

# Dijkstra-Algorithmus

## 1 Grundidee

Der **Dijkstra-Algorithmus** berechnet die **kürzesten Wege** von einem Startknoten zu allen anderen Knoten in einem **gewichteten Graphen mit nicht-negativen Kantengewichten**.

- Greedy-Algorithmus
  - Baut die kürzesten Distanzen schrittweise auf
  - Verwendet typischerweise eine **Prioritätswarteschlange (Min-Heap)**
- 

## 2 Voraussetzungen

- **Gewichteter Graph**
  - **Keine negativen Kantengewichte**
  - Startknoten ist bekannt
- 

## 3 Laufzeiten & Eigenschaften

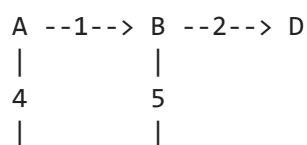
Eigenschaft	Wert
Laufzeit (Heap)	$O((V + E) \log V)$
Laufzeit (Matrix)	$O(V^2)$
Speicherbedarf	$O(V)$
Kürzester Weg	ja
Negative Kanten	 nicht erlaubt

**Hinweis:** Bei negativen Gewichten liefert Dijkstra **falsche Resultate**.

---

## 4 Schritt-für-Schritt-Beispiel

Graph (gerichtet, gewichtet):



V                      V  
C --1--> E

Startknoten: **A**

## Initialisierung

Distanz(A) = 0  
Distanz(B,C,D,E) =  $\infty$

## Ablauf (vereinfacht)

1. A → aktualisiere B=1, C=4
2. B → aktualisiere D=3, E=6
3. D → keine Verbesserung
4. C → aktualisiere E=5

Endresultat:

A: 0  
B: 1  
C: 4  
D: 3  
E: 5

---

## 5 Besonderheiten / Prüfungsrelevante Hinweise

- Dijkstra ist ein **Greedy-Algorithmus**
- Ein Knoten wird „fest“ markiert, sobald seine kürzeste Distanz bekannt ist
- Sehr häufige Prüfungsfrage:
  - *Warum funktionieren negative Kanten nicht?*

---

## 6 Vor- und Nachteile

### Vorteile

- effizient für viele Graphen
- liefert exakte kürzeste Wege
- weit verbreitet (Routing, Netzwerke)

### Nachteile

- keine negativen Kantengewichte

- komplexer als BFS

## Merksatz für die Prüfung

*Dijkstra berechnet kürzeste Wege in gewichteten Graphen ohne negative Kanten mithilfe eines Greedy-Ansatzes.*

## Python-Implementierung

```
In [1]: import heapq

def dijkstra(graph, start):
    # graph: dict {node: [(neighbor, weight), ...]}
    distances = {node: float("inf") for node in graph}
    distances[start] = 0

    pq = [(0, start)] # (distanz, knoten)

    while pq:
        current_dist, current_node = heapq.heappop(pq)

        if current_dist > distances[current_node]:
            continue

        for neighbor, weight in graph[current_node]:
            distance = current_dist + weight
            if distance < distances[neighbor]:
                distances[neighbor] = distance
                heapq.heappush(pq, (distance, neighbor))

    return distances

# Beispiel
graph = {
    "A": [("B", 1), ("C", 4)],
    "B": [("D", 2), ("E", 5)],
    "C": [("E", 1)],
    "D": [],
    "E": []
}

print(dijkstra(graph, "A"))

{'A': 0, 'B': 1, 'C': 4, 'D': 3, 'E': 5}
```