional Interface*[3] und kann damit als Typ für Funktionsausdrücke verwendet werden (wiederholen Sie ggf. die Vorlesung zu *Java Core* ab Folie 48). Die Schnittstelle `UnaryOperator<E>` enthält eine Methode `E apply(E elem)`. Sie können also eine Instanz `UnaryOperator<E> f` mit `f.apply(elem)` auf ein Element `elem` des Typs `E` anwenden und erhalten als Rückgabe ein Element vom Typ `E`.

Dem Konstruktor der `PriorityQueue<E>` Implementierungen soll ein `Comparator<E>` übergeben werden, mit dessen Hilfe Sie die Anordnung der Elemente in der Queue, also deren Priorität, ermitteln können.

Legen Sie Ihre Lösungen der folgenden Aufgaben im Paket `de.tukl.programmierpraktikum2020.p1.a1` ab.

[1] https://en.wikipedia.org/wiki/Set_cover_problem
[2] https://docs.oracle.com/javase/8/docs/api/java/util/function/UnaryOperator.html
[3] https://docs.oracle.com/javase/8/docs/api/java/util/function/package-summary.html

a) In der Vorlesung haben Sie den Ansatz der testgetriebenen Entwicklung (engl. *test-driven development*) kennengelernt. Folgen Sie diesem Ansatz, indem Sie zunächst Testfälle für die Prioritätswarteschlange in der Klasse `PriorityQueueTest` erstellen, bevor Sie mit konkreten Implementierungen in den folgenden Aufgabenteilen beginnen. Sie können dazu als Vorlage die Methode `priorityQueueBeispiel` verwenden.

   Da unsere Prioritätswarteschlangen alle dieselbe Schnittstelle implementieren, müssen wir nicht für jede Prioritätswarteschlange neue Tests schreiben. Stattdessen instanziiert die Methode `getPriorityQueueInstances` die verschiedenen Prioritätswarteschlangen und übergibt diese an den parametrisierten Test (vgl. Annotationen `@ParameterizedTest` und `@MethodSource`).

   Es ist für diese Aufgabe in Ordnung, wenn in der Testklasse einige Beispiele getestet werden.

   Wenn Sie möchten, dürfen Sie aber gerne auch Zufallstests erstellen und für die Tests den vollen Umfang der Standardbibliothek und von JUnit nutzen.

b) Schreiben Sie die Klasse `ListQueue`, welche die Schnittstelle `PriorityQueue` implementiert. In Ihrer Klasse soll die Queue durch eine **sortierte** Liste repräsentiert werden. Verwenden Sie dazu die Klasse `java.util.LinkedList` aus der Standardbibliothek.

c) Implementieren Sie die Klasse `SkewHeap`, welche die Schnittstelle `PriorityQueue` als Skew Heap (selbstbalancierender Heap) implementiert.

   Für den `SkewHeap` ist die `merge` Operation von zentraler Bedeutung. Recherchieren Sie in der Literatur oder im Internet [4] deren Funktionsweise und geben Sie Ihre Quelle im Code an.

d) Implementieren Sie die Schnittstelle `PriorityQueue` als Fibonacci Heap in der Klasse `FibonacciHeap`. Gehen Sie vor wie in der Vorlesung *Algorithmen und Datenstrukturen* beschrieben. Beachten Sie, dass wir statt `increaseKey` eine Methode `update` verwenden, welche die Priorität eines Schlüssels sowohl erhöhen als auch verringern können soll. Eine kurze Beschreibung befindet sich unten.

   Die zusammengefasste Algorithmenbeschreibung für `deleteMax` aus der Vorlesung *Algorithmen und Datenstrukturen*:

   - Phase 1: Das Maximum wird gelöscht und alle Kinder werden neue Wurzeln. Neuen Wurzeln verlieren ihre Markierung, falls vorhanden.
   - Phase 2: Es folgt ein Aufräumschritt, der Bäume zusammenfasst:
     - Es werden wiederholt zwei Bäume miteinander verbunden, deren Wurzeln gleichen Knotengrad haben.
     - Dabei wird die kleinere der beiden Wurzeln Kind der anderen Wurzel.
     - Dies geschieht so lange, bis kein Knotengrad zweimal vorhanden ist.

   Um die Verschmelzungen effizient durchführen zu können, speichert man ein Array von Listen: Für jeden Knotengrad `d` enthält das Array eine Liste d auf alle Wurzeln vom Grad `d`.

[4] z.B. https://de.wikipedia.org/wiki/Skew_Heap#Nicht-rekursive_Verschmelzung_zweier_Heaps_(non-recursive_merge)

Die zusammengefasste Algorithmenbeschreibung für `update` ist an die Operation `increaseKey` aus der Vorlesung *Algorithmen und Datenstrukturen* angelehnt:

- Der Schlüssel wird aktualisiert.
- Danach wird der Schlüssel und sein zugehöriger Teilbaum herausgetrennt, und zu einem neuen Baum des Waldes gemacht.
- Solange der Parent-Knoten bereits eine Markierung hat, wird auch dieser wiederum zur Wurzel eines Baumes gemacht und iterativ zum nächsten Parent übergegangen.
- Sobald der Parent-Knoten keine Markierung hat, bekommt er eine Markierung (es sei denn, er ist eine Wurzel) und der Prozess endet.
- Falls der aktualisierte Eintrag kleinere Priorität hat als der ursprüngliche, sollen zusätzlich noch dessen Kinder abgetrennt und in den Wald eingefügt werden (wie oben beschrieben). Gegebenenfalls muss die Referenz auf das maximale Element aktualisiert werden.

e) Stellen Sie sicher, dass Sie mit Ihren Tests aus Aufgabenteil a) für jede Ihrer `PriorityQueue` Implementierungen mindestens eine Instruction Coverage von 90% erreichen. Erweitern Sie Ihre Tests gegebenenfalls oder fügen Sie neue Tests hinzu.

Aufgabe 2 Spielebundles: Ein Mengenüberdeckungsproblem

Harry Hacker hat sich ein Computerspiel gekauft, das sich durch kostenpflichtige Bundles erweitern lässt. Die Bundles sind unterschiedlich teuer und enthalten bestimmte Features. In vielen Fällen ist dasselbe Feature in mehreren Bundles enthalten.

Zur Maximierung des Spielspaßes braucht Harry Hacker alle Features, aber er möchte gleichzeitig jene Bundles auswählen, die insgesamt am wenigsten kosten.

Wir haben es mit einer Instanz des gewichteten Mengenüberdeckungsproblems zu tun: Es ist eine endliche Menge F von Features gegeben sowie eine Familie $\mathcal{B}$ von Teilmengen ("Bundles") von F, wobei jedes Element von F mindestens in einer Teilmenge enthalten ist (jedes "Feature" ist in mindestens einem "Bundle" enthalten), also

$$F = \bigcup_{S \in \mathcal{B}} S.$$

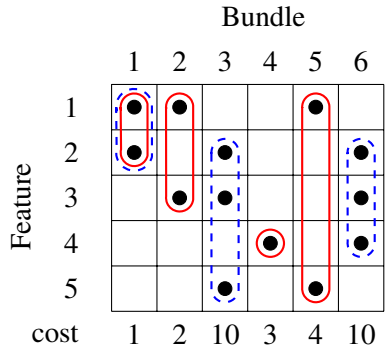
Weiterhin ist jeder Teilmenge $S_i \in \mathcal{B}$ eine Gewichtung w_i zugeordnet. Diese Gewichtung entspricht den effektiven Kosten des Bundles. Ziel ist es nun eine Menge $C \subseteq \mathcal{B}$ auszuwählen, sodass die Gesamtkosten

$$\sum_{S_i \in C} w_i$$

für die Auswahl minimal sind. In unserem konkreten Fall kann jedes Bundle gewichtet werden durch den Preis des Bundles dividiert durch die Anzahl der Features im Bundle, welche noch nicht in der Auswahl enthalten sind (d. h. wir betrachten den Preis pro Feature).

Anhand dieser Gewichtung kann ein Greedy-Algorithmus dann in jedem Auswahlschritt das Bundle mit dem geringsten Preis pro Feature auswählen. Um stets auf das Element mit der niedrigsten Gewichtung zugreifen zu können, verwenden wir die `PriorityQueue`-Implementierungen aus Aufgabe 1. Da wir an minimalen und nicht an maximalen Kosten interessiert sind, müssen wir den Vergleich zweier Zahlen invertieren (mehr dazu später). Da die Gewichte nach jeder Auswahl größer werden oder gleich bleiben und wir mit der umgekehrten Vergleichsoperation das minimale Element suchen, muss die `update`-Methode der `PriorityQueue` insbesondere auch die Priorität verringern können (in *Algorithmen und Datenstrukturen* war nur `increaseKey` gefordert, `update` soll hier nun auch den Schlüssel verringern können).

Beispiel:
In der Abbildung unten sind 6 Bundles und die darin enthalten Features tabellarisch dargestellt. Wir erhalten eine vollständige Überdeckung (also alle 5 Features), wenn wir alle Bundles kaufen, dann wäre der Gesamtpreis jedoch 30€. Wir könnten auch nur die gestrichelt blau umrandeten Bundles 1, 3 und 6 auswählen, dann wären die Gesamtkosten 21€.



Der oben beschriebene Greedy-Algorithmus würde jedoch in jedem Auswahlschritt, wie in folgender Tabelle dargestellt, immer das Bundle mit der kleinsten Gewichtung auswählen (rot markiert in der Tabelle und rot umrandet in der Abbildung).

	Bundle 1	Bundle 2	Bundle 3	Bundle 4	Bundle 5	Bundle 6
Preis	1€	2€	10€	3€	4€	10€
Gewichte Schritt 1	1/2 = 0.5	2/2 = 1.0	10/3 = 3.33 . . .	3/1 = 3.0	4/2 = 2.0	10/3 = 3.33 . . .
Gewichte Schritt 2	- / ∞	2	5	3	4	5
Gewichte Schritt 3	- / ∞	- / ∞	10	3	4	10
Gewichte Schritt 4	- / ∞	- / ∞	10	- / ∞	4	- / ∞

Der Greedy-Algorithmus wählt also die Bundles 1, 2, 4 und 5 mit Gesamtkosten von 10€ aus.

Im Allgemeinen findet der Greedy-Algorithmus nicht immer die optimale Lösung, liefert aber eine gute Approximation.

Legen Sie Ihre Lösungen der folgenden Aufgaben im Paket `de.tukl.programmierpraktikum2020.p1.a2` ab.

a) Implementieren Sie die Klasse `WeightedSetCovering` mit dem Konstruktor:

```
public WeightedSetCovering(Set<E> targetSet, Set<WeightedSet<E>> familyOfSets,
    PriorityQueue<WeightedSet<E>> queue)
```

Die Menge `targetSet` soll mit Elementen aus `familyOfSets` überdeckt werden, sodass die Gesamtkosten möglichst minimal sind.

Die Methode **`public`** `Set<WeightedSet<E>> greedyWeightedCover()` soll diesen Algorithmus mit Hilfe der im Konstruktor übergebenen `PriorityQueue<WeightedSet<E>> queue` implementieren und die Auswahl als `Set<WeightedSet<E>>` zurückgeben. Die Schnittstelle für gewichtete Mengen ist gegeben durch

```
/**
 * Immutable weighted set
 * @param <E> Type of elements contained in WeightedSet
 */
public interface WeightedSet<E> {
    Set<E> getSet();
    double getWeight();
    // returns a WeightedSet without the elements contained in 'other' set
    WeightedSet<E> subtractWeightedSet(WeightedSet<E> other);
}
```

b) Schreiben Sie in der Klasse `WeightedSetCoveringTest` Methode `weightedSetBeispiel` einen Testfall für das in der Abbildung oben aufgeführte Beispiel von Bundles mit zugehörigen Features. Verwenden Sie die in der Vorlage enthalte Klasse **`public class`** `Bundle` **`implements`** `WeightedSet<Integer>`, um die Bundles zu repräsentieren.

Den `Comparator` können Sie Ihrer `PriorityQueue` wie in folgendem Beispiel übergeben:

```
Comparator<WeightedSet<Integer>> cw;
cw = Comparator.comparingDouble(WeightedSet::getWeight);
PriorityQueue<WeightedSet<Integer>> queue = new FibonacciHeap<>(cw.reversed());
```

Da das minimale Element gesucht ist, wird mit `reversed()` der `Comparator` umgekehrt.