universität freiburg

Algorithmen und Datenstrukturen SS 2025

Vorlesung 10: Graphen, Dijkstras Algorithmus

Dienstag, 8. Juli 2025

Prof. Dr. Hannah Bast
Professur für Algorithmen und Datenstrukturen
Institut für Informatik
Albert-Ludwigs-Universität Freiburg

Blick über die Vorlesung heute

- Organisatorisches
  - Erfahrungen mit dem Ü9 — Prio-Q und Adresssuche
  - Vorgucker auf das Ü10 — Dijkstra und Routenplaner
- Inhalt
  - Graphen — Terminologie + Beispiele
  - Dijkstras Algorithmus — Algorithmus + Beispiel
  - Korrektheit + Implementierung — Beweis + diverse Hinweise

Erfahrungen mit dem Ü9 1/2

- Auszüge aus Ihrem Feedback [halbwegs repräsentativ]

"Nachdem ich mir die Folien angeschaut habe, alles sehr einfach"

"Vorlesung wie immer super, schaue ich aktuell immer gerne an"

"Sehr cool, gefühlt wirklich etwas Sinnvolles zu programmieren"

"Sehr gute Mischung aus praktischen Aufgaben und Beweisen"

"Der Code war einfach zu schreiben, Beweise waren schwieriger"

"Bei den Beweisen muss ich immer kräftig überlegen"

"Denkproblem bei heapify → hat sich nach zwei Kopfstößen gelöst"

"Konnte die Aufgaben mit sehr viel Schweiß lösen"

"Bei A3 habe ich gefühlt mehr erklärt als zu beweisen"

"Probleme bei Pylance und autopep8 mit Typannotationen"

Erfahrungen mit dem Ü9 2/2

- Auszüge aus Ihrem Feedback, Fortsetzung

  "Sehr interessant und hilfreich, aber auch etwas schwierig"

  "Leider keine Zeit für die theoretischen Aufgaben gefunden"

  "Besonders hat mir gefallen [...] dass an einer Stelle mal etwas nicht ganz so schön geschriebenes stehen gelassen wurde, und nicht 3 mal hintereinander wegradiert und wieder hingeschrieben wurde"

  "Gefreut, dass wir pünktlich aufgehört haben. Es hat also wirklich geholfen [...] Zeichnungen vorzubereiten und Zeichnungen nicht bis zur Perfektion erneut zu zeichnen. Gerne weiter so"

  "Zum ersten mal diese blöde evaluirungskacke ausgefüllt"

  "Wo ist denn die Umfrage über unser Wohlergehen? Wie soll ich denn nun wissen, ob meine Kommilitonen genauso leiden wie ich?"

Vorgucker auf das Ü10

- Ein praktisches Blatt mit einer tollen Anwendung

  A1 Implementierung der Berechnung kürzester Wege

  Mittels Dijkstras Algorithmus, der in der Vorlesung heute sehr ausführlich erklärt und analysiert wird

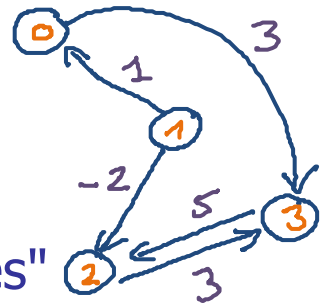
  A2 Implementierung eines ziemlich guten Routenplaners

  Verbindet sehr viel von dem, was Sie bisher gelernt haben

  1. Dijkstras Algorithmus (V10, siehe A1)
  2. Prioritätswürgeschlange zur Implementierung von DA (V9)
  3. Binärer Suchbaum für effiziente Addressuche (V8)
  4. Hash Map und Felder für die Graphdatenstruktur (V5 und V6)
  5. Laufzeitanalyse (V2)
  6. Sortieren braucht man sowieso immer (V1)

  Toller Datensatz (Adressen + Straßennetzwerk), war **viel** Arbeit

Graphen 1/5



- Definition:
  - Ein Graph G besteht aus zwei Mengen V und E

    V = Menge der **Knoten** ... engl. "nodes" oder "**v**ertices"

    E = Menge der **Kanten** ... engl. "**e**dges" oder "arcs"
  - Eine Kante e verbindet jeweils zwei Knoten u und v

    ungerichtete Kante: $e = \{u, v\}$ **Menge**

    gerichtete Kante: $e = (u, v)$ **Tupel**
  - Bei einem **gewichteten** Graph hat man für jede Kante ein Gewicht, auch Länge oder Kosten der Kante genannt

    Für das Ü10: die Reisezeit entlang der Kante (berechnet aus geometrischer Länge und Höchstgeschwindigkeit)

Graphen 2/5

- Repräsentation gerichteter Graph
  - **Adjazenzmatrix** … Platzverbrauch $\Theta(|V|^2)$

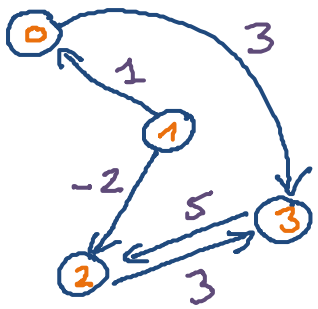
    Sinnvoll bei einem dichten Graphen, d.h. $|E| = \Theta(|V|^2)$
  - **Adjazenzlisten** … Platzverbrauch $\Theta(|V| + |E|)$

    Die Adjazenzlisten implementieren wir jetzt zusammen und den Code können Sie als Grundlage für das Ü10 verwenden

$36 \frac{km}{h} = 36 \cdot \frac{1000 m}{3600 s} = 10 \frac{m}{s}$

$v = \frac{s}{t} \Rightarrow t = \frac{s}{v}$

$|V| = 4$
$|E| = 5$



ADJAZENZMATRIX

	0	1	2	3
0	×	×	×	3
1	1	×	-2	×
2	×	×	×	3
3	×	×	5	×

× = spezieller Wert ≙ keine Kante

ADJAZENZLISTEN

Zielknoten, Gewicht

0 ↦ [3|3]
1 ↦ [0|1] [2|-2]
2 ↦ [3|3]
3 ↦ [2|5]

Hash Map
Key = Knoten
Value = Feld von Paaren (Zielknoten, Gewicht)

Graphen 3/5

- Repräsentation ungerichteter Graph
  - Einen ungerichteten Graphen kann man einfach als gerichteten Graphen darstellen, bei dem es jede Kante in beide Richtungen gibt
  - Falls es Kantenkosten gibt, sind die Kosten dann in beiden Richtungen gleich

    Für das Ü10 bekommen Sie einen gerichteten Graphen

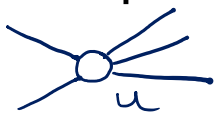
    Für jedes Stück Weg gibt es dort eine Kante in beide Richtungen, wenn man das Stück in beide Richtungen befahren kann, sonst nur in eine Richtung

Graphen 4/5

- Grad, Eingangsgrad, Ausgangsgrad
  - Grad einen Knotens u in einem ungerichteten Graph

    degree(u) = $|\{\{u, v\} : \{u, v\} \in E\}|$

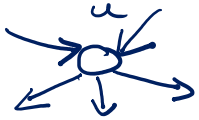
    degree(u) = 5

  - Eingangs- und Ausgangsgrad eines Knotens u in einem gerichteten Graph

    in-degree(u) = $|\{(v, u) : (v, u) \in E\}|$

    out-degree(u) = $|\{(u, v) : (u, v) \in E\}|$

    in-degree(u) = 2

    out-degree(u) = 3

Graphen 5/5

- Pfade
  - Ein Pfad in G ist eine Folge $u_1, u_2, u_3, \ldots, u_k \in V$ mit

    $(u_1,u_2), (u_2,u_3), \ldots, (u_{k-1},u_k) \in E$ gerichteter Graph

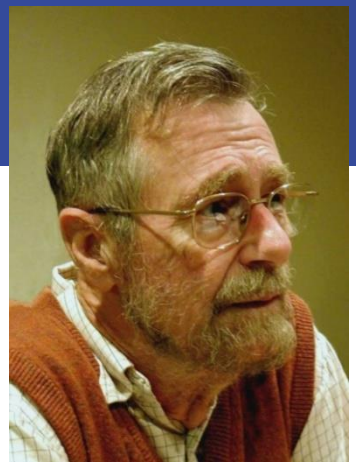
    $\{u_1,u_2\}, \{u_2,u_3\}, \ldots, \{u_{k-1},u_k\} \in E$ ungerichteter Graph
  - Die **Länge** bzw. **Kosten** eines Pfades

    ohne Kantengewichte: Anzahl der Kanten

    mit Kantengewichten: Summe der Gewichte auf dem Pfad
  - Der **kürzeste Pfad** (engl. shortest path) zwischen zwei Knoten u und v ist der Pfad u, ..., v mit minimalen Kosten

    Beispiele auf Folie 14, 24 und 26

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)

    Er war "brilliant, colourful, and opinionated"

    https://www.cwi.nl/about/history/e-w-dijkstra-brilliant-colourful-and-opinionated

    https://en.wikipedia.org/wiki/Edsger_W._Dijkstra
  - Der nach ihm benannte Algorithmus ist aus dem Jahr **1959**

Dijkstras Algorithmus 2/4

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

    unerreicht: $\text{dist}[u] = \infty$

    aktiv: $\text{dist}[u] \geq \boldsymbol{\mu}(s, u)$ aber nicht ∞

    gelöst: siehe nächste Folie

    Auf Englisch: unreached, active, settled

Dijkstras Algorithmus 3/4

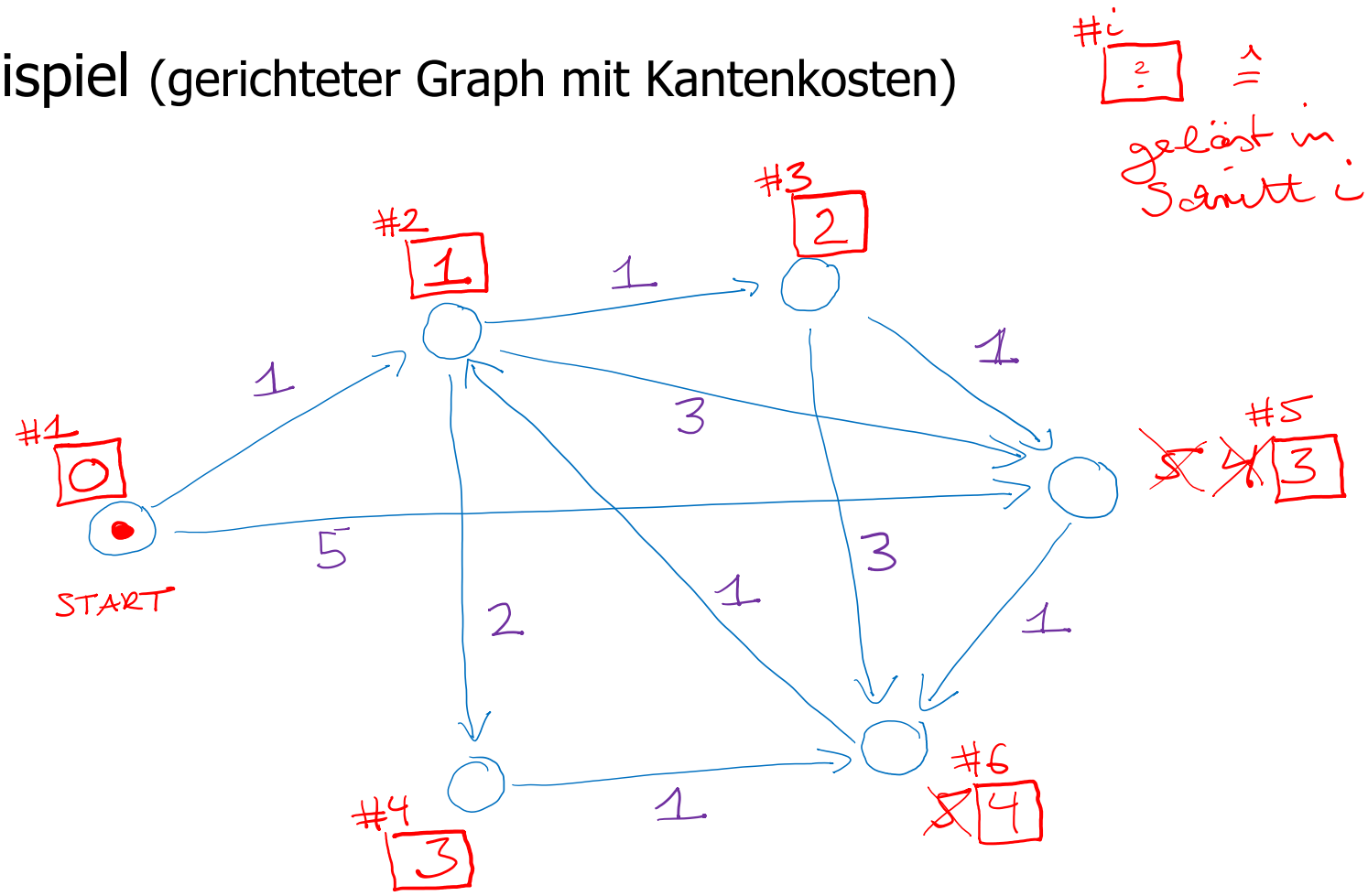
- Algorithmus
  - Zu Beginn nur s aktiv, mit dist[s] = 0 und dist[v] = ∞
  - In jeder Runde holen wir uns den aktiven Knoten u mit dem kleinsten Wert für dist[u]
  - Den Knoten u betrachten wir dann als **gelöst**
  - Für alle (u, v) ∈ E: prüfe ob dist[u] + cost(u, v) < dist[v] und falls ja, setze dist[v]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

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

    Alle gelösten Knoten kennen dann ihre Entfernung von **s**

Dijkstras Algorithmus 4/4

- Beispiel (gerichteter Graph mit Kantenkosten)



Korrektheitsbeweis 1/6

- Annahmen
  - **Annahme 1:** Alle Kantenlängen sind > 0
  - **Annahme 2:** Die $\boldsymbol{\mu}(s, u)$ sind alle **verschieden**

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

    so dass gilt $\boldsymbol{\mu}(s, u_1) < \boldsymbol{\mu}(s, u_2) < \boldsymbol{\mu}(s, u_3) < \ldots$

    Im Beispiel auf Folie 14 ist Annahme 2 **nicht** erfüllt
  - 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 haben unsere Knoten u_1, u_2, ..., u_n so nummeriert, dass $\mu(s, u_1) < \mu(s, u_2) < \ldots < \mu(s, u_n)$
  - Wir wollen zeigen, dass am Ende von Dijkstras Algorithmus $dist[u_i] = \mu(s, u_i)$ für jeden Knoten u_i
  - Im Folgenden zeigen wir, durch Induktion über i
    - Der Knoten u_i wird genau in der i-ten Runde gelöst
    - Ab der i-ten Runde gilt $dist[u_i] = \mu(s, u_i)$

    Wir sehen gleich im Beweis: das kann auch schon in früheren Runden gelten, aber erst dann wissen wir es

Korrektheitsbeweis 3/6

- Induktionsanfang: $i = 1$
  - In Runde 1 ist nur $u_1 = s$ aktiv
  - In der Runde wird also u_1 als einziger aktiver Knoten gelöst
  - Es ist dann $\text{dist}[u_1] = 0 = \boldsymbol{\mu}(s, u_1)$

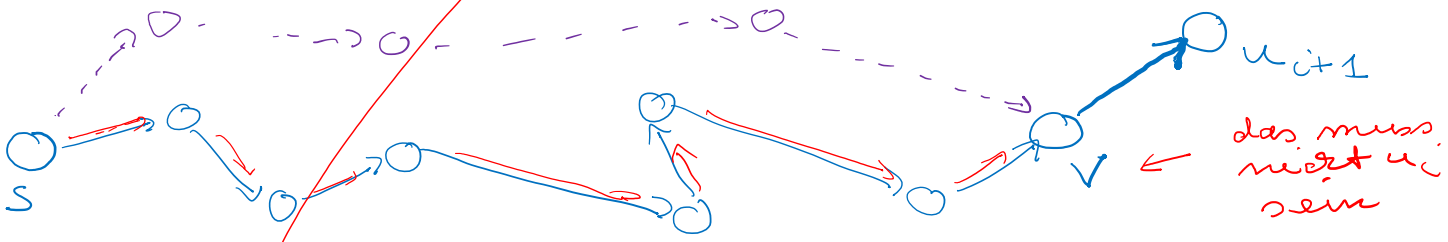
  Siehe Folie 14: zu Beginn dist[s] = 0, dist[v] = ∞ für alle v ≠ s

Korrektheitsbeweis 4/6

- Induktionsschritt: $i \to i + 1$ 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, wir nehmen nur an, dass es ihn gibt

    WICHTIG ZU VERSTEHEN: der Weg von s nach v ist auch ein kürzester Weg SONST hätte man auch einen kürzeren Weg nach u_{i+1}



  - Sei v der Knoten direkt vor u_{i+1} auf diesem Weg … dann:

    $\boldsymbol{\mu}(s, u_{i+1}) = \boldsymbol{\mu}(s, v) + \text{cost}(v, u_{i+1}) > \boldsymbol{\mu}(s, v)$

    > 0 nach ANNAHME 1

    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: $\boldsymbol{\mu}(s, u_1) < \boldsymbol{\mu}(s, u_2) < \ldots$

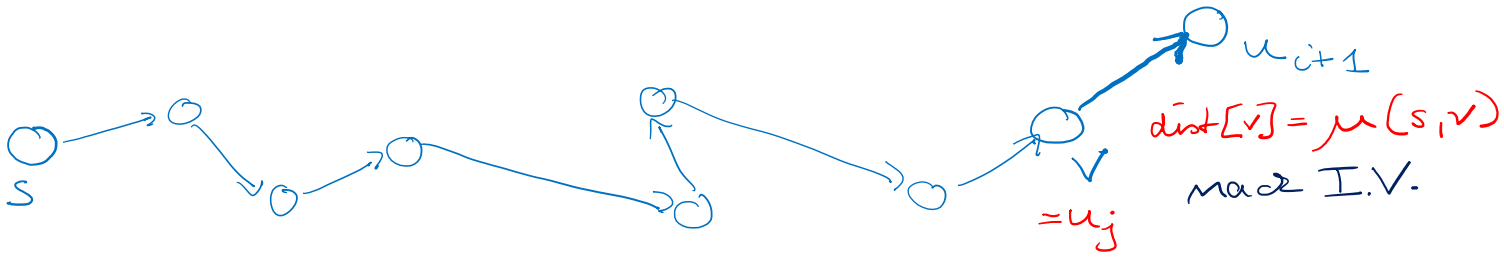
Korrektheitsbeweis 5/6

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

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

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

    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

- Induktionsschritt: $i \to i + 1$ für $i \geq 1$ … Fortsetzung 2
  - Wir müssen noch zeigen, dass in Runde $i + 1$ auch genau u_{i+1} gelöst wird, und nicht u_k mit $k > i + 1$

    Dass nicht $u_1, \ldots, u_i$ gelöst werden, wissen wir, weil sie nach Induktionsvoraussetzung in Runden $1, \ldots, i$ gelöst wurden
  - Für $k > i + 1$ gilt nach Annahme 2 (Monotonie):

    $\boldsymbol{\mu}(s, u_{i+1}) < \boldsymbol{\mu}(s, u_k)$
  - Da $\boldsymbol{\mu}(s, u_{i+1}) = \text{dist}[u_{i+1}]$ und $\boldsymbol{\mu}(s, u_k) \leq \text{dist}[u_k]$ haben wir:

    $\text{dist}[u_{i+1}] < \text{dist}[u_k]$
  - 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 **Priority Queue** 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 PQ verkleinern, ohne dass wir den Knoten rausnehmen
  - Genau dafür gibt es die Operation **change_key**

    Genauer gesagt, braucht man nur ein **decrease_key**, siehe die Bemerkung zu Fibonacci Heaps am Ende der V9
  - Bei vielen PQs, zum Beispiel der std::priority_queue von C++, gibt es aber weder change_key noch decrease_key

    Für das Ü10 sollen Sie die PQ von Ihrem Ü9 verwenden (oder von der Musterlösung, die hat ein change_key); Sie können es aber auch so machen wie auf Folie 23+24 erklärt

Implementierung 3/9

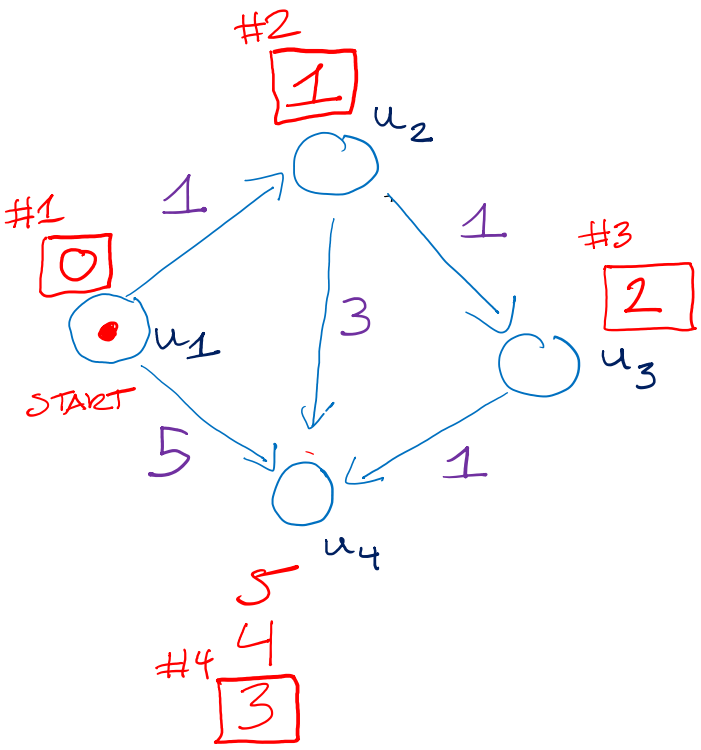
- Implementierung ohne changeKey
  - Statt changeKey macht man einfach ein insert mit dem neuen (niedrigeren) dist Wert

    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

- Beispiel für Dijkstra mit PW ohne changeKey



Zustand der PQ

KEY	VALUE	
~~0~~	~~u_1~~	
~~1~~	~~u_2~~	
5	u_4	*
4	u_4	*
~~2~~	~~u_3~~	
~~3~~	~~u_4~~	

* wenn die dran kommen, haben sie einen größeren dist Wert als den, den wir schon kennen
→ einfach ignorieren

Implementierung 5/9

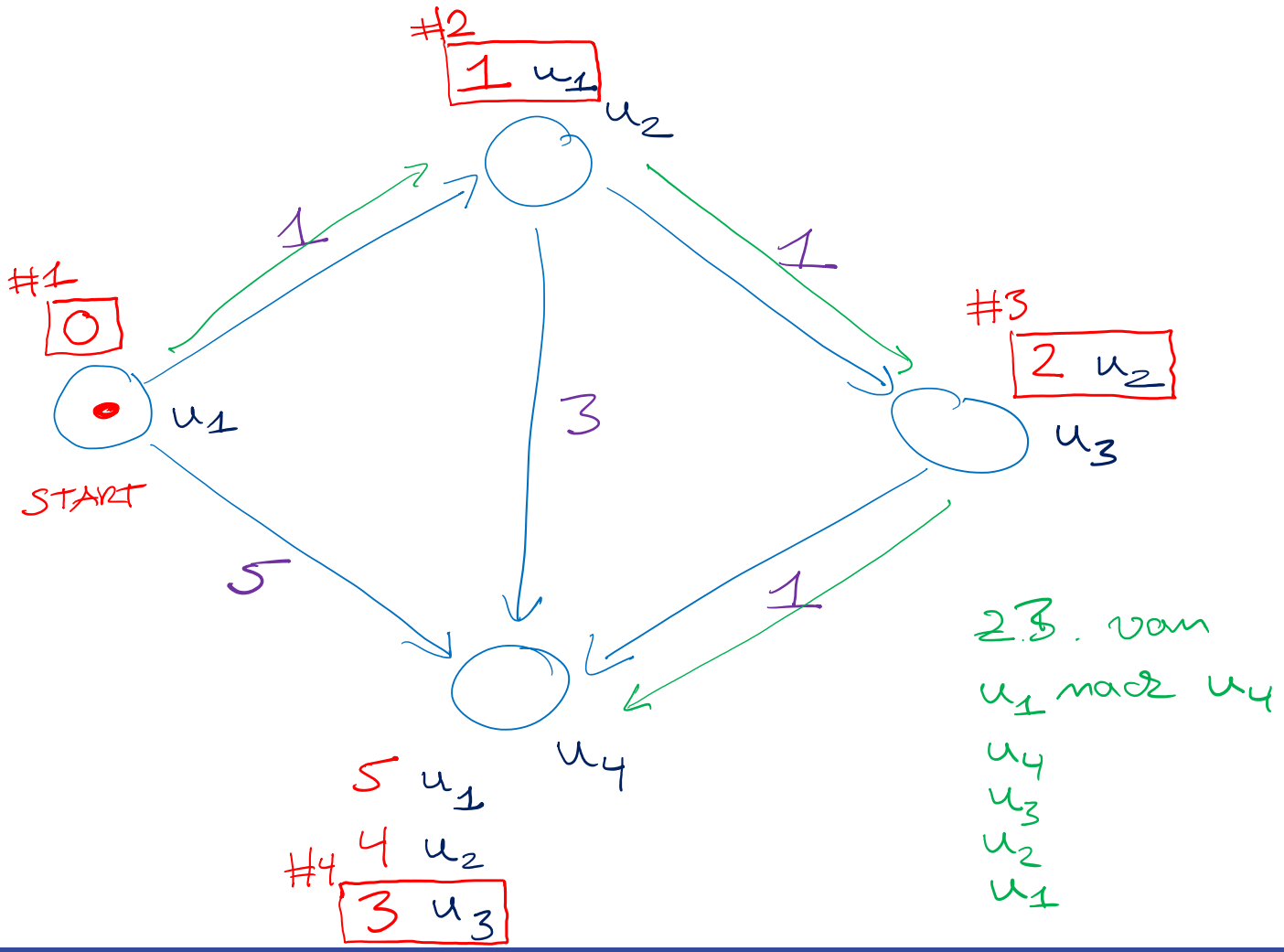
- Berechnung der kürzesten Pfade
  - So wie wir Dijkstras Algorithmus bisher beschrieben haben, berechnet er nur die **Länge** des kürzesten Weges
  - Wenn man sich bei jeder Relaxierung den Vorgängerknoten auf dem aktuell kürzesten Pfad merkt, kriegt man aber auch leicht die tatsächlichen **Pfade**

    Es reicht, sich für jeden Knoten **einen** Vorgängerknoten zu merken, weil jeder Präfix eines kürzesten Weges selber ein kürzester Weg ist
  - Um den kürzesten Weg zu bekommen, kann man dann die Vorgängerknoten bis zum Startknoten zurückverfolgen

    Siehe Beispiel auf der nächsten Folie und das sollen Sie auch für das Ü10 implementieren

Implementierung 6/9

- Berechnung der kürzesten Pfade, Beispiel



Implementierung 7/9

- Visualisierung eines Pfades, Hinweise zum Ü10
  - Für das Ü10 bekommen Sie einen Datensatz mit **Geo-Koordinaten** (Breitengrad, Längengrad) für jeden Knoten
  - Als Ergebnis soll dann eine URL von der folgenden Form ausgegeben werden

    https://ad-teaching.cs.uni-freiburg.de/AlgoDatSS2025/map/?47.99772,7.84251+47.99849,7.84324+47.99874,7.84343+...

    Details siehe Code-Vorlage auf dem Wiki
  - Sie bekommen außerdem eine Variante von dem Datensatz für die Adresssuche von Ü8

    Unterschied: für jede Adresse nicht das betreffende Gebäude, sondern der nächste Knoten des Straßennetzwerkes

Implementierung 8/9

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

    Aber **nicht vorher**, dann kann noch dist[t] > **μ**(s, t) sein
  - Bevor Dijkstras Algorithmus t erreicht, hat er die kürzesten Wege zu **allen** Knoten u mit **μ**(s, u) < **μ**(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 **μ**(s, t) um den Startknoten s abgesucht hat, kann man sicher sein, dass es keinen kürzeren Weg zum Ziel t gibt

Implementierung 9/9

- 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 PQ ist also $O(m)$
  - Die Laufzeit von Dijkstras Algorithmus ist also $\mathbf{O(m \cdot \log n)}$
  - Mit Fibonacci Heaps (V9) geht auch $\mathbf{O(m + n \cdot \log n)}$
  - In der Praxis ist aber oft $m = O(n)$

    Dann ist die asymptotische Laufzeit für Fibonacci Heaps gleich und man nimmt besser den einfachen binären Heap

Literatur / Links

- Graphen und Dijkstras Algorithmus
  - In Mehlhorn/Sanders:

    8 Graph Representation

    9 Graph Traversal

    10 Shortest Paths
  - In Wikipedia

    http://en.wikipedia.org/wiki/Graph_(mathematics)

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