Karlsruher Institut für Technologie

Institut für Theoretische Informatik
T.T.-Prof. Dr. Thomas Bläsius
https://scale.iti.kit.edu

Übungsblatt 11

Algorithmen I - Sommersemester 2024

Abgabe im ILIAS bis 12.07.2024, 18:00 Uhr

Die Abgabe erfolgt alleine oder zu zweit als *eine* PDF-Datei über das Übungsmodul in der Gruppe eures Tutoriums im ILIAS. Beschriftet die Abgabe deutlich mit Matrikelnummer und Name.

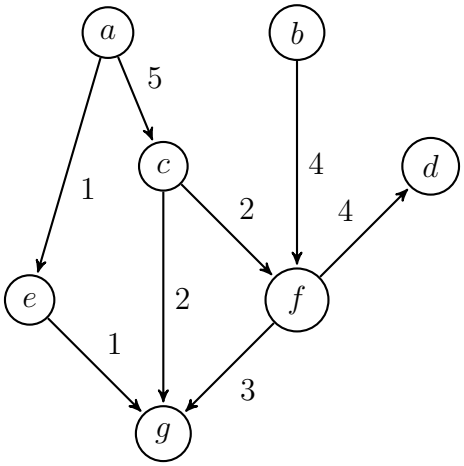
- Achtet bei handschriftlichen Abgaben auf Lesbarkeit.
- Achtet darauf, ob Algorithmen in Worten oder Pseudocode beschrieben werden sollen.
- Es werden immer asymptotisch möglichst effiziente Algorithmen erwartet, wenn nicht anders angegeben.
- Wenn Korrektheits- oder Laufzeitanalysen gefordert sind, behandelt diese separat von der Algorithmenbeschreibung.

Gesamtpunkte: 20 (+ 3 Bonus)

Aufgabe 1 - Dijkstras Angst-Gegner (6 Punkte)

In dieser Aufgabe soll es wieder um kürzeste Pfade gehen. Wir betrachten hierfür ausschließlich gewichtete DAGs mit ganzzahlig aber nicht zwingend positiven Gewichten. Zunächst arbeiten wir mit dem Algorithmus von Bellman-Ford. In einer Iteration von Bellman-Ford, werden alle Kanten einmal relaxiert. Wir nutzen Knotenordnungen, um festzulegen, in welcher Reihenfolge der Algorithmus die Kanten betrachtet. Dabei bleibt die Reihenfolge der Kanten in jeder Iteration gleich. Die Knoten werden dabei in der gegeben Reihenfolge abgearbeitet und es werden jeweils die ausgehenden Kanten des Knoten relaxiert. Die Reihenfolge, in der die ausgehenden Kanten des aktuellen Knotens betrachtet werden, ist ebenfalls durch die Knotenordnung des Zielknotens gegeben.

Für die Analyse nutzen wir den folgenden Beispielgraphen mit Startknoten a, um das Problem zu visualisieren.



a) Führe auf dem Beispielgraphen den Algorithmus von Bellman-Ford (Pseudocode in Vorlesung 10 auf Folie 8) aus. Gib hierfür in jeder Iteration die aktuell kürzesten Wege von a aus an und nutze die Knotenordnung *abcefgd*. Wiederhole dies nun mit der Ordnung *dfgbcea*. Gib für beide Knotenordnungen außerdem an, wie viele Iterationen der Algorithmus benötigt, bis die kürzesten Pfade gefunden wurden.
Hinweis: Um die kürzesten Wege anzugeben, kannst du die folgenden Tabellen nutzen. (2 Punkte)

Iteration	a	b	c	d	e	f	g
1							
2							
3							
4							
5							
6							

Tabelle 1: Knotenordnung *abcefgd*

Iteration	a	b	c	d	e	f	g
1							
2							
3							
4							
5							
6							

Tabelle 2: Knotenordnung *dfgbcea*

b) Wir wollen nun diese Ergebnisse verallgemeinern. Für einen beliebigen Graphen (mit obigen Einschränkungen): Welche Knotenordnung aller Knoten minimiert die Anzahl der Iterationen bis zum Finden der kürzesten Pfade? Welche Knotenordnung maximiert diese? (1.5 Punkte)

Wir nutzen nun die Erkenntnisse der Analyse für unser Problem.

c) Beschreibe in Worten einen Algorithmus, der für einen gegebenen Startknoten die **kürzesten** Pfade[1] zu allen anderen Knoten findet. Begründe die Laufzeit und Korrektheit deines Algorithmus. (2.5 Punkte)

[1] In dieser und der nächsten Teilaufgabe ist es ausreichend das Distanzarray zu bestimmen. Die Pfade können ähnlich wie für BFS auf Blatt 6 Aufgabe 2 b) gesehen rekonstruiert werden.

Aufgabe 2 - Klare Kante zeigen (6 Punkte)

In der Vorlesung hast du im Zusammenhang mit DFS eine Klassifikation von Kanten in Baum-, Rück-, Vor- und Querkanten kennengelernt. Wir erweitern die Kantenklassifikation auf Breitensuche. Wir betrachten dafür den gerichteten Graphen $G = (V, E)$ und den Breitensuchbaum $T = (V, E_T)$, den der Algorithmus ausgibt. Die Kanten aus E_T nennen wir Baumkanten. Für eine Folge von Knoten $v_1, \ldots, v_k$, so dass v_i der Elternknoten von v_{i+1} für alle $i \in \{1, \ldots, k-1\}$ ist, nennen wir $(v_1, v_k) \in E \setminus E_T$ eine Vor- und $(v_k, v_1) \in E \setminus E_T$ eine Rückkante. Alle verbleibenden Kanten klassifizieren wir als Querkante. Außerdem erweitern wir die Klassifikation auf ungerichtete Graphen für BFS und DFS. In diesem Fall sind Vor- und Rückkanten ein und das selbe, wodurch wir nur noch in drei Gruppen einteilen. Alle Graphen sind in dieser Aufgabe ungewichtet.

Betrachte die folgenden Szenarien und gib jeweils an, welche der vier Kanten **nicht** auftreten können. Begründe deine Antwort kurz.

a) Wir führen BFS auf gerichteten Graphen aus. (1.5 Punkte)

b) Wir führen DFS auf ungerichteten Graphen aus. (1.5 Punkte)

c) Wir führen BFS auf gerichteten Bäumen aus. (1.5 Punkte)

d) Wir führen DFS auf DAGs aus. (1.5 Punkte)

Aufgabe 3 - Ausmerzen, wüste Beschimpfungen und viel Söder, Spannungen bei der Union! (8 Punkte)

In der Vorlesung zu Minimalen Spannbäumen hat Dr. Meta Kruskals Algorithmus kennen gelernt. Außerdem wurde ihm erklärt, dass dieser eine Datenstruktur benötigt, die es erlaubt Komponenten zu verwalten. Da Dr. Meta die nächste Vorlesung nicht erwarten kann, beschließt er, selbst eine solche Datenstruktur zu entwickeln.

Die Datenstruktur soll eine *Partitionierung* der Zahlen aus der Grundmenge $G = \{1, \ldots, n\}$ abspeichern. Eine Partitionierung $P = \{P_1, \ldots, P_k\}$ ($P_i \subseteq G$) ist eine Menge von Mengen, sodass jede Zahl der Grundmenge in genau einer Menge enthalten ist. Die Elemente der Partitionierung nennen wir *Partition*. Jede Partition P_i wird durch einen eindeutigen Repräsentanten identifiziert. Insbesondere soll diese Datenstruktur die folgenden Operationen unterstützen:

- `union`($x : \mathbb{N}$, $y : \mathbb{N}$): Vereinigt die Partition, die x enthält, mit der Partition, die y enthält.
- `find`($v : \mathbb{N}$): Gibt den eindeutigen Repräsentanten der Partition, die v enthält, zurück.

Dr. Meta verlangt von dir eine solche Datenstruktur mit den bisher aus der Vorlesung bekannten Datenstrukturen umzusetzen. Du machst dich allerdings mithilfe eines Beispiels erst einmal mit seinen Anforderungen vertraut.

a) Betrachte die folgende Partitionierung der Zahlen $\{1, \ldots, 7\}$ in Mengen.

$$\{1, 2\}, \{4, 6\}, \{3\}, \{5\}, \{7\}$$

Wir nutzen in diesem Beispiel stets die kleinste Zahl als den eindeutigen Repräsentanten einer Menge. Führe die Operationsfolge `find`(6), `union`$(3, 4)$, `find`(6), `union`$(2, 3)$, `find`(6) aus. Gib hierfür nach jeder `union`-Operation die entstehende Partitionierung und nach jeder `find`-Operation den gefundenen Repräsentanten an. (1 Punkt)

Dein erster Implementierungsversuch nutzt ein **Array** der Größe n. Dabei speicherst du am Index i den Repräsentanten der Partition, die die Zahl i enthält.

b) Gib an, wie `union` und `find` umgesetzt werden können. Gib außerdem die Laufzeit der beiden Operationen **ohne Begründung** an. (1.5 Punkte)

Insgesamt ist Dr. Meta deine Datenstruktur noch zu langsam. Wir wollen nun eine **Variante der** $(2, 3)$**-Bäume** verwenden, um von einem Knoten direkt auf den Repräsentanten seiner Partition schließen zu können. Wir bezeichnen diese als *Balanced Tree*. Im Unterschied zu $(2, 3)$-Bäumen ist die Reihenfolge der gespeicherten Elemente nicht relevant und die Schlüssel an jedem Knoten verlieren ihre Bedeutung zur Navigation, werden allerdings beibehalten. Außerdem unterstützen wir folgende Operationen:

- `findRoot`(p: Zeiger): Erhält einen Zeiger auf ein beliebiges Listenelement (eines Balanced Trees) und gibt einen Zeiger auf die Wurzel des Baumes, der zu diesem Listenelement gehört, zurück.
- `merge`(p : Zeiger, q : Zeiger): Erhält Zeiger p und q auf jeweils eine Wurzel eines Balanced Trees und vereinigt die Bäume unter p und q zu einem Balanced Tree.

Wir überlegen uns zunächst, wie wir diese Operationen auf Balanced Trees umsetzen können. Für die Angabe der Umsetzung reicht es nicht, die in der Vorlesung definierten Operationen auf $(2, 3)$-Bäumen zu kombinieren, du musst hier die interne Implementierung auf Listenelement- und Zeigerebene angeben.

c) Beschreibe in Worten, wie du `findRoot` in Zeit $\mathcal{O}(\log(n))$ auf einem balanced tree, der n Elemente speichert, umsetzen kannst. Begründe die Laufzeit deiner Umsetzung. (1 Punkt)

d) Beschreibe in Worten, wie du `merge` in Zeit $\mathcal{O}(\log(n + m))$ für zwei balanced trees mit jeweils n bzw. m Elementen umsetzen kannst. Begründe die Laufzeit deiner Umsetzung und gibt an, wie du sicherstellst, dass der Baum balanciert bleibt. (2.5 Punkte)

Wir nutzen jetzt für jede Partition einen eigenen Balanced Tree. Außerdem halten wir uns ein Array der Größe n welches im Eintrag i einen Zeiger auf ein Listenelement eines Balanced Trees speichert, dass die Zahl i enthält[2]. Als Repräsentant einer Partition

[2]Falls wir die Datenstruktur nutzen, um Kruskals Algorithmus umzusetzen, ist die Initialisierung dadurch gegeben, dass alle Knoten in einer eigenen Menge liegen. Somit beginnen wir mit n Balanced Trees und jeder Arrayeintrag zeigt auf das einzige Element eines Balanced Trees.

wählen wir die Zahl, die in der Wurzel des entsprechenden Balanced Trees gespeichert ist. Wir setzen nun die beiden Operationen der Mengenverwaltungsdatenstruktur um. Hier ist n wie oben definiert die Größe der Grundmenge.

e) Beschreibe in Worten, wie du `find` in Zeit $\mathcal{O}(\log(n))$ umsetzen kannst. (1 Punkt)

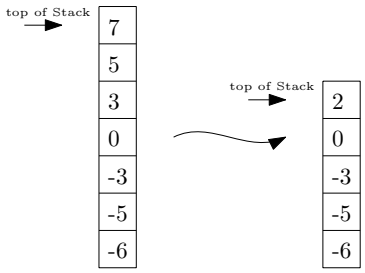
f) Beschreibe in Worten, wie du `union` in Zeit $\mathcal{O}(\log(n))$ umsetzen kannst. (1 Punkt)

Aufgabe 4 - Zum Knobeln (0 Punkte + 3 Bonus)

Ein *geordneter Stack* ist ein Stack in dem die Elemente von unten nach oben in geordneter Reihenfolge auftreten. Diese Datenstruktur unterstützt die folgenden Operationen:

- `pop` (): Entferne das oberste Element vom geordneten Stack und gebe das entfernte Element zurück.
- `pushAndRestoreOrder` (x): Füge x oben auf den geordneten Stack ein. Entferne solange das Element direkt unter x, bis x das größte Element auf dem geordneten Stack ist und die Ordnung wieder hergestellt wurde.

Betrachte das folgende Beispiel, in dem `pushAndRestoreOrder` (2) auf einem geordneten Stack aufgerufen wird. Hier werden $7, 5$ und 3 entfernt, um die Ordnung wiederherzustellen.



Der Stack vor und nach Aufruf der Operation.

Wir implementieren den geordneten Stack analog zum normalen Stack (siehe Blatt 3 Aufgabe 1, aber ohne `spy`) als doppelt verkettete Liste. Sei n die Anzahl an Elementen im Stack.

a*) Gib die worst-case Laufzeit der beiden Operationen in Abhängigkeit von n an und begründe deine Antwort kurz. (1 Punkt)

b*) Zeige, dass die Laufzeit für eine beliebige Folge von Operationen auf einem zu Beginn leeren geordneten Stack bei $\Theta(1)$ pro Operation liegt. Nutze eine Methode deiner Wahl. (2 Punkte)

Eulenfest am 12. Juli 2024

Nächste Woche Freitag findet endlich wieder das Eulenfest statt! Los geht es um 18:30 Uhr, im und um den Infobau. Dieses Jahr gibt es auf der großen Bühne Livemusik von den Bands *JxP* und *Canalopi*. Danach legen unsere DJs *Paul Städter*, *Julicorn und Luis Reissenweber* sowie *queerbeats* auf. Natürlich gibt es auch wieder Karaoke. Außerdem werdet ihr mit Getränken und Essen versorgt. Weitere Infos zum Fest findet ihr unter eulen.party.

Wir suchen auch noch tatkräftige Unterstützung für unser Helferteam. Ob beim Auf- oder Abbau, als Security oder als Betreuer für die zahlreichen Essens- und Getränkestände. Für die Helfer bleibt dabei auch immer was an den Ständen übrig. Tragt euch dafür einfach ins Helfersystem (redseat.de/eulenfest2024) ein.

Mehr Infos und Helfersystem