Graphenalgorithmen: Blatt 3

Lukas Kalbertodt, Elena Resch, Mirko Wagner

8. Mai 2015

Aufgabe 6:

(a) Es handelt sich um einen vollständigen Graphen mit Schlingen mit $V = \{1, \dots, k\}$ als Knoten- und $E := \{(i, j) : i \leq j, \quad i, j \in V\}$ als Kantenmenge. Das Problem entspricht dem Euler-Pfad auf vollständigen Graphen. Die Knoten des Graphen repräsentieren die Zahl, die gerade am Ende der Kette liegt. Eine Kante verbindet immer zwei Zahlen und stellt somit einen Stein dar. Jeder Knoten hat eine Schleife, die aber irrelevant für das Euler-Pfad Problem ist. Der Graph ist vollständig, weil es für jede Kombination von zwei Zahlen $\leq k$ genau einen Spielstein gibt. Daher muss man auch jede Kante exakt einmal besuchen.

(b) Das Problem mit $k = 2, 3$ ist lösbar, für $k = 4$ nicht. $k = 2$ ist dabei ein Sonderfall und eine Ausnahme für die in (c) genannte Regel.

(c) Vollständige Graphen besitzen immer genau dann einen Eulerweg, wenn sie eine ungerade Anzahl von Knoten k haben, weil jeder Knoten $k - 1$ Kanten besitzt und für den Eulerpfad 0 oder 2 Knoten ungeraden Knotengrad haben müssen (Skript Satz 1.3). Der Graph mit $k = 2$ bildet dabei eine Ausnahme, denn seine zwei Knoten besitzen alle Knotengrad 3, da Schlingen doppelt gezählt werden, während für $k > 2$, k gerade gilt, dass alle k Knoten $deg(i) = k - 1, i \in V$ ungerade ist.

(d) Eine geschlossene Kette entspricht einem Eulerkreis im Graphen. Da in dem vollständigen Graphen *alle* Knoten den selben Grad $k - 1$ haben, ist ein Eulerkreis immer dann möglich, wenn der Knotengrad bei allen Knoten gerade ist, also $k - 1$ gerade, insgesamt also k ungerade ist (Skript Satz 1.2).

Aufgabe 7:

Ein „Heap“ ist eine Datenstruktur, die als PriorityQueue genutzt werden kann. Eine PriorityQueue verwaltet eine Menge von Items bestehend aus `key` und `data`. Die PriorityQueue soll jetzt in der Lage sein, möglichst schnell das Objekt mit der höchsten Priorität (niedrigstem Key) zu finden, zurückzugeben und aus der PriorityQueue zu löschen. Die gewünschten Funktionen des ADTs PriorityQueue:

- `min() -> item*`: Gibt eine Referenz auf das Item mit dem kleinsten Key zurück

- `deleteMin() -> item`: Findet und extrahiert das Item mit dem kleinsten Key
- `insert(item)`: Fügt ein Item ein
- `decreaseKey(item*, newValue)`: Verringert den Key eines bestimmten Items
- `delete(item*)`: Löscht ein Item
- `merge(Heap)`: vereint den Heap mit einem anderen (wird nur z.t. gefordert)

„Heap“ ist jetzt der Überbegriff für mehrere Datenstrukturen, die die oben genannten Funktionen schnell ausführen können. Der einfachste und bekannteste Heap ist der BinaryHeap, der alle Items in einem binären Baum organisiert. Innerhalb des Baumes gilt die Invarianz, dass der Key in den beiden Söhnen eines Knoten j größer ist, als der Key in j. Somit ist das Element mit dem kleinsten Key in der Wurzel des Baums. Diese Variante des Heaps ist bereits recht effizient in der Praxis. Eine Abwandlung sind die sog. „d-ary Heaps“, die die selbe asymptotischen Laufzeiten haben (d konstant), aber in der Praxis besser sein können. Hier werden die Items nicht in einem binären Baum, sondern in einem d-ären Baum gespeichert.

Darüber hinaus gibt es aber weitere und sehr exotische Varianten, mit teilweise deutlich besseren asymptotischen Laufzeiten. Zu erwähnen wäre beispielsweise der PairingHeap, der `insert` in $\mathcal{O}(1)$ und `decreaseKey` in armotisiert $o(log\ n)$ schafft und dabei noch praxistauglich ist. Der FibonacciHeap schafft `decreaseKey` zwar in amortisiert $\mathcal{O}(1)$, dies zahlt sich jedoch erst bei sehr großen Probleminstanzen gegenüber dem PairingHeap aus. Der BrodalHeap schafft sogar zusätzlich `decreaseKey` in konstanter Zeit, ist aber in der Praxis absolut untauglich.

Was die Laufzeiten angeht, so kann man eine sinnvolle untere Schranke ableiten: Das Sortierverfahren HeapSort nutzt einen Heap, um per Vergleich zu sortieren. Für vergleichbasierte Sortierverfahren ist ja bereits die untere Schranke $\Omega(n \cdot \log n)$ bekannt. Heapsort ruft je n mal die Funktionen `insert` und `deleteMin` auf. Somit muss entweder `insert` oder `deleteMin` mindestens die Laufzeit $\mathcal{O}(\log n)$ haben. Die restlichen Laufzeiten im Überblick ($*$ armortisiert):

Funktion	Binary	d-ary	Fibonacci	Pairing	Brodal
`min`	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$
`insert`	$\mathcal{O}(\log n)$	$\mathcal{O}(\log_d n)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$
`decreaseKey`	$\mathcal{O}(\log n)$	$\mathcal{O}(\log_d n)$	$\mathcal{O}(1)*$	$o(\log n)*$	$\mathcal{O}(1)$
`deleteMin`	$\mathcal{O}(\log n)$	$\mathcal{O}(d \cdot \log_d n)$	$\mathcal{O}(\log n)*$	$\mathcal{O}(\log n)*$	$\mathcal{O}(\log n)$
`delete`	$\mathcal{O}(\log n)$	$\mathcal{O}(d \cdot \log_d n)$	$\mathcal{O}(\log n)*$	$\mathcal{O}(\log n)*$	$\mathcal{O}(\log n)$