# Algorithmen und Datenstrukturen (ESE)
# Entwurf, Analyse und Umsetzung von Algorithmen (IEMS)
# WS 2012 / 2013

## Vorlesung 13, Dienstag, 29. Januar 2013
(Kürzeste Wege, Dijkstras Algorithmus)

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

UNI FREIBURG

# Blick über die Vorlesung heute

UNI FREIBURG

- Organisatorisches
  - Ihre Erfahrungen mit dem Ü12 (BFS, DFS, Z-Komponenten)
  - Infos zur Klausur (am Samstag, den 23. März, 11 – 13 Uhr)
- Kürzeste Wege
  - Dijkstras Algorithmus zur Berechnung des kürzesten Weges zwischen zwei Knoten (in einem Graphen mit Kantenkosten)
  - Idee + Beispiel + Korrektheitsbeweis
  - Hinweise zur Implementierung und darüber hinaus
  - **Übungsaufgabe (Ü13):** Dijkstras Algorithmus implementieren, und damit den "Durchmesser" des Saarland Graphen berechnen

# Erfahrungen mit dem Ü12 (BFS, DFS, ZK)

UNI FREIBURG

- Zusammenfassung / Auszüge Stand 29. Januar 16:00
  - Aufgabe gut machbar, allerdings Zeitprobleme bei einigen
  - Aufgabe mit realen Daten (GeoNames, Straßengraphen) machen mehr Spaß als zufällige Daten
  - Gut, dass man sich aussuchen konnte ob DFS oder BFS
  - Beim rekursiven DFS Limit für Stackgröße erhöhen

    warum → siehe Vorlesung 9 zu Funktionsaufrufen und Stack
  - Nochmal erlebt wie langsam quadratische Laufzeit ist
  - Einlesen in C++ mit cin, scanf, getline, read ? … ü-nächste Folie
  - Frau Bast verdeckt am Ende Erklärungen zu Ü-Blättern

    ok, ich habe 3 kg abgenommen

# Ergebnisse mit dem Ü12 (BFS, DFS, ZK)

UNI FREIBURG

- Zusammenfassung Ihrer Ergebnisse
  - Daten: OSM Straßengraph des Saarlandes
  - #Knoten im Originalgraph: 1.119.289
  - #Knoten größte Z-Komponente: 213.567
  - Grund 1: Privatwege, Radwege, etc.
  - Grund 2: Nicht mit dem Rest verbundene Subnetzwerke
  - Laufzeit: in < 1 Sekunde machbar (inkl. Einlesen)

# Einlesen von Dateien in C++

UNI FREIBURG

- Speziell von zeilenbasierten Daten
  - Option 1: **FILE*** und **getline**

    Effizient und gut, wenn auch "C Style"
  - Option 2: **ifstream** und **getline**

    "C++ Style", da gab es früher Probleme mit Dateien > 2GB, aber inzwischen genauso gut wie FILE* und getline

    Felderweise << ist aber ineffizient, erst ganze Zeile lesen !
  - Option 3: **fscanf** bzw. **sscanf**

    Fehleranfällig und ineffizient, no-no bei großen Daten
  - Option 4: **FILE*** und **read**

    Alles an einem Stück in einen String einlesen ist natürlich am effizientesten, aber doof wenn nicht alles in den Speicher passt

# Pfade in einem Graphen (Wiederholung)

UNI FREIBURG

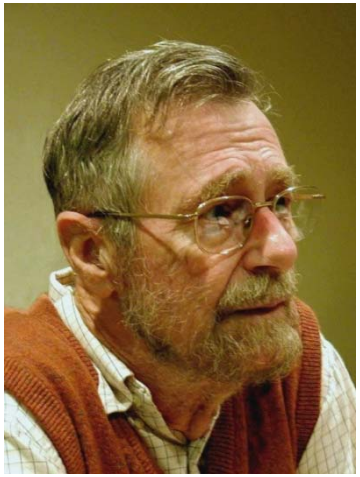
- Für einen Graphen $G = (V, E)$
  - Ein Pfad in $G$ ist eine Folge $u_1, u_2, u_3, \ldots, u_l \in V$ mit
    - $(u_1,u_2), (u_2,u_3), \ldots, (u_{l-1},u_l) \in E$ [gerichteter Graph]
    - $\{u_1,u_2\}, \{u_2,u_3\}, \ldots, \{u_{l-1},u_l\} \in E$ [ungerichteter Graph]
  - Die **Länge des Pfades** (auch: Kosten des Pfades)
    - ohne Kantengewichte: Anzahl der Kanten
    - mit Kantengewichte: Summe der Gewichte auf dem Pfad
  - Der **kürzeste Pfad** (engl. shortest path) zwischen zwei Knoten $u$ und $v$ ist der Pfad $u, \ldots, v$ mit der kürzesten Länge
  - Der **Durchmesser** eines Graphen ist der längste kürzeste Pfad = $\max_{u,v}${Länge von $P$ : $P$ ist ein kürzester Pfad zwischen $u$ und $v$}

# Dijkstras Algorithmus 1/3

UNI FREIBURG

- Idee
  - 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)

    Wir werden gleich sehen: wenn alle Kantenlängen = 1 sind, ist das genau Breitensuche / BFS
- Ursprung
  - 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**

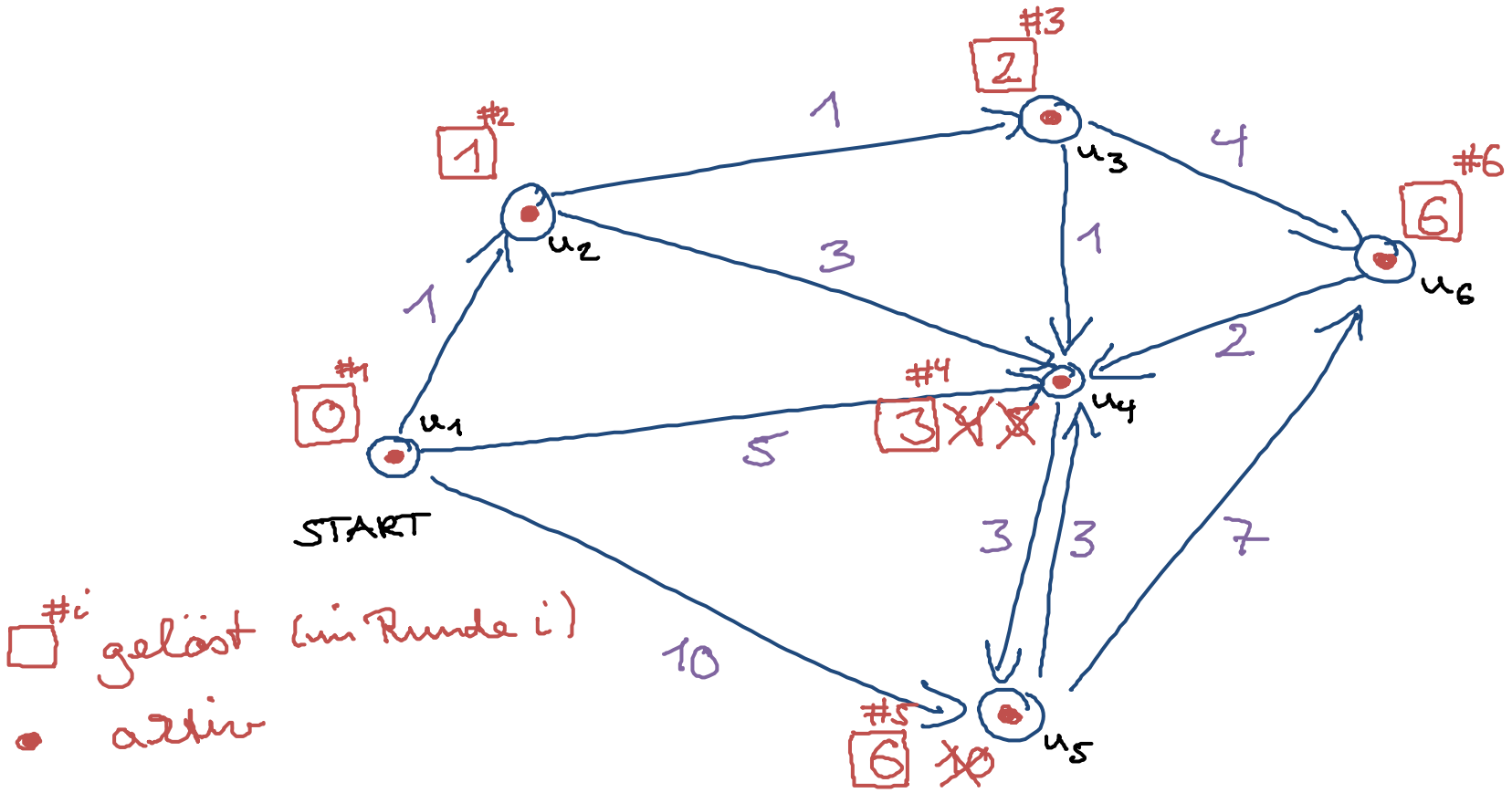


# Dijkstras Algorithmus 2/3

- High-level Beschreibung des Algorithmus
  - Drei Arten von Knoten:
    - Für die **gelösten** Knoten u kennen wir dist(s, u)
    - Für die **aktiven** Knoten haben wir einen Pfad der Länge dist(u) ≥ dist(s,u) (kann optimal sein, muss aber nicht)
    - Die **unerreichten** Knoten haben wir noch nicht erreicht
    - Auf Englisch: settled, active, unreached
  - 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 jeden Nachbarn v von u prüfen wir, ob wir v über u schneller erreichen können als bisher = Relaxieren von (u, v)
  - Nächste Runde, bis es keine aktiven Knoten mehr gibt

# Dijkstras Algorithmus 3/3

- Beispiel



# Korrektheitsbeweis 1/3

beim Beispiel auf der Folie vorher nicht der Fall

UNI FREIBURG

- Argumentationslinie
  - **Annahme 1:** Alle Kantenlängen sind > 0, siehe Folie 19
  - **Annahme 2:** Die dist(s, u) sind **alle verschieden**

    Das erlaubt einen einfacheren und intuitiveren Beweis

    Es geht aber auch ohne, siehe Referenzen … nur bei Interesse
  - Mit A2 gibt es eine Anordnung $u_1$, $u_2$, $u_3$, **...** der Knoten $u_1 = s$

    so dass gilt $dist(s, u_1) < dist(s, u_2) < dist(s, u_3) <$ **...**
  - 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, dass:

    … in der i-ten Runde gilt $dist(u_i) = dist(s, u_i)$

    … in der i-ten Runde wird Knoten $u_i$ gelöst

# Korrektheitsbeweis 2/3

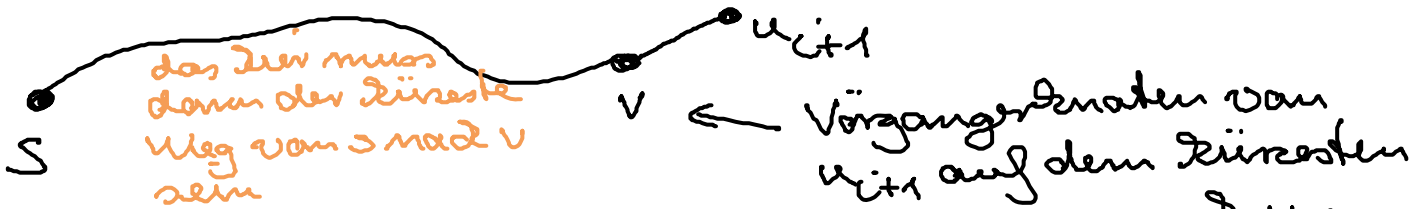
Induktionsanfang $i=1$:

Am Anfang ist nur der Startknoten $s = u_1$ aktiv und $dist(s) = 0$.

Den löst man dann, und dann ist $dist(s) = dist(s, u_1) = 0$

---

Induktionsschritt $1, \dots, i \to i+1$:




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

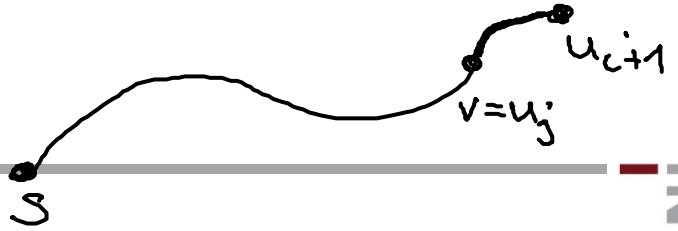
Kosten der Kante von v nach $u_{i+1}$ $> 0$ (wegen Annahme 1)

$\Rightarrow dist(s, v) < dist(s, u_{i+1})$

$\Rightarrow$ Das v ist eins von $u_1, \dots, u_i$

$v = u_j$, $1 \le j \le i$

# Korrektheitsbeweis 3/3




FORTSETZUNG:

Nach Induktionsannahme wurde $v = u_j$ in Runde $j$ gelöst und dabei war $dist(v) = dist(s, u_j)$

Beim Relaxieren in Runde $j$ wurde dann $dist(u_{i+1}) = \underbrace{dist(v)}_{= dist(s, u_j)} + c(v, u_{i+1})$ gesetzt

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

Jetzt müssen wir uns noch $u_k$, $k > i+1$ ansehen:

$dist(u_k) \geq dist(s, u_k) > dist(s, u_{i+1}) = dist(v)$

ALSO wird in Runde $i+1$ das $u_{i+1}$ gelöst und $dist(u_{i+1}) = dist(s, u_{i+1})$

(solang seit Runde $j \leq i$)

# Implementierungshinweise 1/3

UNI FREIBURG

- Einige Hinweise (der Rest ist Übungsaufgabe)
  - 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 **Priori-tätswarteschlange** zu verwalten, mit Schlüssel dist(u)
  - Folgendes Problem taucht dabei auf:
    - Die Länge des aktuell kürzesten Pfades zu einem aktiven Knoten kann sich mehrmals ändern, bevor der Knoten schließlich gelöst wird
    - Wir müssen dann seinen Wert in der PW verkleinern, **ohne** dass wir den Knoten rausnehmen

# Implementierungshinweise 2/3

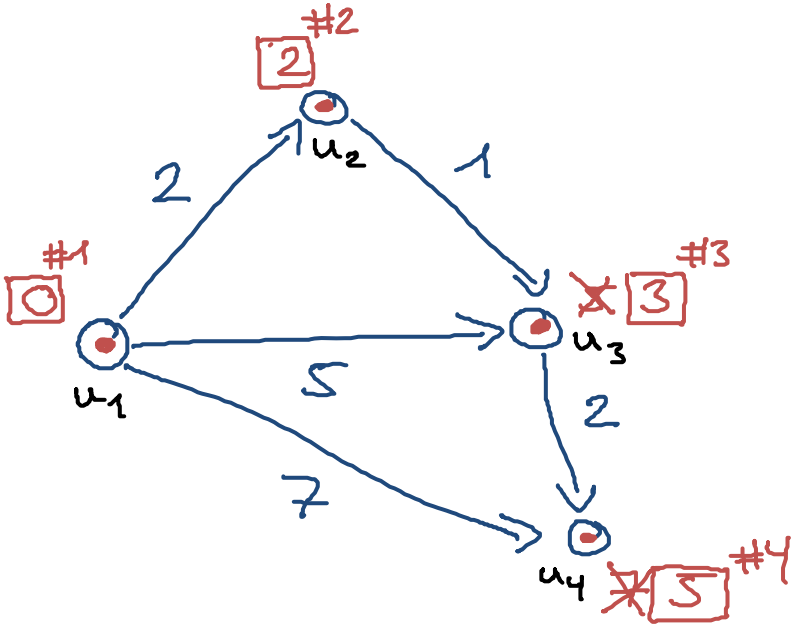
UNI FREIBURG

- Oft gibt es nur insert, getMin und deleteMin
  - Mit so einer PW hat man nur Zugriff auf das jeweils kleinste Element, nicht auf ein beliebiges
  - **Alternative:** Sieht man einen Knoten wieder, mit einem niedrigeren dist Wert, fügt man ihn einfach nochmal ein

    Bei einem gleichen oder höheren dist Wert macht man nichts !
  - Den Eintrag mit dem alten Wert lässt man einfach drin
  - 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**

# Implementierungshinweise 3/3

- Beispiel dazu




Man hat ja ein Feld dist das einem für jeden Knoten die beste Distanz von s gibt, da schaut man nach.

PRIORITÄTS-WARTESCHLANGE

| | | |
|---|---|---|
| #1 | $u_1$ | GELÖST |
| #2 | $u_2$ | GELÖST |
| | $u_3$ 5 | IGNORIERT ! |
| | $u_4$ 7 | IGNORIERT ! |
| #3 | $u_3$ 3 | GELÖST |
| #4 | $u_4$ 5 | GELÖST |

# Laufzeitanalyse

Mit unserem Trick können bis zu m Elemente in der PW sein.

- Für einen Graph mit n Knoten und m Kanten
  - Jeder Knoten wird genau **einmal** gelöst
  - Genau dann werden seine ausgehenden Kanten betrachtet
  - Jede ausgehende Kante führt zu höchstens einem insert
  - Die Anzahl der Operationen auf der PW ist also $O(m)$
  - Die Laufzeit von Dijkstras Algorithmus ist also $O(m \cdot \log m)$
  - Weil $m \leq n^2$ ist das auch $O(m \cdot \log n)$
  - Mit einer aufwändigeren PW geht auch $O(m + n \cdot \log n)$

    Zum Beispiel mit sogenannten **Fibonacci-Heaps**

    Für große und dichte Graphen ($m \sim n^2$) ist das klar besser

    In der Praxis ist aber oft $m = O(n)$ und dann ist der einfache **Binary Heap** die bessere Wahl ... siehe Vorlesung 6

$m \geq n$

$\log n^2 = 2 \cdot \log n$

# Weiterführende Kommentare 1/3

UNI FREIBURG

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

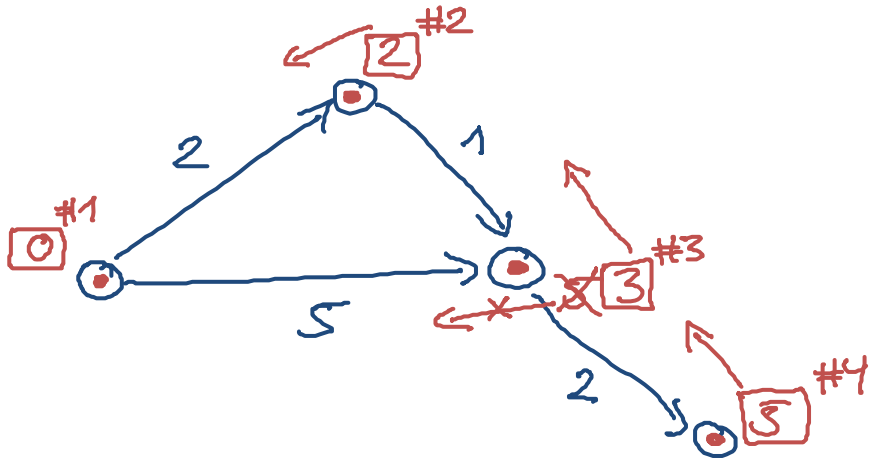
    … aber nicht vorher, dann kann dist(t) > dist(s, t) sein!
  - Bevor Dijkstras Algorithmus t erreicht, hat er die kürzesten Wege zu **allen** Knoten u mit dist(s, u) < dist(s, t) berechnet
  - Dijkstras Algorithmus löst damit nicht nur das sogenannte single source **single** target shortest path Problem, sondern gleich das sogenannte single source **all** targets Problem
  - Das hört sich verschwenderisch an, es gibt aber für allgemeine Graphen keine (viel) bessere Methode

    Intuitiv: 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

# Weiterführende Kommentare 2/3

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**



# Weiterführende Kommentare 3/3


- Erweiterungen
  - In unserem Beweis haben wir benutzt, dass die Kantenlängen alle **nicht-negativ** sind sogar $> 0$
  - Bei negativen Kantenkosten kann es negative Zyklen geben, um mit denen umzugehen braucht man andere Algorithmen
    - Zum Beispiel den Bellman-Ford Algorithmus
    - Wenn der Graph **azyklisch** ist, reicht auch einfach topologisches Sortieren (mit DFS) + Relaxieren der Knoten in der Reihenfolge dieser Sortierung
  - Eine (nicht nur) in der künstlichen Intelligenz häufig benutzte Variante von Dijkstras Algorithmus ist der **A* Algorithmus**:

    Dann zusätzlich gegeben: $h(u)$ = Schätzwert für $dist(u, t)$

# Literatur / Links

UNI FREIBURG

- 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