

Algorithmen und Datenstrukturen (Master)

WiSe 19/20

Benedikt Lüken-Winkels

February 10, 2020

Contents

1 Union-Find	2
2 Hashing	5
3 Amortisierte Analyse	7
4 Planare Graphen	8
5 Split-Find	9
6 Maximum Matching im bipartiten Graphen	10
7 Random Search Tree	12
8 Planare Einbettung	13

1 Union-Find

Beschreiben Sie jeweils eine Lösung für das Union-Find-Problem mit Laufzeit

1. $O(\log n)$ (amortisiert) für UNION und $O(1)$ für FIND
2. $O(1)$ für UNION und $O(\log n)$ für FIND

wobei n die Anzahl der Elemente ist. Begründen Sie in beiden Fällen die entsprechenden Laufzeiten.

Lösung 1.)

Union in $O(\log n)$, Find in $O(1)$. **Idee:** Relabel the smaller half, sodass jedes Element nur maximal $\log n$ geändert wird:

Datenstruktur

name[x] : Name des Blocks, der x enthält
size[A] : Größe des Blocks A (Init 1)
list[A] : Liste der Elemente in Block A

Algorithmus 1 : Initialisierung

```
foreach  $x \in N$  do
    name[x] ← x;
    size[x] ← 1;
    list[x] ← {x};
end
```

Algorithmus 2 : Find(x)

```
return name[x];
```

Algorithmus 3 : Union(A,B)

```
if size[A] ≥ size[B] then
    foreach x ∈ list[B] do
        name[x] ← A;
    size[A] ← size[A]+size[B];
    list[A].append(list[B]);
else
    foreach x ∈ list[A] do
        name[x] ← B;
    size[B] ← size[A]+size[B];
    list[B].append(list[A]);
```

Laufzeit

Find: $O(1)$. Lookup im Array.

Union: $O(\log n)$. Jedes x kann maximal $\log n$ mal seinen Namen ändern, da es sich nach jeder Namensänderung in einer doppelt so großen Liste befindet.

Lösung 2.)

Union in $O(1)$ und Find in $O(\log n)$. **Idee:** Bei Union Anhängen des kleineren Teilbaums an den Größeren.

Das ergibt die Abschätzung $\text{size}[x] \geq 2^{\text{höhe}(x)}$, bzw $\log_2(\text{size}[x]) \geq \text{höhe}(x)$, also wird der Baum nie tiefer, als $\log n$

Datenstruktur

name[x]: Name des Blocks mit Wurzel x (nur, wenn x eine Wurzel relevant)
size[x]: Anzahl der Knoten im Unterbaum mit Wurzel x
wurzel[x]: Wurzel des Blocks mit Namen x
vater[x]: Vaterknoten des Knotens x. 0, wenn x Wurzel

Algorithmus 4 : Initialisierung

```
foreach x ∈ N do
    name[x] ← x;
    size[x] ← 1;
    wurzel[x] ← x;
    vater[x] ← 0;
end
```

Algorithmus 5 : Find(x)

```
while  $vater[x] \neq 0$  do  
  |  $x \leftarrow vater[x]$  ;  
end  
return  $name[x]$ ;
```

Algorithmus 6 : Union(A, B, C)

```
 $a \leftarrow wurzel[A]$ ;  
 $b \leftarrow wurzel[B]$ ;  
if  $size[a] \geq size[b]$  then  
  |  $vater[b] \leftarrow a$ ;  
  |  $name[a] \leftarrow C$ ;  
  |  $wurzel[C] \leftarrow a$ ;  
  |  $size[a] \leftarrow size[a] + size[b]$ ;  
else  
  | analog;  
end
```

Laufzeit

Find: $O(\log n)$. Höhe des Baums bleibt maximal $\log n$, da der kleinere Teilbaum immer an die Wurzel des Größeren gegangen wird und sich die Tiefe des Baums durch seine Größe abschätzen lässt.

Union: $O(1)$. Lediglich die Wurzel muss umgeschrieben werden.

2 Hashing

Entwickeln Sie eine Datenstruktur zur Speicherung von n Schlüsseln aus dem Universum $\{1, \dots, N\}$ (wobei $n \ll N$), die eine Zugriffszeit von $O(1)$ garantiert. Sie dürfen dabei $O(n^2)$ Speicherplatz verwenden.

(Perfektes Hashing) Verbessern Sie die Datenstruktur aus Aufgabe ??, so dass nur noch Speicherplatz $O(n)$ benutzt wird. Hashig durch Verkettung und mit offener Adressierung (Linear Probing: Wie funktioniert Delete())

Lösung

Hashing mit Verkettung Idee: keine Auflösung von Konflikten, sondern mehrere Schlüssel an gleicher Stelle speichern. Tafel T mit m Buckets, Hashfunktion h und Belegungsfaktor $B = \frac{n}{m}$.

Verdopplungsstrategie: Wenn $B > 2$ verdopple m und rehashe alle n mit neuem h .

Lookup(x): Lineare Suche in einer kurzen Liste $T[h(x)]$ in $O(1)$, da durch die Verdopplungsstrategie garantiert wird, dass es genug Platz gibt, um jedes Element in eine eigene Liste zu legen.

Insert(x): Füge x an erste freie Stelle in $T[h(x)]$ ein.

Delete(x): Entferne x aus $T[h(x)]$. Bei $B \leq \frac{1}{2}$ kann nach m Delete halbiert werden.

Hashing mit offener Adressierung Idee: Linear Probing. Ausprobieren einer Reihe von Hashfunktionen h_i . Startpunkt ist $f(x)$, $g(x)$ verschiebt beim Probing. Eine Beispielfunktion wäre (mit $n \leq m$, damit $B \leq 1$):

$$h_i(x) = (x \bmod m + i) \bmod m$$

status[1, ..., m]: Status des Feldes (belegt, frei oder gelöscht)

Lookup(x): Probiere h_0, h_1, h_2, \dots bis freie Stelle oder x gefunden wurde:

Insert(x): Probiere h_0, h_1, h_2, \dots bis freie oder gelöschte Stelle gefunden wurde:

Delete(x): Probiere h_0, h_1, h_2, \dots bis freie Stelle oder x gefunden wurde. Entferne x und markiere status[Position(x)] als gelöscht.

Perfektes Hashing Ziel Speicherplatz $O(n)$ ohne Kollisionen. Idee: 2-Stufen-Hashing

1.Stufe Wähle eine Hashfunktion h_k so dass die Summe der Bucketgrößen in der Tafel T mit $s=n$ Elementen $< 3n$ ist, also:

$$(1) \sum_{i=0}^{n-1} |w_i^k|^2 < 3n$$

h_k muss in diesem Schritt noch nicht injektiv sein. Sei p eine Primzahl mit $p > N$, dann wähle zufällig Kandidaten k aus $\{1, \dots, p-1\}$, bis (1) erfüllt ist. Wir wissen, dass mindestens die Hälfte aller möglichen k geeignet sind.

\Rightarrow Wahrscheinlichkeit $\frac{1}{2}$ und Erwartungswert für Versuche, um k zu finden $= 2$ (Münzwurf).

$\Rightarrow O(2n)$ Tests, bis k gefunden wird.

2.Stufe Für nicht-leere Buckets jeweils eine Tafel s_i mit $2|w_i^k|^2$ Platz und Wahl von k_i so, dass h_{k_i} injektiv auf w_i^k . (Wieder Münzwurf). Platzbedarf:

$$(2) \sum_{i=0}^{n-1} 2|w_i^k|^2 = 2 \sum_{i=0}^{n-1} |w_i^k|^2 < 10n$$

\Rightarrow Gesamtplatzbedarf: $(1) + (2) \rightarrow O(13n)$

3 Amortisierte Analyse

Beschreiben Sie die Technik der amortisierten Analyse einer Folge von Operationen auf einer Datenstruktur D . Demonstrieren Sie diese Technik am Beispiel einer Folge von Increment-Operationen auf einem binären Zähler.

Lösung

Potentialmethode Idee: Bilde den Ablauf eines Algorithmus als Zustände und deren Übergänge ab.

Zustände D', D'', \dots in der Datenstruktur

$pot : D \rightarrow \mathbb{R}$, Methode zur Bestimmung des Potentialwertes eines Zustandes

$op : D \rightarrow D'$, Operation die einen Zustand in den nächsten überführt

$T_{Tats}(op)$, Tatsächliche Laufzeit einer Operation

$T_{Amort}(op) = T_{Tats}(op) + pot(D'') - pot(D') = T_{Tats}(op) + \Delta pot$, Amortisierte Laufzeit einer Operation

Beim **Binärzähler für die Increment Operation** stellt pot die Anzahl der Einsen k dar.

$$T_{Tats}(incr) = 1 + k$$

$\Delta pot = 1 - k$, da die Einsen zu 0 geflippt werden.

$$T_{Amort}(incr) = 1 + k + (1 - k) = 2$$

\Rightarrow Ein Increment kostet $O(1)$

4 Planare Graphen

Sei G ein planarer Graph mit n Knoten und m Kanten. Folgern Sie aus dem Satz von Euler, dass $m \leq 3n - 6$ und dass G einen Knoten vom Grad ≤ 5 besitzt.

Lösung

Eulerformel:

$$n - m + f = 2$$

Proof. $m \leq 3n - 6$

Ein maximaler planarer Graph hat eine Einbettung, in der jedes Face ein Dreieck ist (Triangulierung). Jede Kante liegt damit am Rande von 2 Faces und jedes Face hat 3 Kanten:

$$3f = 2m$$

Eingesetzt in die Eulerformel:

$$\begin{aligned} n - m + \frac{2}{3}m &= 2 \\ \Rightarrow m &= 3n - 6 \end{aligned}$$

Für allgemeine planare Graphen gilt daher:

$$m \leq 3n - 6$$

□

Proof. G hat einen Knoten v mit $\deg(v) \leq 5$

Annahme, dass $\forall v \in V : \deg(v) \geq 6$. Dann wäre

$$m = \sum_{v \in V} \frac{\deg(v)}{2} \geq \frac{6n}{2} = 3n$$

$\Rightarrow m \leq 3n - 6$ ist nicht mehr erfüllt

□

Zusatz: Zeigen Sie, dass für bipartite planare Graphen $m \leq 2n - 4$ gilt.

Lösung

Proof. $m \leq 2n - 4$ für bipartite planare Graphen.

Da bipartite Graphen keine ungeraden Zyklen haben, ist das kleinste Face ein Viereck:

$$2f = m$$

Eingesetzt in die Eulerformel:

$$\begin{aligned} n - m + \frac{1}{2}m &= 2 \\ \Rightarrow m &= 2n - 4 \end{aligned}$$

□

5 Split-Find

Das *Split – Find*-Problem ist wie folgt definiert: Verwalte eine Einteilung der Zahlen $\{1, \dots, n\}$ in disjunkte Intervalle, die am Anfang nur aus dem Intervall $[1, n]$ besteht, unter folgenden Operationen:

- *FIND*(i): liefert das Intervall, das die Zahl i enthält.
- *SPLIT*(i): ersetze das Intervall $[a, b]$ und $[i + 1, b]$

Entwickeln Sie eine Datenstruktur die jede FIND-Operation in Zeit $O(1)$ und jede Folge von SPLIT-Operationen möglichst effizient unterstützt.

Lösung

Idee: Relable the smaller Half für Split.

Datenstruktur

`name[x] = Name des Intervalls von x`

Algorithmus 7 : Initialisierung

```
foreach  $x \in N$  do
  | name[x]  $\leftarrow$  1;
end
```

Algorithmus 8 : Find(x)

```
return name[x];
```

Algorithmus 9 : Split(x)

```
Intervall I  $\leftarrow$  Find(x);
Laufe in I parallel nach links l und rechts r, bis name[x]  $\neq$  name[l] oder name[r];
Benenne kleineres Intervall  $I_{min}$  um;
I =  $[a, b] \rightarrow [a, x], [x+1, b]$ ;
```

Laufzeit

Find(x): $O(1)$

Split(x): $O(\# \text{Namensänderungen}) = O(\log n)$

6 Maximum Matching im bipartiten Graphen

Erklären Sie die Grundidee des Algorithmus zur Bestimmung eines Maximum Matchings in einem bipartiten Graphen.

Lösung

Idee: Berechne eine Folge von Matchings M_0, M_1, \dots, M_i mit

1. $M_0 = \emptyset$
2. $M_{i+1} = M_i \oplus P_i$. P_i ist ein erweiternder Pfad für M_i
3. $|P_i| \leq |P_{i+1}|$. (Länge der erweiternden Pfade ist monoton wachsend)
4. Alle verschiedenen P_i gleicher Länge sind knotendisjunkt. \Rightarrow Es existieren nur \sqrt{s} viele Paare solcher Pfade gleicher Länge.

Initialisierung Konstruiere \hat{G} mit 2 neuen Knoten s und t richte die Kanten von A nach B . Der Abstand (Level) eines Knotens ist die Länge des Pfades von s aus. Hat ein Knoten v $\text{level}[v] = -1$, so ist er nicht mehr von s aus erreichbar.

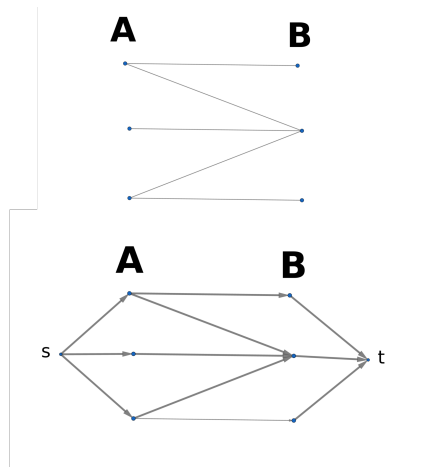


Figure 1: $G \rightarrow \hat{G}$

Innere Schleifen Ausführung:

Solange ein Pfad von s nach t existiert, also $\text{level}[t] \neq -1$

Bestimme maximale Menge S von kürzesten knotendisjunkten erweiternden Pfaden (Phase), sodass für alle Kanten (v, w) von Pfaden in S gilt: $\text{level}[v] = \text{level}[w] - 1$. Also jede Kante immer genau ein Level höher geht.

- * Für jeden Pfad in S streiche die erste und letzte Kante.
- * Drehe alle anderen Kanten um.

Laufzeit $O(\sqrt{nm})$. da $m > n$. \sqrt{n} Für die Anzahl der Phasen und $O(n + m)$ für Breiten und Tiefensuche zur Behandlung der Pfade.

7 Random Search Tree

Beschreiben Sie den Aufbau und die Operationen eines Random Search Trees und analysieren Sie diese.

Lösung

Aufbau Jeder der s Schlüssel aus S erhält einen zufälligen Prioritätswert $prio(x_i) = p_i$.
Initialisierung:

Algorithmus 10 : Initialisierung: RST(N)

$(x, p) \leftarrow \max(\text{prio})$ aus N ;
 $(x, p).left \leftarrow \text{RST}(\{(x_l, p_l) | x_l < x\})$;
 $(x, p).right \leftarrow \text{RST}(\{(x_r, p_r) | x_r > x\})$;

Der Baum ist sehr wahrscheinlich balanciert, da der Worst Case eintritt, wenn $\max \text{prio}$ in der Teilliste gleich $\max \text{Value}$ ist, was bei gleichverteilten Zufallszahlen unwahrscheinlich ist.

Operationen

Lookup(x): Suche nach Schlüssel im balancierten Baum $O(\log n)$

Insert(x): Insert (x, p) als Blatt nach der Schlüsselgröße und rotiere dann nach der Priorität, bis die Heapeigenschaft $prio(x) < prio(\text{parent}(x))$

Delete(x): Runterrotieren des Knotens, bis x ein Blatt ist. Dann entferne das Blatt. $O(1 + |L| + |R|)$ mit

R = linkes Rückgrat des rechten Teilbaums

L = rechtes Rückgrat des linken Teilbaums

Split(x): Insert($(x + \epsilon, \infty)$). Rotiere $(x + \epsilon, \infty)$ bis zur Wurzel, dann entferne die Wurzel.

Join(T_1, T_2): Erstelle T mit Wurzel x mit $\max(S_1) \leq x \leq \max(S_2)$. Dann Hänge T_1, T_2 an x und Delete(x).

\Rightarrow Mit Hilfe der Harmonischen Zahl des jeweiligen Elements lässt sich die Anzahl an nötigen Rotationen bei Insert und Delete als < 2 abschätzen.

8 Planare Einbettung

Geben Sie ein planaren Graphen an, der verschiedene planare Einbettungen besitzt. Geben Sie die entsprechenden Einbettungen jeweils durch Auflistung der Face-Zyklen an. Für welche Graphen ist die planare Einbettung eindeutig?

Lösungen

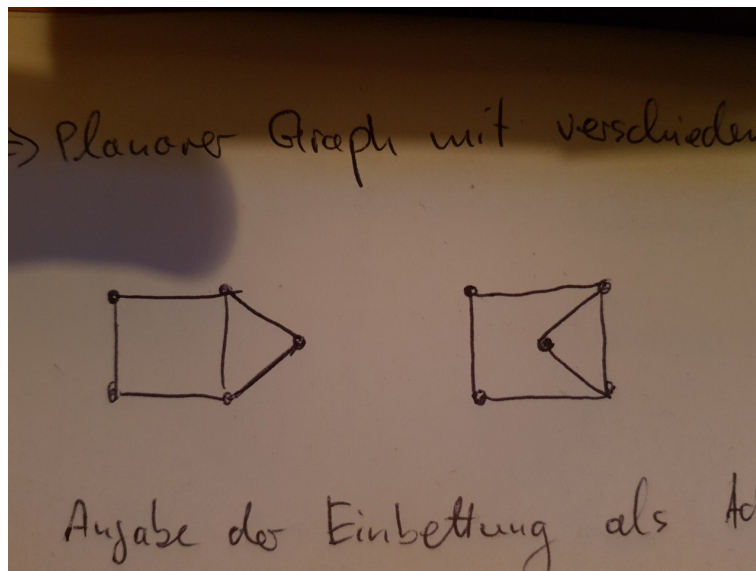


Figure 2: Planarer Graph mit verschiedenen Einbettungen

Ein Graph hat eine eindeutige planare Einbettung, wenn er 3-fach zusammenhängend ist.