Quickselect

Die Idee von Quickselect ist übernommen von Quicksort.
Man wählt ein zufälliges Pivot-Element und teilt das Feld in zwei Teile $a[1,\ldots,m]$ und $a[m+1,\ldots,n]$ auf.
Damit geht es rekursiv genau so weiter, wie eben bei der Medianberechnung in Linearzeit.

Im worst case kann es hier natürlich ebenso wie bei Quicksort passieren, dass man immer das kleinste oder das größte Element als Pivot bekommt – dann ist die Laufzeit auch hier quadratisch!
Wie sieht es im average case aus?

$Q(n)$ sei die Anzahl der Vergleiche, die man im Schnitt (bzgl. Gleichverteilung) bei der Durchführung des oben definierten Verfahrens machen muss. Dann gilt

$$Q(n) \leq 4n \quad \text{(Beweis an der Tafel)}$$

Zum Vergleich: Das vorher vorgestellte Linearzeitverfahren braucht $16n$ Vergleiche...

Der Dijkstra-Algorithmus

Wir wollen uns noch einmal dem Dijkstra-Algorithmus zur Lösung des single-source-all-targets shortest-paths Problems zuwenden. (Vergleiche Folien 5.5 und 5.6)

Wir geben zunächst den Algorithmus relativ detailliert an. Dabei sei $G = (V, E)$ ein gerichteter Graph ohne Schlingen oder Mehrfachkanten, und es sei eine Kantengewichtung gegeben durch $\gamma : E \to \mathbb{N} \setminus \{0\}$. $D[1, \ldots, n]$ mit $n = |V|$ sei ein Array, in dem wir anfangs Schätzwerte und später die genauen Abstandswerte der Knoten $x = x_i$ von u (dem gegebenen Startknoten) abspeichern wollen. Schließlich sei $v[1, \ldots, n]$ ein Array, in dem wir abspeichern wollen, von welchem Knoten $y = v[i]$ wir beim kürzesten Weg zum Knoten x_i gehen.

Der Algorithmus

```
B := {u}; R := ∅; v(u) := undef; D(u) := 0;    (* Init B, R und U *)
FORALL y ∈ V \ {u} mit (u,y) ∈ E DO
        D(y) := γ(u,y); v(y) := u; R := R ∪ {y};
ENDFOR;
U := V \ (R ∪ {u});
WHILE R ≠ ∅ DO
        x := undef; α := ∞;                (* suche x mit D(x) minimal *)
        FORALL y ∈ R DO
                IF D(y) < α THEN x := y; α := D(y) ENDIF
        ENDFOR
        B := B ∪ {x}; R := R \ {x};                (* x von R nach B *)
        FORALL (x,y) ∈ E DO
                IF y ∈ U THEN                    (* Rand aktualisieren *)
                        D(y) := D(x) + γ(x,y);
                        v(y) := x;  U := U \ {y};  R := R ∪ {y};
                ELSIF y ∈ R AND D(x) + γ(x,y) < D(y) THEN
                        D(y) := D(x) + γ(x,y);  v(y) := x;   (* kürzerer Weg *)
                                                             (*    über x    *)
                ENDIF
        ENDFOR
ENDWHILE
```

Terminierung

Um nachzuweisen, dass dieser Algorithmus tut, was wir uns von ihm erwarten, müssen wir nun zweierlei tun:

1. Zeigen, dass der Algorithmus immer terminiert.
2. Zeigen, dass die berechneten Werte immer korrekt sind.

Zunächst zum Terminieren:

Wir betrachten die Menge B, die immer eine Teilmenge von V ist, d.h. sie kann maximal n Elemente haben. Zu Beginn hat B ein Element. In jedem Durchgang der WHILE-Schleife kommt ein weiteres dazu, das aus R entfernt wird. Alle Elemente in R kommen aber aus U, wo wiederum jeder Knoten, der nach R geschoben wird, entfernt wird. Damit kann jeder Knoten höchstens einmal zu B hinzugefügt werden, d.h. maximal gibt es $n - 1$ Schleifendurchgänge.

Korrektheit der Werte

Wie können wir einsehen, dass alle Werte, die im Dijkstra-Algorithmus berechnet werden, korrekt sind? Dazu formulieren und beweisen wir eine sogenannte Schleifeninvariante:

> Nach dem k-ten Schleifendurchlauf ($k = 0, 1, \ldots$) gilt:
> Für jeden Knoten $x \in B$ ist ein kürzester Weg von u nach x und seine Länge bekannt, und es gilt für alle $y \in B$ und alle $z \notin B$, dass der kürzeste Weg von u nach y höchstens so lang ist wie der kürzeste Weg von u nach z.

Für $k = 0$ ist die Invariante offensichtlich korrekt.
Nun müssen wir aus der Annahme, dass sie für k stimmt, schließen, dass sie auch für $k + 1$ korrekt ist.

Korrektheitsbeweis (Forts.)

Zunächst ist klar, dass für jeden Knoten $b \in U$ ein Knoten in R existiert, der einen kürzeren Abstand von u hat als b.

Das heißt, wenn der zu B hinzugenommene Knoten unter allen Knoten in R derjenige ist, der den kleinsten Abstand von u hat, dann ist die Invariante auch nach Hinzunahme dieses Knotens zu B erfüllt.

Nehmen wir also das Gegenteil an: Es gibt einen Knoten $a \in R$, dessen Abstand zu u kleiner ist als der des gewählten Knotens x. Wie sieht der zugehörige Weg aus?

Dieser Weg muss irgendwann einmal aus B herausführen. Dann haben wir aber einen Teilweg (auf dem Weg zu a), der schon länger ist als der gesamte Weg von u zu x.

Das ist (da alle Gewichte positiv sind) ein Widerspruch.

Dijkstra: Analyse der Laufzeit

Die Initialisierung beim Dijkstra-Algorithmus (d.h. alles, was vor der WHILE-Schleife berechnet wird) benötigt bei einem Eingabe-Graph mit n Knoten offenbar nur $O(n)$ viele Rechenschritte.

Die WHILE-Schleife wird, wie schon beobachtet, höchstens $n-1$ mal durchlaufen.

Aber wie hoch ist der Aufwand je Schleifendurchlauf?

Das erste FORALL wird maximal $O(n)$ Fälle haben.

Für das zweite FORALL ist zu bemerken, dass jede Kante aus E hier nur einmal – und nicht in jedem Durchlauf einmal – zu behandeln ist.

Damit ergibt sich als Gesamtaufwand das Maximum von n^2 und $m = |E|$, aber da $m < n^2$ immer gilt, erhalten wir $O(n^2)$.

Dichte und dünne Graphen

Wenn ein Graph mit n Knoten $\Omega(n^2)$ Kanten besitzt, ist er an der oberen Grenze der möglichen Kantendichte.

Einen solchen Graph nennen wir *dicht* und stellen fest, dass für dichte Graphen der Dijkstra-Algorithmus in der gegebenen Form lineare Laufzeit (in der Eingabelänge) hat und daher kaum zu verbessern sein dürfte.

Aber wie sieht es bei dünnen Graphen aus?

Man beachte, dass z.B. planare Graphen immer höchstens linear viele Kanten haben, d.h. sie sind im naheliegenden Sinn *dünne* Graphen!

Dann ist der Dijkstra-Algorithmus in der eben vorgestellten Form quadratisch (in der Eingabelänge), und das ist NICHT optimal.

Die Rolle der Datenstruktur

Die vorgestellte Implementierung ist von Arrays als Datenstruktur für unsere Knotenmengen – insbesondere für die Randmenge R – ausgegangen.

Als Alternative wollen wir uns überlegen, wie sich der Aufwand bei Verwendung eines *abstrakten Datentyps* für die Menge R darstellen lässt.

Wir lassen also die konkrete Implementierung der Menge R offen.

Es gibt drei Stellen, an denen im Algorithmus auf diese Menge zurückgegriffen werden muss: Wir fügen Knoten in die Menge ein, wenn von einem (neuen) Baumknoten bisher unbetrachtete Nachbarn zum Rand hinzukommen. Wir entfernen den Knoten mit minimalem D-Wert. Und schließlich verändern wir den D-Wert einiger Elemente von R.

Die Schnittstelle zum abstrakten Datentyp

Wir definieren drei Routinen:

`insert`$(R, y, D(y))$

x := `delete-min`(R)

`decrease-key`$(R, y, D(y))$

Die Routine `insert` wird (höchstens) n mal ausgeführt. Dasselbe gilt für `delete-min`. Wie oft wird `decrease-key` ausgeführt? Antwort: m mal (m = Anzahl der Kanten im Input-Graph).

Bei Verwendung eines Arrays als Datenstruktur ist der Aufwand für die drei Routinen in $O(1)$ für `insert` und `decrease-key`, aber $O(n)$ für `delete-min`.

Der Gesamtaufwand ist mithin $O(n^2)$.

Analyse: Dijkstra mit Heap als Datenstruktur

Ganz anders verhält es sich bei der Verwendung eines Heaps, wie wir ihn vom Heapsort her kennen. Damit kann man alle drei Routinen mit Aufwand $O(\log n)$ implementieren und erhält als Gesamtaufwand $O(n \log n)$ für alle insert- und delete-min-Operationen. Als Gesamtaufwand der decrease-key-Operationen erhalten wir jetzt aber $m \log n$, und damit bei dichten Graphen (genau genommen für $m > n^2/\log n$) eine längere Laufzeit als für die Array-Implementierung. Für dünne Graphen dagegen, also insbesondere für planare Graphen (und andere mit $m \in O(n)$), ist die Heap-Implementierung bei einer Gesamtkomplexität in $O(n \log n)$ eindeutig vorzuziehen.

Eine Anmerkung zum Schluss: Bei Verwendung eines sogenannten *Fibonacci-Heap* als Datenstruktur wird automatisch die jeweils bessere Komplexität erreicht, egal ob der Eingabegraph dicht oder dünn ist.