Notizen zu Algorithmen II

Jens Ochsenmeier

16. Februar 2018

Inhaltsverzeichnis

1

Fortgeschrittene Datenstrukturen

Wir werden uns in diesem Kapitel mit Prioritätslisten beschäftigen. Es gibt noch viele weitere fortgeschrittene Datenstrukturen, z.B.

- monotone ganzzahlige Prioritätslisten (später im Kapitel "kürzeste Wege")
- perfektes Hashing
- Suchbäume mit fortgeschrittenen Operationen
- externe Prioritätslisten (später im Kapitel "Externe Algorithmen")
- Geometrische Datenstrukturen (siehe Kapitel "Geometrische Algorithmen")

1.1 Adressierbare Prioritätslisten

Eine **adressierbare Prioritätsliste** muss folgene Funktionen implementieren:

BUILD($\{e_1, \ldots, e_n\}$)	$M \coloneqq \{e_1, \ldots, e_n\}$
SIZE	**return** $\lvert M \rvert$
INSERT(e)	$M \coloneqq M \cup \{e\}$
MIN	**return** $\min M$
DELETEMIN	$e \coloneqq \min$; $M \coloneqq M \setminus \{e\}$; **return** e
REMOVE(h : Handle)	$e \coloneqq h$; $M \coloneqq M \setminus \{e\}$; **return** e
DECREASEKEY(h : Handle, k : Key)	key(h) $\coloneqq k$
MERGE(M')	$M \coloneqq M \cup M'$

Adressierbare Prioritätslisten haben viele Anwendungen, beispielsweise im *Dijkstra-Algorithmus* für kürzeste Wege oder in der Graphpartitionierung. Allgemein lassen sich adressierbare Prioritätslisten gut bei Greedy-Algorithmen verwenden, bei denen sich die Prioritäten (begrenzt) ändern.

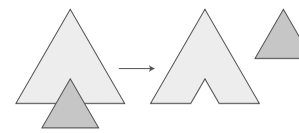
Datenstruktur

Als grundlegende Datenstruktur wird ein Wald heap-geordneter Bäume verwendet. Hier wird also der Binary Heap verallgemeinert

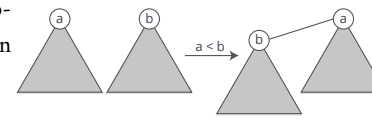
- Baum → Wald
- zwei Kindknoten → beliebig viele Kindknoten

Wir verwenden folgende grundlegende Operationen zur Bearbeitung solcher Wälder:

- **cut**: Teilbaum ausschneiden und als neuen Baum speichern



- **link**: Baum 2 mit größerem Wurzelknoten als Baum 1 an Baum 1 als Kindknoten des Wurzelknotens anhängen



- **union**: union(a, b) = link$(\min(a, b), \max(a, b))$

Dijkstras Algorithmus

Dijkstras Algorithmus kann die Distanz zwischen einem Startknoten s und jedem anderen Knoten des Graphen berechnen.

```
DIJKSTRA(s : Node, T : Tree)
// Initialisieren: Distanz zu jedem Knoten ist ∞, zu Startknoten 0.
d = ⟨∞, ..., ∞⟩
d[s] = 0
// Startknoten zu PQ hinzufügen.
Q.insert(s)
```

- $d = \langle \infty, \ldots, \infty \rangle$

  Zu Beginn ist die Distanz zu jedem Knoten ∞.

- parent$[s] = s, d(s) = 0$

  Der Startknoten wird initialisiert.

- Q.insert(s)

  Prioritätsliste wird mit Startknoten initialisiert.

1.2 Pairing Heaps

Pairing Heaps müssen folgende Funktionen implementieren:

```
INSERTITEM(h : Handle)
  newTree(h)
NEWTREE(h : Handle)
  forest := forest ∪ {h}
  if *h < min then minPtr := h
DECREASEKEY(h : Handle, k : Key)
  key(h) := k
  if h not a root then cut(h) else updateMinPtr(h)
DELETEMIN() : Handle
  m := minPtr
  forest := forest \ {m}
  foreach child h of m do newTree(h)
  pairwiseRootUnion()
  updateMinPtr()
  return m
PAIRWISEROOTUNION()
  // see picture
MERGE(o : AdressablePQ)
  if *minPtr > *(o.minPtr) then minPtr := o.minPtr
  forest := forest ∪ o.forest
  o.forest := ∅
```

Einige Funktionalitäten lassen sich gut veranschaulichen:

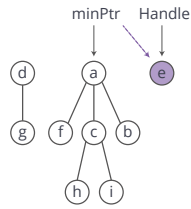


Abbildung 1.1. `newTree`: Element e wird hinzugefügt (ggf. auch ein ganzer Baum) und — falls nötig — der `minPtr` angepasst.

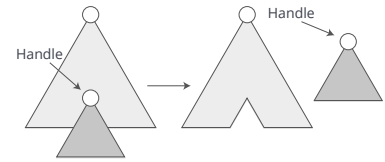


Abbildung 1.2. `decreaseKey`: Durch Herabsetzen des Keys wird eventuell die Heap-Eigenschaft verletzt, deswegen wird der Teilbaum ausgeschnitten und als neuer Baum hinzugefügt.

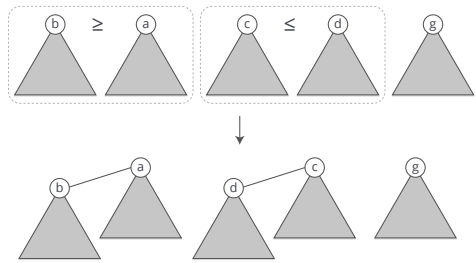


Abbildung 1.3. `pairwiseRootUnion` fügt jeweils zwei Bäume des Waldes zu einem größeren Baum zusammen, indem die beiden Wurzeln miteinander verglichen werden und eine der beiden Wurzeln Kindknoten der anderen wird.

Repräsentation

Meistens speichert man Pairing Heaps als doppelt verkettete Liste der Wurzeln. Die Baum-Items können beispielsweise folgendermaßen gespeichert werden:

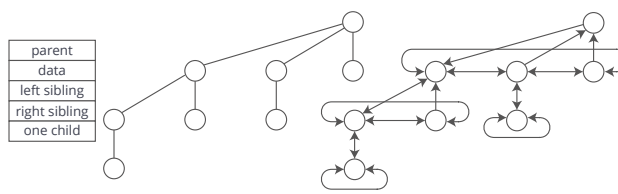


Abbildung 1.4. Speichern der Baum-Items. Man kann hier noch Speicherplatz einsparen, indem man *left sibling* und *parent* zusammenfasst. Allerdings muss man dann alle Geschwisterknoten traversieren, damit man zum Elternknoten kommen kann, da nur der linkeste Kindknoten den Elternknoten speichert.

Analyse

- `insert` und `deleteMin` gehen in $O(1)$.
- `deleteMin` und `remove` gehen jeweils in $O(\log n)$ amortisiert.
- `decreaseKey` ist schwieriger zu analysieren, geht aber amortisiert in $O(\log \log n) \leq T \leq O(\log n)$ und ist in der Praxis sehr schnell.

Wir werden als nächstes *Fibonacci-Heaps* verwenden, um noch mehr Leistung rauszukitzeln.

1.3 Fibonacci-Heaps

Mithilfe von Fibonacci-Heaps erhalten wir eine amortisierte Komplexität von $O(\log n)$ für `deleteMin` und $O(1)$ für alle anderen Operationen.
Fibonacci-Heaps speichern ein paar Zusatzinformationen pro Knoten ab, wodurch neue Hilfsfunktionen kreiert werden könnnen, die diese Beschleunigung ermöglichen:

- **Rank** eines Knotens: Anzahl direkter Kinder
- **Mark**: Knoten, die ein Kind verloren haben, werden markiert
- **Vereinigung nach Rank**: Union nur für gleichrangige Wurzeln
- **Kaskadierende Schnitte**: Knoten, die beide markiert sind (also ein Kind verloren haben), werden geschnitten

Repräsentation

Die Repräsentation ist analog zu der von Pairing Heaps, die Wurzeln werden wieder als doppelt verkettete Liste gespeichert und die Baum-Items als Parameterliste:

parent
data, *rank*, *mark*
left sibling
right sibling
one child

Funktionalität

`insert` und `merge` werden wie gehabt implementiert. `decreaseKey` verwendet die neue `cascadingCut`-Methode und `deleteMin` Union-by-Rank. Wir beschleunigen Union-by-Rank, indem wir ein Feld pro Rank bereitstellen und Knoten in diese

Felder eintragen. Sollte das passende Feld bereits belegt sein, so wird der neue Knoten mit dem Knoten, der sich im Feld befindet gelinkt und entsprechend verschoben.

```
DELETEMIN() : Handle
  m := minPtr
  forest := forest \ {m}
  foreach child h of m do newTree(h)
  while ∃ a, b ∈ forest : rank(a) ≡ rank(b) do
    union(a, b)
  updateMinPtr()
  return m
DECREASEKEY(h : Handle, k : Key)
  key(h) := k
  cascadingCut(h)
CASCADINGCUTh : Handle
  assert h is not a root
  p := parent(h)
  unmark(h)
  cut(h)
  if p is marked then
    cascadingCut(p)
  else mark(p)
```

Eine amortisierte Analyse von `deleteMin` ergibt $\Omega(\log n)$ für vergleichsbasiertes `deleteMin`.

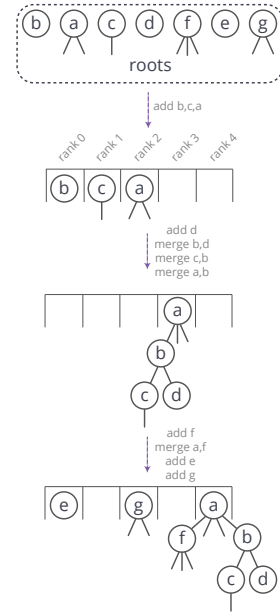


Abbildung 1.5. Fast Union-by-Rank.

2

Kürzeste Wege

Wir betrachten einen Graph $G = (V, E)$ mit Kantengewicht $c : E \to \mathbb{R}$ und Anfangsknoten $s \in V$.
Gesucht ist die Länge $\mu(v)$ des **kürzesten Pfades** von s nach v für alle $v \in V$, wobei

$$\mu(v) := \min \{c(p) : p \text{ ist Pfad von } s \text{ nach } v\}$$

und

$$c(\langle e_1, \ldots, e_k \rangle) := \sum_{i=1}^{k} c(e_i).$$

Oft suchen wir auch eine "geeignete" Repräsentation des kürzesten Pfades.

2.1 Allgemeine Definitionen

Wir benutzen im Allgemeinen zwei Knotenarrays:

- $d[v]$ = aktuelle (= vorläufige) Distanz von s nach v.
  Invariante: $d[v] \geq \mu(v)$.
- parent$[v]$ = Vorgänger von v auf (vorläufigem) kürzesten Pfad von s nach v.
  Invariante: Dieser Pfad bezeugt $d[v]$.

Initial ist

- $d[s] = 0$, parent$[s] = s$ und
- $d[v] = \infty$, parent$[v] = \perp$.

Kern ist das *Relaxieren* der Kanten $(u, v) \in E$:

```
if d[u] + c(u,v) < d[v] then  // z.B. wenn d[v] ≡ ∞
  d[v] := d[u] + c(u,v)
  parent[v] := u
```

Die oben genannten Invarianten werden dadurch nicht verletzt. $d[v]$ kann sich also problemlos mehrmals ändern.

2.2 Dijkstras Algorithmus

Dijkstras Algorithmus ist der wohl einfachste Algorithmus, um dieses Problem zu lösen. In Pseudocode:

```
// d und parent initialisieren
// alle Knoten als ungescannt setzen
while ∃ non-scanned node u with d[u] < ∞ do
  u := non-scanned node v with minimal d[v]
  relax all edges (u,v) out of u
  set u scanned
```

Ist $v \in V$ von s aus erreichbar, so wird v irgendwann gescannt. Wird v gescannt, so ist $\mu(v) = d[v]$.
Am Ende definiert d die optimalen Entfernungen und `parent` die zugehörigen Wege.
Dieser Algorithmus wurde bereits in Algorithmen I ausführlich diskutiert.

Analyse

Es ist

$$T_{\text{Dijkstra}} = O(m \cdot T_{\text{decreaseKey}}(n) + n \cdot (T_{\text{deleteMin}}(n) + T_{\text{insert}}(n))).$$

Nutzen wir also Fibonacci-Heaps, so kriegen wir

$$T_{\text{DijkstraFib}} = O(m + n \log n).$$

2.3 Monotone ganzzahlige Prioritätslisten

Wir beobachten, dass Dijkstras Algorithmus die Prioritätsliste *monoton* benutzt — `insert` und `decreaseKey` benutzen nämlich Distanzen der Form $d[u] + c(e)$. Die Werte nehmen also ständig zu.

Sind alle Kantengewichte $\in [0, C]$, so gilt

$$\forall v \in V : d[v] \leq (n-1)C.$$

Ist insbesondere min der letzte Wert, der aus Q entfernt wurde, so sind in Q *immer* nur Knoten mit Distanzen im Intervall $[\min, \min + C]$.

Bucket-Queue

Eine **Bucket-Queue** ist ein zyklisches Array B von $C + 1$ doppelt verketteten Listen. Knoten der Distanz $d[v]$ werden in $B[d[v] \mod (C + 1)]$ gespeichert.

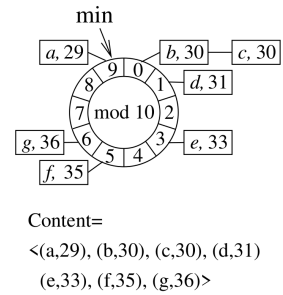


Abbildung 2.1. Bucket-Queue mit $C = 9$.

Folgende Operationen werden auf einer Bucket Queue implementiert:

- **Initialisierung**: $C + 1$ leere Listen werden angelegt, $\min = 0$.
- **`insert(v)`**: fügt v in $B[d(v) \mod (C + 1)]$ ein
  $\Rightarrow O(1)$
- **`decreaseKey(v)`**: schiebt v von seiner Liste nach $B[d(v) \mod (C + 1)]$
  $\Rightarrow O(1)$
- **`deleteMin`**: fängt bei $B[\min \bmod (C + 1)]$ an; falls leer, min := min + 1, ↻
  $\Rightarrow O(nC)$

Mit Bucket-Queues kriegen wir also den Dijkstra-Algorithmus auf $O(m+\text{maxPathLength})$ gedrückt.

Radix-Heaps

Radix-Heaps sind eine Variante von Bucket Queues, die Buckets von -1 bis K für $K = 1 + \lfloor \log C \rfloor$ benutzt. Wie vorhin schon betrachtet sei $\min$ die zuletzt aus Q entfernte Distanz und somit

$$\forall v \in Q : d[v] \in [\min, \ldots, \min + C].$$

Wir betrachten die *binäre Repräsentation* der möglichen Distanzen in Q. Wir speichern v in Bucket $B[i]$, falls sich $d[v]$ und min zuerst an der i-ten Stelle unterscheiden (Ausnahmen: $B[K]$ falls $i > K$, $B[-1]$ falls sie sich nicht unterscheiden).
Das definiert die **Most Significant Digit** (kurz MSD) — das ist die Position der höchstwertigen Ziffer in der Binärdarstellung von a und b, an der sich die beiden unterscheiden. $\text{msd}(a, b)$ kann mit Maschinenbefehlen sehr schnell berechnet werden.
Wir nutzen folgende *Radix-Heap-Invariante*:

$$v \text{ ist gespeichert in Bucket } B[i], \text{ wo } i = \min\{\text{msd}(\min, d[v]), K\}$$

Wir können nun `deleteMin` folgendermaßen implementieren:

```
DELETEMIN(): Element
if B[−1] = ∅ then
  i := min{j ∈ 0...K : B[j] ≠ ∅}
  move min B[i] to B[−1] and to min
  foreach e ∈ B[i] do   // exactly here the invariant is violated!
    move e to B[min{msd(min, d[v]), K}]
return B[−1].popFront()
```

Die `deleteMin`-Laufzeit ist $O(K)$, insgesamt lässt sich also die Laufzeit des Dijkstra-Algorithmus hierdurch auf $O(m + n \log C)$ drücken.

2.4 All-Pairs Shortest Paths

Herausforderung in diesem Abschnitt ist es, nicht die Abstände zu einem festgelegten Startknoten zu berechnen, sondern zwischen allen Knotenpaaren $(u, v) \in V^2$ für $G = (V, E)$. Zusätzlich erlauben wir negative Kantenkosten, allerdings keine negativen Kreise.
Wir werden zwei verschiedene Lösungen erhalten:

1. n mal den *Bellman-Ford-Algorithmus* ausführen
   $\Rightarrow O(n^2 m)$
2. *Knotenpotentiale* verwenden
   $\Rightarrow O(nm + n^2 \log n)$

Bellman-Ford-Algorithmus

Der **Bellman-Ford-Algorithmus** wurde bereits in Algorithmen I behandelt, deswegen hier nur eine kurze Wiederholung. Wie Dijkstras Algorithmus findet er den kürzesten Pfad zwischen einem festgelegten Startknoten und allen anderen Knoten des Graphen, unterstützt aber auch negative Kantengewichte.

1. Distanzen initialisieren: $d[s] = 0$, $d[v] = \infty$ für alle anderen Knoten
2. Von s ausgehend alle Knoten des Graphen durchgehen und pro Knoten v die ausgehenden Kanten betrachten.
   - Eintragen, ob v von s in $< \infty$ erreicht werden kann.
   - Ist $d[v] + c(v,u) < d[u]$ für einen Nachbar von v? Wenn ja, $d[u]$ aktualisieren.

Der zweite Schritt wird maximal $|V| - 1$ mal wiederholt. Sollte sich schon davor bei einem Durchgang nichts mehr ändern, so kann man aufhören.
Die Laufzeit des Bellman-Ford-Algorithmus ist $O(nm)$, nutzt man ihn als Basis für All-Pairs Shortest Paths kriegt man also $O(n \cdot nm) = O(n^2 m)$.

Knotenpotentiale

Jeder Knoten erhält ein Potential $\text{pot}(v)$. Mit diesen Knotenpotentialen lassen sich die **reduzierten Kosten** $\overline{c}(e)$ für eine Kante $e = (u,v) \in E$ als

$$\overline{c}(e) = \text{pot}(u) + c(e) - \text{pot}(v)$$

definieren.
Ist p ein Pfad von u nach v mit Kosten $c(p)$, dann ist

$$\overline{c}(p) = \text{pot}(u) + c(p) - \text{pot}(v).$$

Ist p' ein anderer u-v-Pfad, dann gilt

$$c(p) \leq c(p') \iff \overline{c}(p) \leq \overline{c}(p').$$

Wir berechnen die gewünschten Informationen nun so:

1. Wir fügen einen *Hilfsknoten* s zu G hinzu.
2. Wir fügen (s,v) für alle $v \in V \setminus \{s\}$ mit Kosten 0 hinzu.
3. Berechne die kürzesten Pfade von s aus mit Bellman-Ford.
4. Definiere $\text{pot}(v) := \mu(v)$ für alle $v \in V$.
   Die reduzierten Kosten sind jetzt alle nicht-negativ, also können wir Dijkstra benutzen und ggf. s wieder entfernen.
5. Für eine beliebige Kante $(u,v) \in E$ gilt

$$\mu(u) + c(e) \geq \mu(v) \text{ und deshalb } \overline{c}(e) = \mu(u) + c(e) - \mu(v) \geq 0.$$

```
neuen Knoten s und alle Kanten s,v hinzufügen   // O(n)
pot := μ := BellmanFordSSSP(s, c)   // O(nm)
foreach x ∈ V do
  μ̄(x, ·) := DijkstraSSSP(x, c̄)

// zurück zur ursprünglichen Kostenfunktion
foreach e = (v, w) ∈ V² do   // O(n²)
  μ(v, w) := μ̄(v, w) + pot(w) − pot(v)
```

Die Gesamtlaufzeit beträgt also $O(nm + n^2 \log n)$.

2.5 Distanz zu Zielknoten

Wir haben bisher zwei Fälle diskutiert:

1. Abstände aller Knoten zu einem Startknoten s
2. Abstände zwischen allen Knotenpaaren $\{u, v\} \in V^2$

Als nächstes schauen wir uns an, wie man den kürzesten Pfad zwischen einem Startknoten s und einem Zielknoten t ermittelt.

Trick 0 — Dijkstra abbrechen

Am einfachsten ist es, einfach Dijkstra abzubrechen, wenn t aus Q entfernt wird. Das spart "im Schnitt" die Hälfte des Scans.

Bidirektionale Suche

Idee ist hier, abwechselnd von s und t aus zu suchen. Von s aus sucht man auf $G = (V, E)$, von t aus auf dem zugehörigen *Rückwärtsgraphen* $G^r = (V, E^r)$.
Die vorläufige kürzeste Distanz wird in jedem Schritt gespeichert:

$$d[s,t] = \min\{d[s,t], d_{\text{forward}}[u] + d_{\text{backward}}[u]\}$$

Abgebrochen wird, wenn die Suche einen Knoten scannt, der in die andere Richtung bereits gescannt wurde.

A^*-Suche

Idee der A^*-Suche ist es, "in die Richtung" des Ziels zu suchen. Dazu benötigen wir eine Funktion $f(v)$, die für alle $v \in V$ die eigentliche Funktion $\mu(v, t)$ schätzen kann.

Anschließend können wir $\text{pot}(v) = f(v)$ setzen und $\overline{c}(u,v) = c(u,v) + f(v) - f(u)$.
$f(v)$ muss diese Eigenschaften haben:

- **Konsistenz**: $c(e) + f(v) \geq f(u)\ (\forall e = (u,v))$.
  Die reduzierten Kosten dürfen also nicht negativ sein.
- $f(v) \leq \mu(v,t)\ (\forall v \in V)$.
  Dann ist $f(t) = 0$ und wir können aufhören wenn t aus Q entfernt wird.

Ist p ein beliebiger Pfad von s nach t, so ist $d[t] \leq c(p)$ (alle Kanten auf p seien relaxiert). Wie finden wir jetzt aber so eine Funktion $f(v)$?
Wir benötigen eine Heuristik für $f(v)$.

- Betrachtet man eine Strecke im Straßennetzwerk, so kann $f(v)$ beispielsweise der euklidische Abstand $\|v - t\|_2$ sein. Damit erhält man eine deutliche, aber keine überragende Beschleunigung.
- **Landmarks** sind deutlich geeigneter, benötigen allerdings Vorberechnung.
  Man wähle eine *Landmarkmenge* L. Berechne und speichere $\mu(v,l)$ für alle $l \in L, v \in V$.
  Während einer Query suche man jetzt ein Landmark $l \in L$ "hinter" dem Ziel und benutze die untere Schranke

  $$f_l(v) = \mu(v,l) - \mu(t,l)$$

  Vorteile sind, dass Landmarks konzeptuell einfach sind, eine erhebliche Beschleunigung bringen (um den Faktor 20) und mit anderen Techniken kombinierbar sind.
  Allerdings ist die Landmarkauswahl schwierig und der Platzverbrauch sehr groß (besonders für große V).