Algorithmen und Datenstrukturen

Greedy-Algorithmen:

Münzproblem ♦ Kürzeste Pfade

Einführendes Beispiel:
Münzproblem

Problemstellung

Name: **Münzproblem**

Eingabe: eine ganze Zahl W, die einen Geldwert repräsentiert (in Cent)

Ausgabe: Möglichst kleine Menge von Münzen, deren Wert W ist.

Strategie

- Beginne mit der leeren Menge *M.*
  Sei *s* die Summe der Werte der in *M* enthaltenen Münzen.
- Wähle immer eine Münze mit möglichst großem Wert *v*.
  - Falls $s + v \leq W$, dann füge die Münze *M* hinzu und aktualisiere *s*.
  - Sonst versuche die Münze mit nächst kleinerem Wert.
- Stoppe, falls $s + v = W$.

➢ *Probiere die Münzen also in der Reihenfolge*
2 Euro, 1 Euro, 50 Cent, 20 Cent, 10 Cent, 5 Cent, 2 Cent, 1 Cent

Beispiel

***W* = 392** (für 3,92 Euro)

	s
2 Euro + 2 Euro	200
1 Euro + 1 Euro	300
50 Cent + 50 Cent	350
20 Cent + 20 Cent + 20 Cent	390
10 Cent	
5 Cent	
2 Cent	392

Formulierung als Optimierungsproblem

Eingabe: (endliche) Menge von Münzen, ganze Zahl W

Ausgabe: *kleinste* Teilmenge der Münzen,
so dass folgende Kriterien erfüllt sind:

1. ***Gültigkeitskriterium:***
   Summe der Münzwerte $\leq W$

2. ***Optimalitätskriterium:***
   Summe der Münzwerte maximal.

Kanonischer Greedy-Algorithmus

Verallgemeinerung

Probleme, auf die sich Greedy-Algorithmen anwenden lassen:

- Bestimmung einer (kleinsten) Menge von Elementen aus einer gegebenen Menge (*hier: Vorrat an Münzen*)
- so dass eine Gültigkeitsbedingung erfüllt wird und
- eine Zielfunktion optimiert (*hier: maximiert*) wird.

Allgemeine Greedy-Strategie

- Ordne die Elemente der gegebenen Menge geeignet an (*hier: absteigende Münzwerte*).
  $e_1, e_2, \ldots, e_n$
- Beginne mit der leeren Menge.
- Für $1 \leq k \leq n$: Teste, ob e_k zusammen mit den bisher ausgewählten Elementen eine Lösung ergeben kann (die Gültigkeitsbedingung erfüllt, *hier:* $s + v \leq W$)
  - Dann füge e_k zur Menge hinzu.
  - Sonst wird e_k verworfen.

Konkretisierung

- Es ist aus einer endlichen Menge E auszuwählen.
- Die Elemente von E besitzen Werte (*Gewichte*).
- Es ist eine minimale Teilmenge von E zu finden,
  - die ein Gültigkeitskriterium erfüllt und
  - deren Gesamtgewicht maximal (*minimal*) ist.

Formalisierung: Teilmengensysteme

- Sei E eine <u>endliche</u> Menge und $\mathcal{U}$ ein System von Teilmengen von E.
  - E: enthält Werte, die ausgewählt werden können
  - $\mathcal{U}$: enthält zulässige Lösungsmengen (Gültigkeitskriterium)
- Das Paar $(E, \mathcal{U})$ heißt **Teilmengensystem.**
- Sei $w: E \longrightarrow \mathbb{R}$ eine Gewichtsfunktion.

  Für $M \subseteq E$: $w(M) = \sum_{e \in M} w(e)$
- **Aufgabe:** Ermittlung einer minimalen Teilmenge von E,
  - die in $\mathcal{U}$ enthalten ist und
  - deren Gewichtsfunktion maximal (*oder* minimal) ist.

Kanonischer Greedy-Algorithmus

Fall: Teilmenge mit maximalem Gesamtgewicht

Ordne die Elemente in E nach absteigendem Gewicht:

$w(e_1) \geq w(e_2) \geq \ldots \geq w(e_n)$

$M \longleftarrow \varnothing$

Für k von 1 bis n

  Falls $M \cup \{e_k\} \in \mathcal{U}$

    $M \longleftarrow M \cup \{e_k\}$

Gib M aus

Fall: Teilmenge mit minimalem Gesamtgewicht

analog mit $w(e_1) \leq w(e_2) \leq \ldots \leq w(e_n)$

… am Beispiel Münzproblem

- Man hat von jedem Münzwert zwei Münzen im Portemonnaie, aber sogar drei 20 Cent-Münzen; *geordnet:*
  $E = \{ e_1, \quad e_2, \quad e_3, \quad e_4, \; e_5, \; e_6, \; e_7, \; e_8, \; e_9, e_{10}, e_{11}, \ldots, \; e_{14}, \ldots, e_{17}\}$
  $w(e_i)$: 200, 200, 100, 100, 50, 50, 20, 20, 20, 10, 10, 5, 5, 2, 2, 1, 1
- $\mathcal{U}$ enthält alle Teilmengen von E, deren Elemente eine Summe kleiner oder gleich 392 bilden:
  $M \in \mathcal{U}$ gdw. $w(M) \leq 392$ (M ist gültig)
- Kanonischer Greedy-Algorithmus:
  $\emptyset \longrightarrow \{e_1\} \longrightarrow \cancel{\{e_1, e_2\}} \longrightarrow \{e_1, e_3\}$ **optimale Lösung ?**
  $\ldots \longrightarrow \boxed{\{e_1, e_3, e_5, e_7, e_8, e_{14}\}}$
- Jede weitere Vereinigung führt zu einer Menge nicht in $\mathcal{U}$.

Optimalität

- Nicht immer führt der kanonische Greedy-Algorithmus zu einer optimalen Lösung:

  $E = \{e_1, e_2, e_3\}$, $\mathcal{U} = \{\emptyset, \{e_1\}, \{e_2\}, \{e_3\}, \{e_2,e_3\}\}$,
  $w(e_1) = 3$, $w(e_2) = w(e_3) = 2$
  $M \in \mathcal{U}$ gdw. $w(M) \leq 4$

  Algorithmus liefert $M = \{e_1\}$ mit $w(M) = 3$.
  Optimal (maximal) wäre aber $M' = \{e_2,e_3\}$ mit $w(M')= 4$.

- Die optimale Lösung wird genau dann erreicht, wenn $(E, \mathcal{U})$ ein sogenanntes **Matroid** ist.
- Sonst muss die Optimalität in jedem Fall einzeln bewiesen werden.

Matroide

- Ein Matroid ist ein Teilmengensystem $(E, \mathcal{U})$ mit
  1. $\emptyset \in \mathcal{U}$;
  2. Falls $B \in \mathcal{U}$ und $A \subseteq B$, dann gilt $A \in \mathcal{U}$;
  3. **(Austauscheigenschaft)** Für alle $A, B \in \mathcal{U}$ gilt:
     Falls $|A| < |B|$, dann gibt es ein $x \in B \setminus A$ so dass $A \cup \{x\} \in \mathcal{U}$.

- **Satz:** Sei $(E, \mathcal{U})$ ein Teilmengensystem und $w\colon E \longrightarrow \mathbb{R}^+$ eine beliebige Gewichtsfunktion.
  Der kanonische Greedy-Algorithmus liefert eine optimale Lösung, falls $(E, \mathcal{U})$ ein Matroid ist.

Beweis (Matroid $\Rightarrow$ Optimalität)

Seien: $(E, \mathcal{U})$ Matroid, $w: E \longrightarrow \mathbb{R}^+$ (beliebig),

$B = \{b_1, b_2, \ldots, b_r\}$ eine optimale Lösung, wobei

$w(b_1) \geq w(b_2) \geq \ldots \geq w(b_r)$ und $B_i = \{b_1, b_2, \ldots, b_i\}$.

Sei a_i das i-te vom Greedy-Algorithmus hinzugefügte Element,

$A_0 = \emptyset$, $A_i = \{a_1, a_2, \ldots, a_i\}$ $(1 \leq i \leq r)$.

Vollst. Induktion über *i*: $A_i \in \mathcal{U}$ $(0 \leq i \leq r)$ und $w(a_i) \geq w(b_i)$ $(1 \leq i \leq r)$.

$i = 0$: $\emptyset \in \mathcal{U}$. (Zweite Aussage ist trivial.)

$i\text{-}1 \longrightarrow i$ $(1 \leq i \leq r)$:

Da $|A_{i-1}| < |B_i|$, gibt es ein $x \in B_i \setminus A_{i-1}$ mit $A_{i-1} \cup \{x\} \in \mathcal{U}$.

Der Greedy-Alg. kann also A_{i-1} um ein $a_i \in E$ mit $w(a_i) \geq w(x)$ erweitern.

Da $x \in B_i$ ist, gilt $w(x) \geq w(b_i)$. Also $w(A_r) \geq w(B)$ und A_r ist optimal.

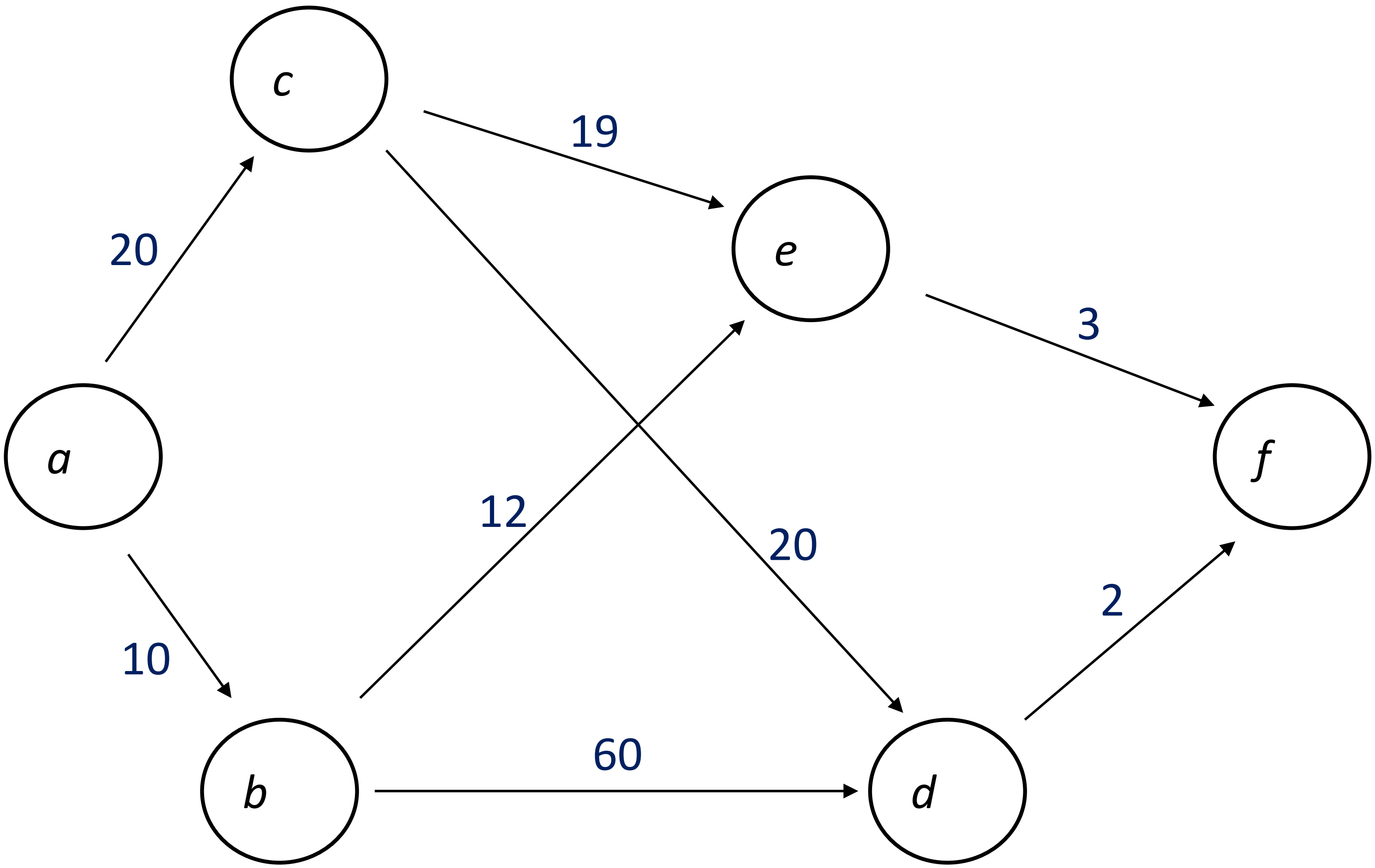
Algorithmen auf Graphen:
Kürzeste Pfade

Routenplanung

- Ziel: Berechnung der Kosten
  - Wegstrecke *oder*
  - Fahrzeit *oder*
  - Fahrpreis

  auf einer Route von Ort u nach Ort v
- *Modell:*
  Gerichteter Graph mit positiven Kantengewichten
  $G = (V, E, w)$, (V, E) gerichteter Graph,
  $$w : E \longrightarrow \mathbb{R}^+$$

Beispiel



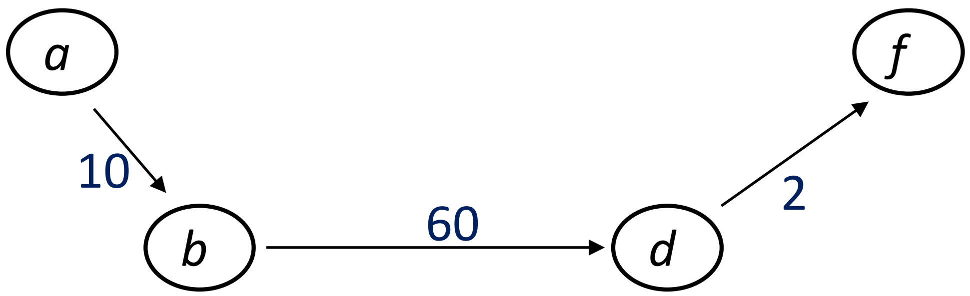
Länge eines Pfades – gewichtet

- Sei $p = v_0, v_1, \ldots, v_n$ ein Pfad in $G = (V, E, w)$.
- Das Pfadgewicht $w(p)$ ist

$$\sum_{i=0}^{n-1} w(v_i, v_{i+1})$$

- *Beispiel:*

  $w(a,b,d,f) = 72$



Kürzeste Pfade - gewichtet

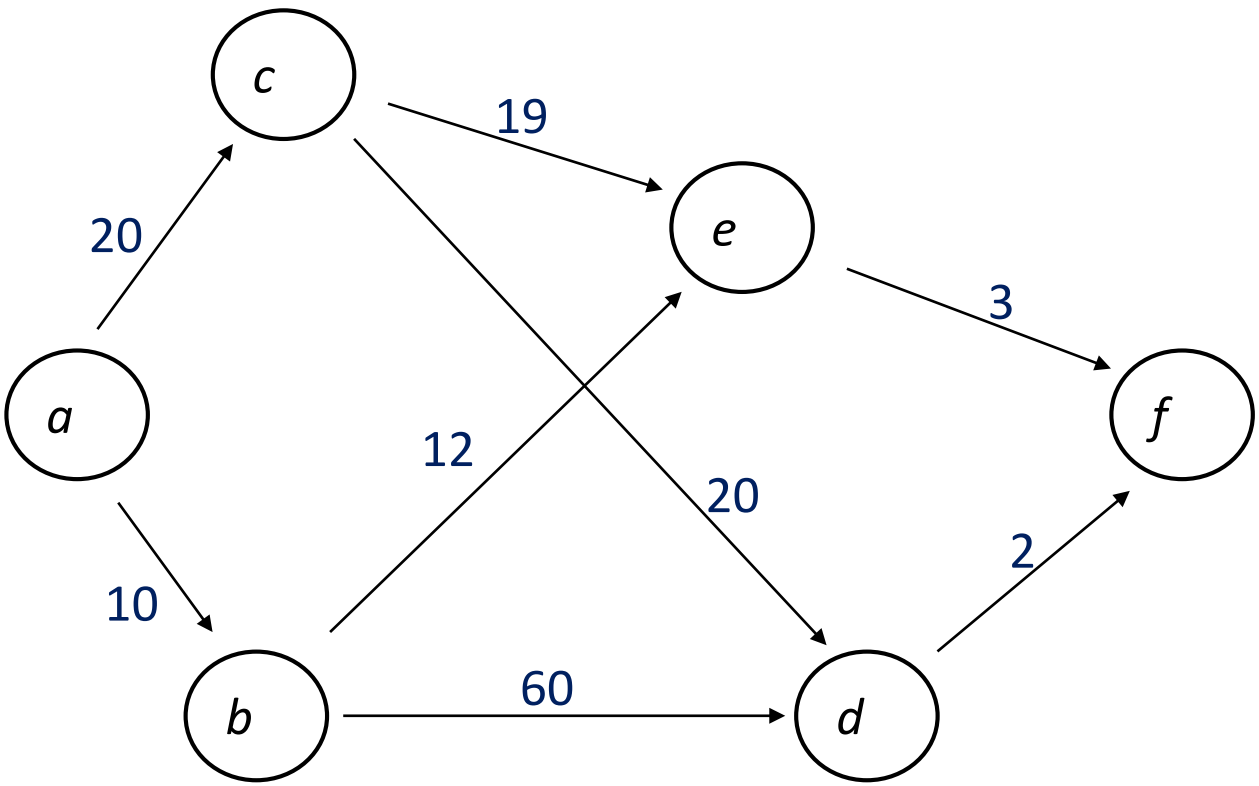
Eingabe: Gerichteter Graph $G = (V,E,w)$ mit positiven Kantengewichten, ein Knoten $\boldsymbol{u} \in V$.

Ausgabe: Menge der Pfade mit minimalem Pfadgewicht von $\boldsymbol{u}$ nach v für alle $v \in V$.

Greedy:

- Sortieren alle *kreisfreien* Pfade von $\boldsymbol{u}$ nach aufsteigender Pfadlänge
- **Gültigkeitsbedingung:** für jeden Knoten v wird nur ein kürzester Pfad von u nach v gefunden
- Starten mit leerer Menge
- Fügen nächsten Pfad hinzu, falls noch kein Pfad mit demselben Endknoten in der Menge ist

Beispiel



Anordnung aller kreisfreien Pfade:

a	0
a,b	10
a,c	20
a,b,e	22
a,b,e,f	25
a,c,e	39
a,c,d	40
a,c,d,f	42
...	

Optimalität

- Betrachten folgendes Teilmengensystem $(E, \mathcal{U})$:
  - E: Menge aller kreisfreien Pfade von u
  - $\mathcal{U}$: System mit $\emptyset$ und allen Pfadmengen, deren Pfade in u beginnen und paarweise verschiedene Endknoten haben
- Wenn $M \in \mathcal{U}$, dann ist jede Teilmenge von M in $\mathcal{U}$.
- Falls A und B aus $\mathcal{U}$ mit $|A| < |B|$,
  - dann gibt es in A genau $|A|$ verschiedene Endknoten,
  - in B befindet sich mindestens ein Pfad p mit einem Endknoten, der nicht in A vorkommt und
  - $A \cup \{p\} \in \mathcal{U}$

➢ $(E, \mathcal{U})$ ist Matroid.

Dijkstra-Algorithmus

Effizienzverbesserung:

- Nicht alle kreisfreien Pfade von *u* werden erzeugt
- Bilden der Pfade ähnlich dem Prinzip der Breitensuche
- Protokollieren dabei in jedem Knoten:
  - bisher gefundene kürzeste Pfadlänge zu diesem Knoten
  - Kante, über die der Knoten auf dem bisher kürzestem Pfad erreicht wird

Dijkstra – Idee

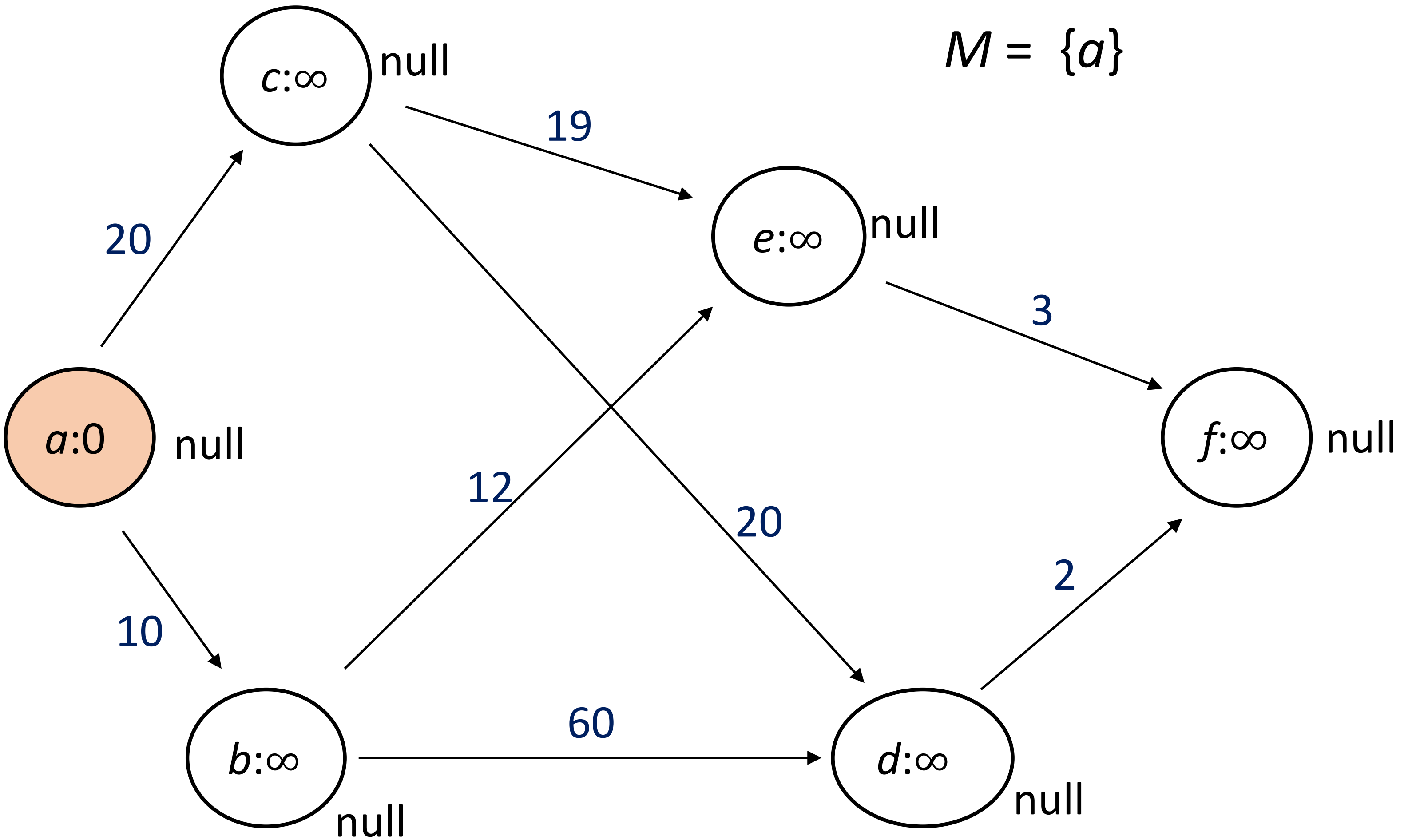
- Bilde Menge *M* bereits aufgesuchter Knoten, die noch „neue“ Nachbarn haben können (solange, bis diese leer ist)
- Initialisiere
  - die Menge *M* mit *u*,
  - die kürzeste Pfadlänge von *u* mit 0,
  - aller anderer Knoten mit ∞ (*noch nicht aufgesucht*)
- Entnehmen aus *M* Knoten *v* mit bislang kürzester Pfadlänge und behandeln alle seine Nachbarn *y* wie folgt:
  - Falls kürzeste Pfadlänge ∞, dann in Menge *M* eintragen
  - Falls durch Pfad über Kante (*v*,*y*) neuer kürzester Pfad zu *y* gefunden: *kürzeste Pfadlänge in y aktualisieren*

Konstruktion des kürzesten Pfades

- Bei jeder Aktualisierung der Pfadlänge:
  Speichern der Kante, die für die kürzeste Pfadlänge in den Knoten führt (**Vorgängerkante**)
- Nach dem Terminieren:
  Zurückverfolgen der Vorgängerkanten bis zum Startknoten u

Dijkstra am Beispiel

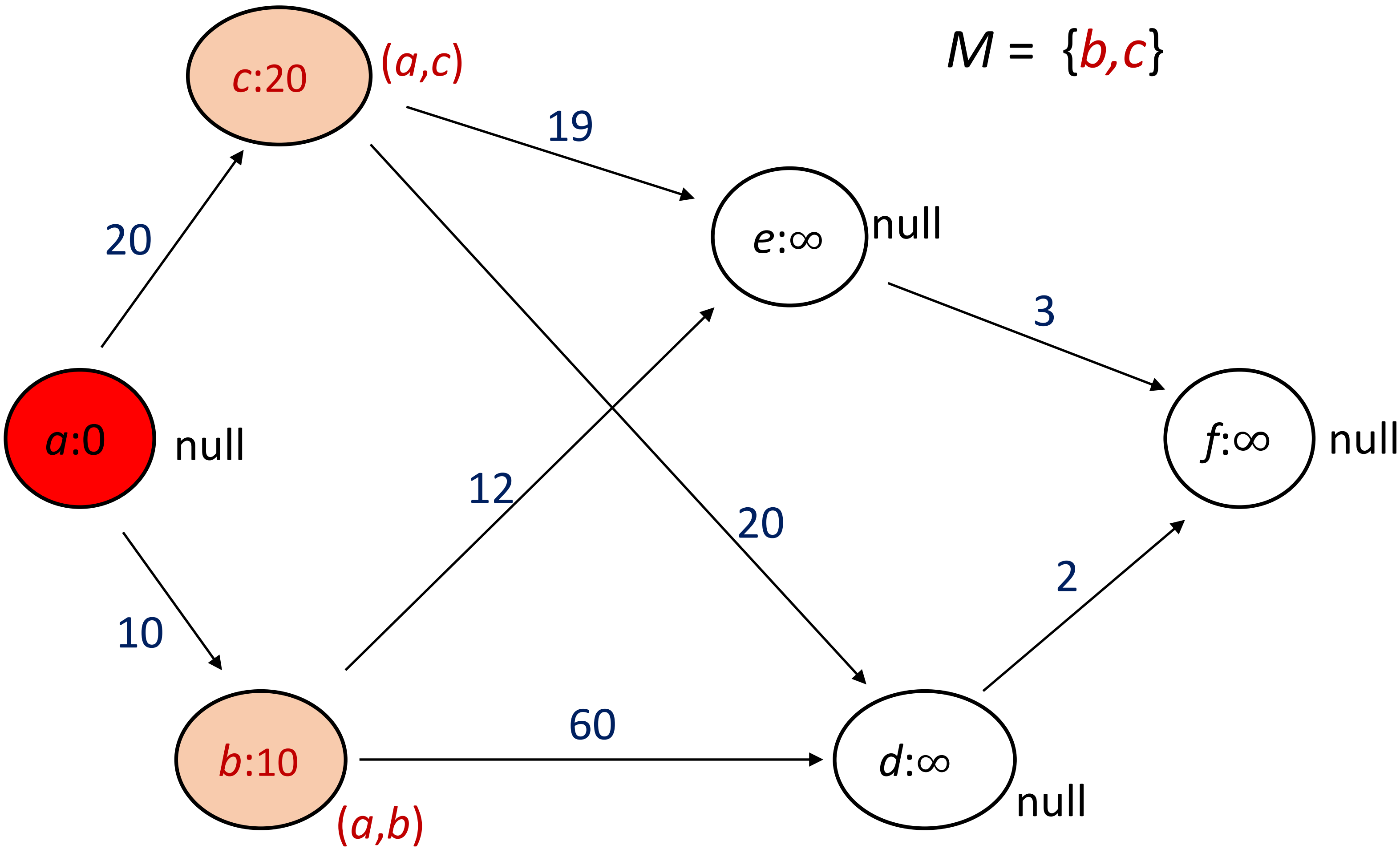
$u = a$
$M = \{a\}$



Dijkstra am Beispiel

$u = a$

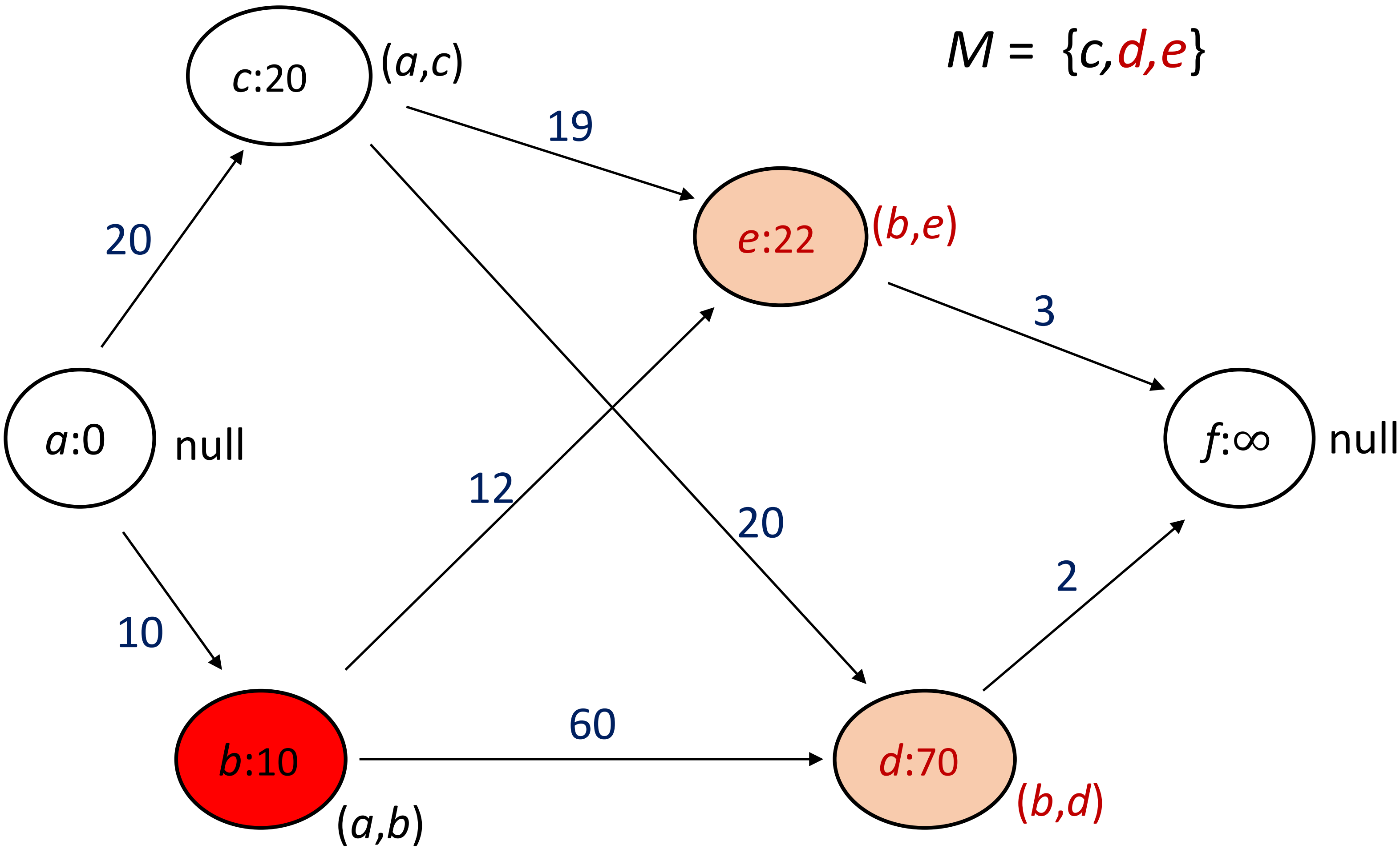
$M = \{b,c\}$



Dijkstra am Beispiel

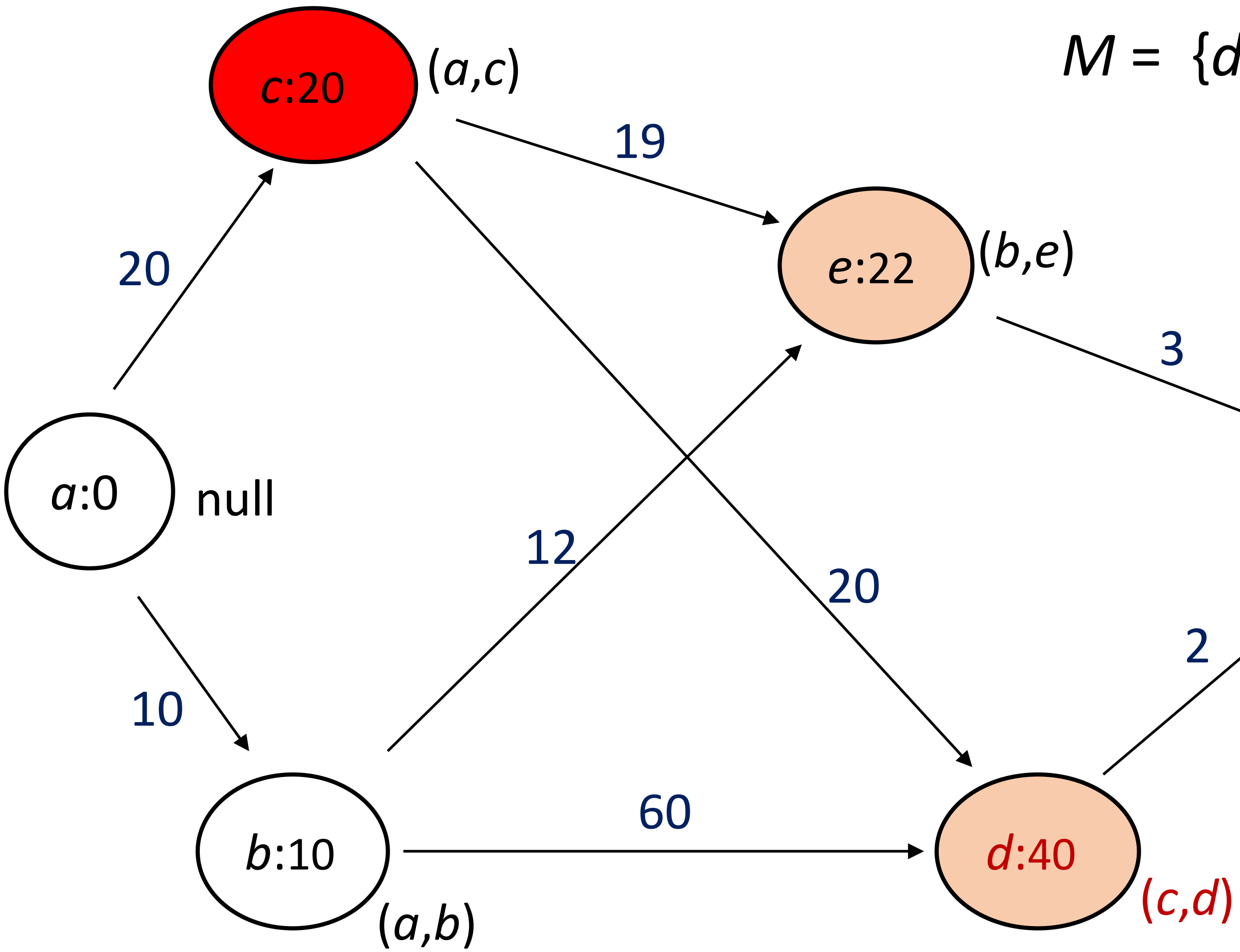
$u = a$

$M = \{c,d,e\}$



Dijkstra am Beispiel

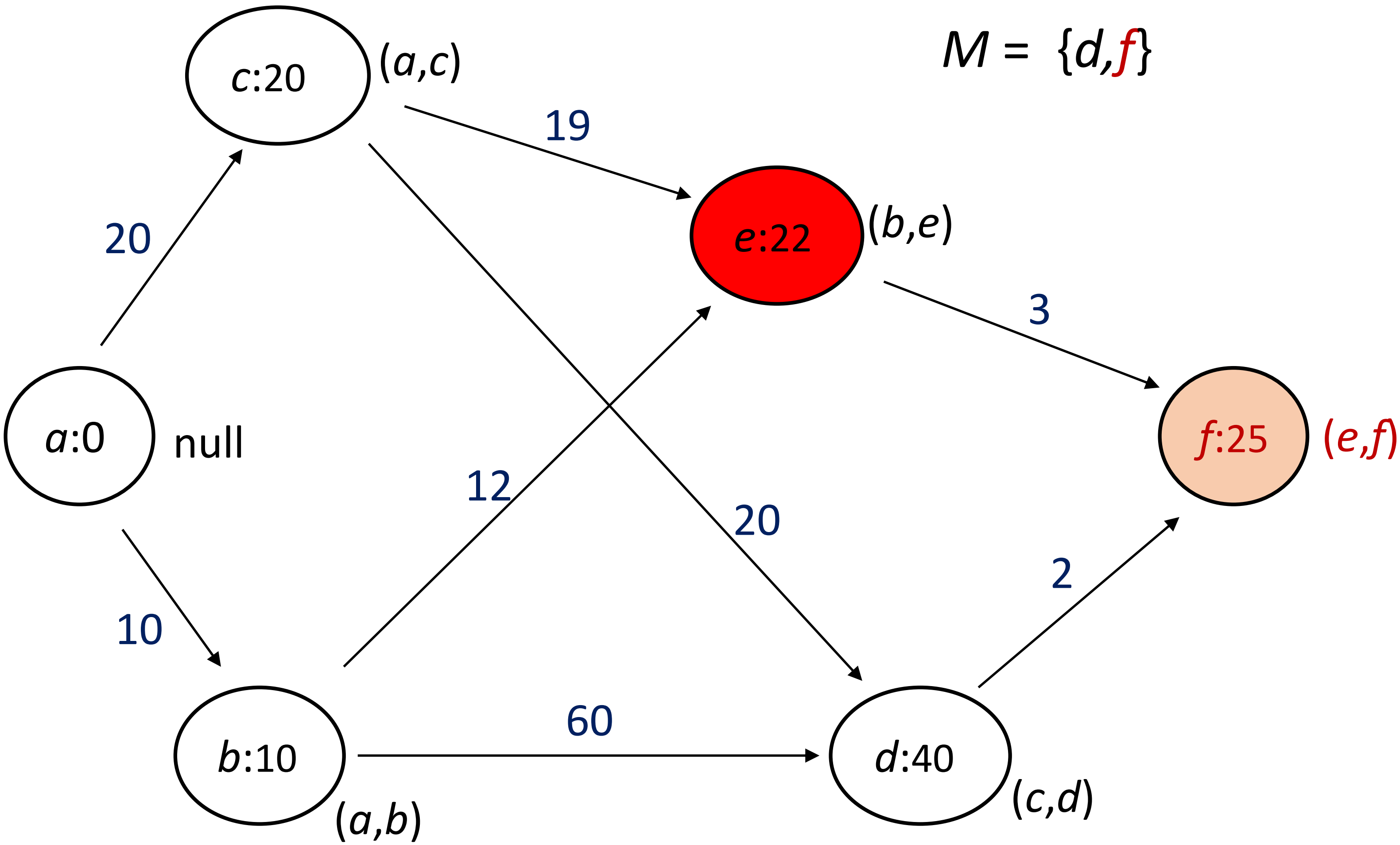
$u = a$
$M = \{d,e\}$



Dijkstra am Beispiel

$u\ =\ a$

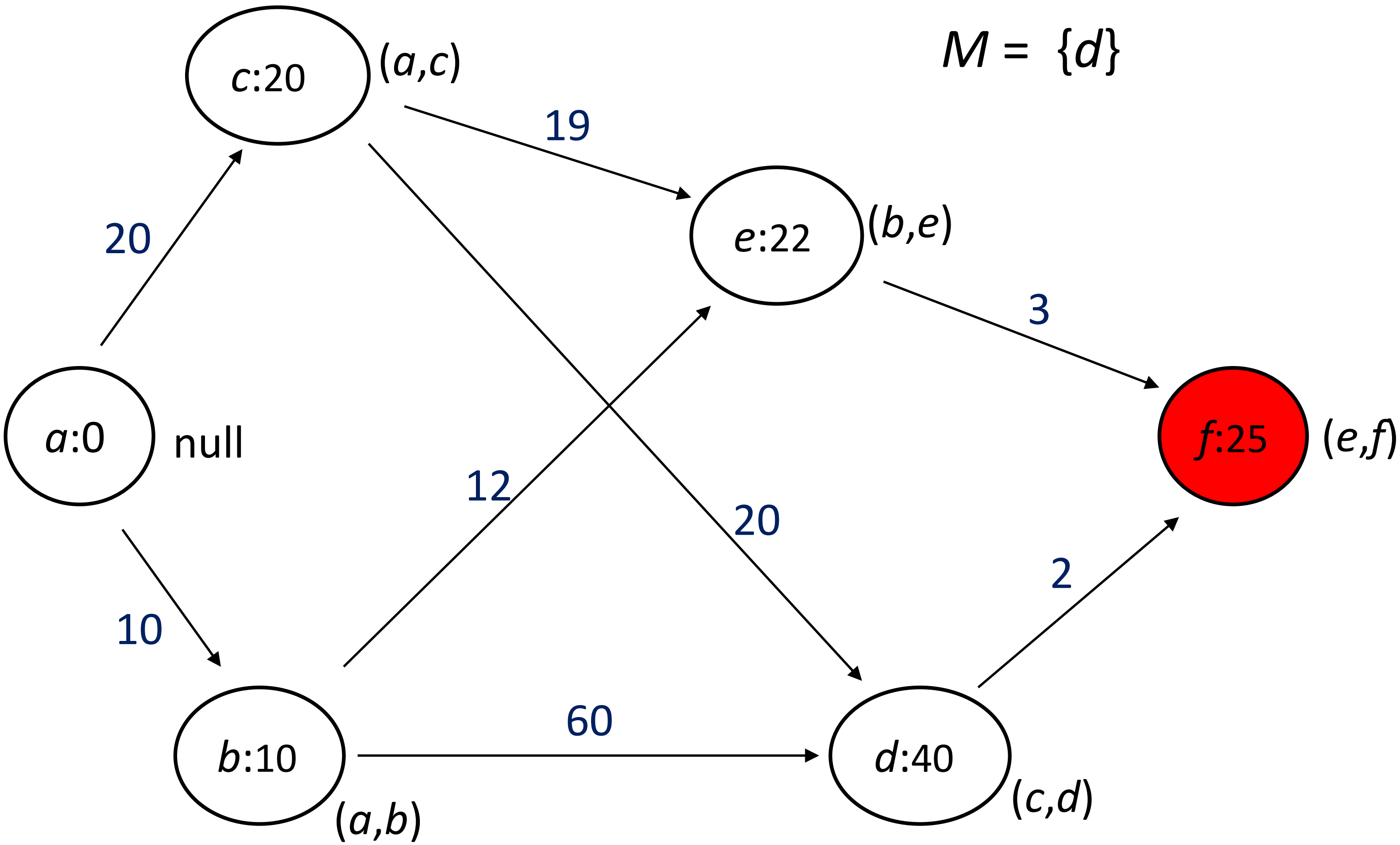
$M = \{d, f\}$



Dijkstra am Beispiel

$u = a$

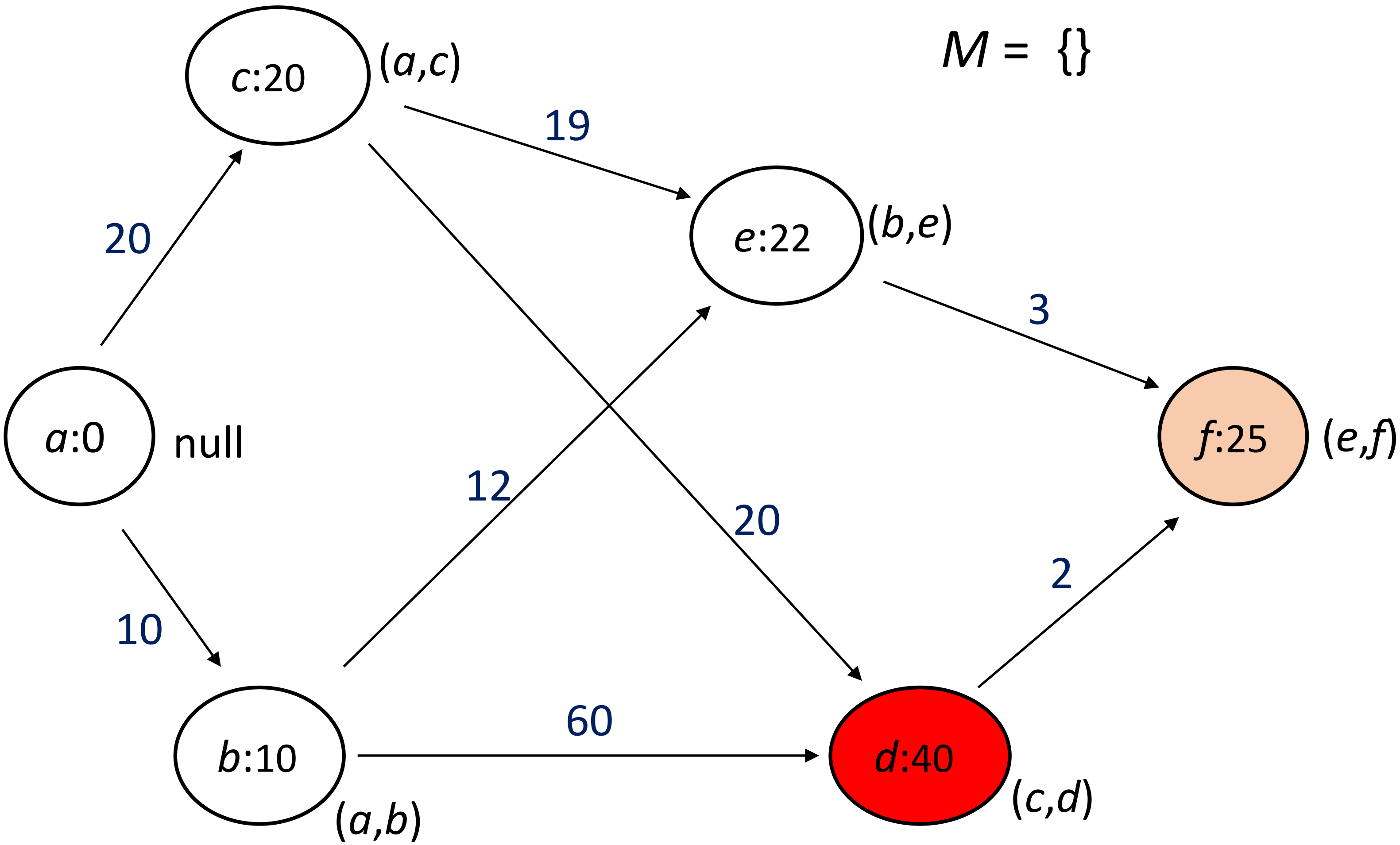
$M = \{d\}$



Dijkstra am Beispiel

$u = a$

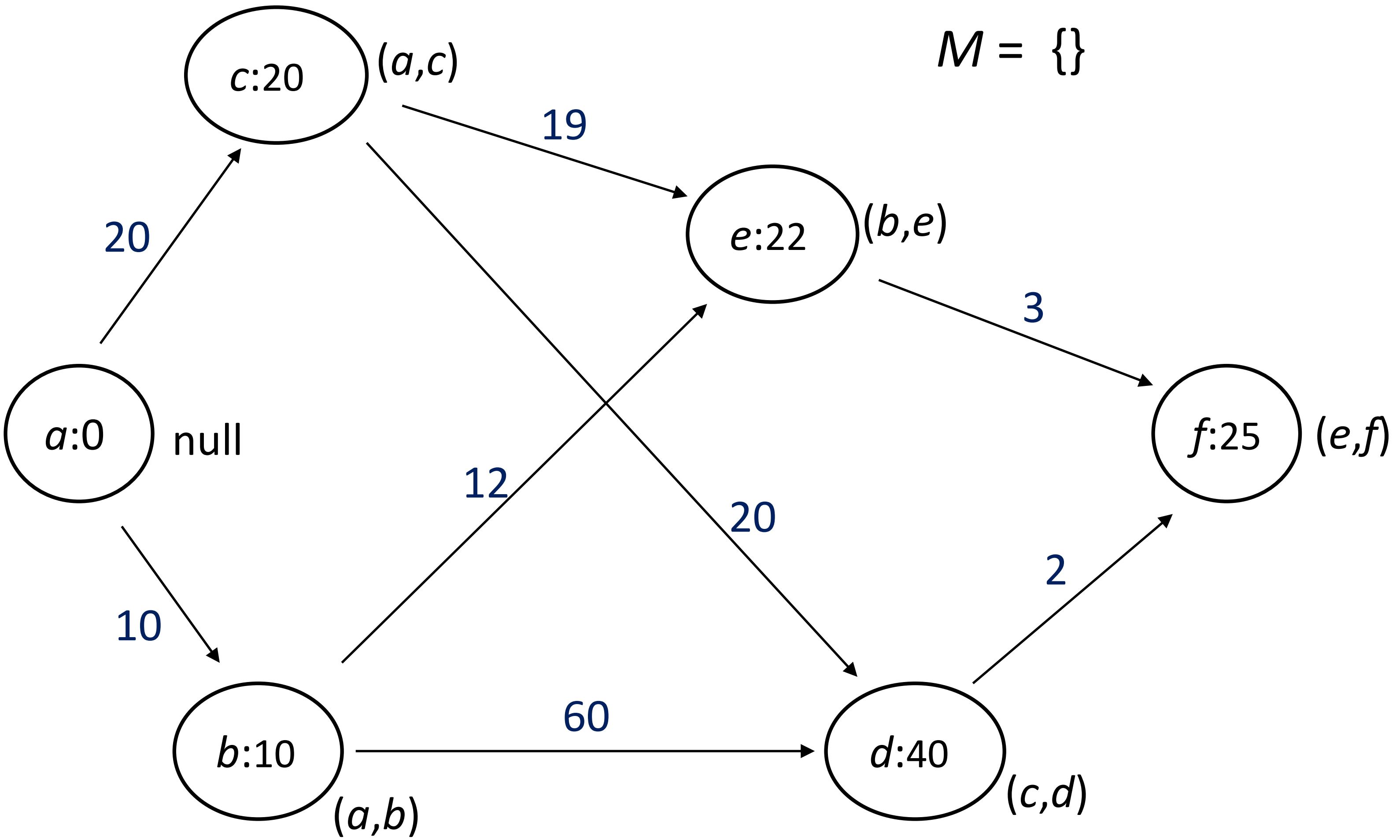
$M = \{\}$



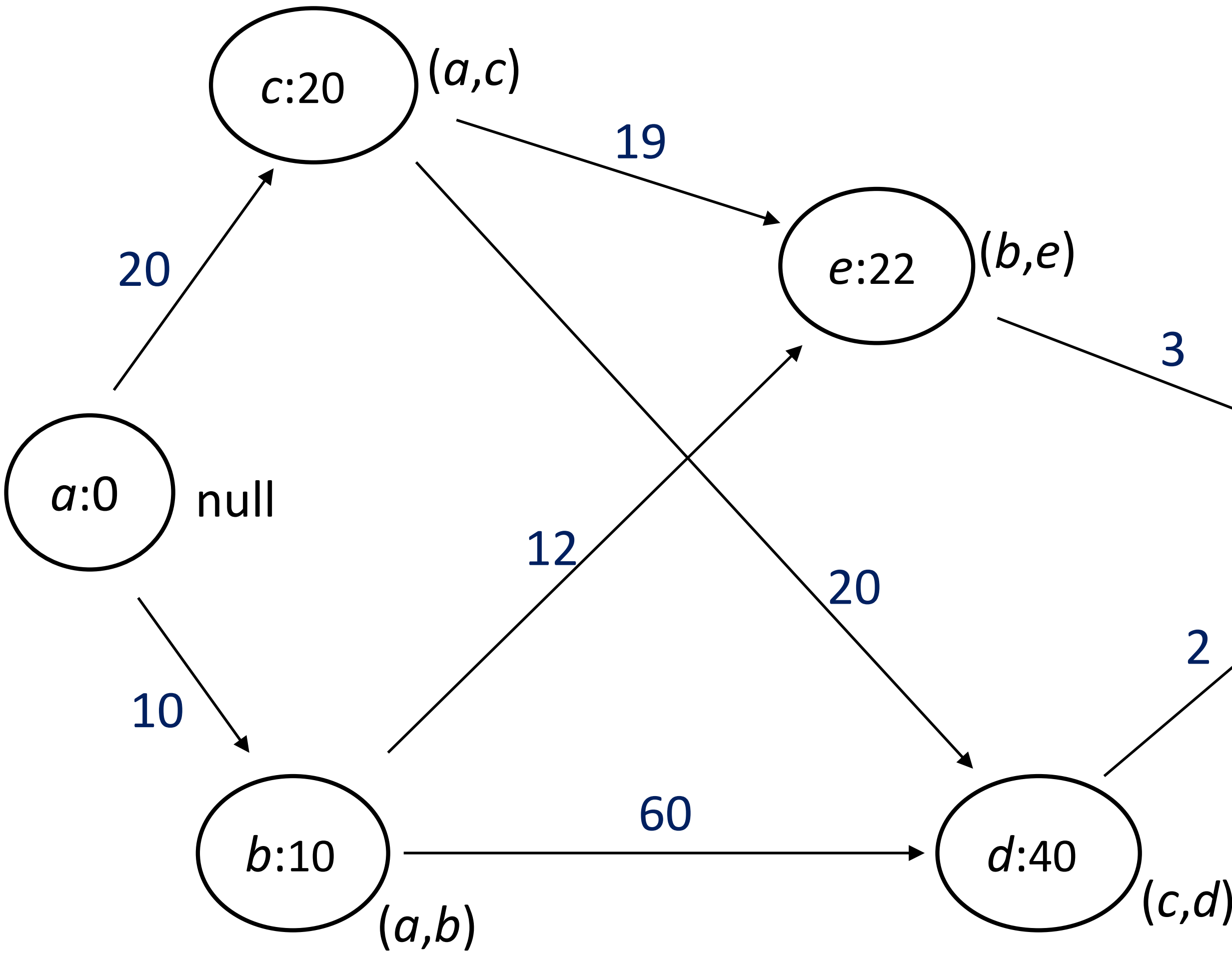
Dijkstra am Beispiel

$u = a$

$M = \{\}$



Kürzeste Pfade am Beispiel



a : a
$a - b$: a, b
$a - c$: a, c
$a - d$: a, c, d
$a - e$: a, b, e
$a - f$: a, b, e, f

Dijkstra-Algorithmus - Implementierung

Anpassung der Datenstruktur

- Entnehmen aus *M* Knoten *v* mit bislang <u>kürzester</u> Pfadlänge …

➢ Verwaltung der Elemente *sortiert*
  - ➢ Hinzufügen von Elementen: sortiertes Einfügen
  - ➢ Entnehmen immer das „kleinste“ Element
  - ➢ Ändert sich die bislang ermittelte kürzeste Pfadlänge eines Knotens, so muss dieser entfernt und neu hinzugefügt werden.

➢ **Prioritätswarteschlange / Priority Queue *PQ***
  - ➢ speichert die Elemente zusammen mit einem Schlüssel
  - ➢ die spezialisierten Operationen:
    - **insert(*y*, *PQ*)** sortiertes Einfügen von *y* in die *PQ*
    - **extractMin(*PQ*)** dequeue des Elements mit min. Schlüssel
    - **delete(*y*,*PQ*)** Entfernen von *y* aus der *PQ*

Implementierungen des ADT PQ

- eine effiziente Implementierung: als **AVL-Baum**

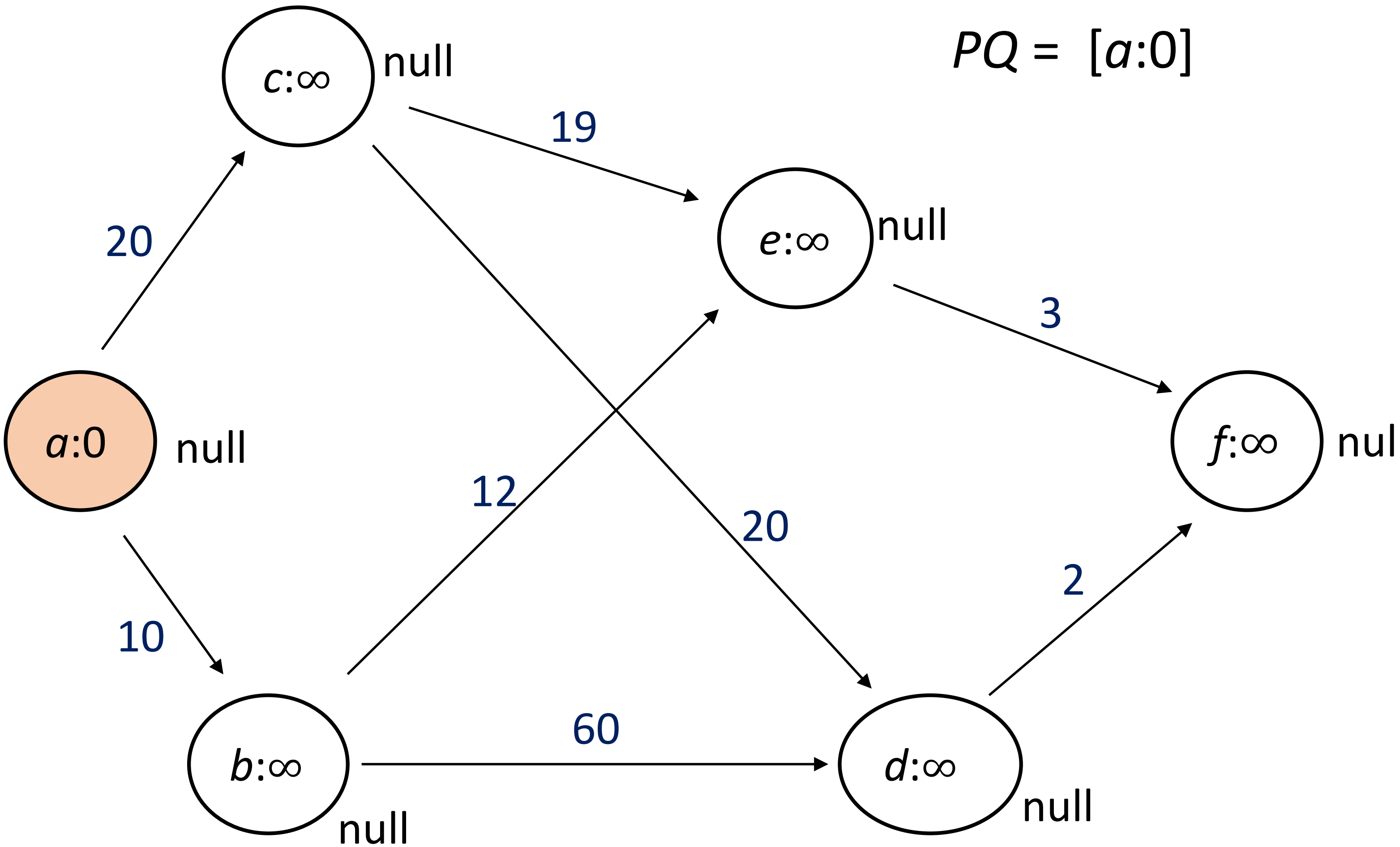
- eine einfache Implementierung: als **sortierte Liste**
  - **insert**: Hinzufügen von Elementen: sortiertes Einfügen (wie **insert** bei **insertionSort**: $O(n)$)
  - **extractMin**: Entnehmen das „kleinste“ Element (wie **dequeue** bei einer **Queue**: $O(1)$**)**
  - **delete**: Suchen des zu löschenden Elements **(binSearch)** und anschließend Löschen: $O(\log n)$
  - einige weitere Operationen für Listen:
    - Erzeugen einer leeren PQ
    - Test, ob eine PQ leer ist

Dijkstra mit einer Priority Queue

- kürzeste Pfadlängen als Schlüssel (aufsteigend sortiert)
- **extractMin** entnimmt immer den Knoten
  mit bislang kürzester Pfadlänge
- Ändert sich der Schlüssel eines Knotens,
  der bereits in der PQ ist, muss ggf. umsortiert werden
  (**delete**, gefolgt von **insert**)

Dijkstra am Beispiel

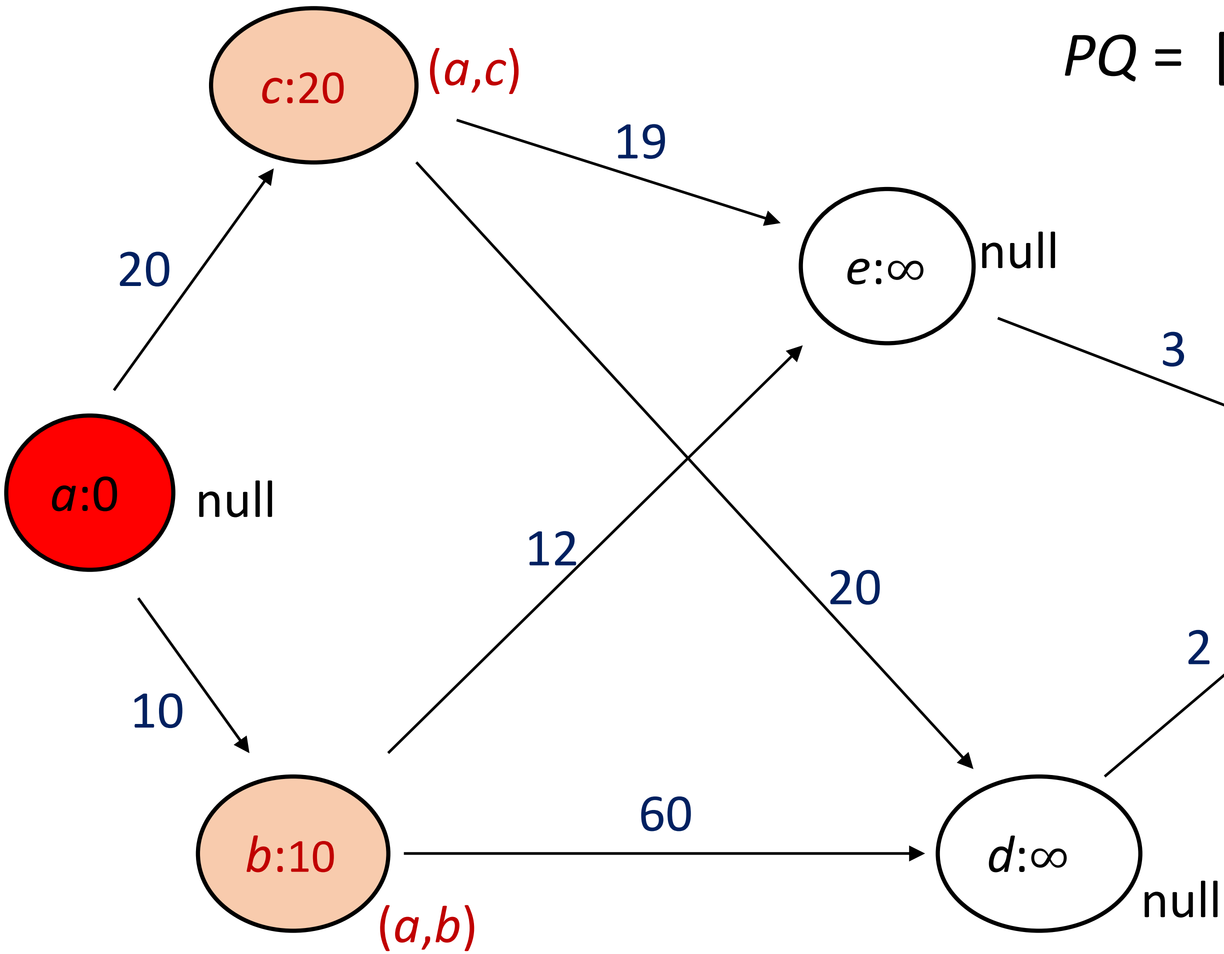
$u = a$
$PQ = [a{:}0]$



Dijkstra am Beispiel

$u = a$

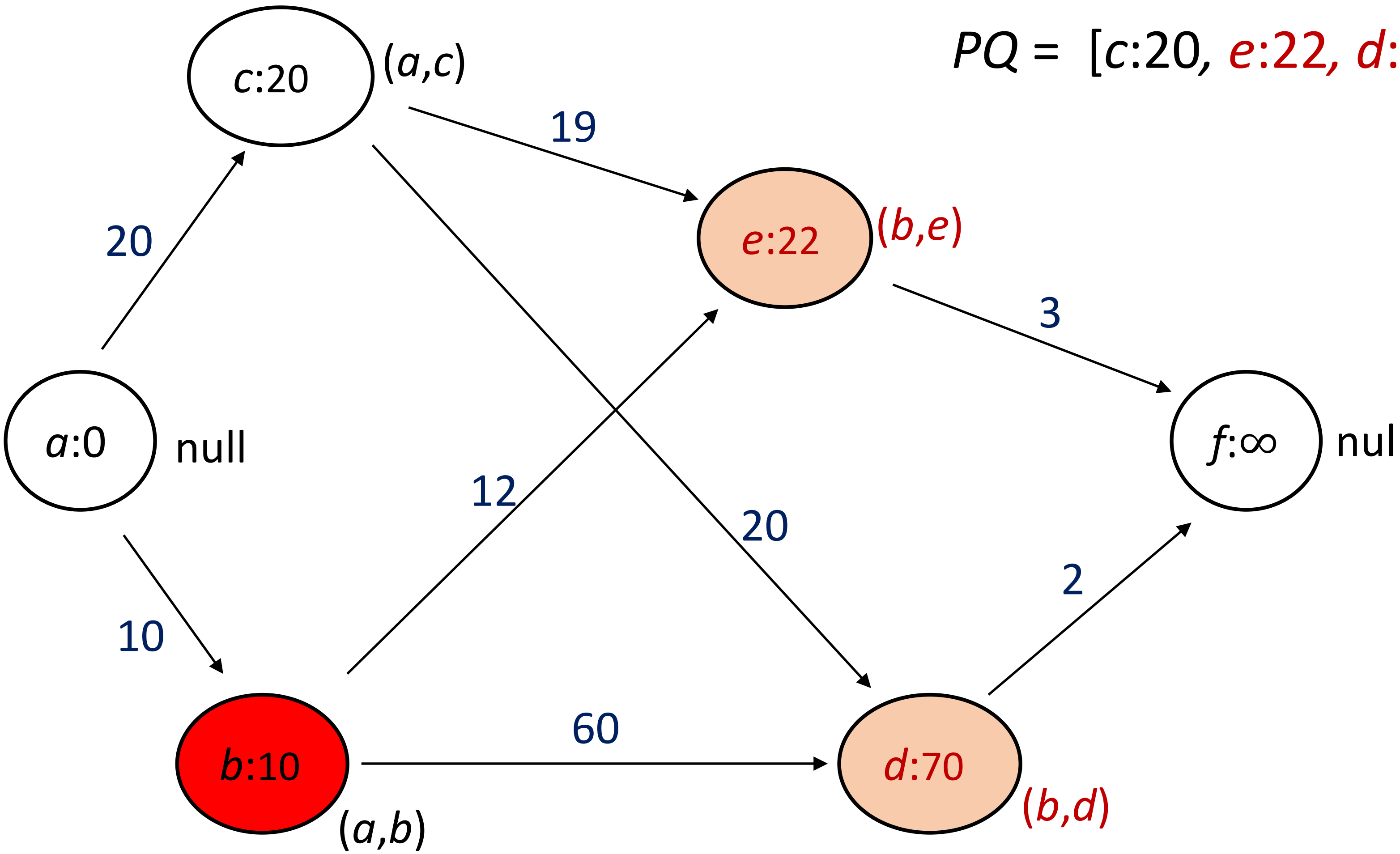
$PQ = [b{:}10,\ c{:}20]$



Dijkstra am Beispiel

$u \;=\; a$

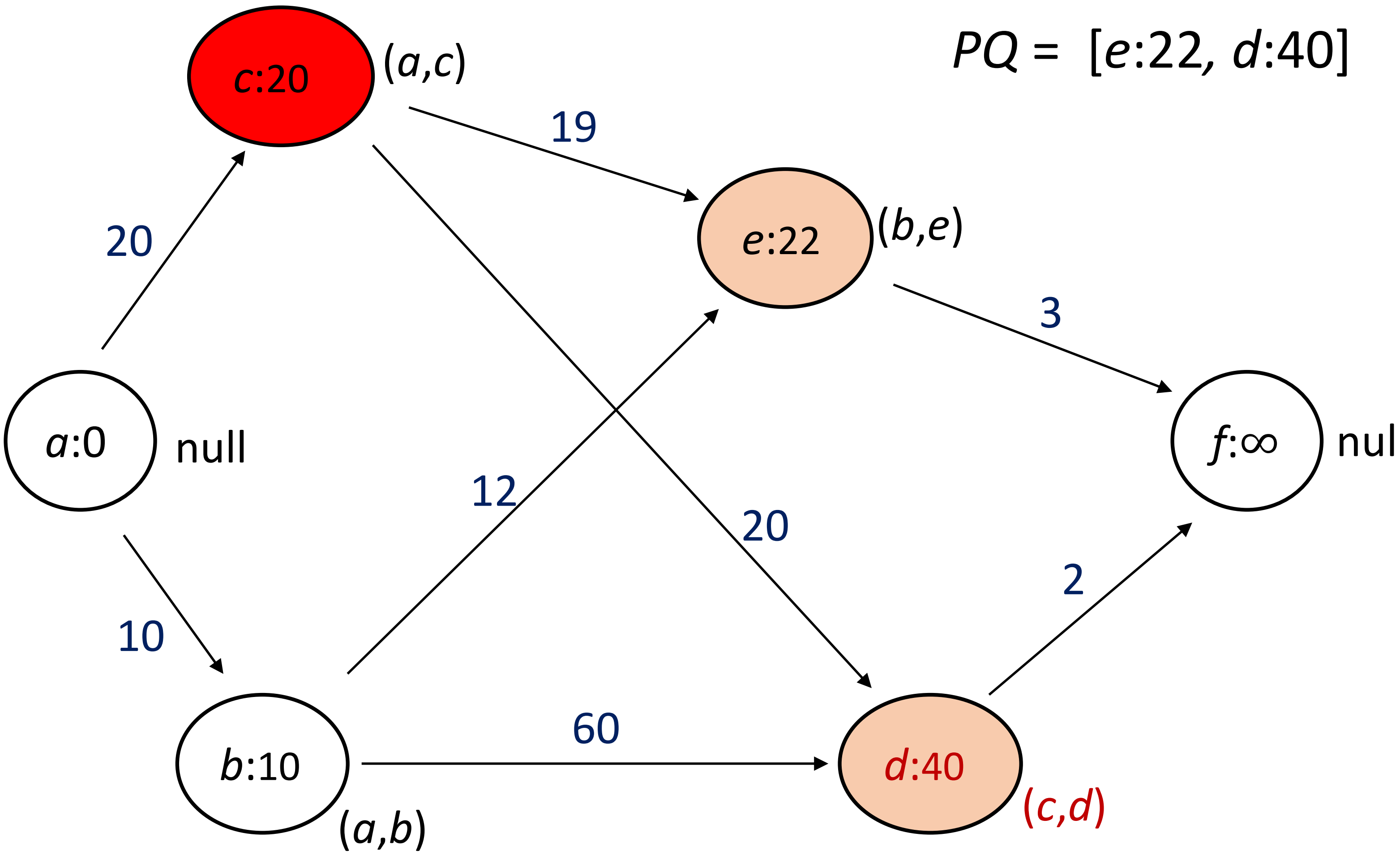
$PQ = [c{:}20,\ e{:}22,\ d{:}70]$



Dijkstra am Beispiel

$u = a$

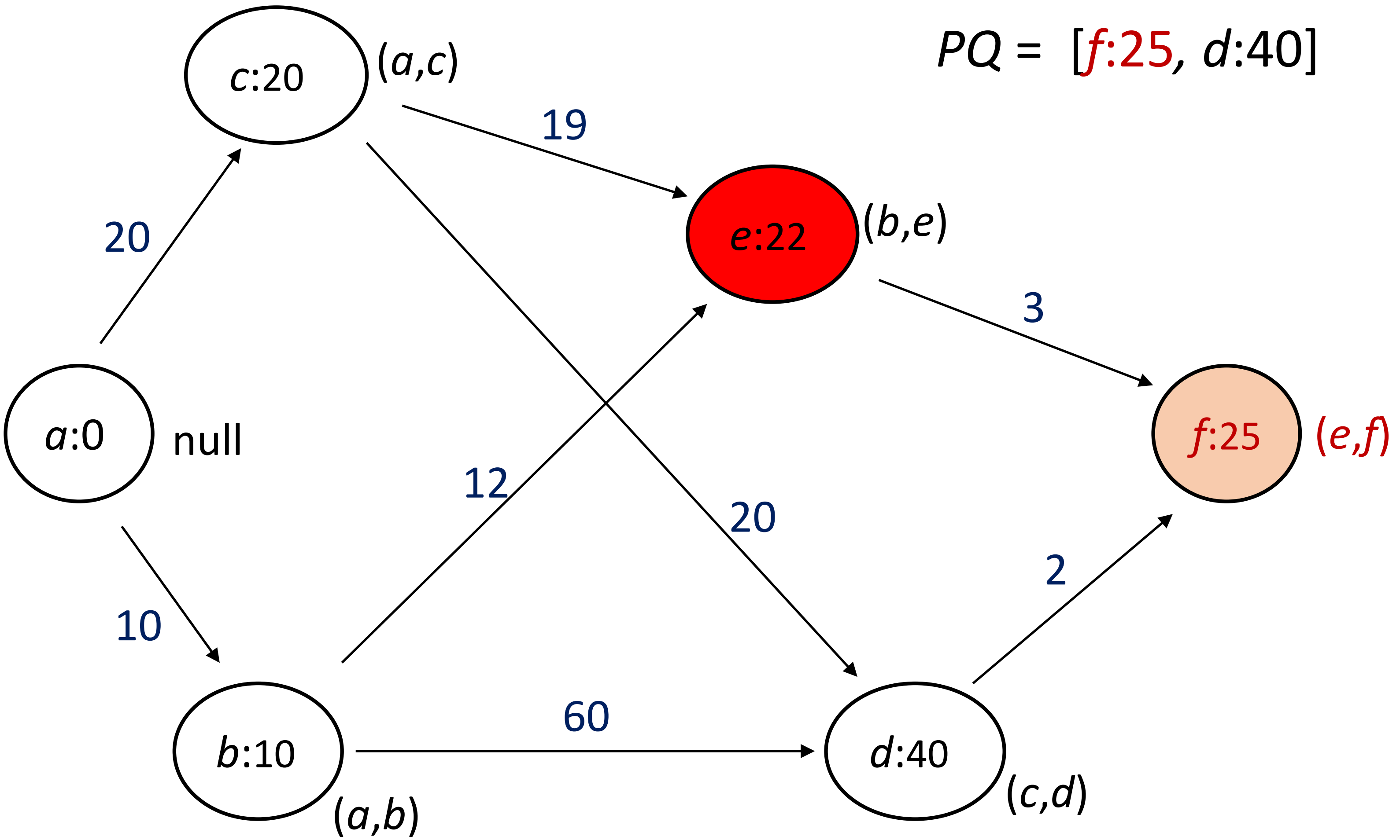
$PQ = [e{:}22,\ d{:}40]$



Dijkstra am Beispiel

$u \;=\; a$

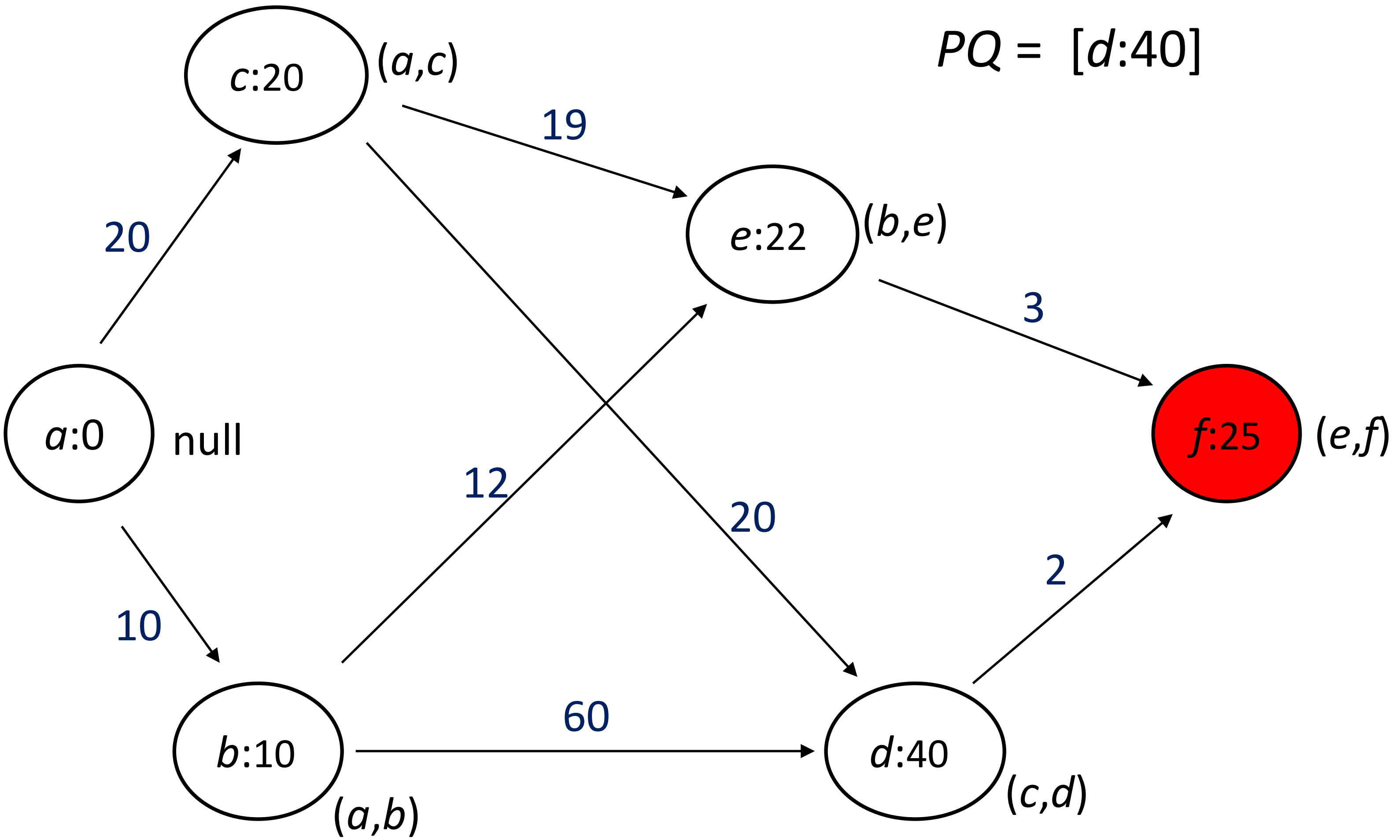
$PQ = \;[f{:}25,\; d{:}40]$



Dijkstra am Beispiel

$u \;=\; a$

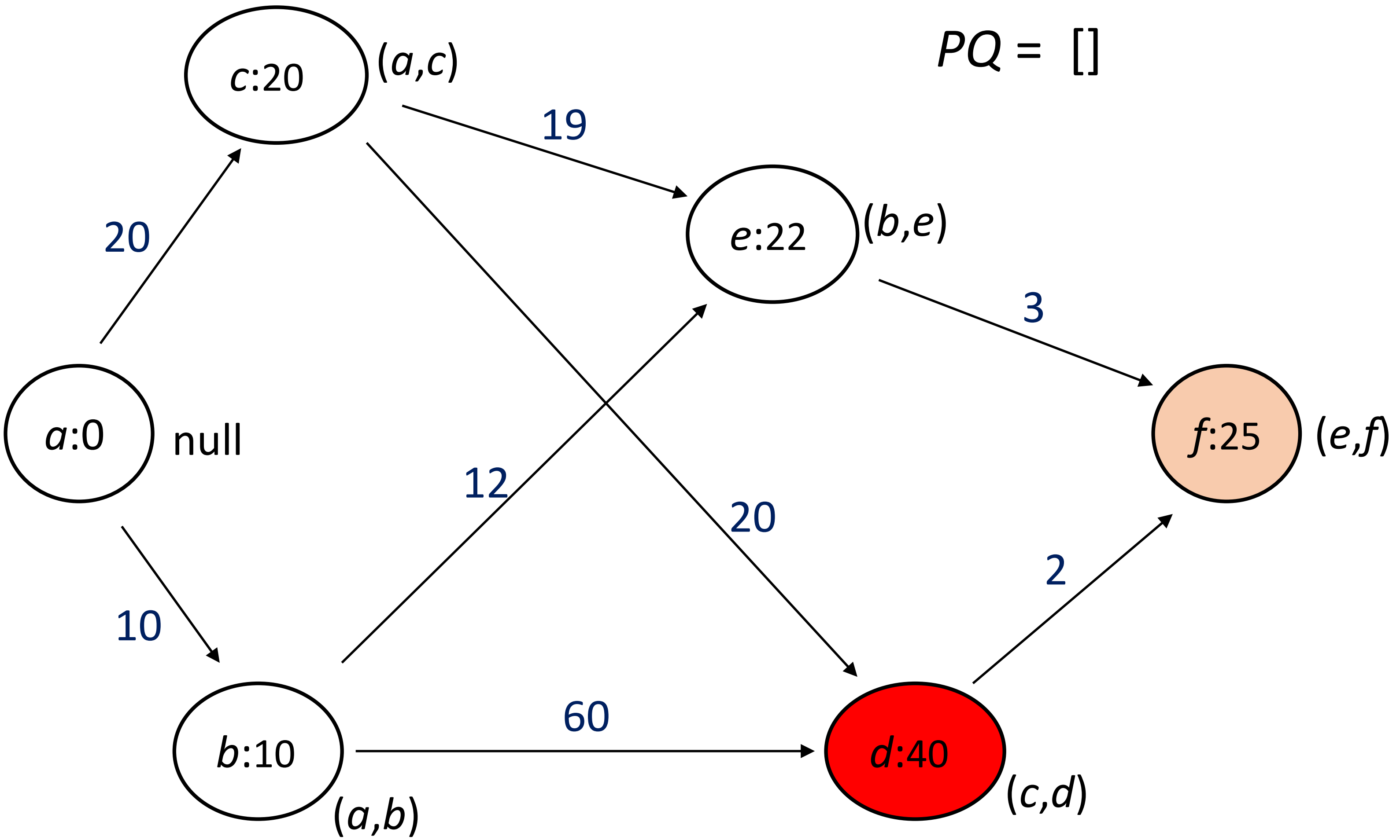
$PQ = \;[d{:}40]$



Dijkstra am Beispiel

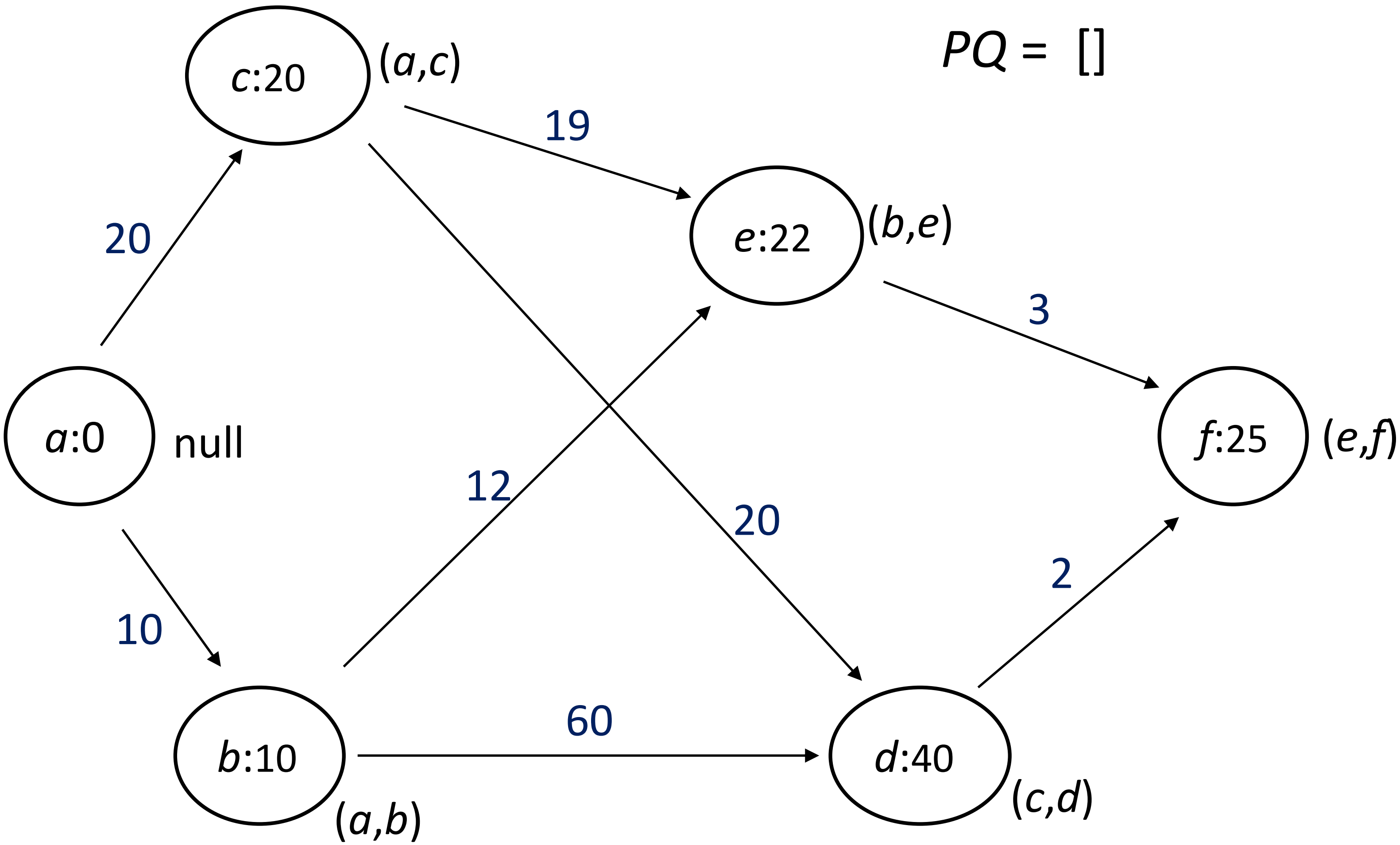
$u = a$

$PQ = []$

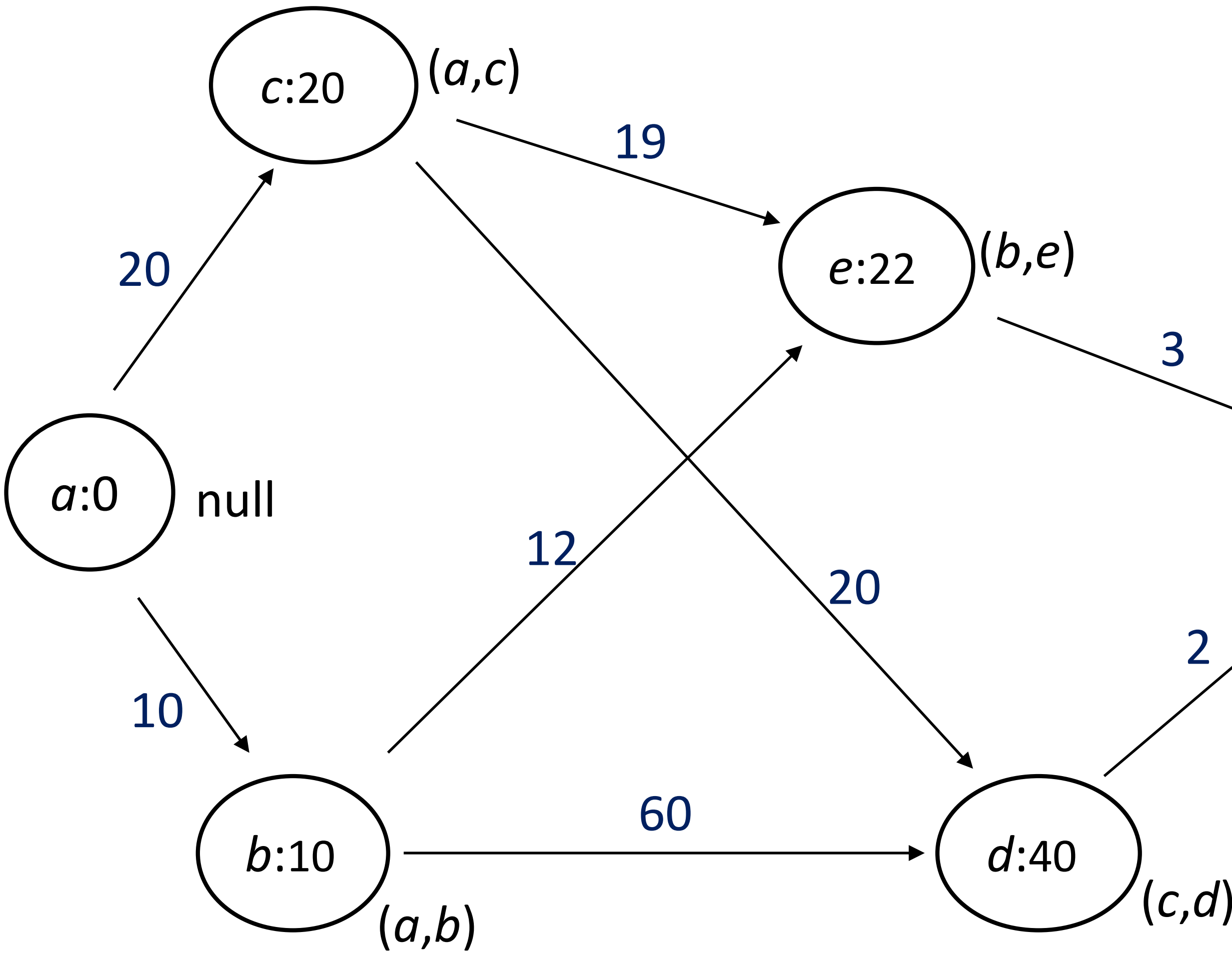


Dijkstra am Beispiel

$u = a$
$PQ = []$



Kürzeste Pfade am Beispiel



a : a
$a - b$: a, b
$a - c$: a, c
$a - d$: a, c, d

$a - e$: a, b, e
$a - f$: a, b, e, f

Pseudocode

Für jedes *v* in *V*

```
        dist(v) ⟵ ∞        # minimale Pfadlänge von u nach v
        pre(v) ⟵ null      # Vorgängerkante auf optimalem Pfad
dist(u) ⟵ 0
PQ ⟵ [u]                   # Prioritätswarteschlange
solange PQ nicht leer
        v ⟵ extractMin(PQ)
        für alle y mit (v,y) in E
                aktualisiere(v,y)
```

aktualisiere(v,y)

falls dist(y) = ∞

    insert(*y*, *PQ*)   *# Einfügen in die PQ*

falls dist(*y*) > dist(*v*) + *w*(*v*,*y*)

    dist(*y*) $\longleftarrow$ dist(*v*) + *w*(*v*,*y*)

    pre(*y*) $\longleftarrow$ (*v*,*y*)

    falls (Schlüssel von *y*) < (Schlüssel des Vorgängers von *y* in *PQ*)

        delete(*y*, *PQ*)   */* Umsortieren von y in der*

        insert(*y*, *PQ*)   *Prioritätswarteschlange */*

Abschließende Bemerkungen

- Sind alle Kantengewichte 1, so entspricht Dijkstra (im wesentlichen) BFS.

- Greedy kann als vereinfachte dynamische Programmierung betrachtet werden:
  - *Wie DP:* Berechnung von Folgewerten allein durch Rückgriff auf bereits berechnete Werte.
  - *Anders als DP:* Nicht die Werte für alle kleineren Teilprobleme berechnen und tabellieren, da die Berechnung auf lokal verfügbaren Informationen basiert