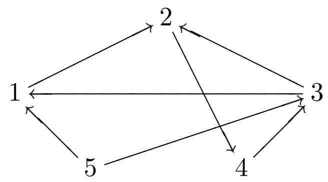
Martin Götze, Bernhard Streit

4. Übung zu Informatik B

Tutorium: Marco Block, Fr 12–14 Uhr

1. **Transitive Hülle mit Warshall**

   Berechnen Sie mit dem Warshall-Algorithmus die transitive Hülle des folgenden Graphen.



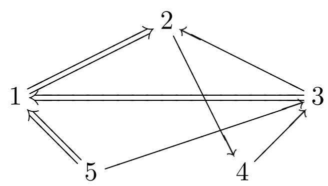
Wir stellen die Adjazenmatrix auf:

$$\begin{pmatrix} 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 \end{pmatrix}$$

Im folgenden sei nun k der aktuell betrachtete Knoten; i ein Knoten, der zu k hinzeigt und j ein Knoten, auf den k zeigt. (Also $i \to k \to j$)

Sei $k = 1$.
Es gilt: $i \in \{3, 5\}$ und $j \in \{2\}$:

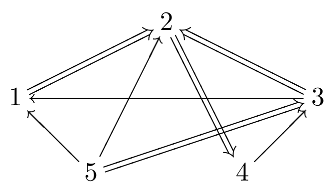
$$\begin{pmatrix} \mathbf{0} & \mathbf{1} & \mathbf{0} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & 0 & 0 & 1 & 0 \\ \mathbf{1} & 1 & 0 & 0 & 0 \\ \mathbf{0} & 0 & 1 & 0 & 0 \\ \mathbf{1} & 0 & 1 & 0 & 0 \end{pmatrix}$$



Die Verbindung $3 \to 2$ existiert bereits; wir fügen $5 \to 2$ hinzu.

Sei $k = 2$.
Es gilt: $i \in \{1, 3, 5\}$ und $j \in \{4\}$:

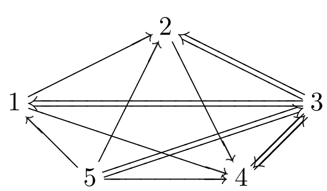
$$\begin{pmatrix} 0 & \mathbf{1} & 0 & 0 & 0 \\ \mathbf{0} & \mathbf{0} & \mathbf{0} & 1 & \mathbf{0} \\ 1 & \mathbf{1} & 0 & 0 & 0 \\ 0 & \mathbf{0} & 1 & 0 & 0 \\ 1 & \mathit{1} & 1 & 0 & 0 \end{pmatrix}$$



Wir fügen hinzu: $1 \to 4, 3 \to 4, 5 \to 4$

Sei $k = 3$.
Es gilt: $i \in \{4, 5\}$ und $j \in \{1, 2, 4\}$:

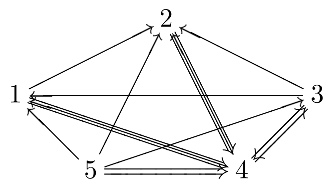
$$\begin{pmatrix} 0 & 1 & \mathbf{0} & \mathit{1} & 0 \\ 0 & 0 & \mathbf{0} & 1 & 0 \\ \mathbf{1} & \mathbf{1} & \mathbf{0} & \mathit{1} & \mathbf{0} \\ 0 & 0 & \mathbf{1} & 0 & 0 \\ 1 & \mathit{1} & \mathbf{1} & \mathit{1} & 0 \end{pmatrix}$$



Wir fügen die Verbindungen $4 \to 1$, $4 \to 2$ und $5 \to 2$ hinzu.

Sei $k = 4$.
Es gilt: $i \in \{1, 2, 3, 5\}$ und $j \in \{1, 2, 3\}$:

$$\begin{pmatrix} 0 & 1 & 0 & \mathit{1} & 0 \\ 0 & 0 & 0 & \mathbf{1} & 0 \\ 1 & 1 & 0 & \mathit{1} & 0 \\ \mathit{1} & \mathit{1} & \mathbf{1} & \mathbf{0} & \mathbf{0} \\ 1 & \mathit{1} & 1 & \mathit{1} & 0 \end{pmatrix}$$

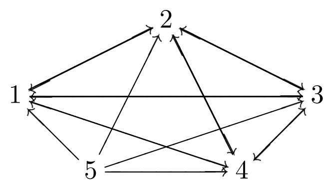


Wir fügen die Verbindungen $1 \to 3$, $2 \to 1$ und $2 \to 3$ hinzu.

Sei $k = 5$.
Es gilt: $i \in \emptyset$, daher verändert dieser Schritt nichts.

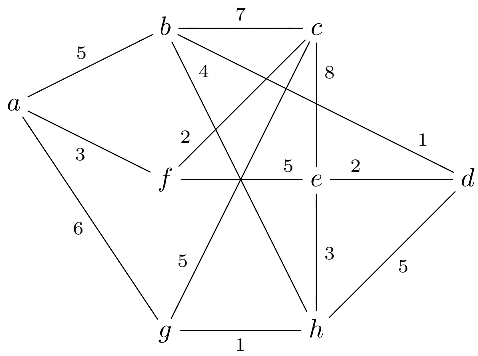
Ergebnis:

$$\begin{pmatrix} 0 & 1 & \mathit{1} & \mathit{1} & 0 \\ \mathit{1} & 0 & \mathit{1} & 1 & 0 \\ 1 & 1 & 0 & \mathit{1} & 0 \\ \mathit{1} & \mathit{1} & 1 & 0 & 0 \\ 1 & \mathit{1} & 1 & \mathit{1} & 0 \end{pmatrix}$$



2. **Prim und MST**

   Berechnen Sie mit dem Prim-Algorithmus einen MST für folgenden Graphen bei Start in Knoten a. Beschreiben Sie den zeitlichen Verlauf der Prioritätswarteschlange.



Wir haben

$$Q = \{(a,\infty),(b,\infty),(c,\infty),(d,\infty),(e,\infty),(f,\infty),(g,\infty),(h,\infty)\}$$

sowie

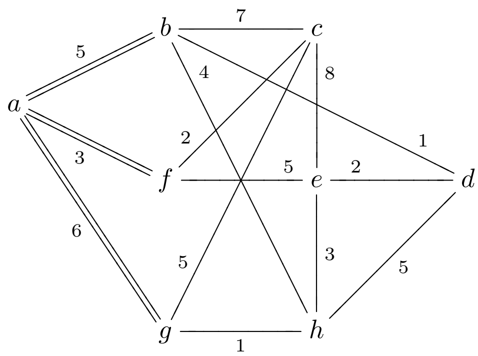
$$\Pi = \{(a,\text{nil}),(b,\text{nil}),(c,\text{nil}),(d,\text{nil}),(e,\text{nil}),(f,\text{nil}),(g,\text{nil}),(h,\text{nil})\}$$

Sei $\mathrm{w}(x,y)$ das Gewicht der Kante zwischen a und b, $\mathrm{key}(x) = y$ mit $(x,y) \in Q$ und $\pi(x)$ der π-Zeiger von x.

Wir starten bei Knoten a und löschen diesen aus Q. Nachbarn von a sind b, f, g.

- b: Es ist $b \in Q$ und $\mathrm{w}(a,b) = 5 < \infty = \mathrm{key}(b)$. Also setzen wir $\pi(b) = a$ und $\mathrm{key}(b) = 5$.
- f: Es ist $f \in Q$ und $\mathrm{w}(a,f) = 3 < \infty = \mathrm{key}(f)$. Also setzen wir $\pi(f) = a$ und $\mathrm{key}(f) = 3$.
- g: Es ist $g \in Q$ und $\mathrm{w}(a,g) = 6 < \infty = \mathrm{key}(g)$. Also setzen wir $\pi(g) = a$ und $\mathrm{key}(g) = 6$.

Wir erhalten



Wir haben

$$Q = \{(b,5),(c,\infty),(d,\infty),(e,\infty),(f,3),(g,6),(h,\infty)\}$$

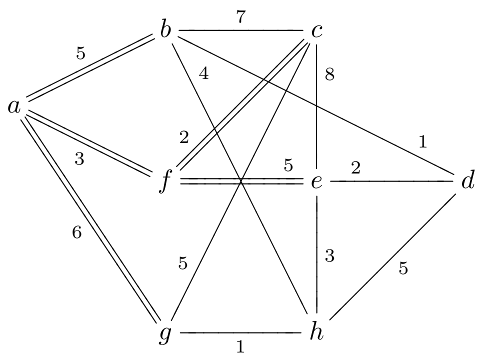
sowie

$$\Pi = \{(a,\mathrm{nil}),(b,a),(c,\mathrm{nil}),(d,\mathrm{nil}),(e,\mathrm{nil}),(f,a),(g,a),(h,\mathrm{nil})\}$$

Als nächstes löschen wir f aus Q, da minimales Element in Q. Nachbarn von f sind a, c, e.

- a: a ist nicht mehr in Q.
- c: Es ist $c \in Q$ und $\mathrm{w}(f,c) = 2 < \infty = \mathrm{key}(c)$. Also setzen wir $\pi(c) = f$ und $\mathrm{key}(c) = 2$.
- e: Es ist $e \in Q$ und $\mathrm{w}(f,e) = 5 < \infty = \mathrm{key}(e)$. Also setzen wir $\pi(e) = f$ und $\mathrm{key}(e) = 5$.

Wir erhalten



Wir haben nun

$$Q = \{(b,5),(c,2),(d,\infty),(e,5),(g,6),(h,\infty)\}$$

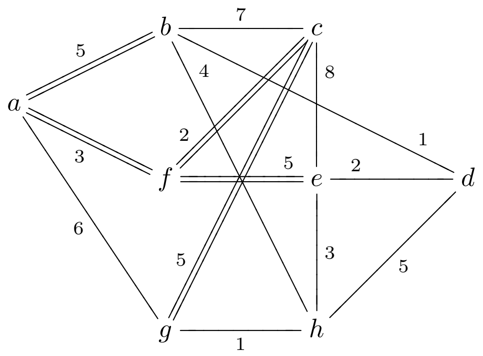
sowie

$$\Pi = \{(a,\mathrm{nil}),(b,a),(c,f),(d,\mathrm{nil}),(e,f),(f,a),(g,a),(h,\mathrm{nil})\}$$

Als nächstes löschen wir c aus Q, da minimales Element in Q. Nachbarn von c sind $b, e, f, g,$.

- b: Es ist $b \in Q$, aber $\mathrm{w}(c,b) = 7 \not< 5 = \mathrm{key}(b)$.
- e: Es ist $e \in Q$, aber $\mathrm{w}(c,e) = 8 \not< 5 = \mathrm{key}(e)$.
- f: Es ist $f \notin Q$.
- g: Es ist $g \in Q$ und $\mathrm{w}(c,g) = 5 < 6 = \mathrm{key}(g)$. Also setzen wir $\pi(g) = c$ und $\mathrm{key}(g) = 5$.

Wir erhalten



Wir haben nun

$$Q = \{(b,5), (d,\infty), (e,5), (g,5), (h,\infty)\}$$

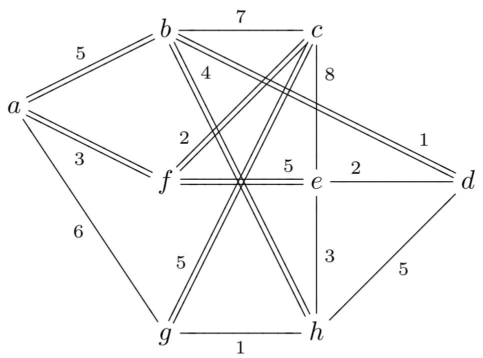
sowie

$$\Pi = \{(a, \mathrm{nil}), (b,a), (c,f), (d, \mathrm{nil}), (e,f), (f,a), (g,c), (h, \mathrm{nil})\}$$

Nun löschen wir b aus Q, da minimales Element in Q. Nachbarn von b sind a, c, d, h.

- a: Es ist $a \notin Q$.
- c: Es ist $c \notin Q$.
- d: Es ist $d \in Q$ und $\mathrm{w}(b,d) = 1 < \infty = \mathrm{key}(d)$. Also setzen wir $\pi(d) = b$ und $\mathrm{key}(d) = 1$
- h: Es ist $h \in Q$ und $\mathrm{w}(b,h) = 4 < \infty = \mathrm{key}(h)$. Also setzen wir $\pi(h) = b$ und $\mathrm{key}(h) = 4$.

Wir erhalten



Wir haben nun

$$Q = \{(d,1), (e,5), (g,5), (h,4)\}$$

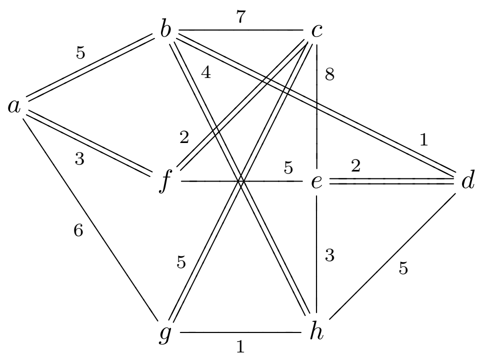
sowie

$$\Pi = \{(a, \mathrm{nil}), (b,a), (c,f), (d,b), (e,f), (f,a), (g,c), (h,b)\}$$

Nun löschen wir d aus Q, da minimales Element in Q. Nachbarn von d sind b, e, h.

- b: Es ist $b \notin Q$.
- e: Es ist $e \in Q$ und $\mathrm{w}(d, e) = 2 < 5 = \mathrm{key}(e)$. Also setzen wir $\pi(e) = d$ und $\mathrm{key}(e) = 2$
- h: Es ist $h \in Q$, aber $\mathrm{w}(d, h) = 5 \not< 4 = \mathrm{key}(h)$.

Wir erhalten



Wir haben nun

$$Q = \{(e, 2), (g, 5), (h, 4)\}$$

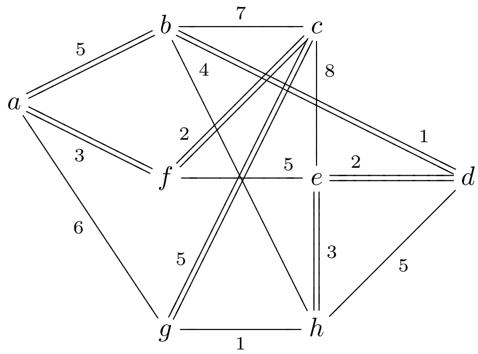
sowie

$$\Pi = \{(a, \mathrm{nil}), (b, a), (c, f), (d, b), (e, d), (f, a), (g, c), (h, b)\}$$

Als nächstes löschen wir e aus Q, da minimales Element in Q. Nachbarn von e sind c, d, f, h.

- c: Es ist $c \notin Q$.
- d: Es ist $d \notin Q$.
- f: Es ist $f \notin Q$.
- h: Es ist $h \in Q$ und $\mathrm{w}(e, h) = 3 < 4 = \mathrm{key}(h)$. Also setzen wir $\pi(h) = e$ und $\mathrm{key}(h) = 3$

Wir erhalten



Wir haben nun

$$Q = \{(g, 5), (h, 3)\}$$

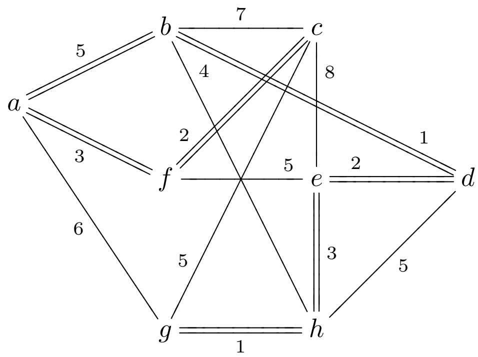
sowie

$$\Pi = \{(a, \mathrm{nil}), (b, a), (c, f), (d, b), (e, d), (f, a), (g, c), (h, e)\}$$

Nun löschen wir h aus Q, da minimales Element in Q. Nachbarn von h sind b, d, e, g.

- b: Es ist $b \notin Q$.
- d: Es ist $d \notin Q$.
- e: Es ist $e \notin Q$.
- g: Es ist $g \in Q$ und $\mathrm{w}(h, g) = 1 < 5 = \mathrm{key}(g)$. Also setzen wir $\pi(g) = h$ und $\mathrm{key}(g) = 1$

Wir erhalten



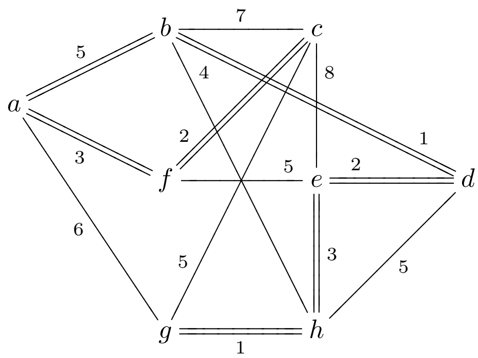
Wir haben nun

$$Q = \{((g, 1)\}$$

sowie

$$\Pi = \{(a, \mathrm{nil}), (b, a), (c, f), (d, b), (e, d), (f, a), (g, h), (h, e)\}$$

Zuletzt nehmen wir g aus Q; Nachbarn von g sind a, c, h. Da $Q = \emptyset$, ändert sich nichts; wir erhalten



3. O(**Kruskal**)

   Analysieren Sie die Laufzeit des Kruskal-MST-Algorithmus in O-Notation. Jede Operation MakeSet und FindSet hat konstante Kosten. Die Kosten aller UnionFind-Operationen zusammen sind $O(|V|log_2|V|)$.

   Betrachte folgenden Pseudocode:

```
1    A ← ∅
2    for alle v ∈ V do
3        makeSet (v)
4    SortiereAufsteigend (Kanten)
5    for alle Kante(u,v) in Kanten do
6        if FindSet(u) ≠ FindSet(v) then
7            A ← A ∪ {(u,v)}
8            Union(u,v)
```

   Wir wissen: MakeSet $= O(1)$ und FindSet $= O(1)$. Die Kosten aller UnionFind-Operationen sind $O(|V| \log_2 |V|)$.

   - Zeile 1: $O(1)$
   - Zeile 2: $|V|$ Zuweisungen $\rightarrow O(|V|)$
   - Zeile 3: n mal konstanter Aufwand $\rightarrow O(|V|)$
   - Zeile 4: Sortierung hat maximal Aufwand $O(|E| \log_2 |E|)$.
   - Zeile 5: |Kanten| Zuweisungen $\rightarrow O(|E|)$
   - Zeile 6: Beide Findset und if haben $O(1)$, also insgesamt $O(|E|)$
   - Zeile 7: Zuweisung hat $O(1)$, also folgt $O(|E|)$
   - Zeile 8: Unionfunktion hat insgesamt $O(|V| \log_2 |V|)$.

   Daraus folgt: Kruskal hat $O(1) + O(|V|) + O(|V|) + O(|E| \log_2 |E|) + O(|E|) + O(|E|) + O(|E|) + O(|V| \log_2 |V|) = O(|V| \log_2 |V| + |E| \log_2 |E|) = O(|V|^2 \log_2 |V|)$ (da $|E| \leq |V|^2$).

4. **ÖPNV**

Modelliere Graph und finde Algorithmus (in Pseudocode) für ÖPNV-Problem, der für Starthaltestelle billigste Verbindung zu allen anderen Strecken ausgibt.

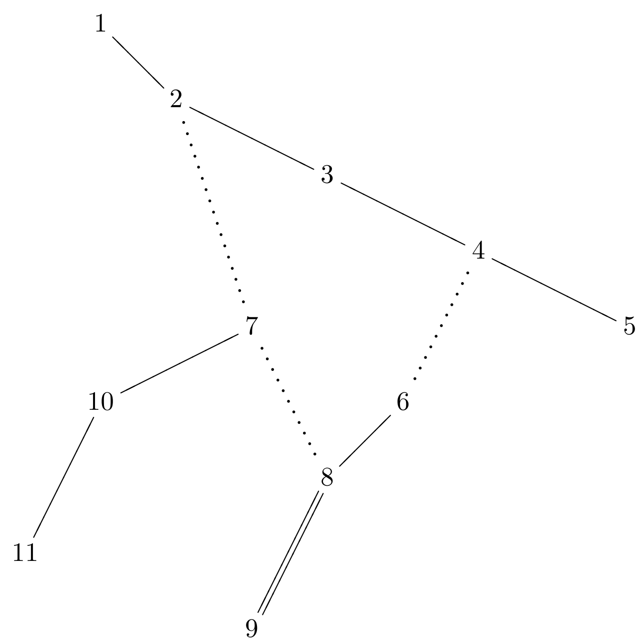
Wir haben Graph $G = (V, E)$ mit $V = \{x_1, x_2, x_3, ..., x_n\}$ Haltestellen und $E = \{(x_i, x_j) \mid \text{existiert Linie zwischen } x_i \text{ und } x_j\}$.
Die Linien sind gespeichert in $L = \{l_1, l_2, ..., l_m\}$. Für die l_i gilt: $l_i = \{x_i | x_i \text{ in Linie } l_i \text{ enthalten}\}$.

Das Programm funktioniert in zwei Schritten:

(a) Eingabe der Starthaltestelle, Belegung der Kanten, Berechnung des MST

(b) (Beliebig oft) Eingabe einer Zielhaltestelle, Berechnung des Preises

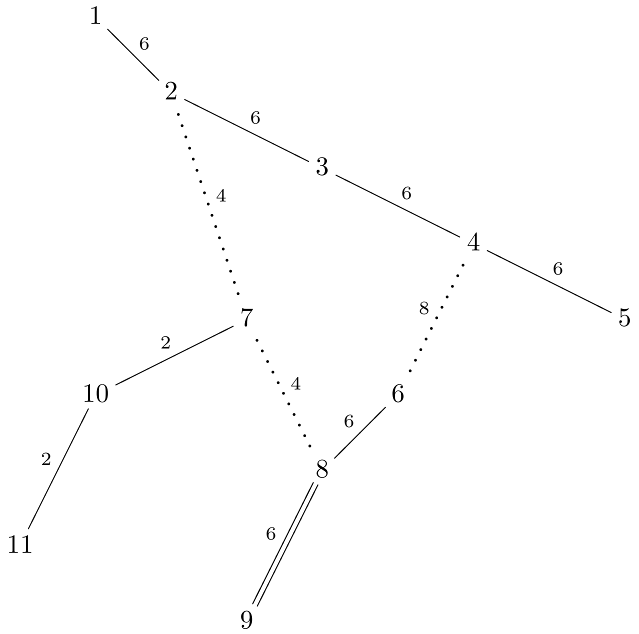
Bei Eingabe der Starthaltestelle werden die Kanten des Graphen mit einem Wert versehen; dieser gibt an, um wieviel sich die Reisekosten steigern, wenn man diesen Weg wählt. Ein Beispiel:



Wir haben hier die Linien $L = \{l_1, l_2, l_3, l_4, l_5\}$ mit $l_1 = \{1, 2, 3, 4, 5\}$, $l_2 = \{2, 7, 8, 9\}$, $l_3 = \{7, 10, 11\}$, $l_4 = \{6, 8, 9\}$, $l_5 = \{4, 6\}$.
Startknoten sei 10.

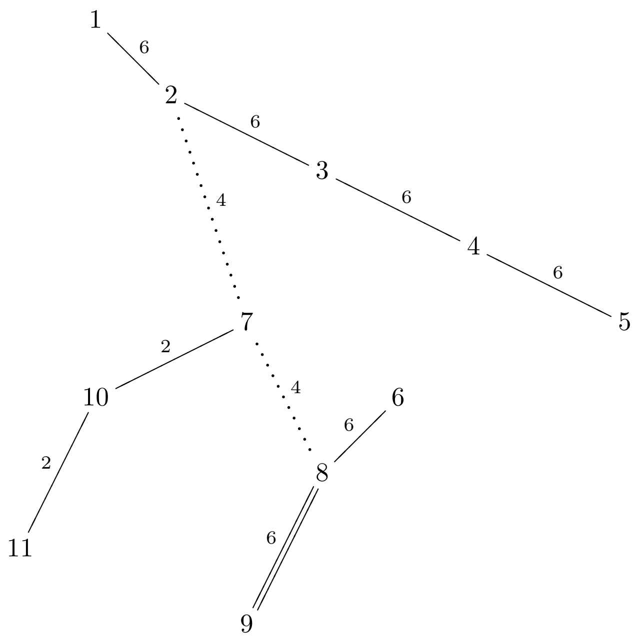
Zuerst werden (von der 10 ausgehend) alle Kanten belegt nach folgendem Schema: Sind x_1 und x_2 miteinander verbunden, so gibt der Wert der Kante an, was es zusätzlich kosten wird, von x_1 nach x_2 zu kommen. Dabei wird in einer Art Tiefensuche vorgegangen: Von einem Knoten ausgehend schaut man, ob es andere Linien gibt, die ihn berühren; falls ja, so wechselt man zu der anderen Linie und belegt alle Punkte der Linie mit Preis+2. Das macht man für alle Linien, die den Punkt berühren. Danach - und falls nein - setzt man die Suche fort mit der aktuellen Linie (und nicht mit Preis+2, sondern mit Preis). Dabei ist es wichtig, höhere oder gleiche Preise auf den Kanten zu überschreiben; bei niedrigeren Preisen stoppt man.

Es wird also folgendes berechnet:



Nun ermittelt man z.B. mit Prim den MST.

Nach dieser (aufwendigeren) Initialisierung erhält man einen MST; für eine Zielanfrage führt man einfach eine Suche durch den MST durch und gibt den Wert der letzten Kante aus. Der MST wäre in diesem Fall:



Hier der erste Teil des Algorithmus:

```
funktion suche (Knoten, Linien, Preis)
    wenn |Linien| > 1:
        Linien2 = Linien
        solange
```

```
5                 l = extractLinie(Linien2)
6             bis  Knoten in l oder Linien2 = ∅
7
8             wenn (L ohne l) ≠ ∅ dann
9                 suche(Knoten, (L ohne l), Preis)
10
11
12        wenn Knoten in l dann
13            macheGrau (Knoten)
14            für  alle  y in  Adj(Knoten)
15                wenn weiss(y) und (y in l) und (w(Knoten, y) ≥ Preis) dann
16                    w(Knoten, y) = Preis
17                    suche (y, L, Preis)
18                sonst wenn weiss(y) und (y nicht in l) und (w(Knoten, y) ≥ Preis) dann
19                    w(Knoten,y) = Preis + 2
20                    suche(y,L ohne l, Preis+2)
21
22            macheWeiss(Knoten)
```

Aufgerufen wird der Code mit suche (Startknoten, AlleLinien, 2).

Das Prinzip ist folgendes:

(a) Der Algorithmus wird mit dem Startwert, einer Liste mit allen Linien und dem Preis 2 EUR (Mindestpreis) aufgerufen.

(b) Zuerst wird geschaut, welche von den übergebenen Linien an dem Punkt vorbeikommen (Zeilen 1 bis 6).

(c) Für jede der Linien, die durch den Punkt laufen, wird der Algorithmus rekursiv aufgerufen, aber ohne die entsprechende Linie.
Und zwar solange rekursiv, bis keine Linie mehr übrig ist, die durch den Punkt läuft.(Zeile 8)

(d) Nun wird die aktuelle Haltestelle zunächst grau gefärbt (Zeile 13), um zu verhindern, dass wir im Kreis laufen.

(e) Dann wird ein Nachbar vom aktuellen Punkt ermittelt; ist dieser eine Haltestelle der aktuellen Linie, wird der Fahrpreis auf die Kante gesetzt und zu dem Nachbarn gesprungen. (Zeilen 16,17)

(f) Ist der Nachbar kein Punkt der Linie, wird der Preis erhöht und trotzdem weitergemacht. (Zeilen 19,20)
BEACHTE dass dies auch geschehen kann, wenn der Nachbar über eine andere Linie verbunden ist, so dass eigentlich keine Kostenerhöhung entstehen müsste. In diesem Fall wird das von einer der früheren Rekursionsinstanzen wieder reviediert.
Es ist dennoch NÖTIG so zu verfahren, man denke z.B. an folgenden Fall:
Eine Linie, die über Punkt a, b und c verläuft, eine Linie, die nur zwischen Punkt c und d verläuft, eine Linie, die über b, c, d und e geht:

$$a \longrightarrow b \longrightarrow c = d \qquad e$$

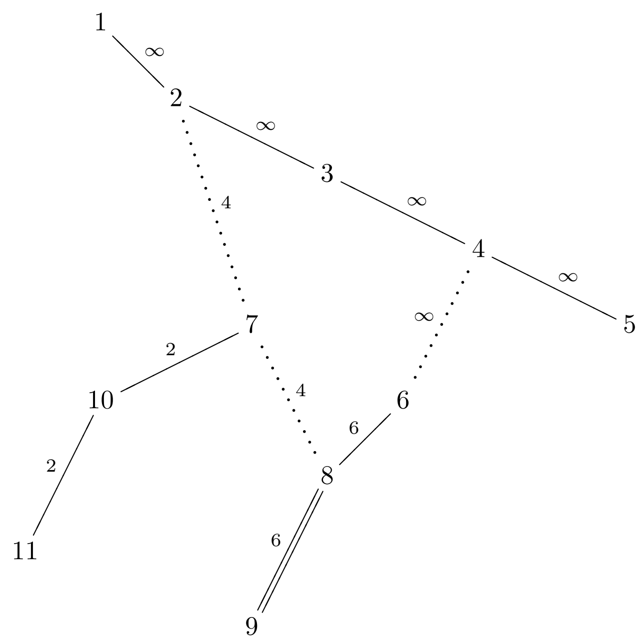
$$\xrightarrow{\qquad\qquad\qquad}$$

Würde man bei a anfangen, über b gehen, bei c landen, hätte man 2 EUR bezahlt. Wenn man zu d ginge, wäre man bei 4. Von d aus wäre es zu e aber dann schon 6 EUR, und dass verhindert man dadurch, dass man alle Linien durchgeht - dann werden die 6 EUR später zu 4 EUR gemacht.

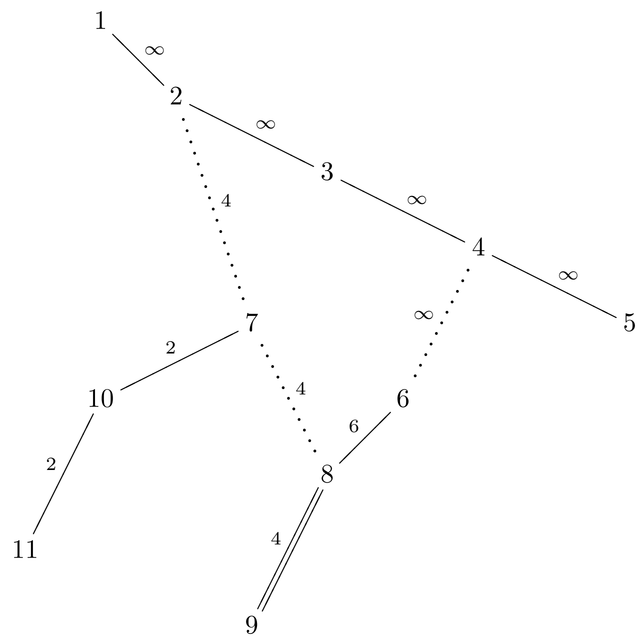
(g) Am Ende wird der Knoten wieder weiss gemacht, um bei späteren Suchen nicht an der falschen Stelle zu blockieren.

Prinzipiell also wie eine Tiefensuche. Der Trick ist halt, dass wir an jeder Haltestelle in jede Linie einmal umsteigen, und somit den Graph zuerst mit den teuersten Werten zu füttern, diese dann aber Stück für Stück zu revidieren. z.B. könnte der Graph so aussehen an zwei Zeitpunkten t_1 und t_2 mit $t_1 < t_2$:

Zeitpunkt t_1:



Zeitpunkt t_2:



Hier wurde die 6 zwischen 8 und 9 ersetzt durch eine 4.

Der Algorithmus ist natürlich sehr aufwendig, da er - je nach Liniennetz - manche Teile öfters durchgeht; dennoch wird so abgesichert, dass alle Kanten die billigsten sind, und außerdem ist die Berechnung ja nur einmalig - danach wird ja direkt auf den MST zugegriffen, der hat dann eine Performance von mindestens $O(log_2 n)$ und höchstens $O(n)$...

Für die Suche empfielt es sich, je nach Liniennetz und daraus entstehenden MST-Baum eine handelsübliche Breiten- oder Tiefensuche zu verwenden. Die Fahrtkosten sind dann die Kosten, die auf der letzten benutzten Kante standen.