Week 10 — Sheet 9

Algorithms and Data Structures

27.11.2023 — Georg Hasebe

Debriefing of Submissions

Graph Theory

- Be more formal and rigorous!
- Graphs are mathematical objects like e.g. functions and so if tasked to prove something for graphs fulfilling certain properties it doesn't suffice to think of a few small examples. Instead we have to be more general!
- v_{cut}? Two or more ZHK!
- Ex. 8.1.: Pigeonhole? What is it? Where did you use it?

Feedback

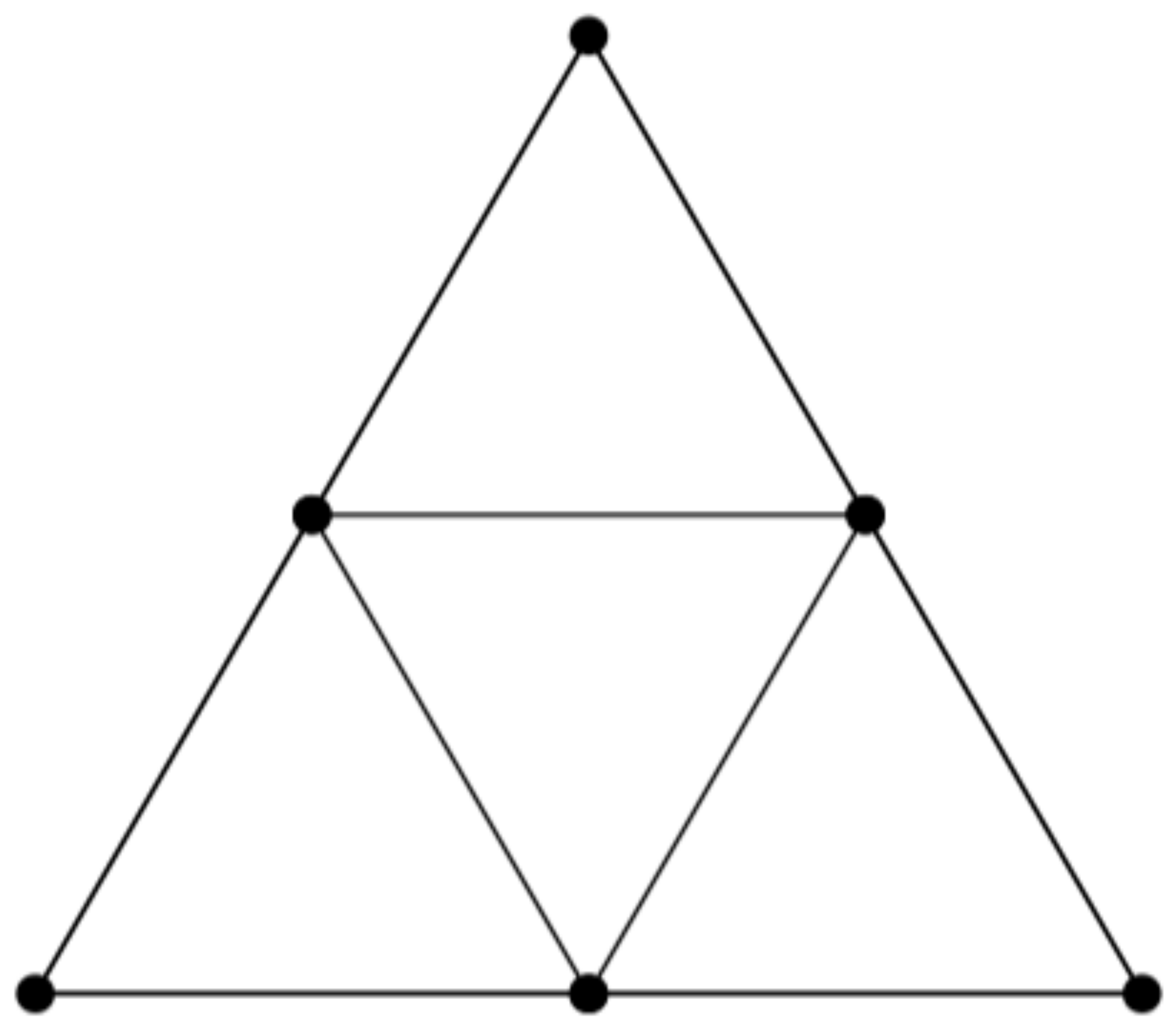
- Not happy with your feedback? Come and talk to me.
- Sometimes more comments, sometimes none.

Proof by induction?

(e) Suppose in a graph G every pair of vertices v, w has a common neighbour (i.e., for all distinct vertices v, w, there is a vertex x such that $\{v, x\}$ and $\{w, x\}$ are both edges). Then there exists a vertex p in G which is a neighbour of every other vertex in G (i.e., p has degree $n - 1$).

~~Proof by induction?~~ Counterexample

(e) Suppose in a graph G every pair of vertices v, w has a common neighbour (i.e., for all distinct vertices v, w, there is a vertex x such that $\{v, x\}$ and $\{w, x\}$ are both edges). Then there exists a vertex p in G which is a neighbour of every other vertex in G (i.e., p has degree $n-1$).



Exercise Sheet 9

Debriefing of Exercise Sheet 9

Theory Recap

Dijkstra’s algorithm

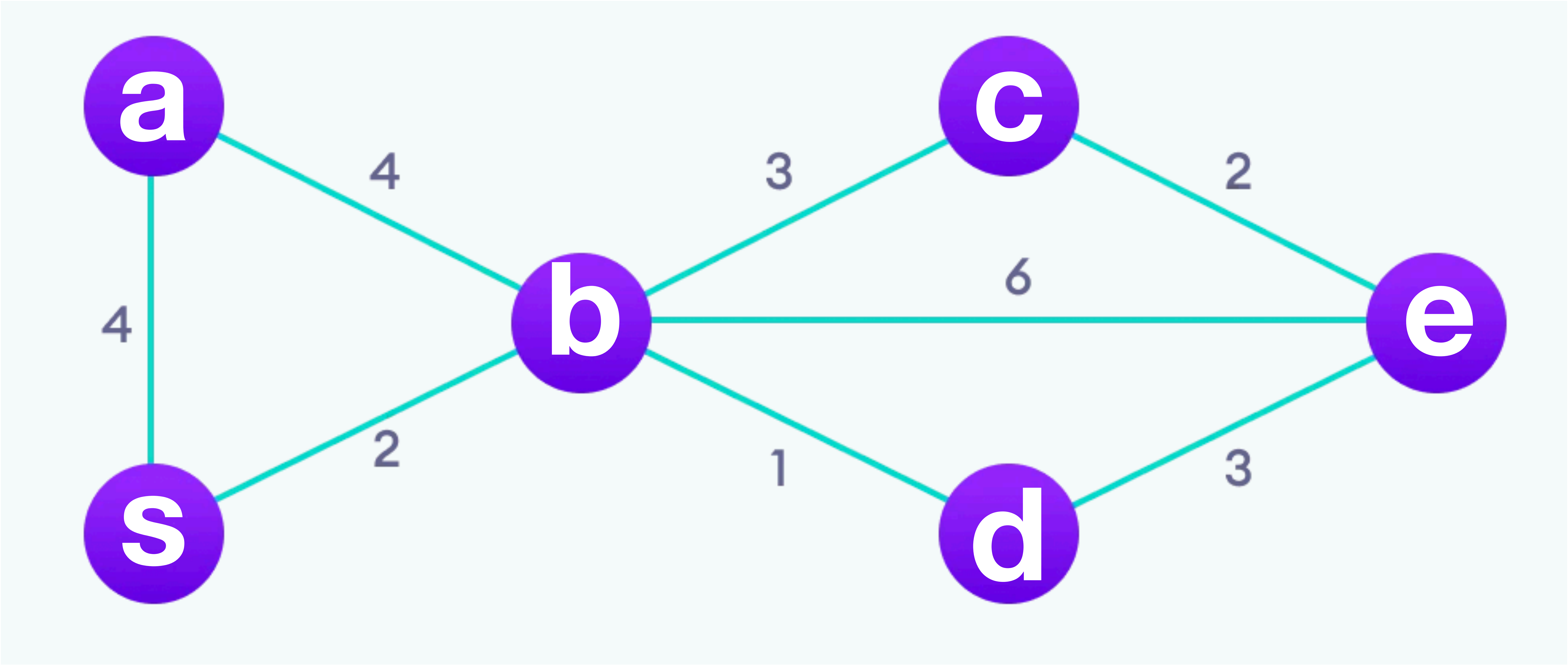
Dijkstra's algorithm

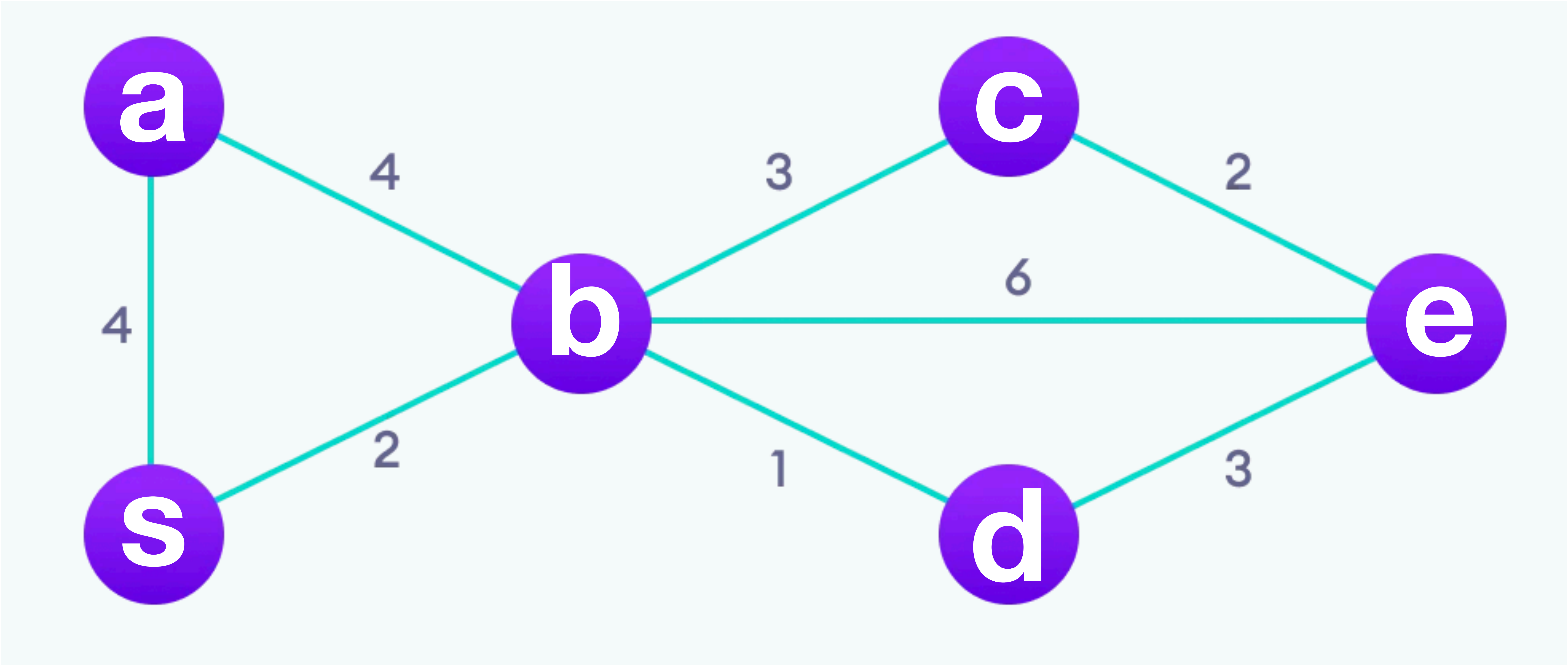
- **We have:** a weighted Graph $G = (V, E)$ with nonnegative weights and a starting vertex $s \in V$.
- **We want:** shortest paths in G starting from s.

Dijkstra’s algorithm

- You can think of Dijkstra’s algorithm as generalizing breadth-first search to weighted graphs.
- A wave emanates from the source, and the first time that a wave arrives at a vertex, a new wave emanates from that vertex.
- Whereas breadth-first search operates as if each wave takes unit time to traverse an edge, in a weighted graph, the time for a wave to traverse an edge is given by the edge’s weight.
- Because a shortest path in a weighted graph might not have the fewest edges, a simple, first-in, first-out queue won’t suffice for choosing the next vertex from which to send out a wave.

Example





To	a	b	c	d	e
Shortest Path from s	4	2	5	3	6

Confusion

Many different variants

```
Dijkstra(s):
d[s] ← 0 , d[v] ← ∞ für v ∈ V \ {s}
S ← ∅     [H ← make-heap(V), decrease-key(H,s,0)]
WHILE S ≠ V:
    wähle v* ∈ V \ S mit d[v*] minimal
    [v* ← extract-min(H)]
    S ← S ∪ {v*}
    FOR (v*,v) ∈ E, v ∉ S:
        d[v] ← min{ d[v], d[v*] + c(v,v*) }
        [decrease-key(H, v, d[v])]
```

Lecture Notes

```
function Dijkstra(Graph, source):
    dist[source] ← 0                           // Initialization

    create vertex priority queue Q

    for each vertex v in Graph.Vertices:
        if v ≠ source
            dist[v] ← INFINITY                 // Unknown distance from source to v
            prev[v] ← UNDEFINED                // Predecessor of v

        Q.add_with_priority(v, dist[v])


    while Q is not empty:                      // The main loop
        u ← Q.extract_min()                    // Remove and return best vertex
        for each neighbor v of u:              // Go through all v neighbors of u
            alt ← dist[u] + Graph.Edges(u, v)
            if alt < dist[v]:
                dist[v] ← alt
                prev[v] ← u
                Q.decrease_priority(v, alt)

    return dist, prev
```

Wikipedia

```
DIJKSTRA(G, w, s)
INITIALIZE-SINGLE-SOURCE(G, s)
S = Ø
Q = Ø
for each vertex u ∈ G.V
    INSERT(Q, u)
while Q ≠ Ø
    u = EXTRACT-MIN(Q)
    S = S ∪ {u}
    for each vertex v in G.Adj[u]
        RELAX(u, v, w)
        if the call of RELAX decreased v.d
            DECREASE-KEY(Q, v, v.d)
```

Introduction to Algorithms Book

```
DIJKSTRA(G = (V, E), s)
for each v ∈ V\{s} do
    d[v] ← ∞; p[v] ← null
d[s] ← 0; p[s] ← null
Q ← ∅
INSERT(s, 0, Q)
while Q ≠ ∅ do
    u ← EXTRACT-MIN(Q)
    for each (u, v) ∈ E do
        if p[v] = null then
            d[v] ← d[u] + w((u, v))
            p[v] ← u
            ENQUEUE(v, d[v], Q)
        else if d[u] + w((u, v)) < d[v] then
            d[v] ← d[u] + w((u, v))
            p[v] ← u
            DECREASE-KEY(v, d[v], Q)
```

Lecture Script

What should I learn?

What should I learn for theory?

- Work with the lecture notes and the script, since that’s what you are examined on.
- Additional material can often help with understanding in case the before mentioned material is confusing, but you should generally never use things/ theorems/runtimes etc. that we didn’t cover in lectures.

Dijkstra

Pseudocode

Runtime:

$O((|V|+|E|)\log|V|)$

(see script for proof; or see next slide which I found to be more understandable)

```
DIJKSTRA(G = (V, E), s)

for each v ∈ V\{s} do                           ▷ Initialisiere für alle Knoten die
    d[v] ← ∞; p[v] ← null                       ▷ Distanz zu s sowie Vorgänger
d[s] ← 0; p[s] ← null                           ▷ Initialisierung des Startknotens
Q ← ∅                                           ▷ Leere Prioritätswarteschlange Q
INSERT(s, 0, Q)                                 ▷ Füge s zu Q hinzu
while Q ≠ ∅ do
    u ← EXTRACT-MIN(Q)                          ▷ Aktueller Knoten
    for each (u, v) ∈ E do                      ▷ Inspiziere Nachfolger
        if p[v] = null then                     ▷ v wurde noch nicht entdeckt
            d[v] ← d[u] + w((u, v))             ▷ Berechne obere Schranke
            p[v] ← u                            ▷ Speichere u als Vorgänger von v
            ENQUEUE(v, d[v], Q)                 ▷ Füge v zu Q hinzu
        else if d[u] + w((u, v)) < d[v] then   ▷ Kürzerer Weg zu v entdeckt
            d[v] ← d[u] + w((u, v))             ▷ Aktualisiere obere Schranke
            p[v] ← u                            ▷ Speichere u als Vorgänger von v
            DECREASE-KEY(v, d[v], Q)            ▷ Setze Priorität von v herab
```

Dijkstra’s Runtime Analysis

```
INITIALIZE-SINGLE-SOURCE(G, s)

for each vertex v ∈ G.V
    v.d = ∞
    v.π = NIL
s.d = 0
```

```
RELAX(u, v, w)

if v.d > u.d + w(u, v)
    v.d = u.d + w(u, v)
    v.π = u
```

with a binary min-heap that includes a way to map between vertices and their corresponding heap elements. Each EXTRACT-MIN operation then takes $O(\lg V)$ time. As before, there are $|V|$ such operations. The time to build the binary min-heap is $O(V)$. (As noted in Section 21.2, you don’t even need to call BUILD-MIN-HEAP.) Each DECREASE-KEY operation takes $O(\lg V)$ time, and there are still at most $|E|$ such operations. The total running time is therefore $O((V + E) \lg V)$, which is $O(E \lg V)$ in the typical case that $|E| = \Omega(V)$. This running time improves

Introduction to Algorithms, 22.3 Dijkstra’s Algorithm Analysis

Q eingefügt und genau einmal aus Q entfernt wird. Wurde ein Knoten u aus Q entfernt, dann wird er niemals wieder in Q eingefügt, und sein Wert $d[u]$ wird niemals wieder verändert. Der Grund ist, dass für alle später aus Q entfernten Knoten x der Wert $d[x]$ mindestens so gross wie $d[u]$ ist; folglich ist der Vergleich in Schritt 13 niemals erfüllt. Da jeder

Lecture Script

```
DIJKSTRA(G, w, s)
INITIALIZE-SINGLE-SOURCE(G, s)
S = Ø
Q = Ø
for each vertex u ∈ G.V
    INSERT(Q, u)
while Q ≠ Ø
    u = EXTRACT-MIN(Q)
    S = S ∪ {u}
    for each vertex v in G.Adj[u]
        RELAX(u, v, w)
        if the call of RELAX decreased v.d
            DECREASE-KEY(Q, v, v.d)
```

Introduction to Algorithms, 22.3 Dijkstra’s Algorithm Analysis

Example on Blackboard

```
DIJKSTRA(G = (V, E), s)
for each v ∈ V\{s} do                              ▷ Initialisiere für alle Knoten die
    d[v] ← ∞; p[v] ← null                          ▷ Distanz zu s sowie Vorgänger
d[s] ← 0; p[s] ← null                              ▷ Initialisierung des Startknotens
Q ← ∅                                              ▷ Leere Prioritätswarteschlange Q
INSERT(s, 0, Q)                                    ▷ Füge s zu Q hinzu
while Q ≠ ∅ do
    u ← EXTRACT-MIN(Q)                             ▷ Aktueller Knoten
    for each (u, v) ∈ E do                         ▷ Inspiziere Nachfolger
        if p[v] = null then                        ▷ v wurde noch nicht entdeckt
            d[v] ← d[u] + w((u, v))                ▷ Berechne obere Schranke
            p[v] ← u                               ▷ Speichere u als Vorgänger von v
            ENQUEUE(v, d[v], Q)                    ▷ Füge v zu Q hinzu
        else if d[u] + w((u, v)) < d[v] then       ▷ Kürzerer Weg zu v entdeckt
            d[v] ← d[u] + w((u, v))                ▷ Aktualisiere obere Schranke
            p[v] ← u                               ▷ Speichere u als Vorgänger von v
            DECREASE-KEY(v, d[v], Q)               ▷ Setze Priorität von v herab
```

What should I learn for CodeExpert?

- The variants differ mostly in the usage of the data structure storing the vertices
  - Normal queue, priority queue, start with all vertices in the queue, only insert the start vertex in the queue at the start, decrease key, re-insert vertices…

Java’s PriorityQueue

OVERVIEW PACKAGE **CLASS** USE TREE DEPRECATED INDEX HELP

PREV CLASS **NEXT CLASS** FRAMES NO FRAMES ALL CLASSES

SUMMARY: NESTED | FIELD | CONSTR | METHOD DETAIL: FIELD | CONSTR | METHOD

compact1, compact2, compact3
java.util

Class PriorityQueue<E>

java.lang.Object
  java.util.AbstractCollection<E>
    java.util.AbstractQueue<E>
      java.util.PriorityQueue<E>

Type Parameters:

`E - the type of elements held in this collection`

All Implemented Interfaces:

`Serializable`, `Iterable<E>`, `Collection<E>`, `Queue<E>`

All Methods | **Instance Methods** | **Concrete Methods**

Modifier and Type	Method and Description
`boolean`	`add(E e)` Inserts the specified element int
`void`	`clear()` Removes all of the elements fro
`Comparator<? super E>`	`comparator()` Returns the comparator used to
`boolean`	`contains(Object o)` Returns `true` if this queue conta
`Iterator<E>`	`iterator()` Returns an iterator over the ele
`boolean`	`offer(E e)` Inserts the specified element int
`E`	`peek()` Retrieves, but does not remove,
`E`	`poll()` Retrieves and removes the head
`boolean`	`remove(Object o)` Removes a single instance of the
`int`	`size()` Returns the number of elements
`Spliterator<E>`	`spliterator()` Creates a ***late-binding*** and *fail*
`Object[]`	`toArray()` Returns an array containing all
`<T> T[]`	`toArray(T[] a)` Returns an array containing all

No decrease key method?🤨

https://docs.oracle.com/javase/8/docs/api/java/util/PriorityQueue.html

Re-Insertion instead decrease key

- Instead of decreasing the key, we could also just reinsert the node again (allowing for the same node appearing multiple times in the priority queue) but with the decreased priority.
- Questions that naturally arise are: How does this re-insertion affect the runtime? How does it change the pseudocode and concrete implementation?

Re-insertion runtime analysis (informally)

- We add nodes when we reach them through an edge (inner for loop). Thus we have at most $|E|$ insert operations.
- The while loop breaks only if the queue is empty. Thus we also have $|E|$ dequeue operations (extract and delete from queue).
- Denoting the time it takes for an insert operation with T_i and the time it takes for an dequeue operation with T_d we get $O(|E| \cdot T_i + |E| \cdot T_d)$.
- If n is the size of a heap, we already learned that both dequeue and insert is in $O(\log n)$ (recall of repair heap).
- Now what is the size of the heap? Since we have at most $|E|$ insert operations that is also our size.

Re-insertion runtime analysis (informally)

- Both T_i and T_d are in $O(\log|E|)$, thus $O(|E| \cdot T_i + |E| \cdot T_d)$ = $O(|E| \cdot \log|E|)$
- Notice that for simple connected graphs $O(\log|E|) = O(\log|V|)$ since $|V| - 1 \leq |E| \leq \binom{|V|}{2} \leq |V|^2$
- Therefore we have $O(|E| \cdot \log|E|)$ = $O(|E| \cdot \log|V|)$ for simple connected graphs

Re-insertion vs. decrease key runtime

- The re-insertion version has actually been found to achieve faster computing times in practice, which this paper shows:

Priority Queues and Dijkstra's Algorithm *

Mo Chen † Rezaul Alam Chowdhury ‡ Vijaya Ramachandran §
David Lan Roche ¶ Lingling Tong ‖

UTCS Technical Report TR-07-54

October 12, 2007

1.2 Summary of Experimental Results

Briefly here are the conclusions of our experimental study:

- During in-core computations involving real-weighted sparse graphs such as undirected $\mathcal{G}_{n,m}$, road networks, and directed power-law, implementations based on DIJKSTRA-NODEC ran faster (often significantly) than DIJKSTRA-DEC implementations. However, this performance gap narrowed as the graphs became denser.

https://www3.cs.stonybrook.edu/~rezaul/papers/TR-07-54.pdf

Java Implementation (from my github repo)

```java
public int[] dijkstra(ArrayList<ArrayList<Edge>> G, int n, int start) {
    int[] P = new int[n]; // store predecessors of nodes
    int[] D = new int[n]; // store distance from start to nodes

    for (int i = 0; i < n; ++i) D[i] = Integer.MAX_VALUE;
    D[start] = 0;

    PriorityQueue<Node> PQ = new PriorityQueue<>();
    PQ.add(new Node(start, 0));

    while (!PQ.isEmpty()) {
        Node u = PQ.poll();

        if (D[u.key] < u.dist) continue; // already found shorter distance

        for (Edge edge : G.get(u.key)) {
            int v = edge.to;
            int d = D[u.key] + edge.weight;

            if (d < D[v]) {
                D[v] = d;
                P[v] = u.key;
                PQ.add(new Node(v, D[v]));
            }
        }
    }

    return D;
}
```