Datenstrukturen und effiziente Algorithmen
Übung 11

Markus Vieth, David Klopp, Christian Stricker

25. Januar 2016

Markus Vieth, David Klopp, Christian Stricker

Aufgabe 1

0.1 Mergebale Heap

`insert, Min, ExtractMin, Union, Make-Heap`

zusätzlich `DecreaseKey, Delete`

	`Make-Heap`	`Insert`	`Min`	`Extr.Min`	`Union`	`Dec.-Key`	`Del`
Fib-Heap	1	1	1	$\log n$	1	1	$\log n$
Bin-Heap	1	$\log n$	1	$\log n$	n	$\log n$	$\log n$

Zyklische doppelt verkettete Liste von Wurzelknoten
Darunter Bäume

- Kinder eines Knoten ebenfalls zylisch doppelt verkettete Liste
- Jeder Knoten Pointer zum Vater (außer Wurzel) und zum Kind/ zur Kinderliste

keine Sortierung in Wurzelliste/Kindliste
Heap hat zudem Attribute `min` und `size`

∀ **Knoten:** #Kinder(direkt) ”rank”/”degree”

bool marked (false zu Beginn)

insert: neuer Node in Wurzelliste einfügen, H.min und H.size anpassen ⇒ konstant

Min: min-pointer zurückgeben ⇒ konstant

Union: Wurzellisten zusammenführen, Min und Size anpassen ⇒ konstant

Extract-min

Für size < 2 Min zurückgeben wenn vorhanden, danach leerer Heap

Für size ≥ 2 gib min zurück, setze min auf anderen Knoten in Wurzelliste, consolidate, min und size updaten

⇒ $\log n$

Consolidate

Ziel Nach Consolidierung sollen in der Wurzellsite keine zwei Knoten mit dem selben Rang sein.
Node-Pointer-Array mit Größe $\log(size)$

decreaseKey Setze Key auf neuen Wert, wenn Heapeigenschaft verletzt:

```
1  do {
2    Nehme x in Wurzellsite auf, entferne x aus der Kindesliste seines
         Parents
3    Wenn x.p nicht makietr war:
4      makiere x.p
5      break
6    sonst x = x.p
7    Nehme x in Wurzellsite auf
8  }while true
```

$\Rightarrow$ amort konstant

Delete decreaseKey mit $-\infty$ (∞ bei Maxheap) extractMin $\Rightarrow \quad \log n$

Aufgabe 3

a)

Hilfsmethode

```
1  int left(int i) {
2    return 2i+1;
3  }

5  int right(int i) {
6    return 2i+2;
7  }

9  int parent(int i) {
10   return floor((i-1)/2);
11 }

13 // Heap-Eigenschaft herstellen
14 void heapify(Array b, int i) {
15   // kleinstes Element des Asts finden
16   int min = i
17   if (left(i) < b.size && b[left(i)].key < b[min].key)
18     min = left(i)
19   if (right(i) < b.size && b[right(i)].key < b[min].key)
20     min = right(i)

22   // wenn ein kleines Element existiert dann stelle die Heap-Eigenschaft her
23   if (min != i) {
24     b.swap(i, min)
25     // verfahre kaskadenartig weiter
26     heapify(b, min)
27   }
28 }
```

decreaseKey

```
1  bool decrease(Array b, int i, newKey) {
2    // neuer Schlüssel muss kleiner sein als alter
3    if (newKey > b[i].key)
4      return false;

6    // Schlüssel setzten
7    b[i].key = newKey;

9    // Heap-Eigenschaft herstellen
10   while (i > 0 && b[i].key < b[parent(i)].key)) {
11         b.swap(i, parent(i));
12         i = parent(i);
13   }
14   return true;
15 }
```

deleteMin

```
1  int deleteMin(Array b) {
2    if (i<b.size)
3      return -1; // Index out of bounds
4    min = b[0];
5    b.swap(i, -1); // sei -1 der Index des letzten Elements
6    b.size -= 1 // verringere die Array Größe um 1

8    // Heap-Eigenschaft herstellen
9    if ((b.size > 0)
10     heapify(b, i)
11   return min;
12 }
```

insert

```
1  void insert(Array b, newKey) {
2    int i = b.size
3    b.size += 1;
4    // setzte b auf den höchst möglichen Wert
5    b.key = MAX_INT
6    // Update den Wert und die Heap-Eigenschaft mit decrease
7    decrease(b, i, newKey)
8  }
```

b)

Die Laufzeit für die decreaseKey Operation beträgt $O(Baumtiefe) = O(\log(n))$. Unter der Annahme, dass das Resizen des Arrays in konstanter Zeit realisiert werden kann, beträgt somit die Laufzeit für insert ebenfalls $O(\log(n))$. Ausschlaggebend für die deleteMin Operation ist die heapify Methode, da sich der Rest der Implementierung in konstanter Zeit realisieren lässt. Die Laufzeit für heapify und somit auch für deleteMin beträgt ebenfalls $O(Baumtiefe) = O(\log(n))$.

c)

Füge alle Elemente unsortiert in ein Array ein. Laufzeit: $O(1)$.
Wende nun Radix-Sort auf das Array an, um es zu sortieren. Laufzeit: $O(n)$. (Siehe hierzu VL 11)
Die Heap-Eigenschaft ist nun für alle Knoten erfüllt.

Zusatz 1

a)

Behauptung $(E, M) = (\{a, b, c, d\}, \{\emptyset, \{a\}, \{b\}, \{c\}\{d\}\})$ ist ein Matroid.

Beweis

zu zeigen: Nicht-Leerheit

$$\emptyset \in M = \{\emptyset, \{a\}, \{b\}, \{c\}\{d\}\}$$

zu zeigen: Vererbung Die Teilmengen einer einelementigen Menge sind die Menge selbst und die leere Menge. Somit ist offensichtlich, dass auch diese Bedingung erfüllt ist.

zu zeigen: Austausch Sei X eine der einelementigen Mengen aus M. Das einzige Element mit geringerer Kardinalität ist die leere Menge.

$$\Rightarrow \exists x \in X \setminus \emptyset = X | \{x\} \cup \emptyset = X \in M$$

$\Rightarrow (E, M)$ ist ein Matroid.

b)

Nich-Leerheit M muss die leere Menge enthalten $\Rightarrow M' := \{\emptyset, \{a, b\}, \{c, d\}\} \subseteq M$.

Vererbung Sei $\mathcal{P}(X)$ die Potenzmenge der Menge X

$$M' \cup \mathcal{P}(\emptyset) \cup \mathcal{P}(\{a, b\}) \cup \mathcal{P}(\{c, d\}) = \{\emptyset, \{a, b\}, \{c, d\}, \{a\}, \{b\}, \{c\}, \{d\}\} =: M'' \subseteq M$$

Austausch Wie in Teilaufgabe a) erklärt, muss die leere Menge nicht weiter betrachtet werden, da mit dieser lediglich die einelementigen Mengen erzeugt werden können.

$\boldsymbol{A = \{v\}, B = \{v, w\}}$ **mit** $\boldsymbol{v, w \in E}$ $B \setminus A = \{w\}, \{w\} \cup A = \{v, w\}$

$\boldsymbol{A = \{a\}, B = \{c, d\}}$ $B \setminus A = \{c, d\} \Rightarrow \{c\} \cup \{a\} = \{a, c\} \in M \veebar \{d\} \cup \{a\} = \{a, d\} \in M$

$\boldsymbol{A = \{b\}, B = \{c, d\}}$ $B \setminus A = \{c, d\} \Rightarrow \{c\} \cup \{b\} = \{b, c\} \in M \veebar \{d\} \cup \{b\} = \{b, d\} \in M$

$\boldsymbol{A = \{c\}, B = \{a, b\}}$ $B \setminus A = \{a, b\} \Rightarrow \{a\} \cup \{c\} = \{a, c\} \in M \veebar \{b\} \cup \{c\} = \{b, c\} \in M$

$\boldsymbol{A = \{d\}, B = \{a, b\}}$ $B \setminus A = \{a, b\} \Rightarrow \{a\} \cup \{d\} = \{a, d\} \in M \veebar \{b\} \cup \{d\} = \{b, d\} \in M$

$\Rightarrow M = \{\emptyset, \{a, b\}, \{c, d\}, \{a\}, \{b\}, \{c\}, \{d\}, \{a, c\}, \{b, d\}\} \veebar M = \{\emptyset, \{a, b\}, \{c, d\}, \{a\}, \{b\}, \{c\}, \{d\}, \{a, d\}, \{b, c\}\}$

$\Rightarrow (E, M)$ ist der kleinste Matroid, welcher den Anforderungen entspricht.

q.e.d.

Markus Vieth, David Klopp, Christian Stricker

Zusatz 2

a)

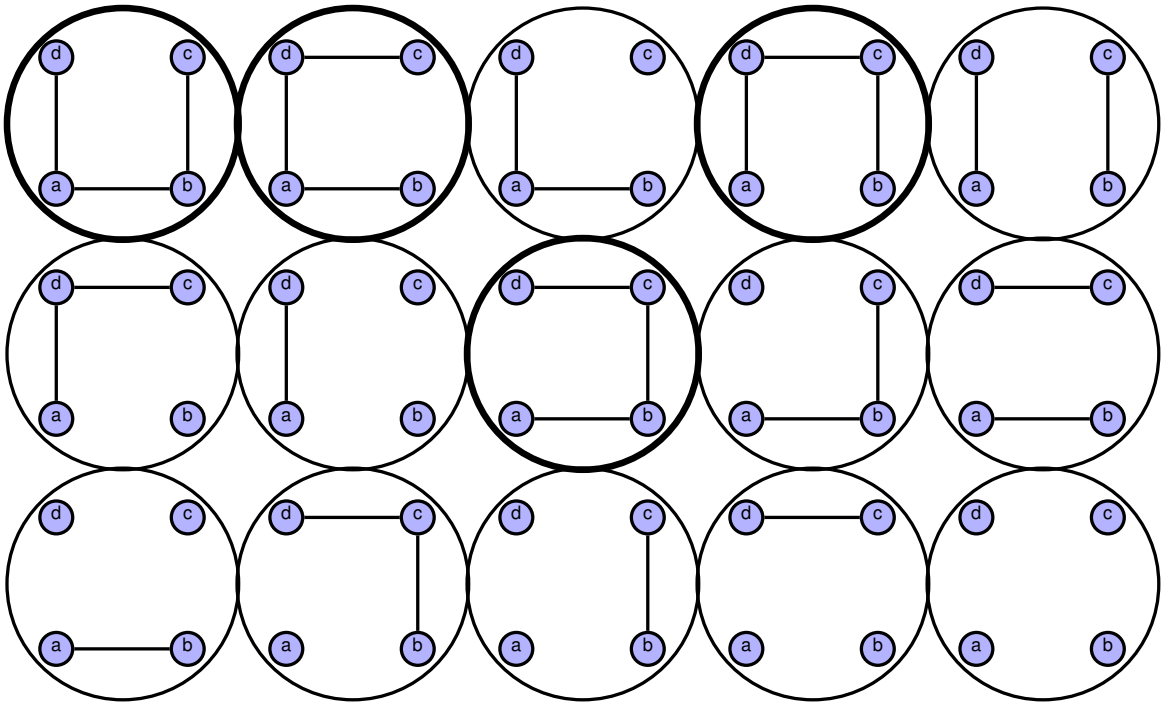

Abbildung 1: graphischer Matroid

Die Teilmengen größter Kardinalität sind $M_1 := \{(a,b),(b,c),(c,d)\}, M_2 := \{(a,b),(b,c),(d,a)\}$, $M_3 := \{(a,b),(c,d),(d,a)\}, M_4 := \{(b,c),(c,d),(d,a)\}$. Es handelt sich dabei um die möglichen Spannbäume des Graphen G.

b)

Es handelt sich um den Algorithmus von Kruskal. Der Algorithmus gibt $B = \{(a,b),(b,c),(c,d)\}$ aus. Die Abfrage verhindert das Bilden von Zyklen.

c)

Das Sortieren der Kanten hat im Allgemeinen (also ohne Radix-Sort) eine Laufzeit in $\mathcal{O}(|E|\log|E|)$. Die Schleife läuft über die Menge aller Kanten, also $|E|$ mal. Innerhalb der Schleife gibt es die Abfrage mit der Laufzeit $f(|E|)$ und die Einfüge-Operation, welche mit passender Datenstruktur in konstanter Zeit ausgeführt werden kann. Somit ergibt sich eine Laufzeit in $\mathcal{O}(|E|\log|E| + |E|f(|E|))$.

d)

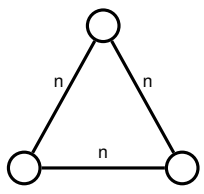

Abbildung 2: Sei n eine beliebigen Länge

Nein, der Algorithmus ist nicht eindeutig. Man denke sich einen dreieckigen Graphen mit den Kanten $E := \{(a,b),(b,c),(c,a)\}$ welche ein einheitliches Kantengewicht aufweisen. Das Ergebnis ist nun vom Sortierverfahren abhängig. So liefern allein stabile und instabile Algorithmen verschiedene Ergebnisse. Mögliche Lösungen wären $L_1 := \{(a,b),(b,c)\}, L_2 := \{(a,b),(c,a)\}, L_3 := \{(b,c),(c,d)\}$