Algorithmen und Datenstrukturen (ESE)
Entwurf, Analyse und Umsetzung von Algorithmen (IEMS)
WS 2013 / 2014

Vorlesung 13, Donnerstag, 30. Januar 2013
(Kürzeste Wege, Dijkstras Algorithmus)

Junior-Prof. Dr. Olaf Ronneberger
Image Analysis Lab
Institut für Informatik
Universität Freiburg

[basiert auf der letztjährigen Veranstaltung von Prof. Dr. Hannah Bast, Lehrstuhl für Algorithmen und Datenstrukturen, mit Folien aus „Informatik 2" von Prof. Dr. Matthias Teschner]

Blick über die Vorlesung heute

UNI FREIBURG

- Organisatorisches
  - Ihre Ergebnisse vom Ü12 (BFS, DFS, Z-Komponenten)
- 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

Ergebnisse mit dem Ü12 (BFS, DFS, ZK)

UNI FREIBURG

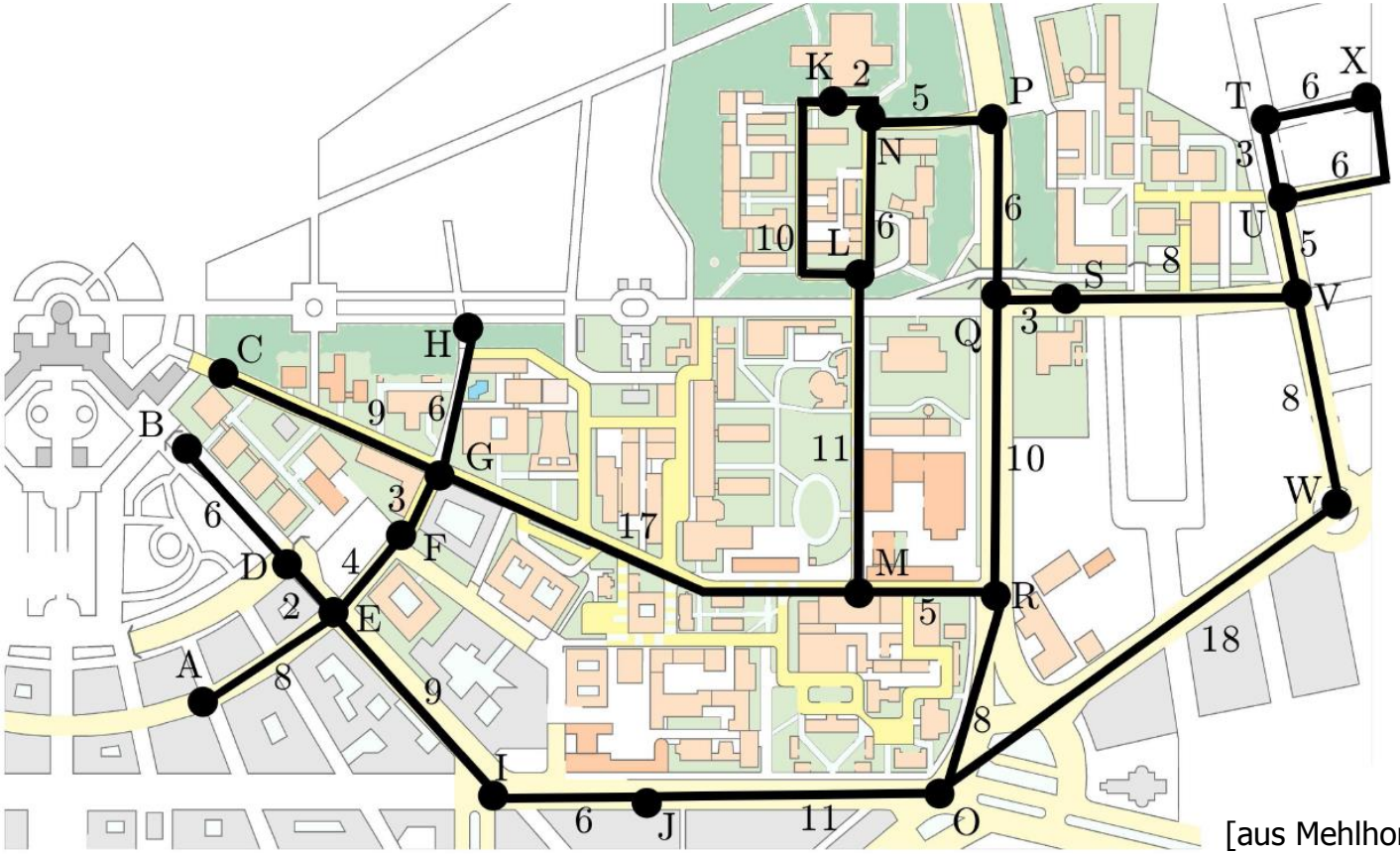
- Zusammenfassung Ihrer Ergebnisse
  - Daten: OSM Straßengraph des Saarlandes
  - #Knoten im Originalgraph: 1.119.289
  - #Knoten größte Z-Komponente: 213.567
  - Grund 1: Privatwege, Radwege, etc.
  - Grund 2: Nicht mit dem Rest verbundene Subnetzwerke
  - Laufzeit: in < 1 Sekunde machbar (inkl. Einlesen)

Pfade in einem Graphen (Wiederholung)

UNI FREIBURG

- Für einen Graphen $G = (V, E)$
  - Ein Pfad in G ist eine Folge $u_1, u_2, u_3, ..., u_l \in V$ mit
    - $(u_1,u_2), (u_2,u_3), ..., (u_{l-1},u_l) \in E$ [gerichteter Graph]
    - $\{u_1,u_2\}, \{u_2,u_3\}, ..., \{u_{l-1},u_l\} \in E$ [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 $u, ..., v$ 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\}

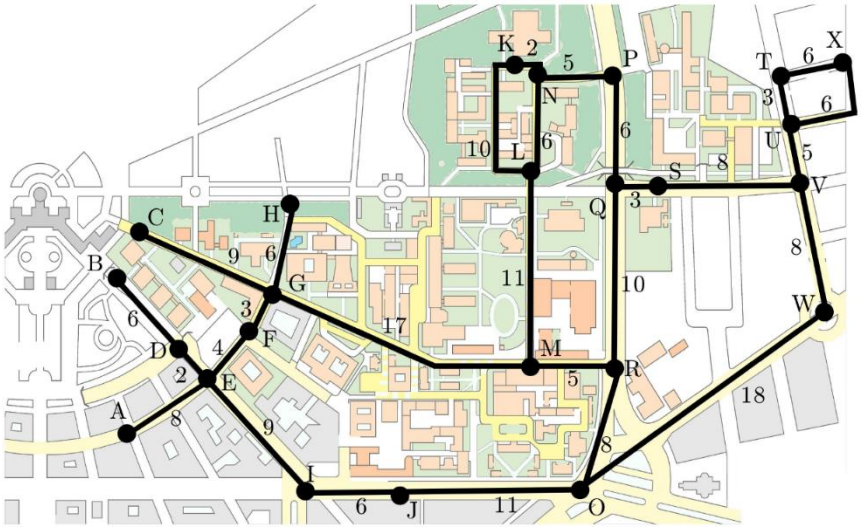
Shortest Path Algorithmus ohne Computer

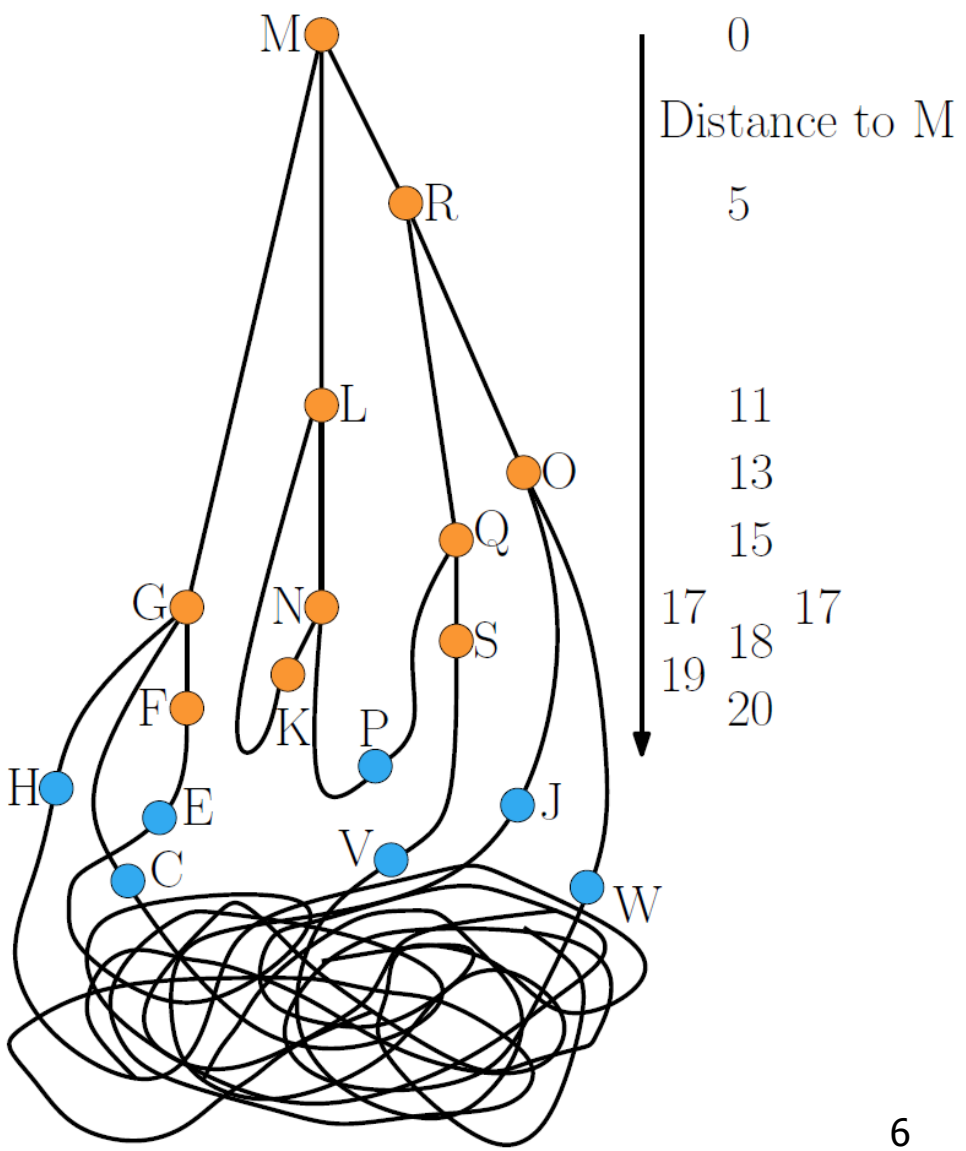


[aus Mehlhorn/Sanders Kapitel 10]

- Gesucht: Kürzester Pfad von Punkt M zu allen anderen Punkten
- Faden entlang der Wege spannen mit Knoten (z.B. Perlen) an Kreuzungen

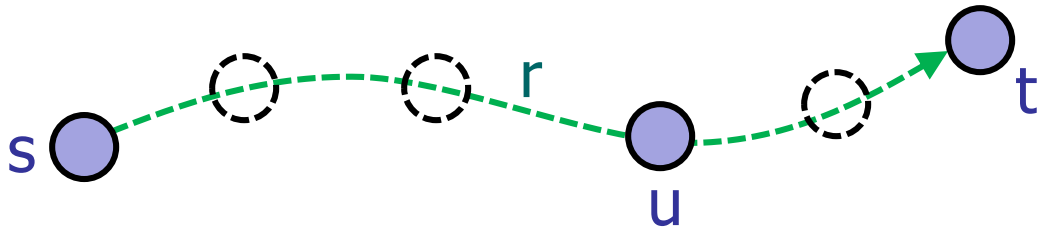
Shortest Path Algorithmus ohne Computer





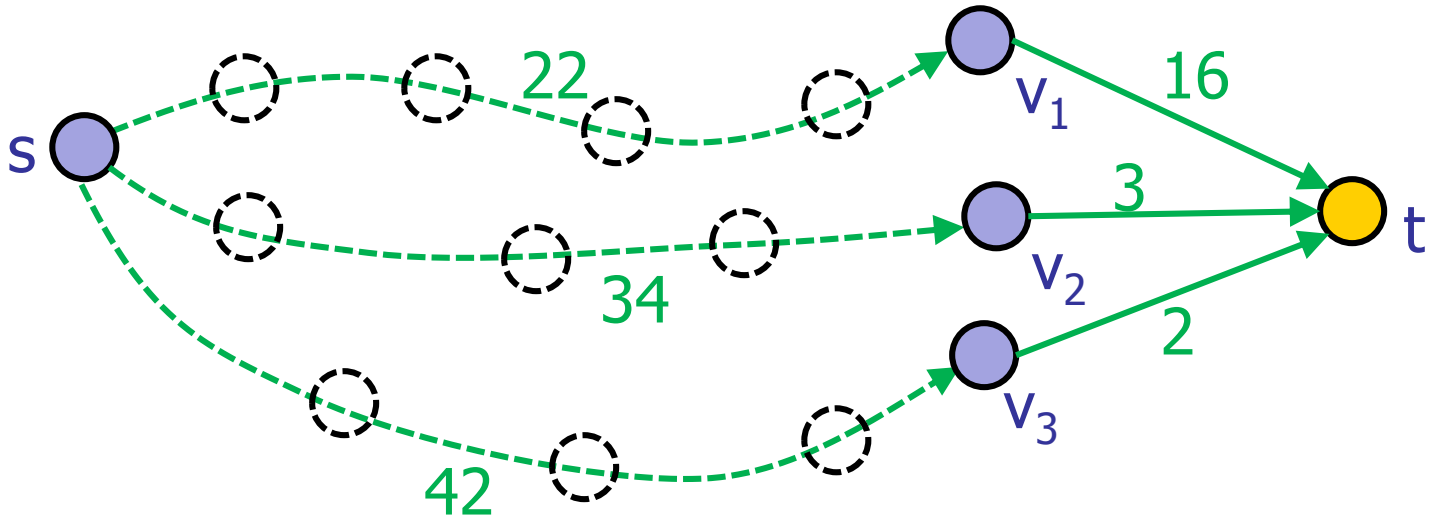
- Das Netz an Punkt M anfassen und hochheben, bis es in der Luft hängt
- Jeder Knoten (Perle) hängt jetzt in einer bestimmten Höhe.
- Der Abstand zu M ist genau der kürzeste Pfad

Eigenschaften von kürzesten Pfaden



- Sei r der kürzeste Pfad von s nach t
- Für jeden Knoten u auf diesem Pfad ist der Teilpfad von s nach u ebenfalls der kürzeste Pfad
- **Beweis**: Wenn es einen kürzeren Pfad von s nach u gäbe, dann könnte man den nehmen um schneller nach t zu kommen → dann wäre r nicht der kürzeste Pfad von s nach t
- Das gilt sogar für jeden beliebigen Teilpfad dazwischen: Also wenn der kürzeste Pfad von s nach t über u_1 und u_2 geht, dann ist der Teilpfad (u_1,u_2) der kürzeste Pfad von u_1 nach u_2

Eigenschaften von kürzesten Pfaden



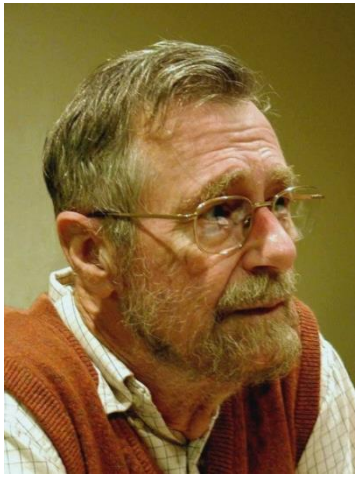
- D.h. sobald wir die kürzesten Pfade von s zu den Vorgängerknoten (hier: v_1, v_2, und v_3) von t kennen, können wir sehr einfach den kürzesten Pfad zu t bestimmen (hier Kosten für die 3 möglichen Pfade vergleichen)
- Idee: Die Kosten für den kürzesten Pfad im Knoten speichern
- Information über die Kanten schicken (message passing, Relaxieren)
- Jetzt brauchen wir nur noch eine Strategie, wie wir die Knoten abarbeiten

Dijkstras Algorithmus 1/3

UNI FREIBURG

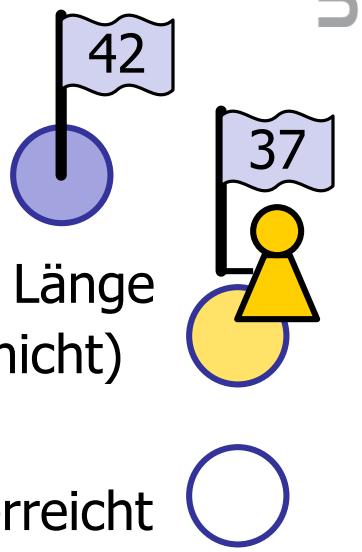
- Idee
  - Sei s der Startknoten und sei dist(s,u) die Länge des kürzesten Pfades von s nach u, für alle Knoten u
  - Besuche die Knoten in der Reihenfolge der dist(s,u)

    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**



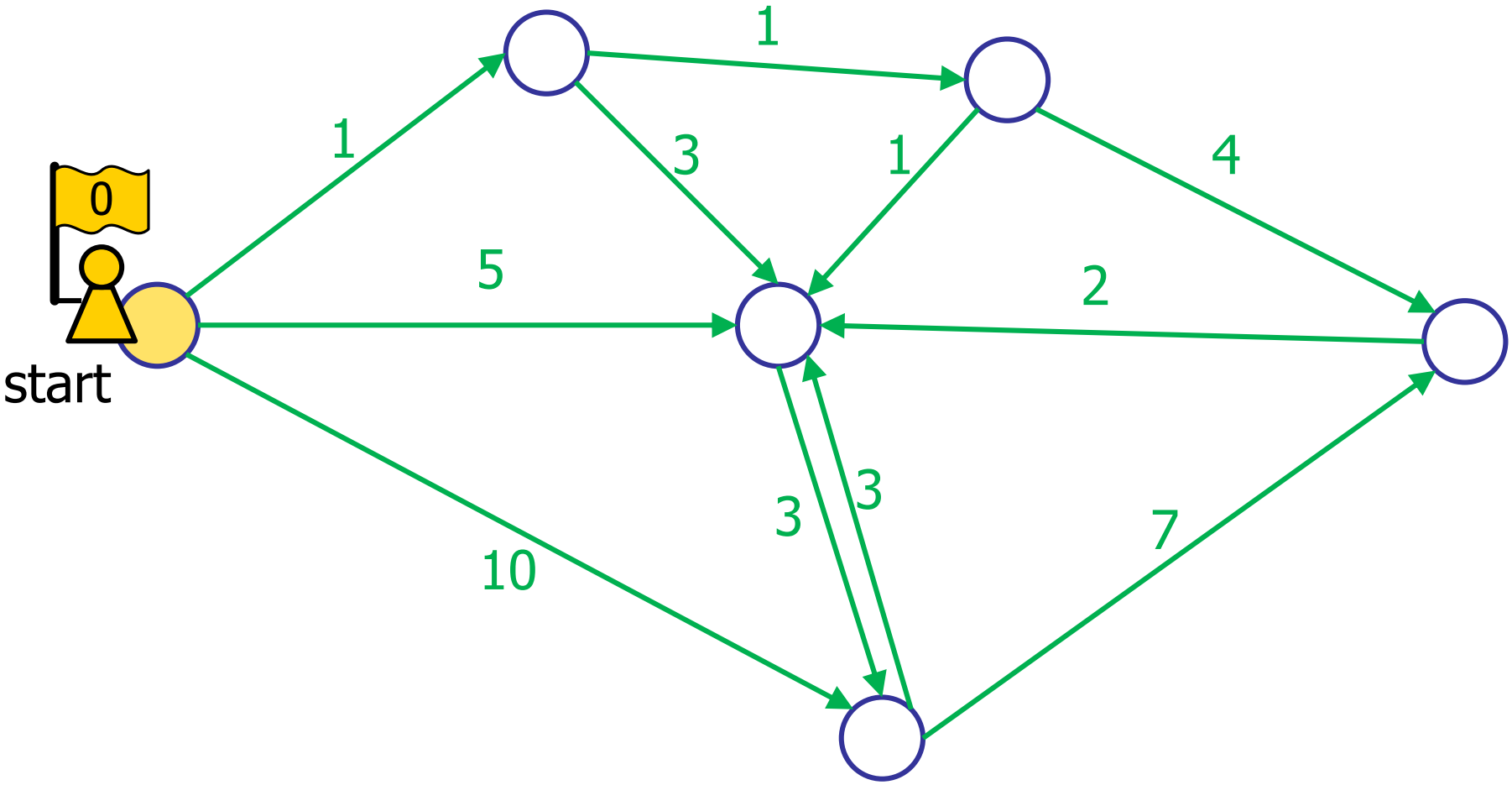
Dijkstras Algorithmus 2/3

- 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 td(u) ≥ dist(s,u) (kann optimal sein, muss aber nicht) (td für „tentative distance“)
    - Die **unerreichten** Knoten haben wir noch nicht erreicht
    - Auf Englisch: settled, active, unreached

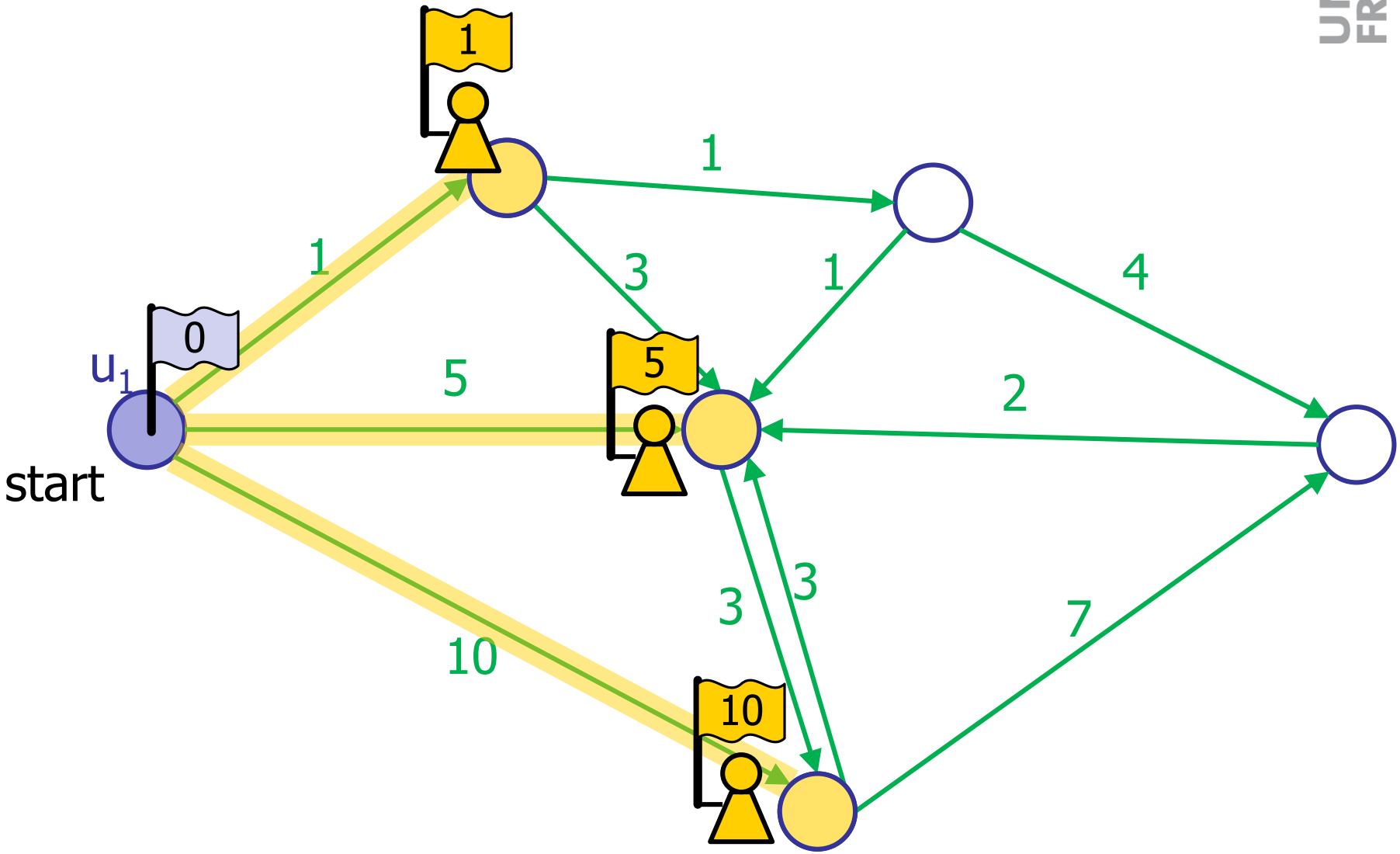


  - In jeder Runde holen wir uns den **aktiven** Knoten u mit dem **kleinsten** Wert für td(u)
  - Den Knoten u betrachten wir dann als **gelöst**
  - Für jeden Nachbarn v von u prüfen wir, ob wir v über u schneller erreichen können als bisher = Relaxieren von (u, v)
  - Nächste Runde, bis es keine aktiven Knoten mehr gibt

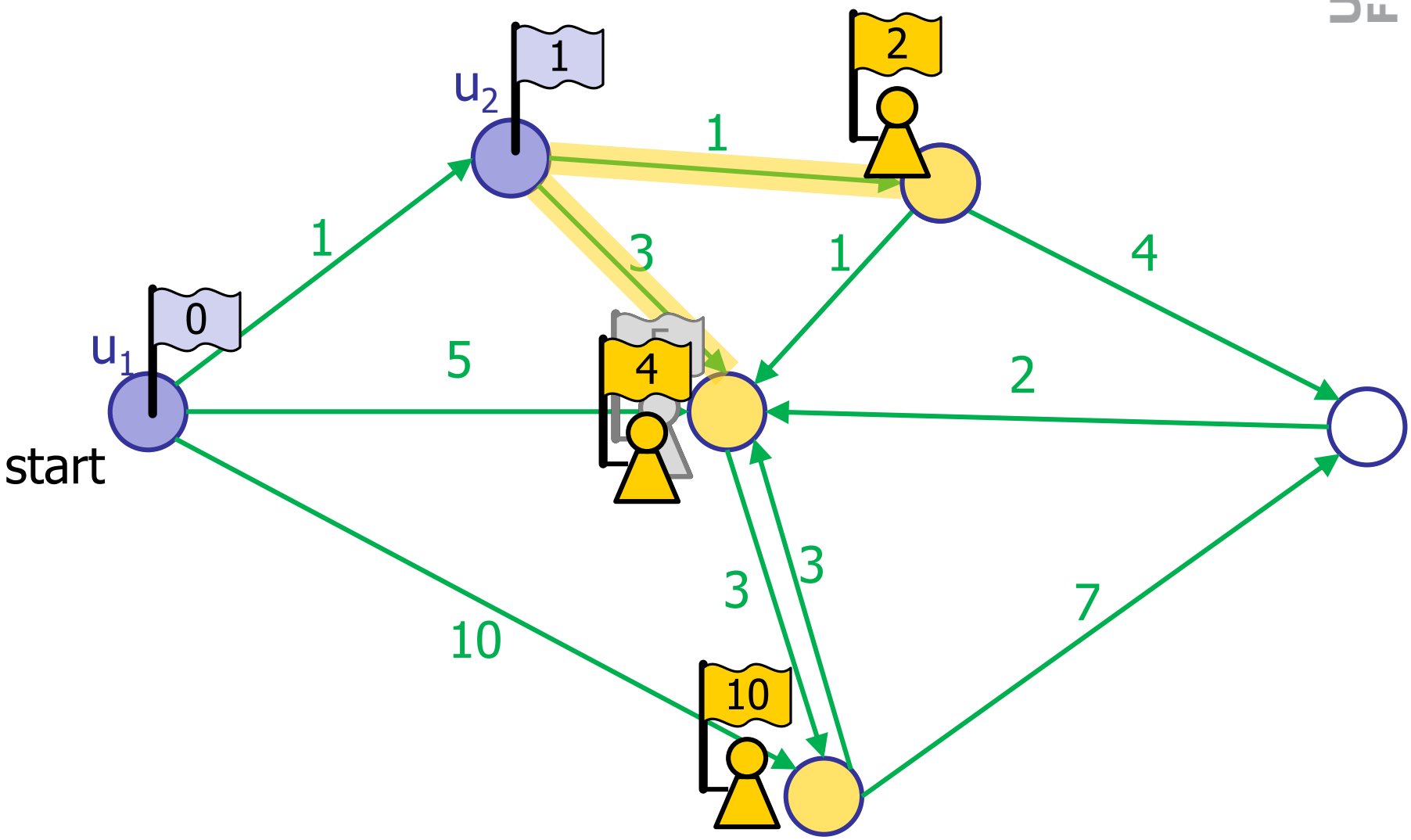
Dijkstras Algorithmus: Start



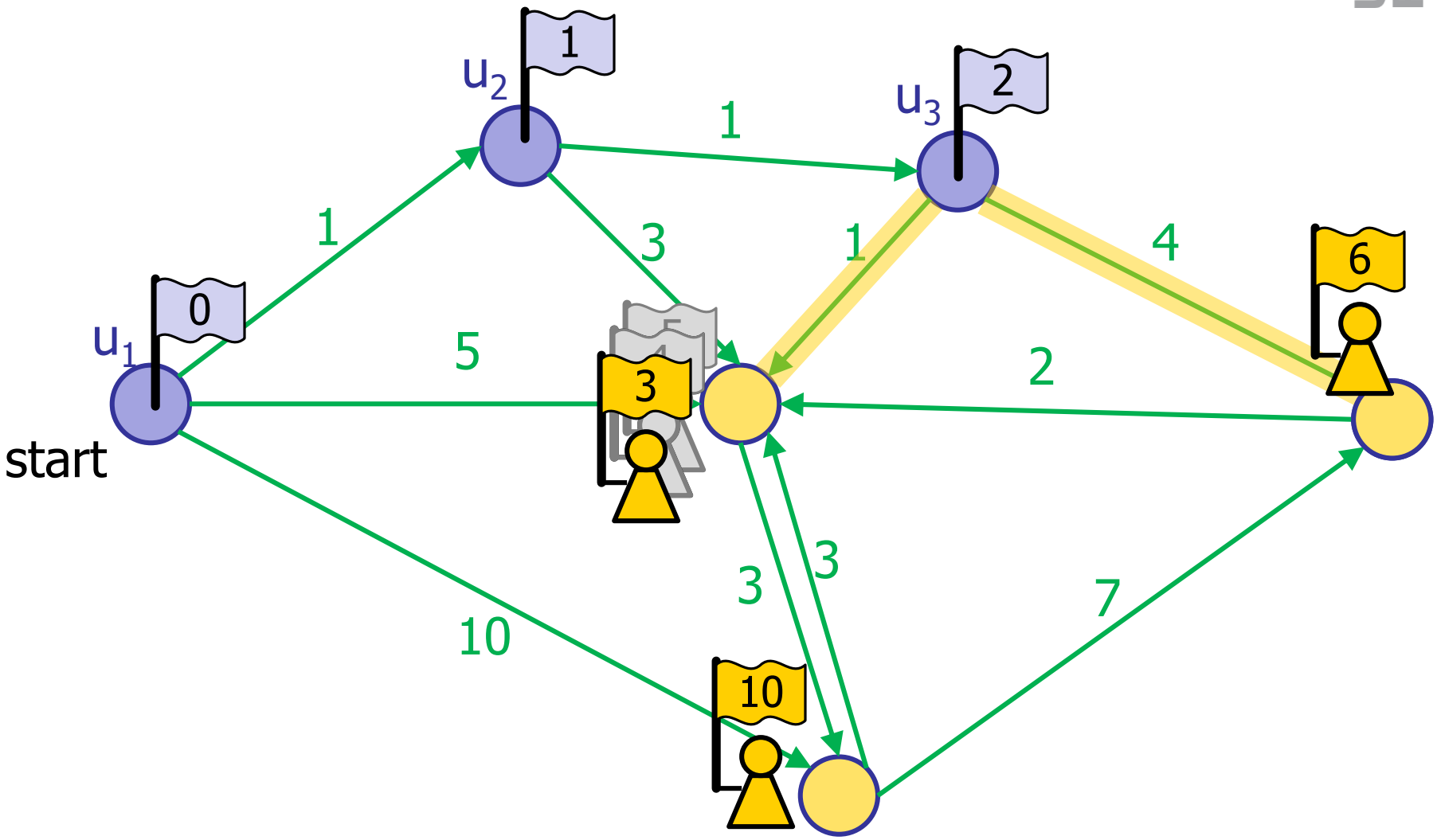
Dijkstras Algorithmus: Runde 1



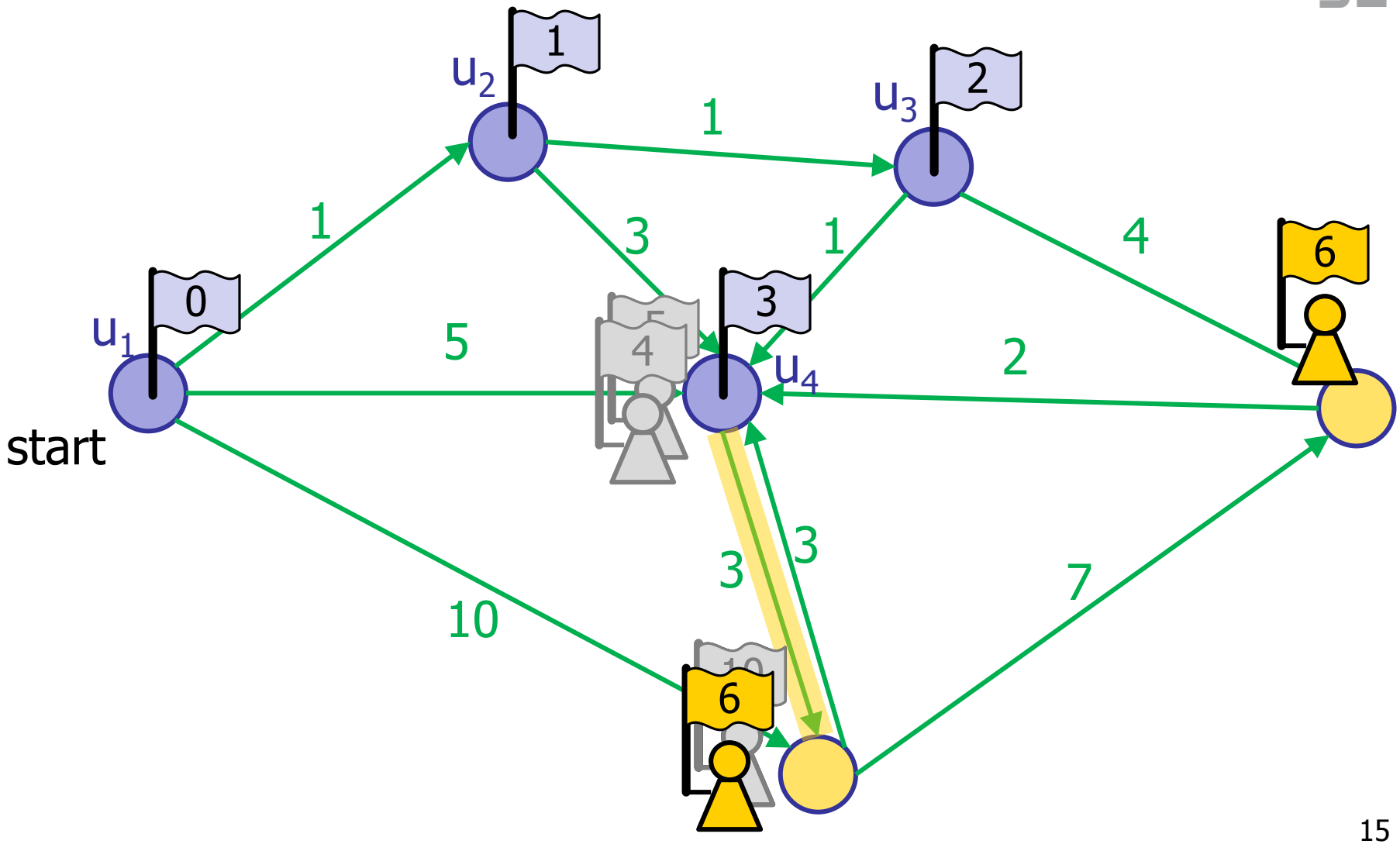
Dijkstras Algorithmus: Runde 2



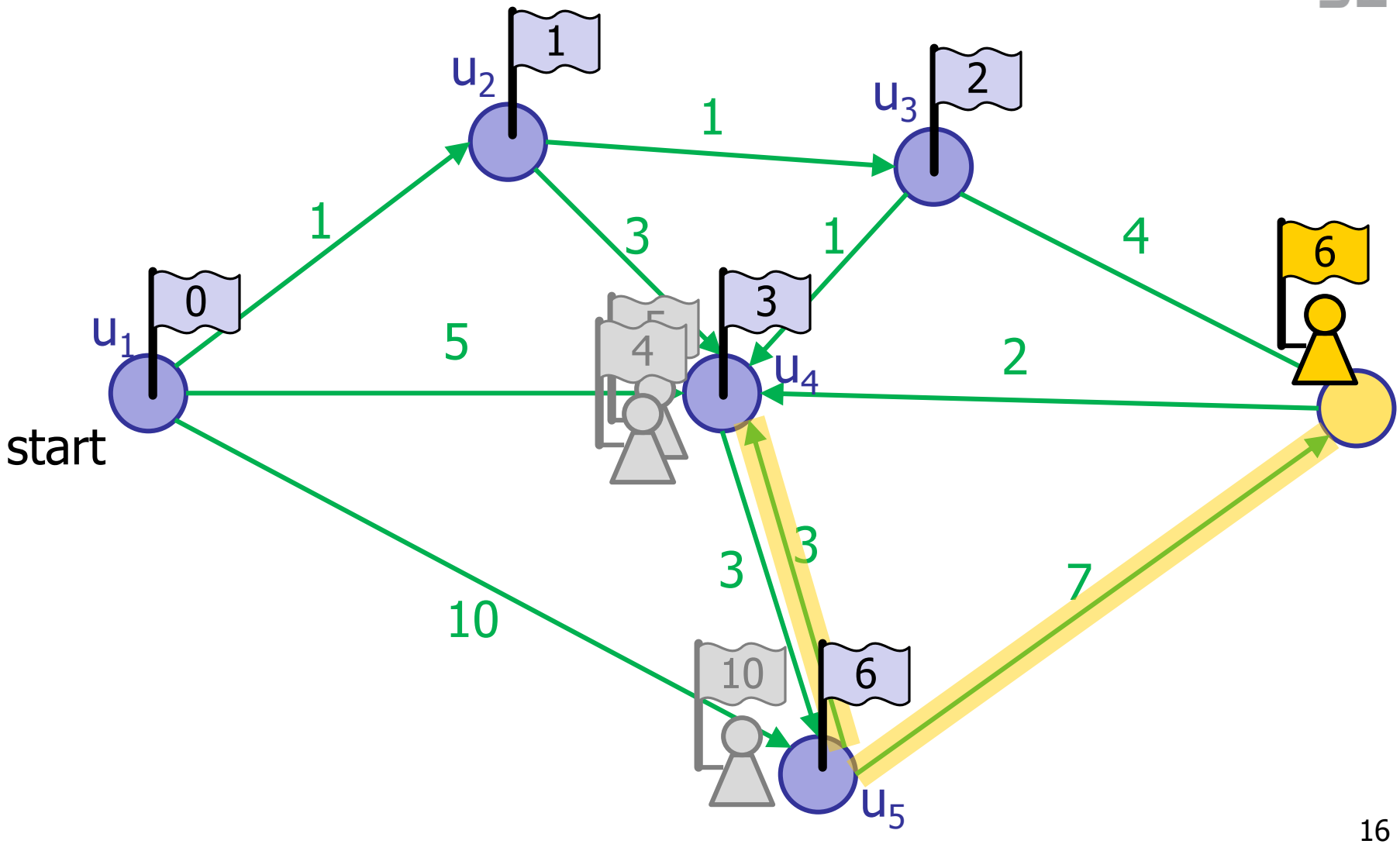
Dijkstras Algorithmus: Runde 3



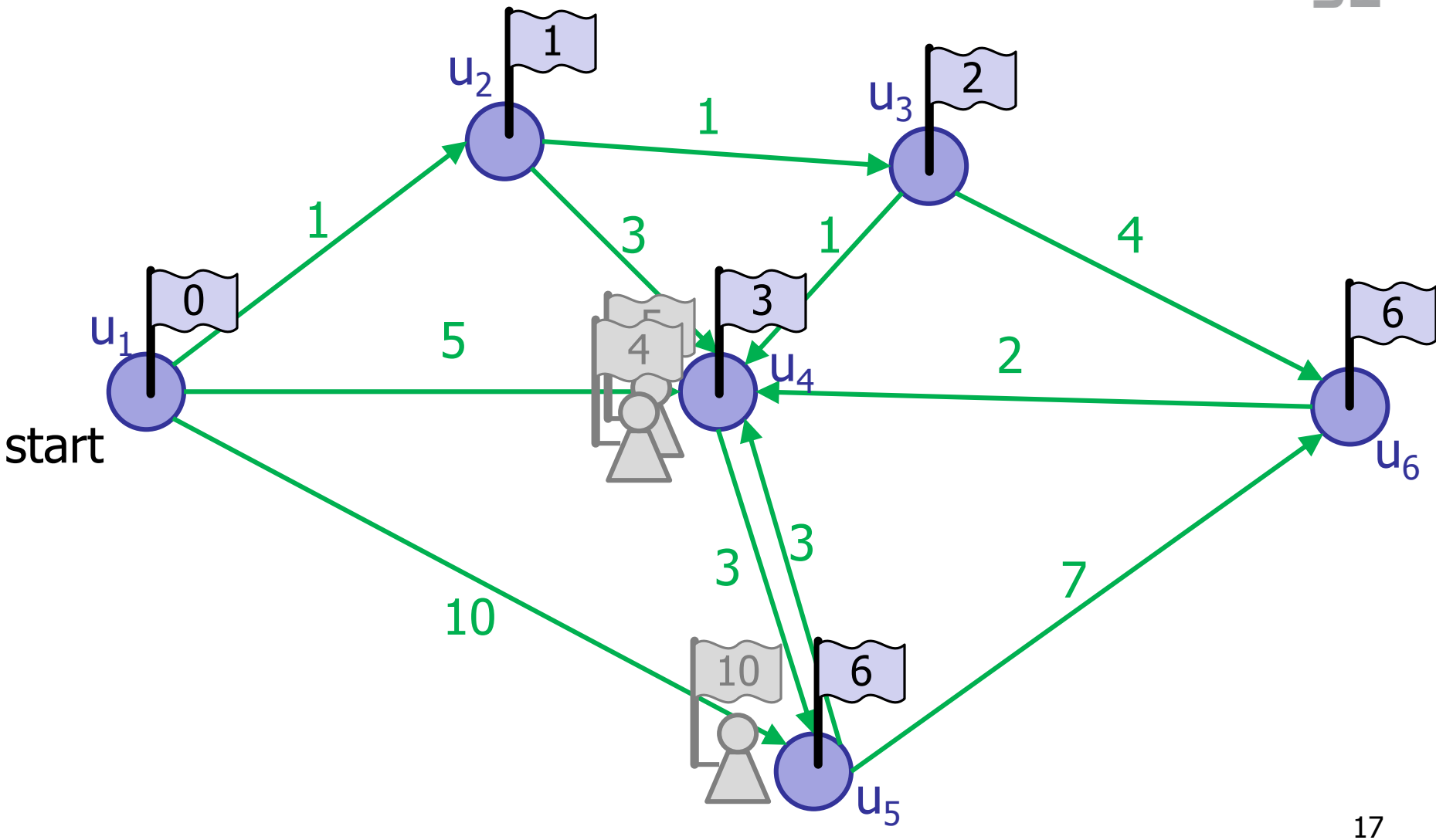
Dijkstras Algorithmus: Runde 4



Dijkstras Algorithmus: Runde 5



Dijkstras Algorithmus: Runde 6



Korrektheitsbeweis 1/3

UNI FREIBURG

- Argumentationslinie
  - **Annahme 1:** Alle Kantenlängen sind > 0, siehe Folie 36
  - **Annahme 2:** Die $dist(s, u)$ sind **alle verschieden** (das war bei dem Beispiel auf der vorherigen Folie nicht der Fall)

    Das erlaubt einen einfacheren und intuitiveren Beweis

    Es geht aber auch ohne, siehe Referenzen … nur bei Interesse
  - Mit A2 gibt es eine Anordnung $u_1, u_2, u_3, \ldots$ der Knoten (wobei $s = u_1$), so dass gilt:

    $$dist(s, u_1) < dist(s, u_2) < dist(s, u_3) < \ldots$$
  - Wir wollen zeigen, dass Dijkstras Algorithmus für jeden Knoten den kürzesten Pfad findet, also am Ende $td(u_i) = dist(s, u_i)$ für jeden Knoten u_i
  - Außerdem zeigen wir, dass die Knoten der Distanz nach gelöst werden, also in der Anordnung $u_1, u_2, u_3, \ldots$ D.h. Knoten u_i wird in Runde i gelöst

Korrektheitsbeweis 1/3

UNI FREIBURG

Zu zeigen: In der i-ten Runde wird Knoten u_i gelöst:

1. Er enthält bereits die korrekte Distanz zum Startknoten, also $td(u_i) = dist(s, u_i)$, und er ist aktiv
2. Von allen aktiven Knoten hat er den kleinsten Wert für td und wird deswegen vom Dijkstra Algorithmus ausgewählt.

Induktionsanfang $i = 1$:

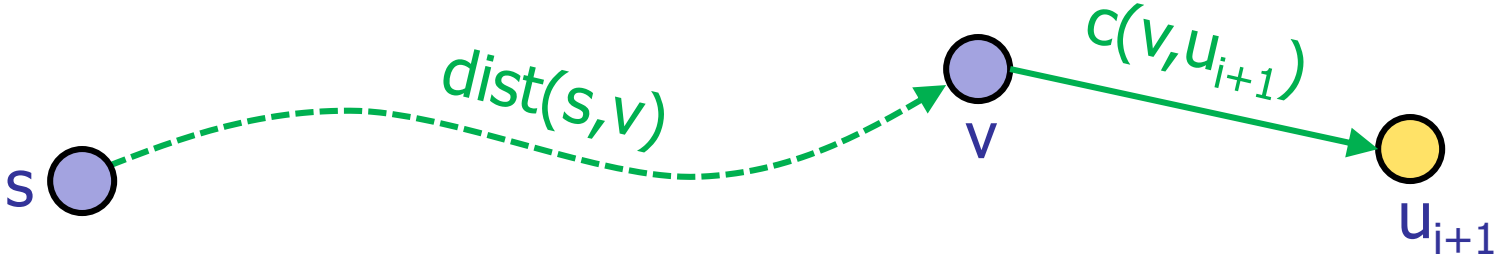
- Zu 1)
  - Am Anfang ist nur der Startknoten $s = u_1$ aktiv und $td(s) = 0$
  - Den löst man dann, und dann ist $td(s) = dist(s,u_1) = 0$
- Zu 2)
  - Es gibt nur diesen einen aktiven Knoten

Korrektheitsbeweis 1/3

UNI FREIBURG

Induktionsschritt (aus 1,...,i folgt i+1)

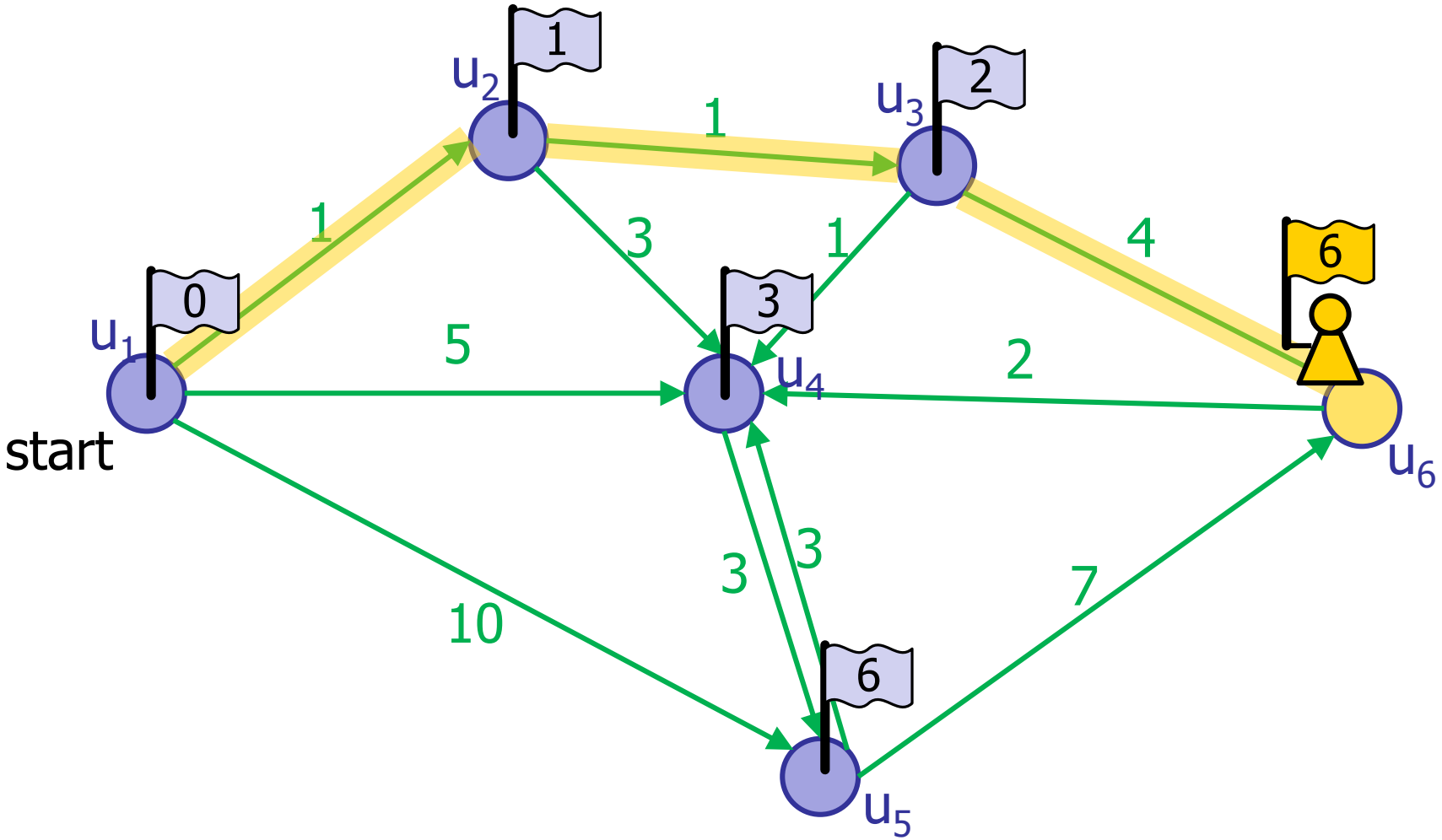
- Zu 1) zu zeigen: Wenn Knoten u_{i+1} an der Reihe ist, enthält er bereits die korrekte Distanz zum Startknoten:
  - Auf dem kürzesten Pfad von s nach u_{i+1} gibt es einen Vorgängerknoten v, so dass gilt:
    $dist(s, u_{i+1}) = dist(s, v) + c(v,u_{i+1})$
    (c sind die Kosten der Kante)



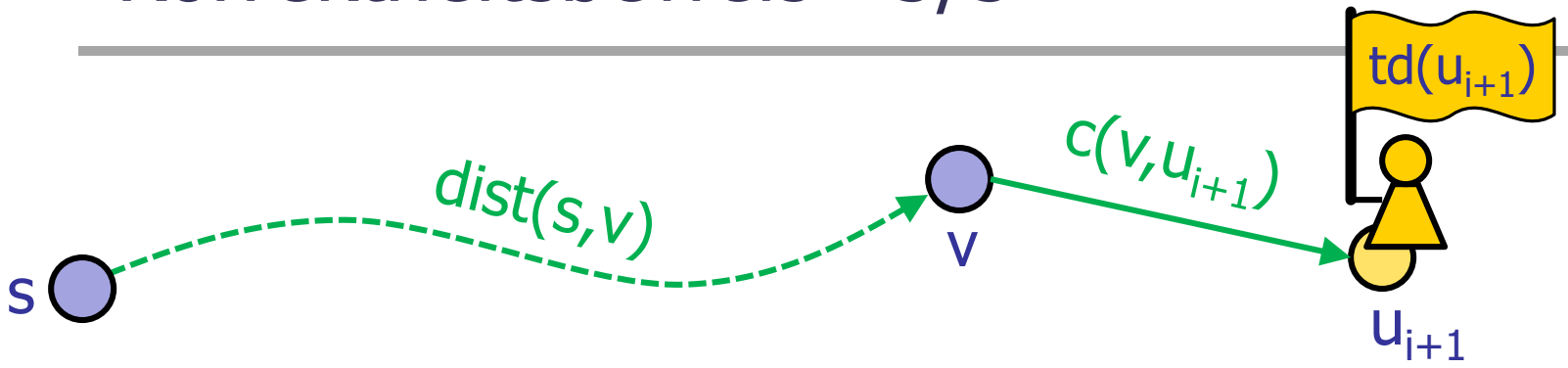
  - somit ist $dist(s,v) < dist(s,u_{i+1})$, da $c > 0$
  - Da u_{i+1} gerade an der Reihe ist, muss v einer der Knoten u_1, ..., u_i sein, also $v = u_j$ mit $1 \leq j < i$

Induktionsschritt Beispiel aus Runde 6

- Vorgänger auf kürzestem Pfad von Knoten u_6 ist Knoten $v = u_3$
- In Runde 3 wurde $td(u_6) = 2 + 4 = 6$ gefunden



Korrektheitsbeweis 3/3



- Nach Induktionsannahme enthält v die richtige Distanz und somit wurde in Runde j der kürzeste Pfad (über Kante v nach u_{i+1}) ausprobiert und steht jetzt in $td(u_{i+1})$
- u_{i+1} ist aktiv, da der Vorgängerknoten gelöst ist.

- Zu 2) zu zeigen: Wenn Knoten u_{i+1} an der Reihe ist, wird er unter allen aktiven Knoten ausgewählt:
  - Alle Knoten mit kleineren dist wurden bereits vorher gelöst
  - Alle anderen Knoten u_k mit $k>i+1$ haben eine größere $dist(s,u_k)$ und somit ist die aktuelle $td(u_k)$ entweder gleich oder größer
  - Somit ist u_{i+1} der Knoten mit der kleinsten td und wird vom Dijkstra Algorithmus ausgewählt □

Implementierungshinweise

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 td(u)
  - Es bietet sich also an, die aktiven Knoten in einer **Priori-tätswarteschlange** zu verwalten, mit Schlüssel td(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

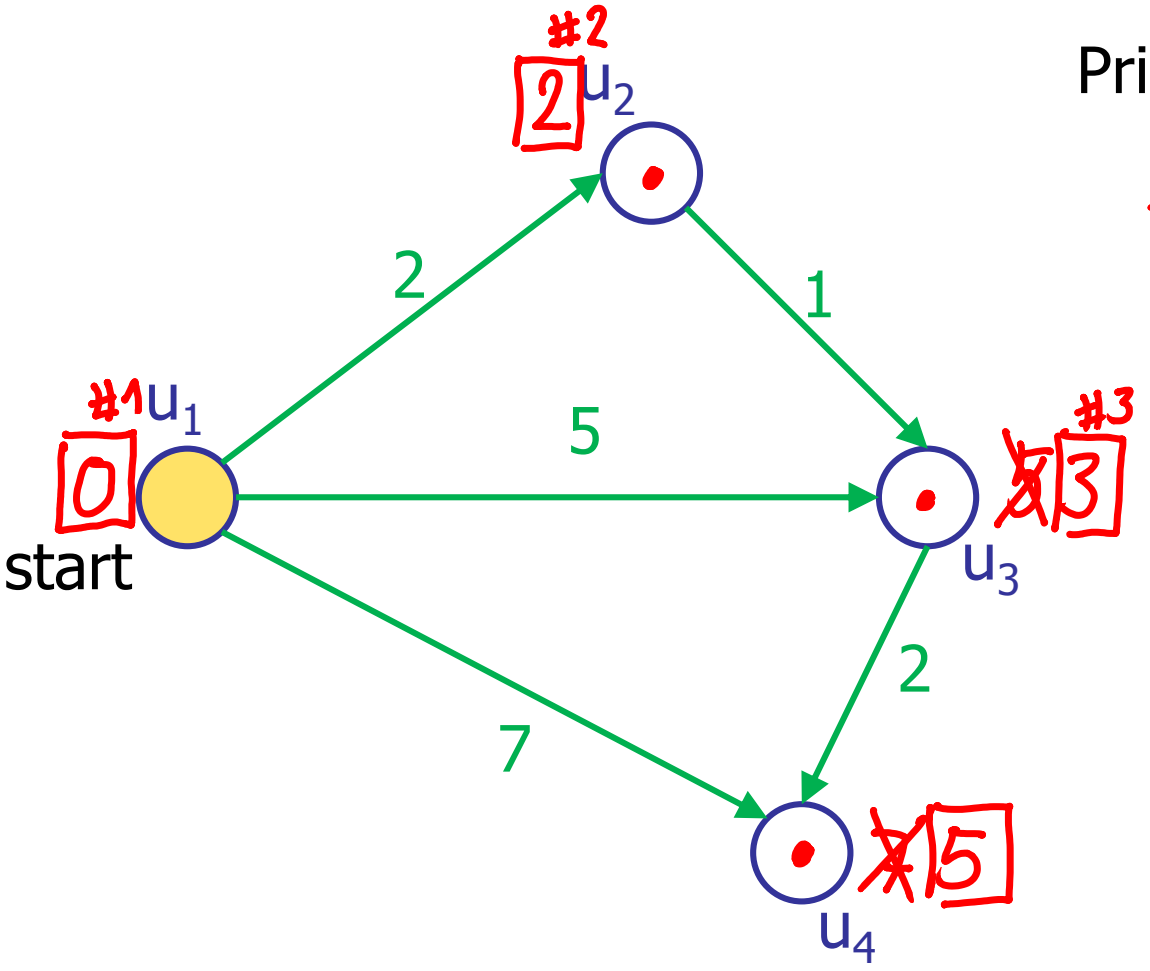
Implementierungshinweise

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**

Beispiel für Prioritätswarteschlange



Prioritätswarteschlange:

~~$(0, u_1)$~~	gelöst #1
~~$(2, u_2)$~~	gelöst #2
~~$(5, u_3)$~~	ignoriert #4
~~$(7, u_4)$~~	ignoriert #5
~~$(3, u_3)$~~	gelöst #3
~~$(5, u_4)$~~	gelöst #4

Laufzeitanalyse

UNI FREIBURG

- Für einen Graph mit n Knoten und m Kanten ($m \geq n$)
  - 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 $O(m \cdot \log m)$
    - ($\log m$, da bis zu m Elemente in der PW sein können)
  - Weil $m \leq n^2$ ist das auch $O(m \cdot \log n)$, da $\log n^2 = 2 \log n$
  - 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

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

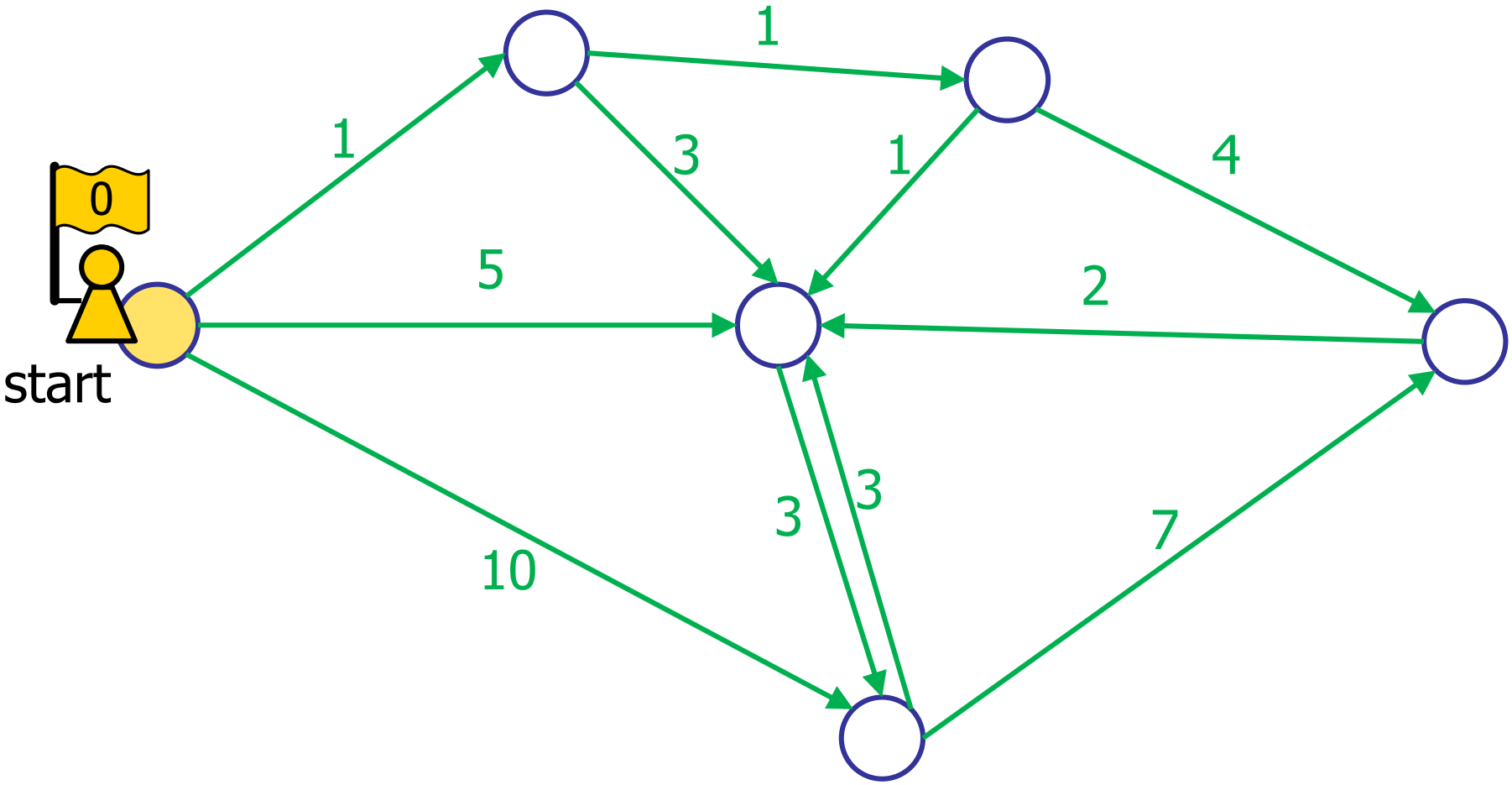
    Intuitiv: erst wenn man **alles** im Umkreis von dist(s, t) um den Startknoten s abgesucht hat, kann man sicher sein, dass es keinen kürzeren Weg zum Ziel t gibt

Weiterführende Kommentare 2/3

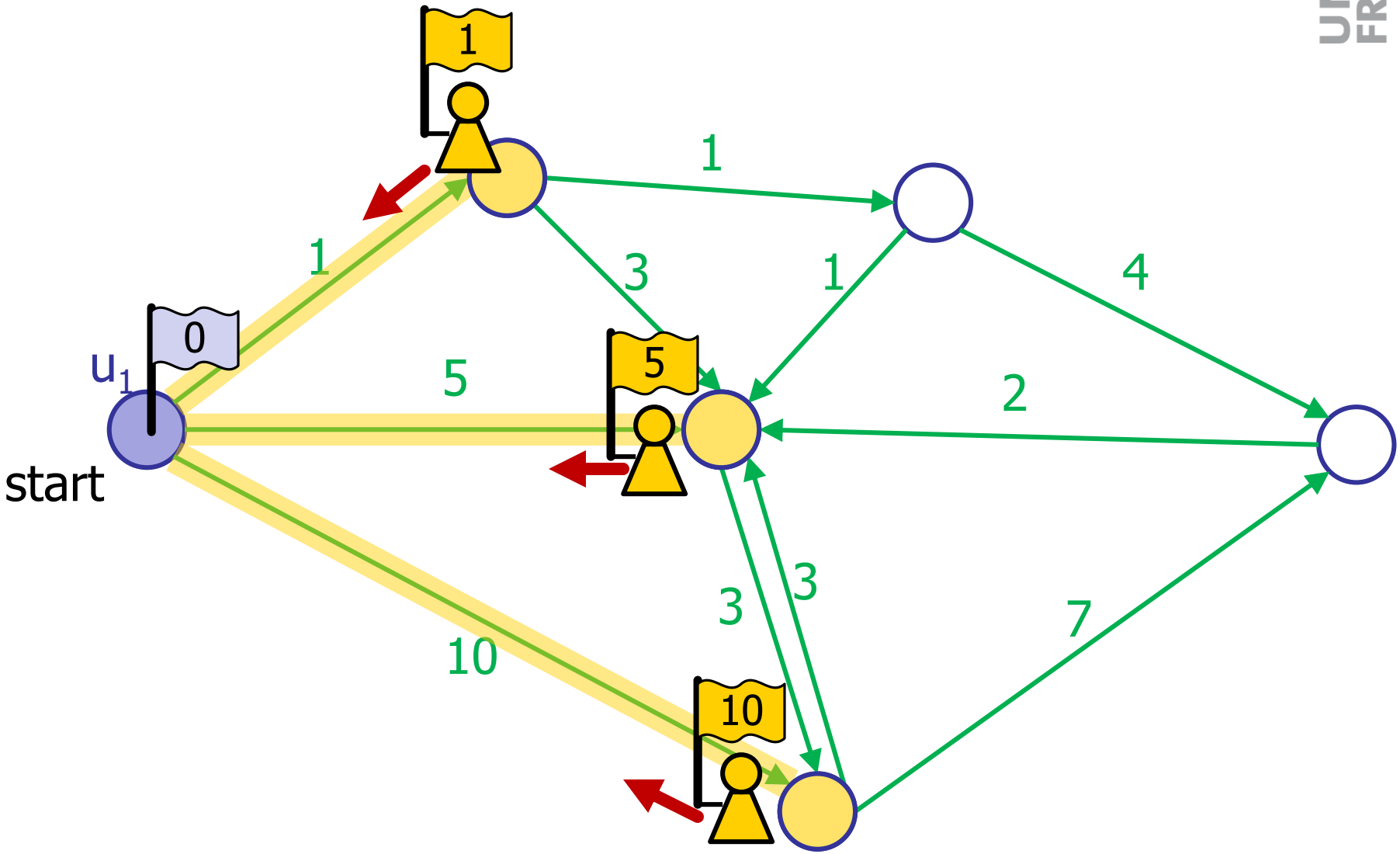
UNI FREIBURG

- 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**

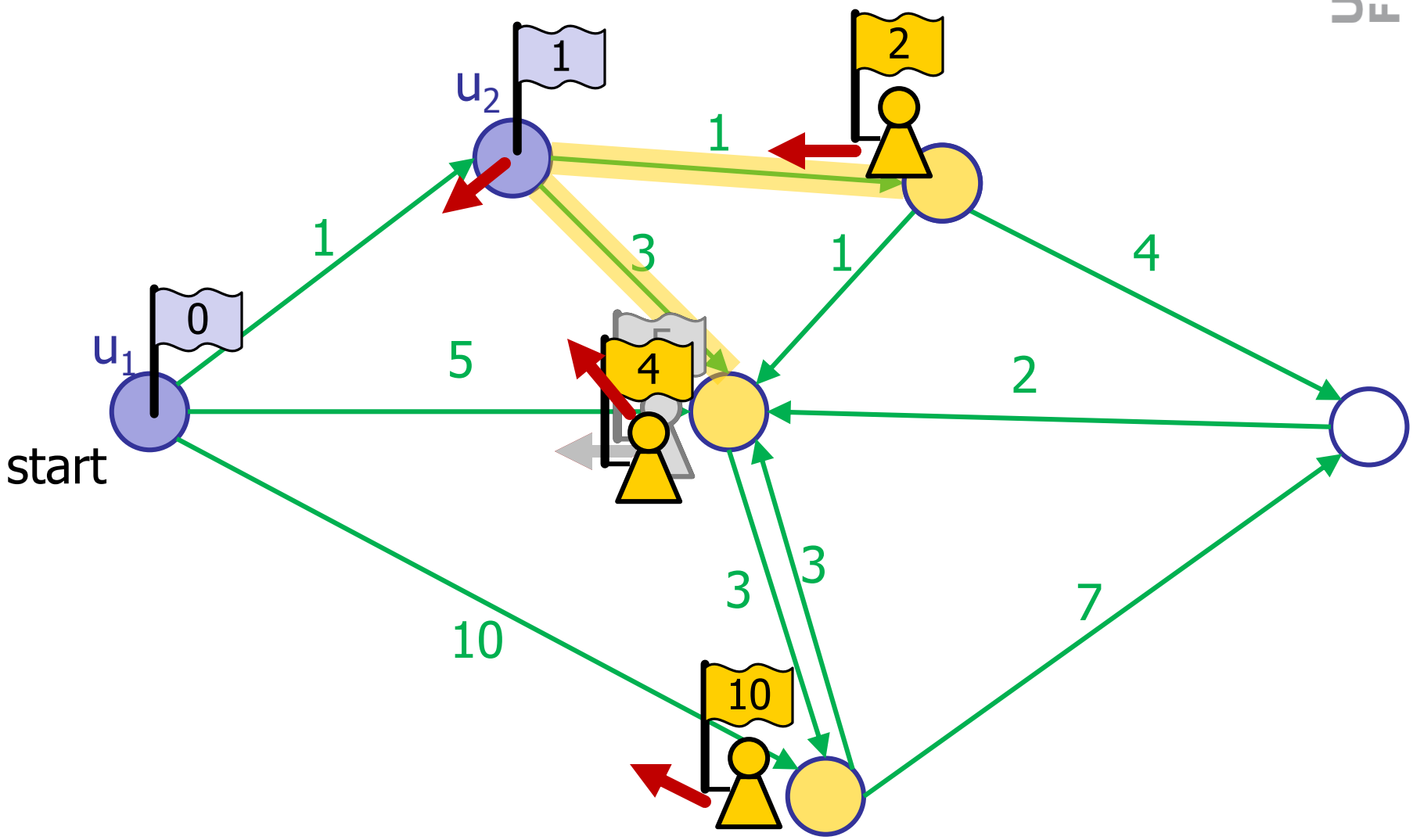
Dijkstras Algorithmus: Start



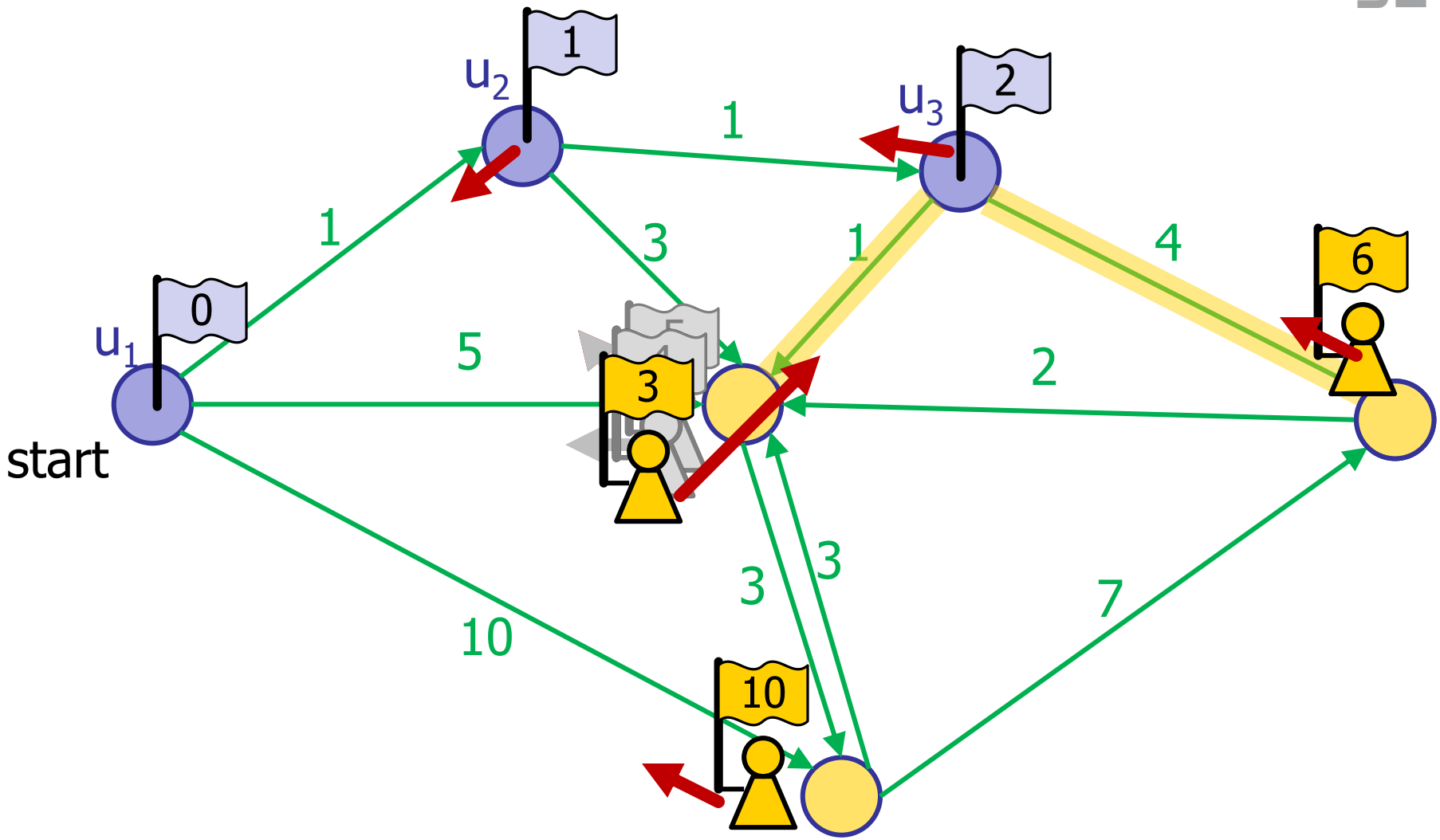
Dijkstras Algorithmus: Runde 1



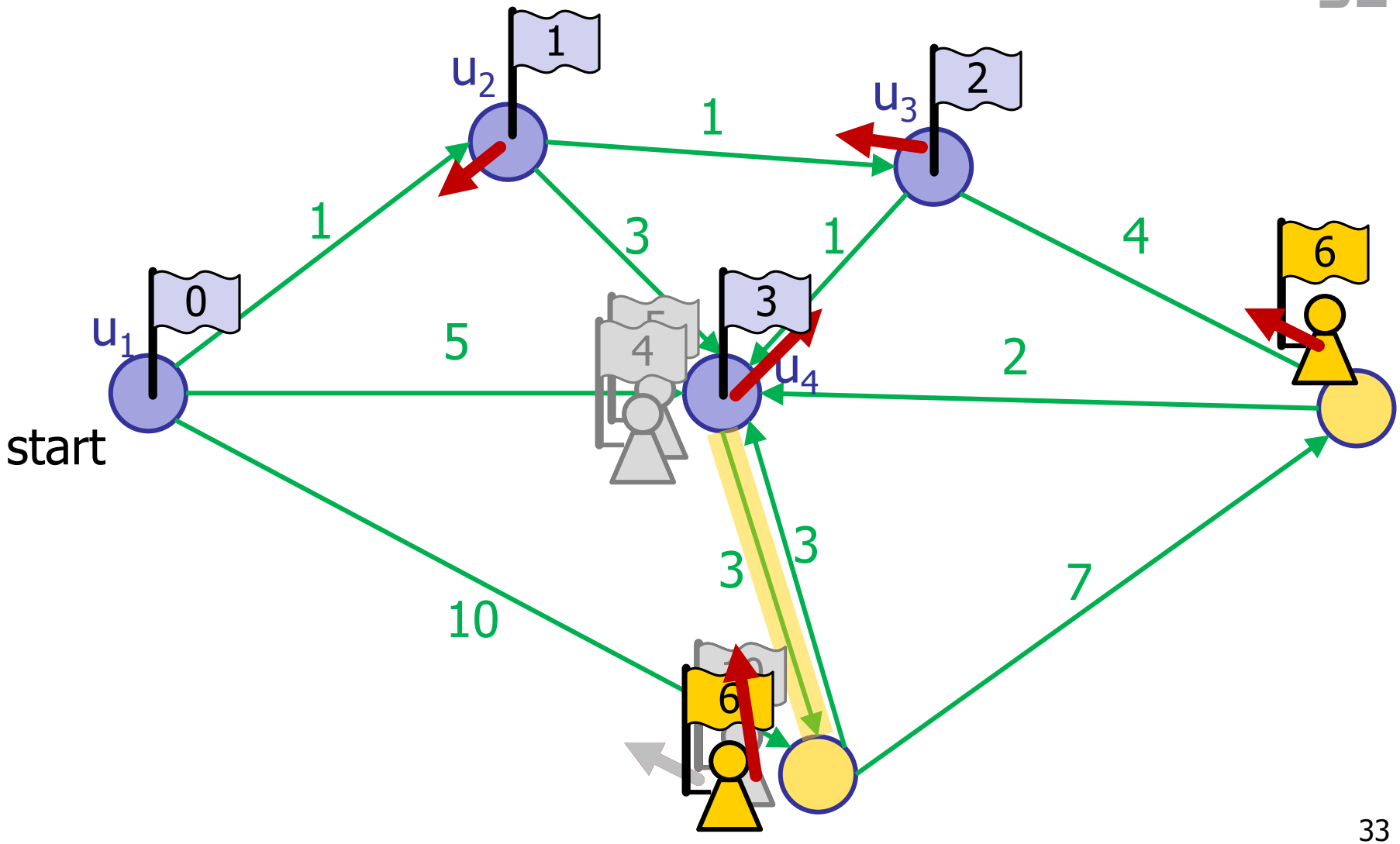
Dijkstras Algorithmus: Runde 2



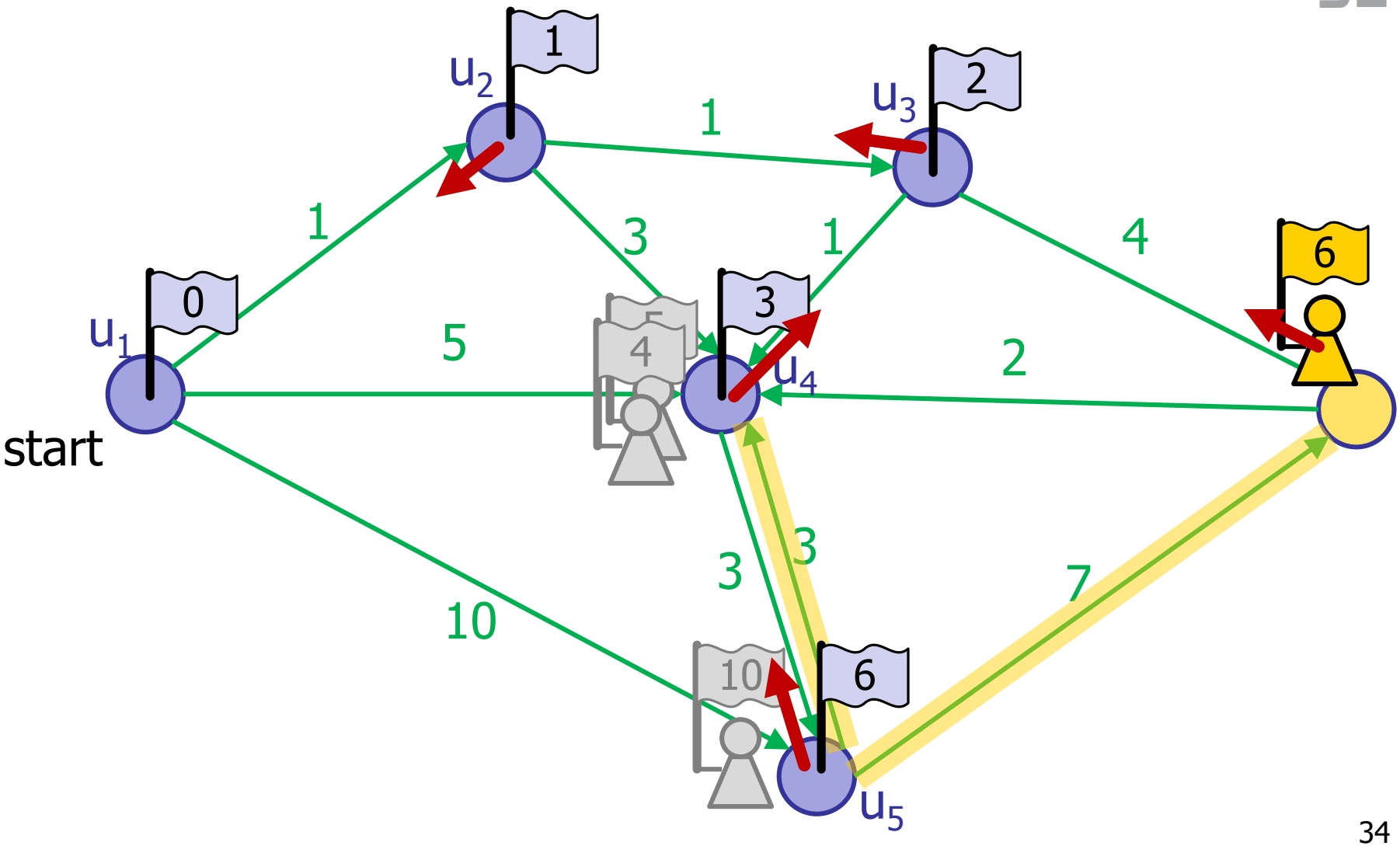
Dijkstras Algorithmus: Runde 3



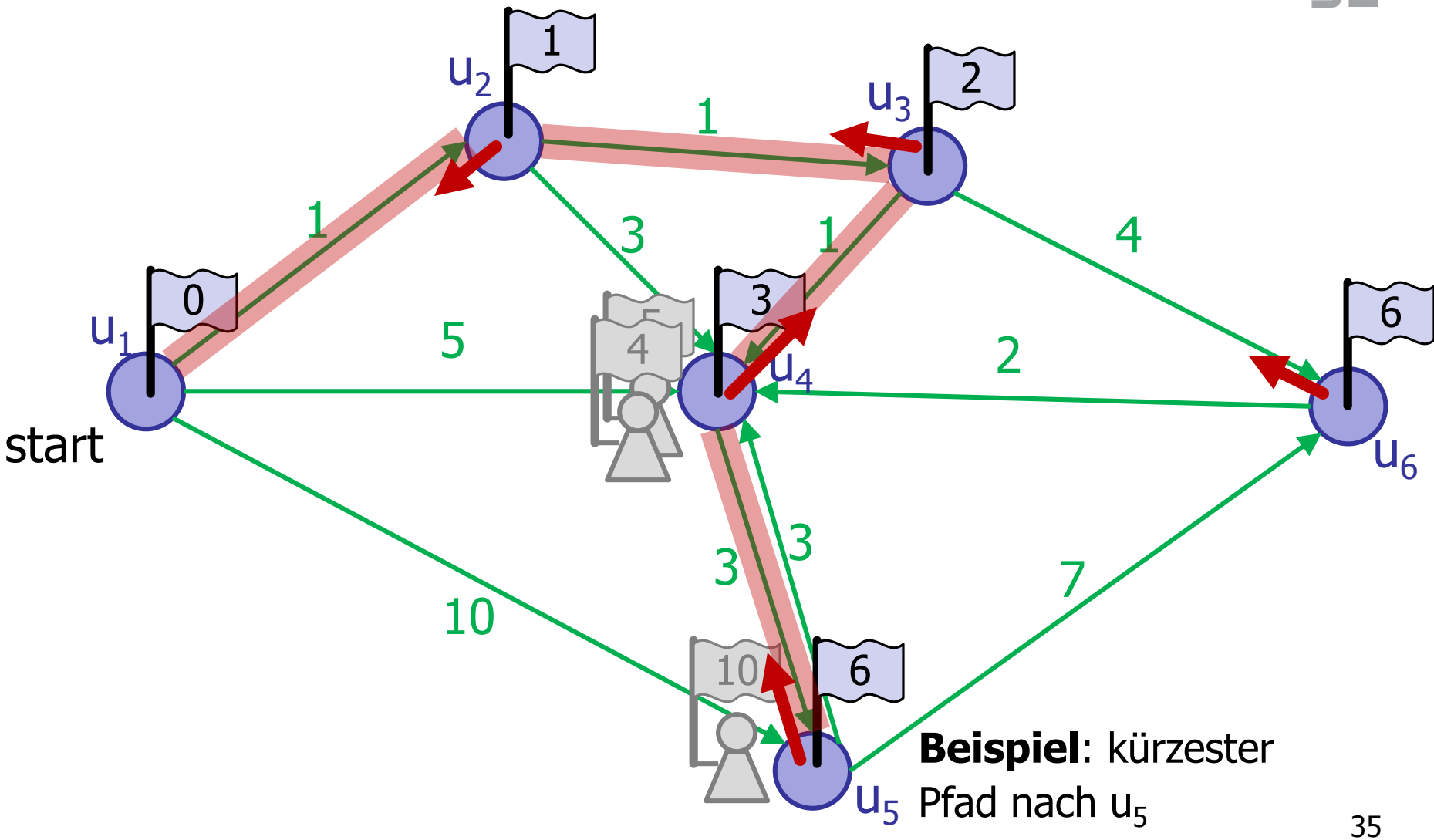
Dijkstras Algorithmus: Runde 4



Dijkstras Algorithmus: Runde 5

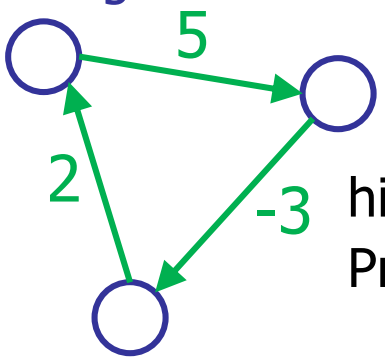


Dijkstras Algorithmus: Runde 6

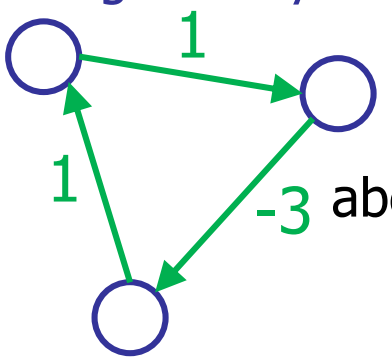


Weiterführende Kommentare 3/3

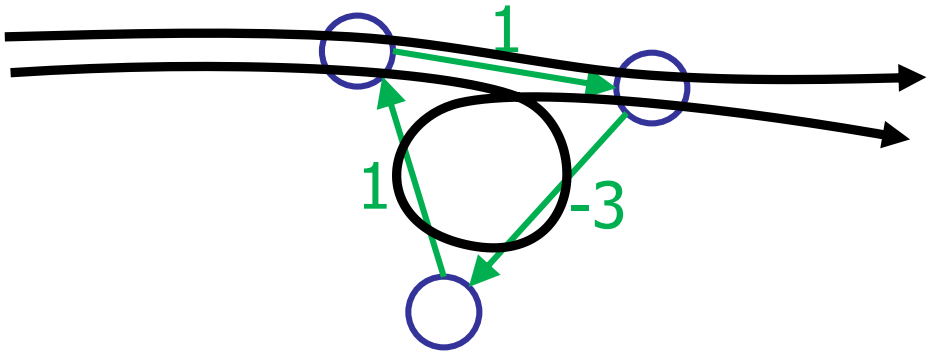
- Erweiterungen
  - In unserem Beweis haben wir benutzt, dass die Kantenlängen alle **nicht-negativ** sind (sogar > 0)
  - Bei negativen Kantenkosten kann es negative Zyklen geben:



hier kein Problem



aber hier



Kein Zyklus: Kosten 1

1 Zyklus: Kosten 0

2 Zyklen: Kosten -1

3 Zyklen: Kosten -2

...

Weiterführende Kommentare 3/3

UNI FREIBURG

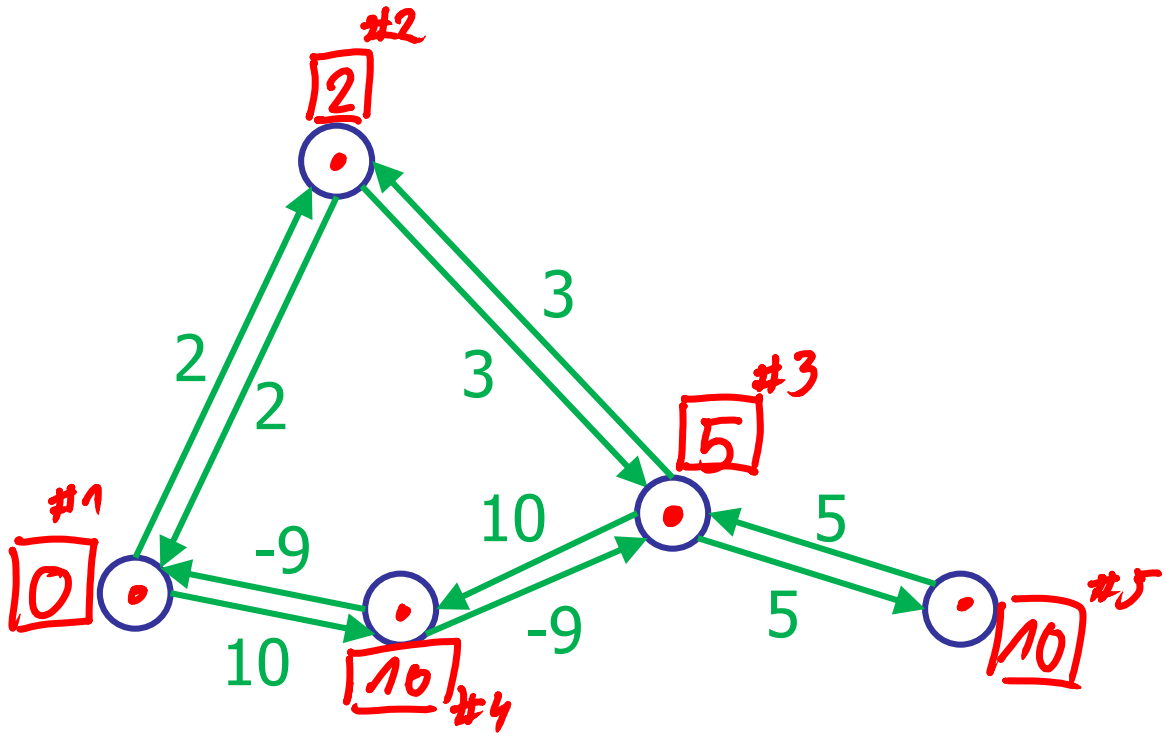
- Erweiterungen
  - um mit negativen Kanten umzugehen braucht man andere Algorithmen
    - Zum Beispiel den Bellman-Ford Algorithmus
    - Wenn der Graph **azyklisch** ist, reicht auch einfach topologisches Sortieren (mit DFS) + Relaxieren der Knoten in der Reihenfolge dieser Sortierung
  - Eine (nicht 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)

Beispiel negative Kosten (E-Auto Verbrauch)

Dijkstra Algorithmus:

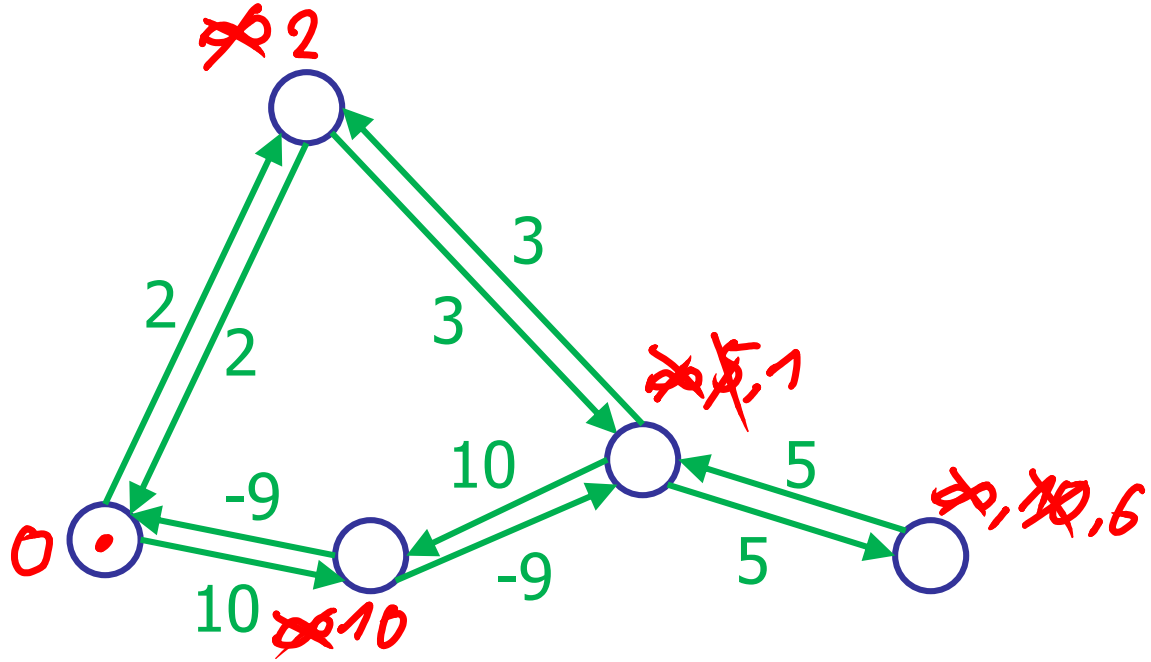
Message passing nur von gelösten Knoten.



Beispiel negative Kosten (E-Auto Verbrauch)

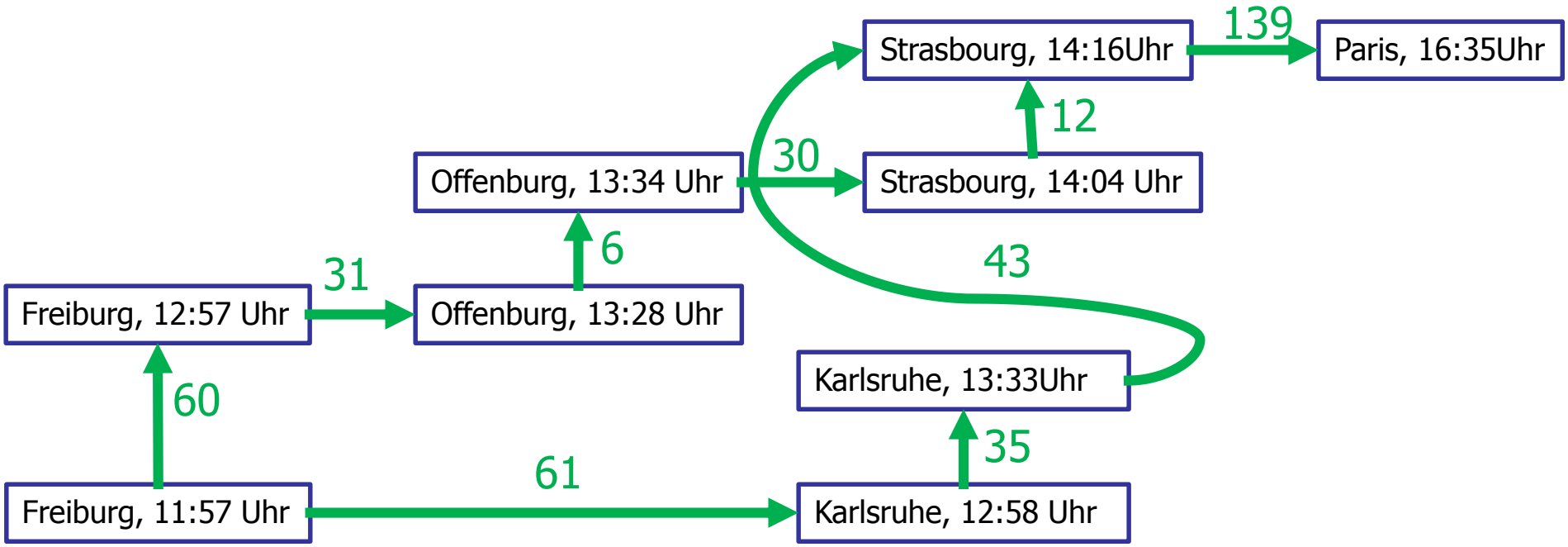
Bellman-Ford Algorithmus:

Message passing von allen Knoten bis sich nichts mehr ändert.

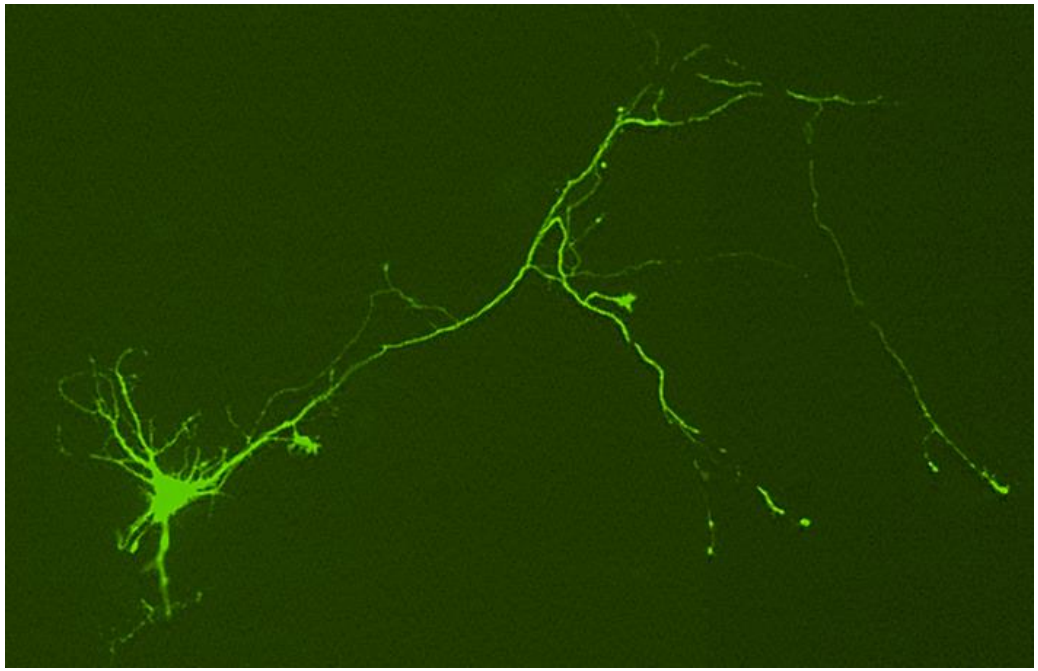


Anwendungen

- Routenplaner für Autofahrten (Übungsblatt)
- Routenplaner für Bahn/Bus-Verbindungen. Wie könnte der Graph dafür aussehen?
  - Am einfachsten Knoten in Raum-Zeit:

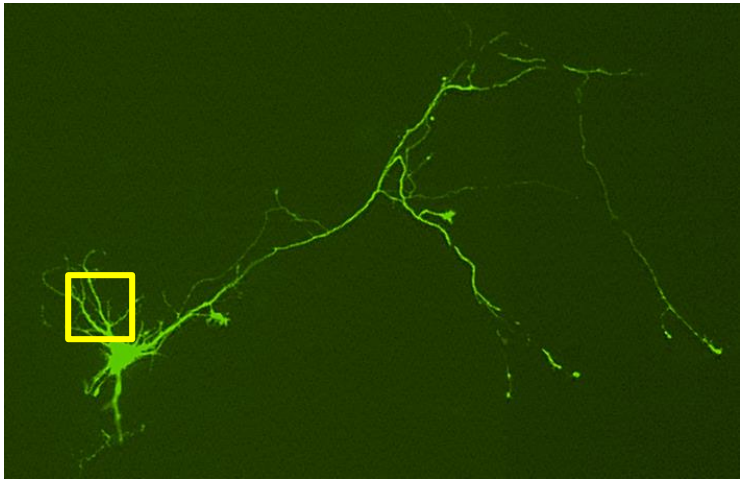


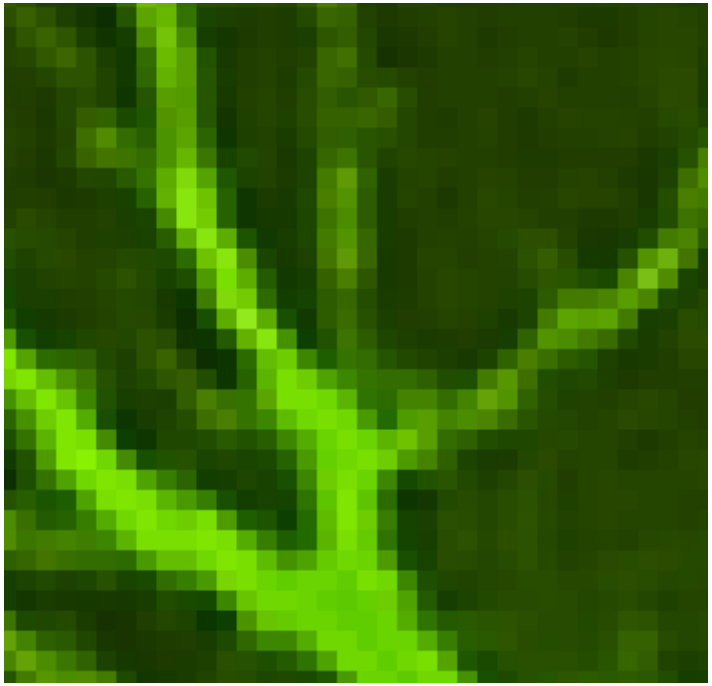
Anwendung in Bildverarbeitung



- Neuronen unter einem Fluoreszenzmikroskop
- Aufgabe: Länge der Axone (das sind die Verbindungen von Neuronen)
- Demo mit ImageJ Plugin NeuronJ
  http://www.imagescience.org/meijering/software/neuronj/

Anwendung: Axone verfolgen





- Bild als Graph: Jedes Pixel ist ein Knoten
- Implizite Kanten: jedes Pixel hat eine Kante zu den 8 Nachbarn … muss nicht gespeichert werden
- Kosten in den Knoten (nicht in den Kanten): helle Pixel kosten wenig, dunkle kosten viel.

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