Informatik II: Algorithmen und Datenstrukturen SS 2017

Vorlesung 10b, Mittwoch, 5. Juli 2017
(Dijkstras Algorithmus)

Prof. Dr. Hannah Bast
Lehrstuhl für Algorithmen und Datenstrukturen
Institut für Informatik
Universität Freiburg

UNI FREIBURG

Blick über die Vorlesung heute

- Drumherum
  - Evolution Homo Sapiens — die entscheidende Mutation?
- Inhalt
  - Dijkstras Algorithmus — Algorithmus + Beispiel
  - Korrektheitsbeweis — endlich wieder Mathe :-)
  - Laufzeit + Implementierung — Analyse + Tipps
  - ÜB10: Implementieren Sie Dijkstras Algorithmus zur (einfachen) Routenplanung auf Baden-Württemberg

Evolution zum Homo Sapiens 1/5

- Entscheidende Mutation? Ihre Kommentare
  - Die Entstehung von Mehrzellern vor 2-3 Milliarden Jahren
  - Aufrechter Gang + Hände frei, vor 5-7 Millionen Jahren
  - Zunahme des Gehirnvolumen durch erhöhten Fleischkonsum
  - Entdeckungsdrang, im Gegensatz zu den Neanderthalern
  - Entwicklung zu sprechendem sozialen Wesen / von Kultur
  - "Die Erfindung von TrapRap bei Lil Waynes Geburt"
  - "Da ich Informatiker bin: die Erfindung der Pornografie"
  - "Der Moment als die Frauen kompliziert wurden und die Männer sie nicht mehr verstanden"

Evolution zum Homo Sapiens 2/5

- Ein paar wichtige Stationen
  - Sauerstoffkatastrophe ~ 2.4 Milliarden Jahre
  - Erste Wirbeltiere ~ 525 Millionen Jahre
  - Übergang Wasser → Land ~ 400 Millionen Jahre
  - Perm-Trias-Massensterben ~ 252 Millionen Jahre
  - Erste Säugetiere ~ 225 Millionen Jahre
  - Dinosaurier futsch ~ 65 Millionen Jahre
  - Erste Primaten ~ 50 Millionen Jahre
  - Aufrechter Gang ~ 4 Millionen Jahre
  - Homo Sapiens ~ 200 Tausend Jahre

Evolution zum Homo Sapiens 3/5

- Analogie: ein Menschenleben
  - Schauen Sie sich an, wie Sie heute sind und aussehen
  - Wann war der entscheidende Moment in Ihrem Leben der diesen Zustand hervorgebracht hat?

Geburt	0 Jahre
Laufen	~ 1.5 Jahre
Sprechen	~ 2 Jahre
Ich-Bewusstein	~ 3 Jahre
Erstes Smartphone	?
Einsetzen der Pubertät	12 – 50 Jahre

Evolution zum Homo Sapiens 4/5

- Menschenleben Zeitraffer
  - Es gibt zwar Zeiten, in denen in relativ kurzer Zeit relativ viel passiert
  - Aber insgesamt ist es ein **fließender Übergang**

    Es ändert sich in jedem Augenblick und diese kontinuierliche Änderung von Augenblick zu Augenblick kann über einen längeren Zeitraum beliebig viel verändern

    https://www.youtube.com/watch?v=iVEiAU_F2qw

Evolution zum Homo Sapiens 5/5

- Übergangsformen ("Transitional Forms")
  - In der Evolution ist es tatsächlich ganz genauso
  - Wenn man sich das ganze in Zeitraffer anschauen würde, sähe man eine kontinuierliche Veränderung, wie beim Altern
  - Schlagender Beweis dafür sind Fossilien von Übergangsformen zwischen Lebewesen und ihren ganz andersartigen Ahnen, z.B.

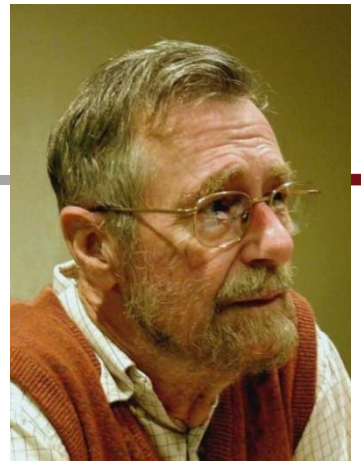
    Pakicetus (wolfsähnlich) → Wal

    Reptilkiefer + Ohr → Säugetierkiefer + Ohr

    Dinosaurier → Vögel

    Gorilla → aufrechter Gang

Dijkstras Algorithmus 1/4



- Ursprung
  - Benannt nach **Edsger Dijkstra** (1930 – 2002)

    Niederländischer Informatiker, einer der wenigen Europäer, die den Turing-Award gewonnen haben

    (für seine Arbeiten zur strukturierten Programmierung)
  - Der Algorithmus ist aus dem Jahr **1959**

Dijkstras Algorithmus 2/4

- Grundidee und Terminologie
  - Sei s der Startknoten und sei dist(s, u) die Länge des kürzesten Pfades von s nach u, für alle Knoten u
  - Besuche die Knoten in der Reihenfolge der dist(s, u)
  - Für jeden Knoten wird während der Ausführung eine vorläufige Distanz dist[u] gespeichert, zu Beginn ∞
  - Es gibt dann drei Arten von Knoten

    unerreicht: dist[u] = ∞

    aktiv: dist[u] ≥ dist(s, u) aber nicht ∞

    gelöst: siehe nächste Folie

    Auf Englisch: unreached, active, settled

Dijkstras Algorithmus 3/4

im Code braucht man NICHT d.ä.

- Algorithmus
  - Zu Beginn nur s aktiv, mit dist[s] = 0 und dist[v] = ∞
  - In jeder Runde holen wir uns den aktiven Knoten u mit dem kleinsten Wert für dist[u]
  - Den Knoten u betrachten wir dann als **gelöst**
  - Für alle (u, v) ∈ E: prüfe ob dist[u] + cost(u, v) < dist[v] und falls ja, setze dist[v] = dist[u] + cost(u, v)

    Das nennt man **Relaxieren** von (u, v)
  - Wiederhole, bis es keine aktiven Knoten mehr gibt

    Alle gelösten Knoten kennen dann ihre Entfernung von s

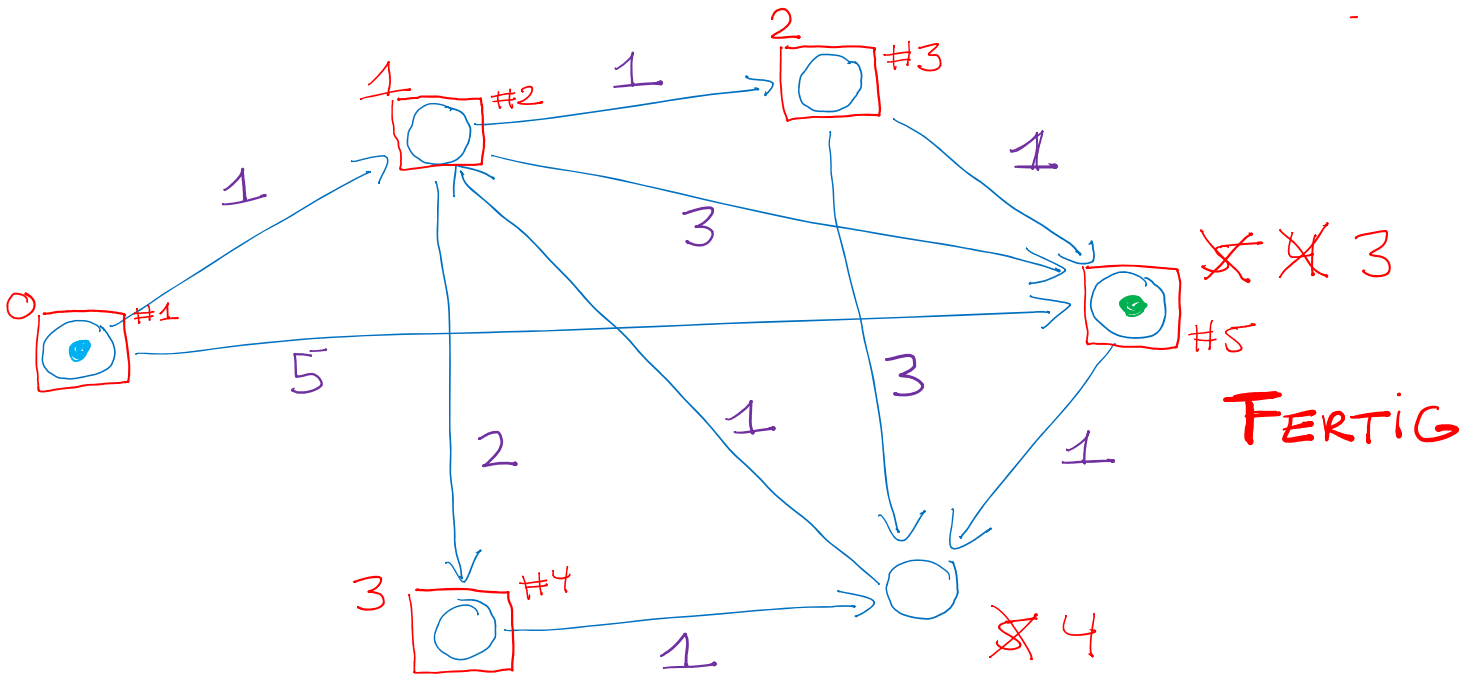
    Falls alle Kantenkosten 1 sind, ist das **genau** BFS

Dijkstras Algorithmus 4/4

Kantenkosten in LILA
START
ZIEL
∞ schreiben nur nichts dran

□ #i gelöst in Schritt i

- Beispiel



#4 wäre auch für den Zielknoten gegangen (auch Kosten 3 → freie Wahl)

Korrektheitsbeweis 1/6

ein BSP auf Folie
wäre nicht der Fall

- Annahmen
  - **Annahme 1:** Alle Kantenlängen sind > 0
  - **Annahme 2:** Die dist(s, u) sind alle **verschieden**

    Es gibt dann eine Anordnung $u_1, u_2, u_3, \ldots$ der Knoten

    so dass gilt $dist(s, u_1) < dist(s, u_2) < dist(s, u_3) < \ldots$
  - Es geht auch mit Kantenlängen ≥ 0 und ohne Annahme 2

    Beweis dazu siehe Referenzen (Mehlhorn/Sanders)

    Mit den Annahmen ist der Beweis einfacher und intuitiver und enthält trotzdem alles Wesentliche

Korrektheitsbeweis 2/6

- Argumentationslinie
  - Wir wollen zeigen, dass am Ende von Dijkstras Algorithmus $dist[u_i] = dist(s, u_i)$ für jeden Knoten u_i
  - Im Folgenden zeigen wir, durch Induktion über i
    - In der i-ten Runde gilt $dist[u_i] = dist(s, u_i)$
    - In der i-ten Runde wird Knoten u_i gelöst

die orange Zahl an Knoten u_i auf Folie 11

die Kosten des „kürzesten" Weges

Korrektheitsbeweis 3/6

- Induktionsanfang: $i = 1$
  - In Runde 1 ist nur $u_1 = s$ aktiv
  - $dist[u_1] = 0 = dist(s, u_1)$
  - u_1 wird als einziger aktiver Knoten gelöst

Korrektheitsbeweis 4/6

- Induktionsschritt: $i \rightarrow i + 1$ für $i \geq 1$
  - Wir betrachten einen kürzesten Weg von s nach u_{i+1}

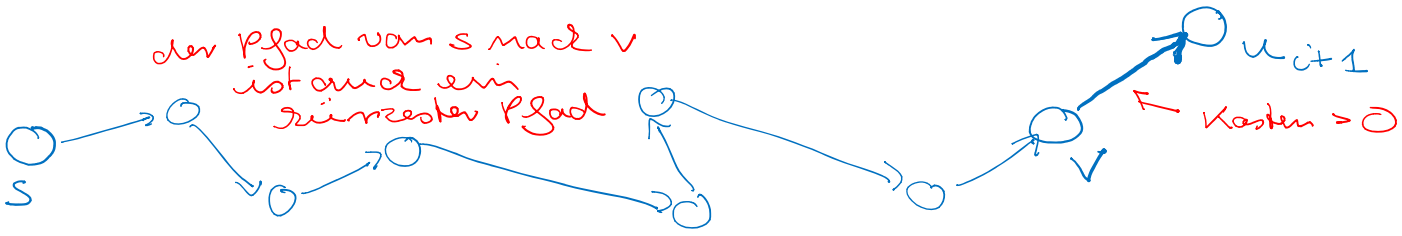
    Wir nehmen nicht an, dass unser Algorithmus diesen Weg kennt, aber wir können ihn im Beweis trotzdem betrachten
  - Sei v der Knoten direkt vor u_{i+1} auf diesem Weg … dann:

    $dist(s, u_{i+1}) = dist(s, v) + cost(v, u_{i+1}) > dist(s, v)$

    $cost(v, u_{i+1}) > 0$

    Das benutzt Annahme 1: alle Kantenkosten sind positiv
  - v muss also einer von $u_1, \ldots, u_i$ sein (aber nicht unbedingt u_i)

    Das benutzt Annahme 2: $dist(s, u_1) < dist(s, u_2) < \ldots$



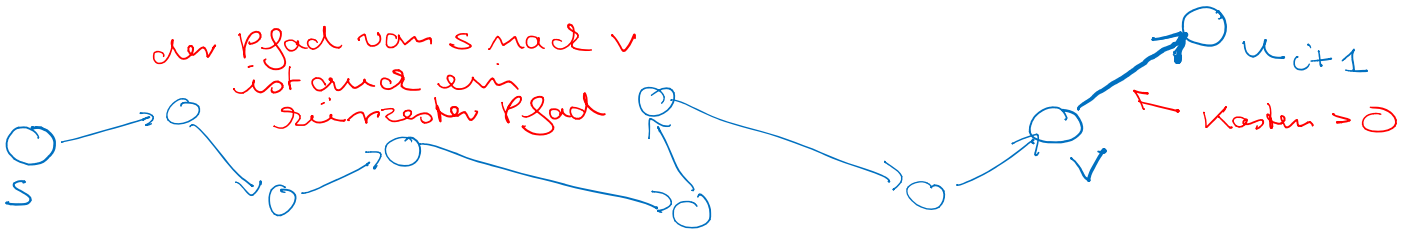
Korrektheitsbeweis 5/6

- Induktionsschritt: $i \rightarrow i + 1$ für $i \geq 1$ … Fortsetzung
  - Es ist also $v = u_j$ wobei $j \in 1 .. i$
  - Nach Induktionsvoraussetzung gilt seit spätestens Runde j

    $dist[u_j] = dist(s, u_j)$
  - In der Runde hat man dann, nach Relaxieren von (u_j, u_{i+1})

    $dist[u_{i+1}] = dist(s, u_j) + cost(u_j, u_{i+1}) = dist(s, u_{i+1})$

    Das gilt schon seit Runde j, aber erst in Runde i + 1 kann sich der Algorithmus sicher sein, dass es nicht besser geht



Korrektheitsbeweis 6/6

weil die Zwischenkosten immer einem tatsächlichen Pfad entsprechen

- Induktionsschritt: $i \rightarrow i + 1$ für $i \geq 1$ … Fortsetzung 2
  - Wir müssen noch zeigen, dass in Runde $i + 1$ auch u_{i+1} gelöst wird, und nicht u_k mit $k > i + 1$
  - Aber für $k > i + 1$ gilt nach Annahme 2 (Monotonie):

    $dist[u_k] \geq dist(s, u_k) > dist(s, u_{i+1})$
  - Also ist u_{i+1} in Runde $i+1$ der aktive Knoten mit dem kleinsten dist Wert und wird also in der Runde gelöst

Implementierung 1/9

- Grundprinzip
  - Wir müssen die Menge der aktiven Knoten verwalten
  - Ganz am Anfang ist das nur der Startknoten
  - Am Anfang jeder Runde brauchen wir den aktiven Knoten u mit dem kleinsten Wert für dist[u]
  - Es bietet sich also an, die aktiven Knoten in einer **Prioritätswürgeschlange** zu verwalten

    Mit Schlüssel dist[u] und Wert u

Implementierung 2/9

- Update von dist[u]
  - Beobachtung: der **dist** Wert eines aktiven Knotens kann sich mehrmals ändern, bevor er schließlich gelöst wird

    Wir müssen dann seinen Wert in der PW verkleinern, ohne dass wir den Knoten rausnehmen
  - Genau dafür gibt es die Operation **changeKey**

    Allerdings steht diese Operation nicht bei allen PWs zur Verfügung, z.B. bei der std::priority_queue von C++

Implementierung 3/9

- Implementierung ohne changeKey
  - Statt changeKey macht man einfach ein insert mit dem neuen (niedrigeren) dist Wert

    Den Eintrag mit dem alten Wert lässt man einfach drin

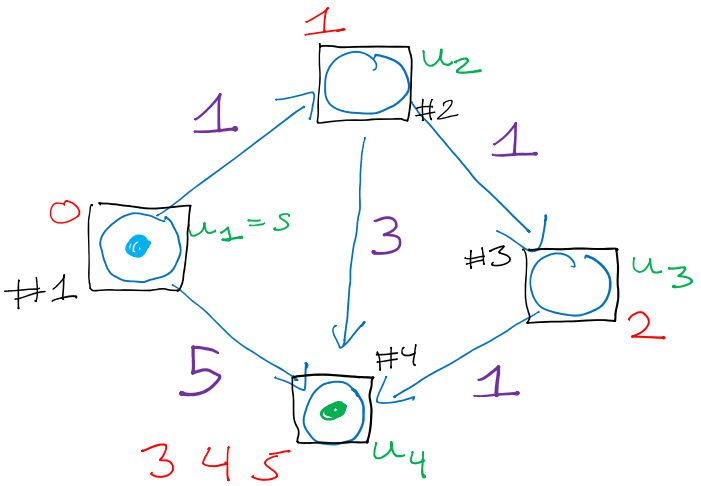
    Bei gleichen oder höheren dist Wert macht man nichts
  - Wenn der Knoten gelöst wird, dann mit dem niedrigsten Wert mit dem er in die PW eingefügt wurde
  - Wenn man dann später nochmal auf den Knoten trifft, mit höherem dist Wert, nimmt man ihn einfach heraus und macht **nichts**

Implementierung 4/9

$dist(s, u_1) = 0$
$dist(s, u_2) = 1$
$dist(s, u_3) = 2$
$dist(s, u_4) = 3$

siehe ANNAHME 2. vom Beweis

- Beispiel für Dijkstra mit PW ohne changeKey



ZUSTAND DER PW

#1
0 u_1
#2
1 u_2
5 u_4
IGNORIEREN
4 u_4
IGNORIEREN
#3
2 u_3
#4
3 u_4

das ist leicht weil man ein Feld für die dist Werte hat

Implementierung 5/9

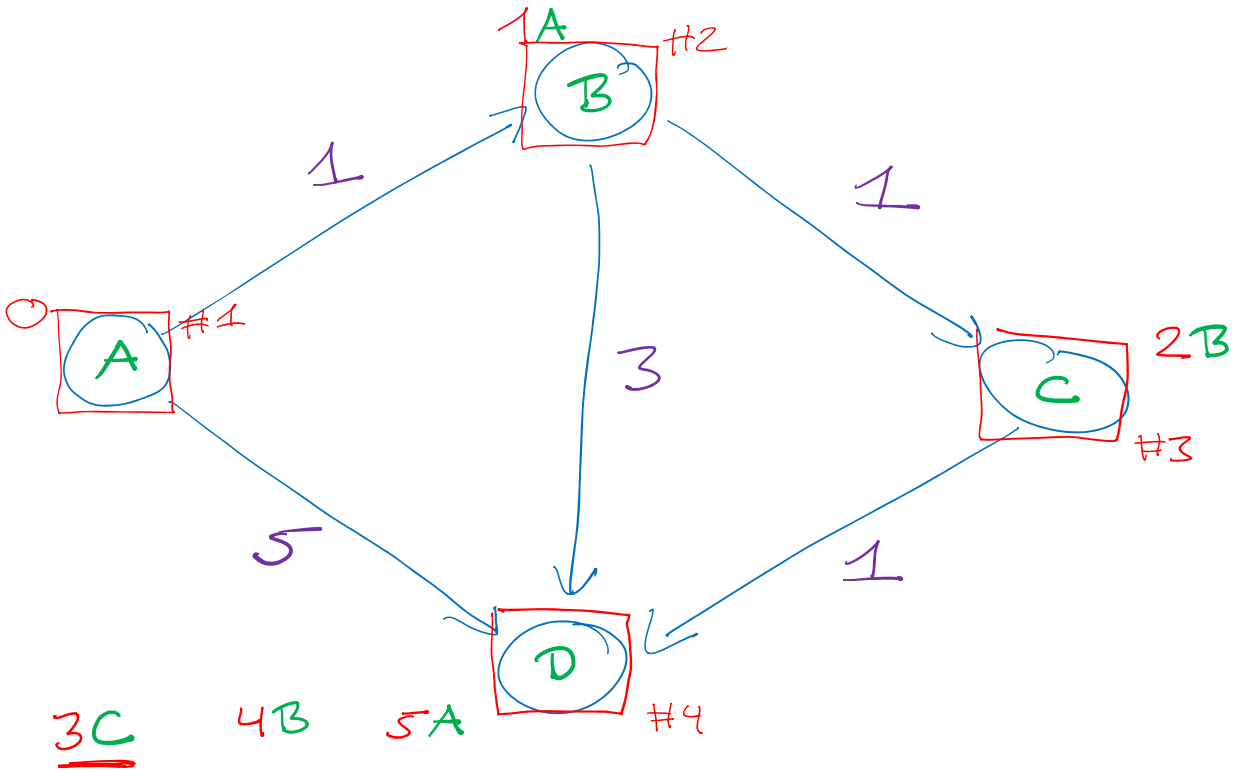
UNI FREIBURG

- Berechnung der kürzesten Pfade
  - So wie wir Dijkstras Algorithmus bisher beschrieben haben, berechnet er nur die **Länge** des kürzesten Weges
  - Wenn man sich bei jeder Relaxierung den Vorgängerknoten auf dem aktuell kürzesten Pfad merkt, kriegt man aber auch leicht die tatsächlichen **Pfade**
  - Es reicht für jeden Knoten ein Zeiger, weil jeder Präfix eines kürzesten Weges selber ein kürzester Weg ist
  - Um den kürzesten Weg zu bekommen, kann man dann einfach die Zeiger bis zum Startknoten zurückverfolgen

Implementierung 6/9

START = A
ZIEL = D

- Berechnung der kürzesten Pfade, Beispiel



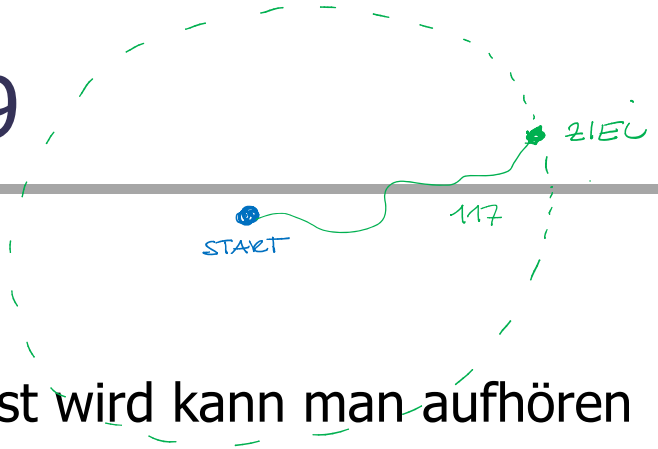
Implementierung 7/9

- Visualisierung eines Pfades mit MapBBCode
  - Für das **ÜB10** bekommen Sie einen Datensatz mit Geo-Koordinaten für jeden Knoten
  - Man kann einen Pfad dann also auf einer Karte malen
  - Das geht sehr einfach mit MapBBCode

    http://share.mapbbcode.org

    Ich mache das jetzt mal an einem einfachen Beispiel vor

Implementierung 8/9



- Abbruchkriterium
  - Sobald der Zielknoten t gelöst wird kann man aufhören

    Aber nicht vorher, dann kann noch dist[t] > dist(s, t) sein
  - Bevor Dijkstras Algorithmus t erreicht, hat er die kürzesten Wege zu **allen** Knoten u mit dist(s, u) < dist(s, t) berechnet
  - Das hört sich verschwenderisch an, es gibt aber für allgemeine Graphen keine (viel) bessere Methode

    Grund: erst wenn man alles im Umkreis von dist(s, t) um den Startknoten s abgesucht hat, kann man sicher sein, dass es keinen kürzeren Weg zum Ziel t gibt

Implementierung 9/9

für das ÜB10 können Sie auch Ihre solche Bucket Queue nehmen (müssen Sie aber nicht)

- Laufzeit dieser Implementierung
  - Jeder der **n** Knoten wird genau **einmal** gelöst
  - Genau dann werden seine ausgehenden Kanten betrachtet
  - Jede der **m** Kanten führt also zu höchstens einem insert
  - Die Anzahl der Operationen auf der PW ist also O(m)
  - Die Laufzeit von Dijkstras Algorithmus ist also **O(m · log n)**
  - Mit einer komplizierteren PW geht auch **O(m + n · log n)**
  - In der Praxis ist aber oft m = O(n)

    Dann ist die asymptotische Laufzeit für die kompliziertere PW nicht besser und man nimmt besser die einfachere PW

Literatur / Links

- Kürzeste Wege und Dijkstras Algorithmus
  - In Mehlhorn/Sanders:

    10 Shortest Paths
  - In Wikipedia

    http://en.wikipedia.org/wiki/Shortest_path_problem

    http://en.wikipedia.org/wiki/Dijkstra's_algorithm