

4 Graphen

4.1 Motivation: Wozu braucht man Graphen?

Viele reale Beziehungen lassen sich durch Graphen darstellen:

- Beziehungen zwischen Personen:
 - Person A kennt Person B
 - Person A spielt gegen Person B
- Verbindungen zwischen Punkten:
 - Straßen
 - Eisenbahn
 - Telefonnetz
 - elektronische Schaltungen
 - ...

Bezogen auf einen Graphen ergeben sich spezielle Aufgaben und Fragen:

- Existiert eine Verbindung zwischen A und B ? Existiert eine zweite Verbindung, falls die erste blockiert ist?
- Wie lautet die kürzeste Verbindung von A nach B ?
- Wie sieht ein minimaler Spannbaum zu einem Graphen aus?
- Wie plane ich eine optimale Rundreise? (*Traveling Salesman Problem*)

Begriffe:

- Knoten (“Objekte”)
- Kanten
- gerichtet/ ungerichtet
- gewichtet/ ungewichtet

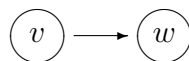
4.2 Definitionen und Graph-Darstellungen

Definition: Ein *gerichteter Graph* (engl. digraph = „directed graph“) ist ein Paar $G = (V, E)$ mit einer endlichen, nichtleeren Menge V , deren Elemente Knoten (nodes, vertices) heißen, und einer Menge $E \subseteq V \times V$, deren Elemente Kanten (edges, arcs) heißen.

Bemerkungen:

- $|V| =$ Knotenanzahl
- $|E| \leq |V|^2 =$ Kantenanzahl
(wenn man alle Knoten-Paare zulässt, manchmal auch: $|E| = \frac{|V|(|V| - 1)}{2}$)
- Meist werden die Knoten durchnummeriert: $i = 1, 2, \dots, |V|$

Graphische Darstellung einer Kante e von v nach w :



Begriffe:

- v ist *Vorgänger* von w
- w ist *Nachfolger* von v
- v und w sind *Nachbarn* bzw. *adjazent*

Weitere Definitionen:

- *Grad* eines Knotens := Anzahl der ein- und ausgehenden Kanten
- Ein *Pfad* ist eine Folge von Knoten v_1, \dots, v_n mit $(v_i, v_{i+1}) \in E$ für $1 \leq i < n$, also eine Folge von „zusammenhängenden“ Kanten.

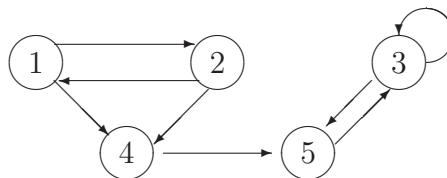


Abbildung 4.1: Beispiel-Graph G_1 (aus Güting, S.145-146). Das Beispiel wird später für den Graph-Durchlauf benötigt werden.

- *Länge eines Pfades* := Anzahl der Kanten auf dem Pfad
- Ein Pfad heißt *einfach*, wenn alle Knoten auf dem Pfad paarweise verschieden sind.
- Ein *Zyklus* ist ein Pfad mit $v_1 = v_n$ und Länge $= n - 1 \geq 2$.
- Ein *Teilgraph* eines Graphen $G = (V, E)$ ist ein Graph $G' = (V', E')$ mit $V' \subseteq V$ und $E' \subseteq E$.

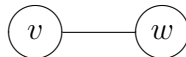
Man kann Markierungen oder Beschriftungen für Kanten und Knoten einführen.
Wichtige Kostenfunktionen oder -werte für Kanten sind:

- $c[v, w]$ oder $cost(v, w)$
- Bedeutung: Reisezeit oder -kosten (etwa die Entfernung zwischen v und w)

Definition: Ein *ungerichteter Graph* ist ein gerichteter Graph, in dem die Relation E symmetrisch ist:

$$(v, w) \in E \Rightarrow (w, v) \in E$$

Graphische Darstellung (ohne Pfeil):



Bemerkung: Die eingeführten Begriffe (Grad eines Knoten, Pfad, ...) sind analog zu denen für gerichtete Graphen. Bisweilen sind Modifikationen erforderlich, z.B. muß ein Zyklus hier mindestens drei Knoten haben.

4.2.1 Graph-Darstellungen

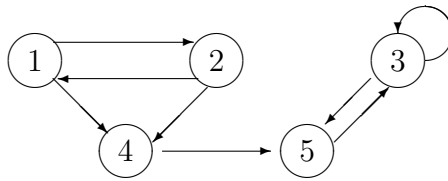
Man kann je nach Zielsetzung den Graphen knoten- oder kanten-orientiert abspeichern, wobei erstere Darstellungsform weitaus gebräuchlicher ist und in vielen verschiedenen Variationen existiert. Man identifiziert zur Vereinfachung der Adressierung die Knoten mit natürlichen Zahlen.

Sei daher $V = \{1, 2, \dots, |V|\}$.

- Bei der *knoten-orientierten Darstellung* existieren folgende Darstellungsformen:
 - Die **Adjazenzmatrix** A eine boolesche Matrix mit:

$$A_{ij} = \begin{cases} true & \text{falls } (i, j) \in E \\ false & \text{andernfalls} \end{cases}$$

Eine solche Matrix A_{ij} läßt sich als Array $A[i, j]$ darstellen. Damit folgt für den Beispiel-Graph G_1 mit der Konvention $true = 1, false = 0$:

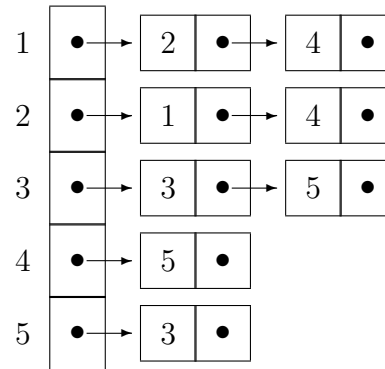
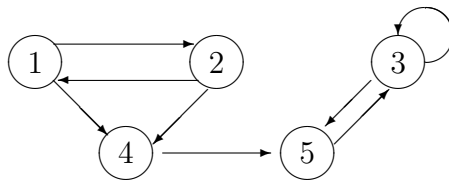


		nach				
		1	2	3	4	5
v o n	1		1		1	
	2	1			1	
	3			1		1
	4					1
	5			1		

Vor- und Nachteile:

- + Entscheidung, ob $(i, j) \in E$ in Zeit $O(1)$
- + erweiterbar für Kantenmarkierungen und Kosten(-Funktionen)
- Platzbedarf stets $O(|V|^2)$, ineffizient falls $|E| \ll |V|^2$
- Initialisierung benötigt Zeit $O(|V|^2)$
- Im Fall der **Adjazenzliste** wird für jeden Knoten eine Liste der Nachbarknoten angelegt (Vorgänger: "invertierte" Adjazenzliste).

Es ergibt sich für den Graphen G_1 :



Vor- und Nachteile:

- + geringer Platzbedarf von $O(|V| + |E|)$
- Entscheidung, ob $(i, j) \in E$ in Zeit $O(|E|/|V|)$ im Average Case
- *kanten-orientierte Darstellung*: eigener Index e für Kanten erforderlich. Prinzip: Adressierung jeder Kante mittels $e \in E$. Dabei ist für jede Kante gespeichert:
 - Vorgänger-Knoten
 - Nachfolger-Knoten
 - ggfs.: Markierung/ Kosten

Methode: meist statische Darstellung, seltener dynamische Listen.

Man unterscheidet verschiedene Typen von Graphen:

- *vollständiger* (complete) Graph: $|E| = |V|^2$ oder $|E| = \frac{|V|(|V| - 1)}{2}$

- *dichter* (dense) Graph: $|E| \simeq |V|^2$
- *dünnere* (sparse) Graph: $|E| \ll |V|^2$

Die Dichte eines Graphen ist das Hauptkriterium bei der Auswahl einer geeigneten Darstellungsform (mit dem Ziel einer optimalen Speicherplatz- und Zugriffs-Effizienz).

4.2.2 Programme zu den Darstellungen

Zunächst widmen wir uns der Implementation der Adjazenzmatrix-Darstellung für einen ungewichteten, gerichteten Graphen. Dabei sei V die Anzahl der Knoten und E die der Kanten.

```
PROGRAM adjmatrix (input, output);
  CONST maxV = 50;
  VAR j, x, y, V, E : INTEGER;
      a : ARRAY[1..maxV,1..maxV] OF BOOLEAN;
BEGIN
  readln (V, E);
  FOR x:= 1 TO V DO
    FOR y := 1 TO V DO a[x,y] := FALSE;
  FOR x:= 1 TO V DO a[x,x] := TRUE; (* spezielle Konvention *)
  FOR j := 1 TO E DO
    BEGIN
      readln(v1, v2); (* Kante wird eingelesen *)
      x := index(v1); y := index(v2); (* berechnet Index fuer jeden Knoten *)
      a[x,y] := TRUE;
      a[y,x] := TRUE; (* ungerichteter Graph *)
    END
  END;
END;
```

Anmerkung: Die Funktion `index(vertex)` ist immer dann nötig, wenn die Knoten nicht einfach durchnummeriert werden (andernfalls gilt `index(v)=v`).

Implementierung der Adjazenzliste:

```

PROGRAM adjlist (input, output);
  CONST maxV = 1000;
  TYPE link = REF node;
       node = RECORD
           v: INTEGER;
           next: link
       END;
  VAR j, x, y, V, E : INTEGER;
      t, z : link;
      adj : ARRAY[1..maxV] OF link; (* Listenbeginn fuer jeden Knoten *)
BEGIN
  readln (V,E);
  z := NEW(link); z↑.next := z;
  FOR j := 1 TO V DO adj[j] := z;
  FOR j:= 1 TO E DO
    BEGIN
      readln(v1,v2);
      x := index(v1); y:= index(v2);
      t := NEW(link); t↑.v := x; t↑.next := adj[y]; adj[y] := t;
      t := NEW(link); t↑.v := y; t↑.next := adj[x]; adj[x] := t; (* Symmetrie *)
    END
  END;

```

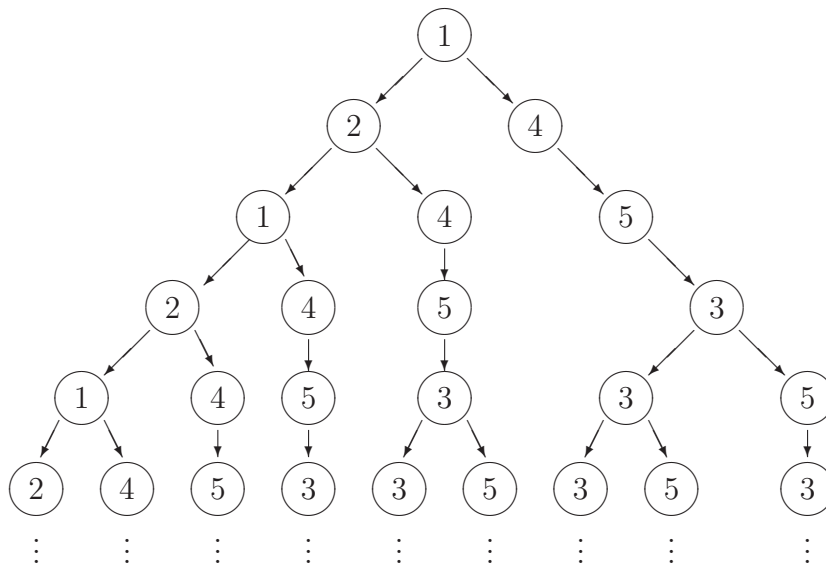
4.3 Graph-Durchlauf

Definition: Die *Expansion* $X_G(v)$ eines Graphen G in einem Knoten v ist ein Baum, der wie folgt definiert ist:

1. Falls v keine Nachfolger hat, ist $X_G(v)$ nur der Knoten v .
2. Falls v_1, \dots, v_k die Nachfolger von v sind, ist $X_G(v)$ der Baum mit der Wurzel v und den Teilbäumen $X_G(v_1), \dots, X_G(v_k)$.

Beachte:

- Die Knoten des Graphen kommen in der Regel mehrfach im Baum vor.

Abbildung 4.2: Beispiel Graph G_1 (s. Absch. 4.1) in seiner Expansion $X_{G_1}(v)$ mit $v = 1$.

- Ein Baum ist unendlich, falls der Graph Zyklen hat.
- Der Baum $X_G(v)$ stellt die Menge aller Pfade dar, die von v ausgehen.

4.3.1 Programm für den Graph-Durchlauf

Gleiches Konzept wie Baum-Durchlauf (vgl. Abschn. 1.3.5)

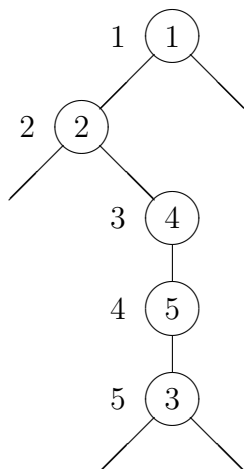
- Tiefendurchlauf: preorder traversal (depth first) mittels stack (oder rekursiv)
- Breitendurchlauf level order traversal (breadth first) mittels queue

Wichtige Modifikation:

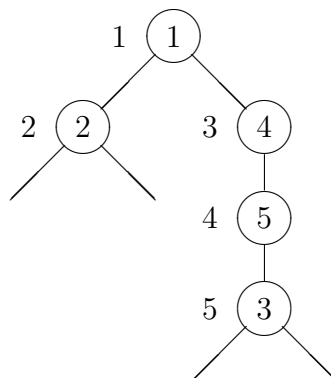
- Das Markieren der schon besuchten Knoten ist notwendig, weil Graph-Knoten im Baum mehrfach vorkommen können.
- Abbruch des Durchlaufs bei schon besuchten Knoten.

Beispiel G_1 mit Startknoten: $v = 1$

Tiefendurchlauf



Breitendurchlauf



Programm-Ansatz:

1. Initialisierung alle Knoten als „not visited“
2. Abarbeiten der Knoten
 - if** („node not visited“) **then**
 - bearbeite
 - markiere: „visited“

Interpretation: Graph-Durchlauf:

Während des Graph-Durchlaufs werden die Graph-Knoten v in 3 Klassen eingeteilt:

- Ungesehene Knoten (unseen vertices):
Knoten, die noch nicht erreicht worden sind:
 $val[v] = 0$
- Baum-Knoten (tree vertices):
Knoten, die schon besucht und abgearbeitet sind. Diese Knoten ergeben den „Suchbaum“:
 $val[v] = id > 0$
- Rand-Knoten (fringe vertices):
Knoten, die über eine Kante mit einem Baum-Knoten verbunden sind:
 $val[v] = -1$

Die Verallgemeinerung für ein beliebiges Auswahlkriterium führt zu folgendem Programm-Schema:

- Start: Markiere den Startknoten als Rand-Knoten und alle anderen Knoten als ungesehene Knoten.
- Schleife: **repeat**
 - Wähle einen Rand-Knoten x mittels eines Auswahlkriteriums (depth first, breadth first, priority first).
 - Markiere x als Baum-Knoten und bearbeite x .
 - Markiere alle ungesehenen Nachbar-Knoten von x als Rand-Knoten.
- ... **until** (alle Knoten abgearbeitet)

Setzt man dieses Vorgehen in ein Programm um, so erhält man für den Graph-Durchlauf folgendes Programm-Gerüst (angelehnt an Sedgewick 88).

```

PROCEDURE GraphTraversal;
  VAR id, k : INTEGER;
      val : ARRAY[1..maxV] OF INTEGER;
BEGIN
  id := 0;
  stackinit / queueinit; (* select one *)
  FOR k := 1 TO V DO val[k] := 0; (* val=0 means: not yet visited *)
  FOR k := 1 TO V DO
    IF val[k] = 0 THEN visit(k)
  END;

```

Die Unterschiede der verschiedenen Durchlauf- und Abspeicherungs-Arten zeigen sich in der Prozedur `visit(k)`.

Rekursiver Tiefendurchlauf mit Adjazenzliste

```

PROCEDURE visit (k : INTEGER);
  VAR t : link;
BEGIN
  id := id + 1; val[k] := id;
  t := adj[k];
  WHILE t ≠ z DO BEGIN
    IF val[t↑.v] = 0 THEN visit(t↑.v); (* val=0 means: not yet visited *)
    t := t↑.next
  END
END;

```

Rekursiver Tiefendurchlauf mit Adjazenzmatrix

```

PROCEDURE visit (k : INTEGER);
  VAR t : INTEGER;
BEGIN
  id := id + 1; val[k] := id;
  FOR t := 1 TO V DO
    IF a[k,t]
      THEN IF val[t] = 0 THEN visit(t);
  END;

```

Iterativer "Tiefendurchlauf" mit Stack und Adjazenzliste

```

PROCEDURE visit (k : INTEGER);
  VAR t : link;
BEGIN
  push (k);
  repeat
    k := pop; id := id + 1; val[k] := id;
    t := adj[k];
    WHILE t ≠ z DO
      BEGIN
        IF val[t↑.v] = 0 (* val=0 means: not yet visited *)
          THEN BEGIN
            push(t↑.v); val[t↑.v] := -1; (* val=-1: fringe vertex *)
          END;
        t := t↑.next
      END
    UNTIL stackempty
  END;

```

Komplexität des Tiefendurchlaufs (in allen Fällen):

- Zeit bei Verwendung einer Adjazenzliste: $O(|E| + |V|)$.
- Zeit bei Verwendung einer Adjazenzmatrix: $O(|V|^2)$.

Iterativer Breitendurchlauf mit Queue und Adjazenzliste

```

PROCEDURE visit (k : INTEGER);
  VAR t : link;
BEGIN
  put(k);
  REPEAT
    k := get; id := id + 1; val[k] := id;
    t := adj[k];
    WHILE t ≠ z DO
      BEGIN
        IF val[t↑.v] = 0 (* val=0 means: not yet visited *)
          THEN BEGIN
            put(t↑.v); val[t↑.v] := -1; (* val=-1: fringe vertex *)
          END;
        t := t↑.next
      END
    UNTIL queueempty
  END;

```

Hinweise zur nicht-rekursiven Implementation:

Knoten, die schon erreicht („touched“), aber noch nicht abgearbeitet sind, werden auf dem Stack oder in der Queue gehalten [rekursiv: lokale Variable t].

Implementierung mit Hilfe des Arrays val (vgl. Absch. 4.3.1)

$$val[v] = \begin{cases} 0 & \text{falls Knoten } v \text{ noch nicht erreicht} \\ -1 & \text{falls Knoten } v \text{ auf Stack / Queue} \\ id > 0 & \text{falls Knoten } v \text{ abgearbeitet} \end{cases}$$

Beachte: Die Reihenfolge ist nicht exakt dieselbe wie im rekursiven Programm.

4.4 Kürzeste Wege

Zu den wichtigen Verfahren auf Graphen gehören Verfahren zum Auffinden kürzester Wege (von einem Knoten zu einem anderen, von einem Knoten zu allen anderen oder auch von allen Knoten zu allen anderen).

4.4.1 Dijkstra-Algorithmus (Single-Source Best Path)

Der Dijkstra-Algorithmus berechnet den kürzesten Weg von einem Startknoten zu jedem anderen Knoten, d.h. den Weg mit minimalen Kosten. Dijkstra publizierte die 3-seitige (!) Originalarbeit im Jahre 1959.

Bemerkung: Als *kürzesten* Weg werden wir in diesem Zusammenhang der Weg mit minimalen *Kosten* bezeichnen, unabhängig von der Pfadlänge (= Anzahl der Kanten des Pfades).

Gegeben sei ein gerichteter Graph G mit der Bewertungsfunktion

$$c[v, w] \quad \begin{cases} \geq 0 & \text{falls eine Kante von } v \text{ nach } w \text{ existiert,} \\ = \infty & \text{falls keine Kante von } v \text{ nach } w \text{ existiert,} \\ = 0 & \text{für } w = v \text{ (spezielle Konvention, falls nötig.)} \end{cases}$$

Aufgabe: Der Start-Knoten v_0 und Endknoten w seien vorgegeben. Finde den Pfad $u_1^j := u_1, u_2, \dots, u_j$ mit minimalen Kosten von v_0 nach w , d.h. den Pfad

$$v_0 = u_0 \rightarrow u_1 \rightarrow u_2 \rightarrow \dots \rightarrow u_j = w$$

mit Startknoten $v_0 = u_0$ und Endknoten $w = u_j$ mit:

$$C[w] := \min_{\substack{j, u_0^j: \\ u_0=v_0, u_j=w}} \sum_{i=1}^j c[v_{i-1}, v_i]$$

Es wird sich zeigen, daß die folgende Verallgemeinerung nicht mehr Aufwand erfordert:

Sei der Start-Knoten v_0 vorgegeben. Bestimme die besten Pfade von v_0 zu *allen* (anderen) Knoten $v \in V$.

Wir sortieren die Knoten v nach den Kosten $C[v]$ und erhalten die Folge (paarweise verschiedener Knoten) $v_1, v_2, \dots, v_{k-1}, v_k, \dots, v_{n-1}$ ($|V| = n$) mit

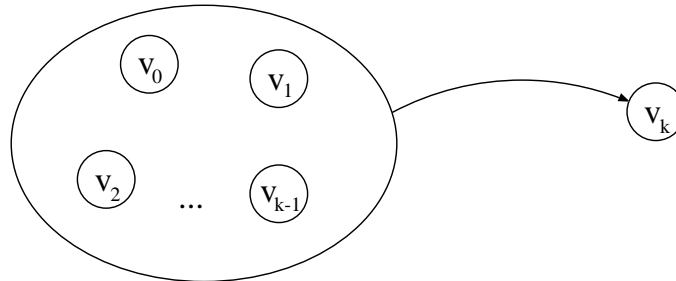
$$C[v_1] \leq C[v_2] \leq \dots \leq C[v_{k-1}] \leq C[v_k] \leq \dots \leq C[v_{n-1}]$$

Terminologie: Der Knoten v_k hat in der Folge v_0, v_1, \dots, v_{n-1} den *Rang* k .

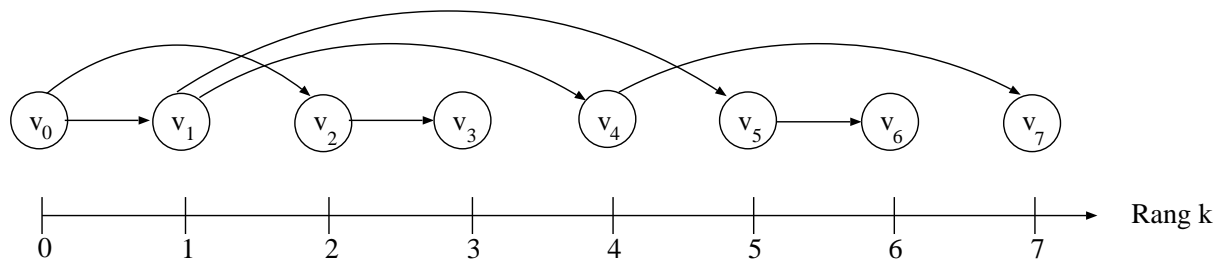
- a) Es muß gelten: Jeder Knoten v_k hat auf seinem besten Pfad einen *direkten* Vorgänger $\text{Pre}(v_k)$ mit

$$\text{Pre}(v_k) \in \{v_0, v_1, \dots, v_{k-1}\}$$

Veranschaulichung:



- b) Die Menge der besten Pfade zu v_1, v_2, \dots, v_{n-1} kann man als Baum darstellen.



Erläuterungen: Die besten Pfade werden systematisch aufgebaut, indem wir bereits bekannte beste Pfade um neue Kanten wachsen lassen:

- Durch Hinzunahme einer Kante können die Kosten nur wachsen.
- Sei w Knoten. Falls die besten Pfade von v_0 zu allen anderen Knoten ungleich w höhere Kosten haben, ist der beste Pfad von v_0 nach w gefunden.
- Der beste Pfad kann keinen Zyklus haben, da die Kosten des Zyklus größer Null sind. Sind diese Kosten gleich Null, so gibt es einen besten Pfad ohne Zyklus mit denselben Kosten.
- Der beste Pfad zu jedem Knoten hat maximal $(|V| - 1)$ Kanten.

Arbeitsweise des Algorithmus (Aho 83, S.205)

Der Dijkstra-Algorithmus baut die gesuchten kürzeste Pfade sukzessive aus schon bekannten kürzesten Pfaden auf. Das entspricht etwa einer äquidistanten Welle um den Startpunkt, für die während der Ausbreitung jeder Weg zu einem gerade erreichten Knoten vermerkt wird.

Sei V die Menge der Knoten.

Sei der Startknoten v_0 gegeben.

Wir bestimmen die Folge der Knoten v_1, v_2, \dots, v_{n-1} rekursiv.

Definition: $k = 0, 1, \dots, |V| - 1$

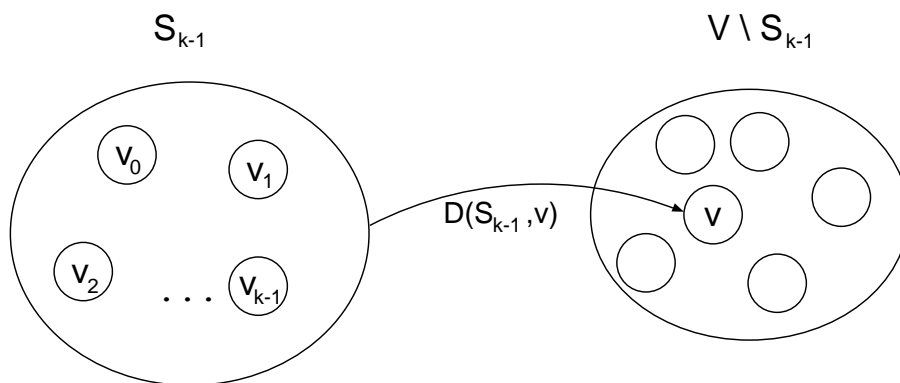
S_k := $\{v_0, v_1, \dots, v_k\}$
 = die Menge der Knoten v mit den k besten Pfaden $v_0 \rightarrow v_j$, $j = 1, \dots, k$
 $D(S_k, v)$:= die Kosten des besten Pfades $v_0 \rightarrow \dots \rightarrow v$ mit $\text{Pre}(v) \in S_k$

Dijkstra-Algorithmus:

- (1) INITIALIZATION
- (2) $S_0 := \{v_0\};$
- (3) $D(S_0, v) := c[v_0, v] \quad \forall v \in V \setminus S_0$
- (4) FOR each rank $k = 1$ TO $|V| - 1$ DO BEGIN
- (5) $v_k := \arg \min\{D(S_{k-1}, v) : v \in V \setminus S_{k-1}\};$
- (6) $S_k := S_{k-1} \cup \{v_k\};$
- (7) FOR each vertex $v \in (V \setminus S_k) \cap \text{Adj}(v_k)$ DO
- (8) $D(S_k, v) := \min\{D(S_{k-1}, v), D(S_{k-1}, v_k) + c[v_k, v]\}$
- (9) END;

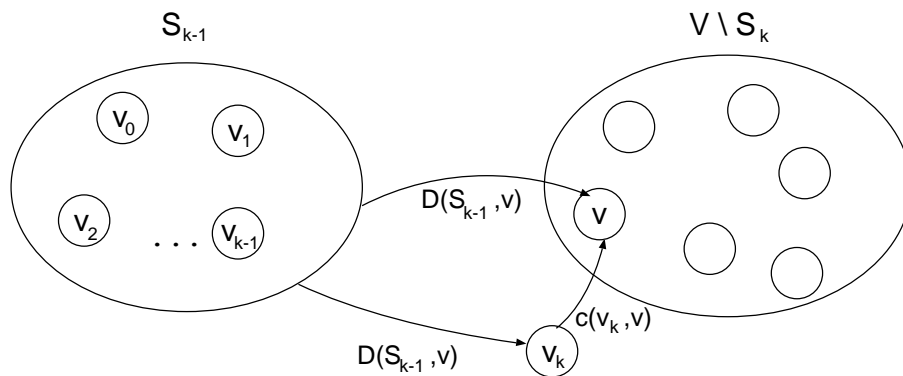
Ergebnis: minimale Kosten $C[v_k] = D(S_{k-1}, v_k)$

Veranschaulichung der beiden entscheidenden Operationen des Dijkstra-Algorithmus:



$$v_k := \arg \min_{v \in V \setminus S_{k-1}} D(S_{k-1}, v)$$

$$S_k := S_{k-1} \cup \{v_k\}$$



$$D(S_k, v) = \min\{D(S_{k-1}, v), D(S_{k-1}, v_k) + c[v_k, v]\}$$

Durch den Dijkstra-Algorithmus wird für jeden Rang $k = 0, \dots, |V| - 1$ die Knotenmenge V in drei disjunkte Teilmengen zerlegt:

- a) S_k : abgearbeitete Knoten,
d.h. der global beste Pfad zu diesen Knoten ist schon gefunden.
- b) $\{v \in V \setminus S_k : D(S_k, v) < \infty\}$
Randknoten, die von S_k aus erreicht worden sind.
- c) $\{v \in V \setminus S_k : D(S_k, v) = \infty\}$
Unerreichte Knoten, d.h. zu diesem Knoten existiert noch kein Pfad.

Eine Verallgemeinerung dieses Konzeptes führt zu dem sogenannten A^* -Algorithmus und verwandten heuristischen Suchverfahren, wie sie in der Künstlichen Intelligenz verwendet werden.

Der Index k wird letztlich nicht benötigt, so daß die abgearbeiteten Knoten v_k in *einer* Menge S (ohne Index) verwaltet werden können:

Seien $V := \{1, \dots, n\}$ und $S := \{\text{Knoten, die bereits abgearbeitet sind.}\}$

PROCEDURE Dijkstra;

{ Dijkstra computes the cost of the shortest paths from vertex 1
to every vertex of a directed graph }

BEGIN

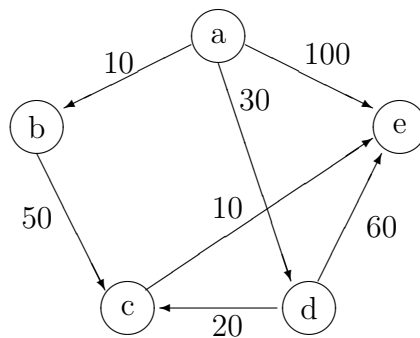
- (1) $S := \{1\};$
- (2) FOR $i := 2$ TO n DO
- (3) $D[i] := c[1, i];$ { initialize D }
- (4) FOR $i := 1$ TO $n - 1$ DO BEGIN
- (5) choose a vertex $w \in V \setminus S$ such that $D[w]$ is a minimum;
- (6) add w to S ;
- (7) FOR each vertex $v \in V \setminus S$ DO

(8) $D[v] := \min(D[v], D[w] + c[w, v])$
 END
 END; { Dijkstra }

Anmerkungen zum Dijkstra-Algorithmus:

- Der Dijkstra-Algorithmus liefert keine Näherung, sondern garantiert Optimalität.
- Bei negativen Bewertungen ist er nicht anwendbar. Existieren Zyklen mit negativer Bewertung, so gibt es keinen eindeutigen Pfad mit minimalen Kosten mehr.
- Frage bei einem vollständigen Graphen:
 Ist eine niedrigere Komplexität vorstellbar, wobei jede Kante weniger als einmal betrachtet wird?

Beispiel: [Aho 83, S.205]



k	v_k	S_k	$D(S_k, b)$	$D(S_k, c)$	$D(S_k, d)$	$D(S_k, e)$
0	—	{a}	10	∞	30	100
1	b	{a, b}	—	60	30	100
2	d	{a, b, d}	—	50	—	90
3	c	{a, b, d, c}	—	—	—	60
4	e	{a, b, d, c, e}	—	—	—	—

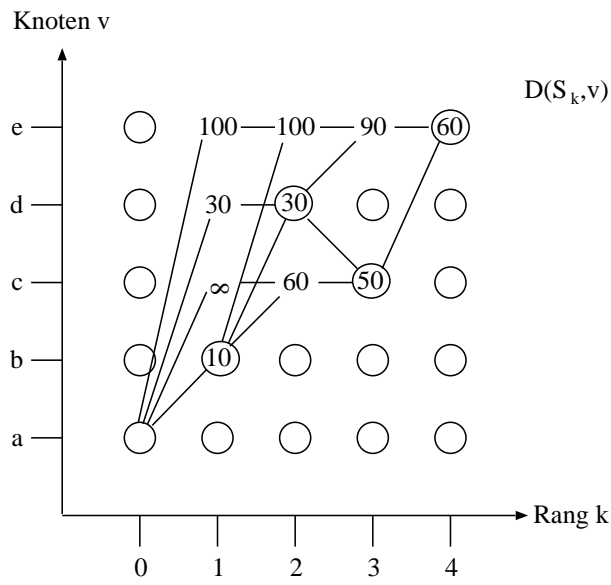
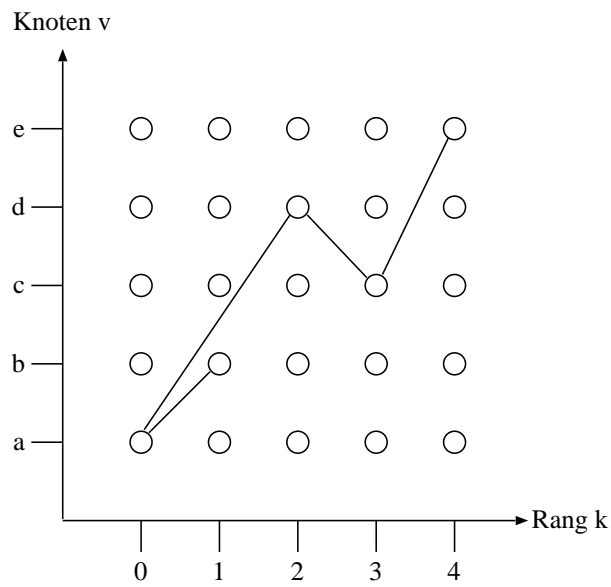


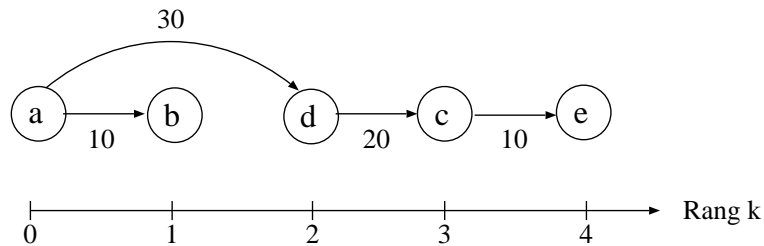
Tabelle der Kosten $D(S_k, v)$



Zurückverfolgung der besten Pfade

Reihenfolge der erreichten Knoten: a, b, d, c, e

Baumdarstellung der besten Pfade:



Übung: Erstelle das vollständige Programm und rekonstruiere den optimalen Pfad (Vorgängerknoten 'merken')

Komplexität des Dijkstra-Algorithmus:

Operation:

- A) Initialisierung: Zeilen (1-3): $O(|V|)$: vernachlässigbar gegen B und C
- B) Minimum-Operation: Zeilen (4-6)
- C) Update-Operation: Zeilen (4,7,8)

Wir betrachten drei Arten der Implementierung:

- Adjazenz-Matrix:
 - B: $O(|V|^2)$
 - C: $O(|V|^2)$ oder $O(|E|)$
 - insgesamt: $O(|V|^2)$
- Adjazenz-Liste und Priority-Queue:

Die Menge $V \setminus S_k$ der nicht abgearbeiteten Knoten mit den Abständen $D(S_k, v)$ wird in einer (erweiterten) Priority Queue verwaltet.

 - B: Es gibt $O(|V|)$ Durchgänge je Zeit $O(\log |V|)$:
 $O(|V| \log |V|)$:
 - C: über alle $O(|V|)$ Durchgänge zusammen müssen insgesamt $O(|E|)$ viele Werte aktualisiert werden mit je Zeit $O(\log |V|)$ (Einzelheiten ausarbeiten)
 - insgesamt: $O((|V| + |E|) \log |V|)$
- Adjazenz-Listen und Fibonacci-Heap (ohne Einzelheiten):
 - B: $O(|V| \log |V|)$
 - C: $O(|E|)$ im Mittel
 - insgesamt: $O(|E| + |V| \log |V|)$

4.4.2 DAGs und kürzeste Wege

DAG := directed acyclic graph
 = d.h. ein gerichteter Graph ohne Zyklen
 weighted DAG = gewichteter DAG

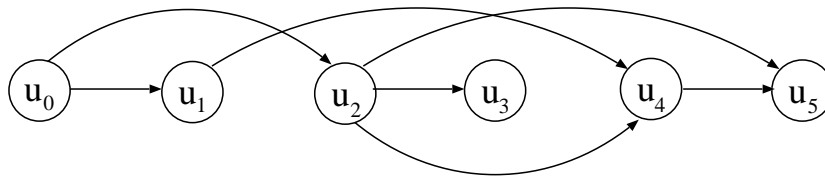
Definition: Eine topologische Sortierung eines gewichteten Graphen $G = (V, E)$ ist eine Durchnummerierung (Permutation) seiner Knoten

$$u_0, u_1, \dots, u_{n-1}$$

so daß für alle Kanten (u_i, u_j) gilt:

$$i < j$$

Anschauliche Interpretation: Durch topologische Sortierung werden die Knoten eines Graphen in eine "lineare Ordnung von links nach rechts" gebracht:



Es gibt keine Kanten, die rückwärts laufen.

Anmerkung: Da Zyklen ausgeschlossen sind, sind auch negative Kosten kein Problem.

Satz: Ein Graph ist azyklisch genau dann, wenn eine topologische Sortierung möglich ist.

Beweis: " \Leftarrow ": klar

" \Rightarrow ": Sei G ein azyklischer Graph. Dann muß G einen Startknoten v , also einen Knoten ohne Vorgänger, haben. (Ansonsten läßt sich ein Zyklus konstruieren, indem man von Vorgänger zu Vorgänger läuft.) Wir geben dem Knoten die Nummer 0, also $v = u_0$. Dann entfernen wir u_0 samt seiner ausgehenden Kanten aus G und wenden dieselbe Methode auf den Restgraphen an, um u_1 zu bestimmen usw. *q.e.d.*

In diesem Beweis wird *eine mögliche* Implementierung einer topologischen Sortierung betrachtet. Eine andere Methode verwendet den Tiefendurchlauf:
 resultierende Komplexität $O(|V| + |E|)$.

Bestimmung des besten Pfades: Annahme: Der DAG ist bereits topologisch sortiert:

$$0, 1, 2, 3, \dots, n - 1$$

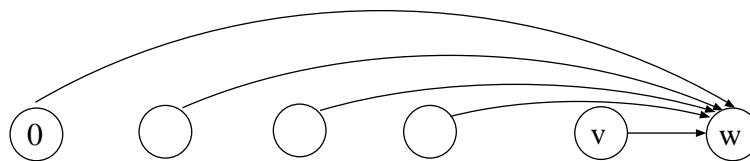
Sei der Startknoten 0. Betrachte den Pfad $u_0^j := u_0, u_1, \dots, u_j$ (mit $u_{i-1} < u_i$, $i = 1, \dots, j$).

Ansatz: dynamische Programmierung mit der Hilfsgröße:

$$\begin{aligned} C[w] &:= \text{Kosten des besten Pfades von } v_0 \text{ nach } w \\ &= \min_{\substack{j, u_1^j: \\ u_j = w}} \sum_{i=1}^j c[u_{i-1}, u_i] \end{aligned}$$

Zerlegung:

$$\begin{aligned} C[w] &= \min_{\substack{j, u_1^j: \\ u_j = w}} \left\{ \sum_{i=1}^{j-1} c[u_{i-1}, u_i] + c[u_{j-1}, u_j] \right\} \\ &= \min_v \left\{ \min_{\substack{j-1, u_1^{j-1}: \\ u_{j-1} = v}} \sum_{i=1}^{j-1} c[u_{i-1}, u_i] + c[v, w] \right\} \\ &= \min_v \{C[v] + c[v, w]\} \end{aligned}$$



Die rekursive Auswertung für $w = 0, \dots, n - 1$ ist einfach wegen der topologischen Sortierung des Graphen.

Frage: Gilt diese Rekursionsgleichung auch für das Dijkstra-Problem?

Ja, aber die effiziente Auswertung ist das Problem.

Implementierung:

a) mittels Adjazenz-Matrix:

$$O(|V|^2)$$

b) mittels Adjazenz-Liste für Vorgängerknoten:

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

Das heißt: Bei einem *dünnen* DAG ist dieser DP-Algorithmus deutlich effizienter als der Dijkstra-Algorithmus (zumindest in der Standardversion).

4.4.3 Floyd-Algorithmus (All-Pairs Best Path)

Im Jahre 1962 stellte Floyd einen Algorithmus vor, der alle kürzesten Pfade zwischen zwei beliebigen Knoten eines Graphen berechnet.

Prinzipiell machbar:

Dijkstra-Algorithmus für jeden Knoten anwenden, ergibt Zeitkomplexität $O(|V| \cdot |V|^2) = O(|V|^3)$.

Andere Möglichkeit: Floyd-Algorithmus (*dynamischer Programmierung*).

Sei $G = (V, E)$ ein gerichteter Graph mit $V = \{1, 2, \dots, n\}$, $n = |V|$ und nicht-negativer Bewertung

$$c[v, w] \quad \begin{cases} \geq 0 & \text{falls Kante } v \rightarrow w \text{ existiert} \\ = \infty & \text{andernfalls} \end{cases}$$

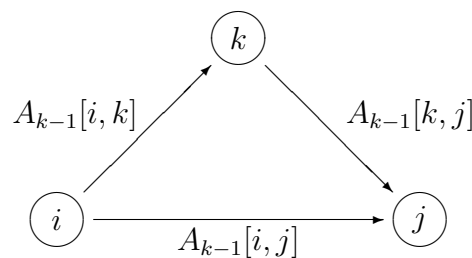
Wir definieren zunächst eine $|V| \times |V|$ -Matrix A_k mit den Elementen:

$$A_k[i, j] := \begin{array}{l} \text{minimale Kosten um vom Knoten } i \text{ zum Knoten } j \\ \text{zu gelangen über Knoten aus } \{1, \dots, k\}. \end{array}$$

Aufgrund der Definition gilt folgende Rekursionsgleichung (Gleichung der DP):

$$A_k[i, j] := \min\{A_{k-1}[i, j], A_{k-1}[i, k] + A_{k-1}[k, j]\}$$

Visualisierung der DP-Gleichung:



Vereinfachung:

Es ändert sich kein Element $A_k[i, j]$, wenn i oder j gleich k sind:

- $A_k[i, k] = A_{k-1}[i, k]$
- $A_k[k, j] = A_{k-1}[k, j]$

Man benötigt also nicht mehrere Matrizen $A_k[i, j]$, $k = 1, \dots, n$, sondern nur eine einzige Matrix $A[i, j]$.

Initialisierung: $A[i, j] = c[i, j] \quad \forall i, j \in \{1, \dots, |V|\}$

Programm zum Floyd-Algorithmus

Die Matrix C repräsentiert den Graphen in der Adjazenz-Matrix-Darstellung. Die Matrix A ist oben definiert. Die Matrix P speichert die jeweiligen Vorgänger (**P**redecessor) eines jeden Knotens auf dem besten Pfad und dient also der Rekonstruktion dieses Pfades [Aigner].

```

PROCEDURE Floyd (VAR A: ARRAY [1..n,1..n] OF REAL;
                  c: ARRAY [1..n,1..n] OF REAL;
                  P: ARRAY [1..n,1..n] OF INTEGER);
  VAR i, j, k : INTEGER;
BEGIN
  FOR i := 1 TO n DO
    FOR j := 1 TO n DO BEGIN
      A[i,j] := c[i,j];
      P[i,j] := 0;
    END;
  FOR i := 1 TO n DO A[i,i] := 0;
  FOR k := 1 TO n DO
    FOR i := 1 TO n DO
      FOR j := 1 TO n DO
        IF A[i,k] + A[k,j] < A[i,j]
          THEN BEGIN
            A[i,j] := A[i,k] + A[k,j];
            P[i,j] := k
          END
        END
      END
    END;
END;

```

Komplexität:

- Programm-Struktur: Schleifen über i , j und k
- Zeit $O(|V|^3)$
- Platz $O(|V|^2)$

4.4.4 Warshall-Algorithmus (1962)

Spezialfall des Floyd-Algorithmus für ‘unbewertete’ Graphen:

$$c[v, w] = \begin{cases} \mathbf{true} & \text{falls Kante } v \rightarrow w \text{ existiert} \\ \mathbf{false} & \text{andernfalls} \end{cases}$$

Es wird also die Existenz einer Verbindung zwischen jedem Knotenpaar geprüft. Die Matrix A beschreibt hier die *transitive Hülle* des Graphen.

Definition: Zwei Knoten v und w eines ungerichteten Graphen heißen *verbunden*, wenn es einen Pfad von v nach w gibt.

Definition: Zwei Knoten v und w eines gerichteten Graphen heißen *stark verbunden*, wenn es einen Pfad von v nach w und einen Pfad von w nach v gibt.

Definition: Gegeben sei ein gerichteter Graph $G = (V, E)$. Die *transitive Hülle* (transitive closure) von G ist der Graph $\bar{G} = (V, \bar{E})$, wobei \bar{E} definiert ist durch:

$$(v, w) \in \bar{E} \iff \text{es gibt in } G \text{ einen Pfad von } v \text{ nach } w.$$

Die entsprechende Modifikation des Floyd-Algorithmus ergibt den Algorithmus von Warshall, wobei statt der Real/Integer-Kosten-Matrix eine boolesche Matrix verwendet wird. Das Element $A[i, j]$ der booleschen Matrix gibt dabei an, ob ein Pfad von i nach j existiert.

```

PROCEDURE Warshall (VAR A: ARRAY [1..n,1..n] OF BOOLEAN;
                   c: ARRAY [1..n,1..n] OF BOOLEAN);
  VAR i, j, k : INTEGER;
BEGIN
  FOR i := 1 TO n DO
    FOR j := 1 TO n DO A[i,j] := c[i,j];
  FOR k := 1 TO n DO
    FOR i := 1 TO n DO
      FOR j := 1 TO n DO
        IF NOT A[i,j]
          THEN A[i,j] := A[i,k] AND A[k,j];
  END;

```

4.5 Minimaler Spannbaum

Alternative Bezeichnungen: Minimum Spanning Tree (MST), Spannbaum mit minimalen Kosten

ähnlich: Kruskal-Algorithmus 1957 (und auch Dijkstra nach Uminterpretation)

4.5.1 Definitionen

Gegeben sei ein ungerichteter Graph $G = (V, E)$:

- Ein Knotenpaar (v, w) heißt *verbunden*, wenn es in G einen Pfad von v nach w gibt. Der Graph G heißt *verbunden* (oder *zusammenhängend*), falls alle Knotenpaare verbunden sind.
- Ein verbundener Graph ohne Zyklen heißt *freier Baum*. Wählt man einen Knoten daraus als Wurzel, so erhält man einen (normalen) Baum.
- Ein *Spannbaum* ist ein freier Baum, der alle Knoten von G enthält und dessen Kanten eine Teilmenge von E bilden.
Ist G bewertet, dann definieren wir die Kosten eines Spannbaums von G als die Summe über die Kosten der Kanten des Spannbaums.

Definition: Sei G ein ungerichteter, bewerteter Graph. Dann heißt ein Spannbaum, für den die Summe über die Kosten seiner Kanten minimal ist, *minimaler Spannbaum* (Minimum Spanning Tree) von G .

Anwendungen: Minimale Spannbäume haben Anwendungen bei der Optimierung von Telefonnetzen, Straßennetzen und anderen Netzwerken.

Um einen minimalen Spannbaum zu einem gegebenen Graph zu berechnen, macht sich der **Prim-Algorithmus** (1957) eine Eigenschaft des minimalen Spannbaums (MST Property) zunutze, die im wesentlichen auf der additiven Zerlegbarkeit der Kostenfunktion beruht.

4.5.2 MST Property

Sei $G = (V, E)$ ein verbundener, ungerichteter Graph mit der Bewertungsfunktion $c[v, w] \geq 0$ für eine Kante (v, w) . Sei $U \subset V$.

Gilt für die Kante (u, v)

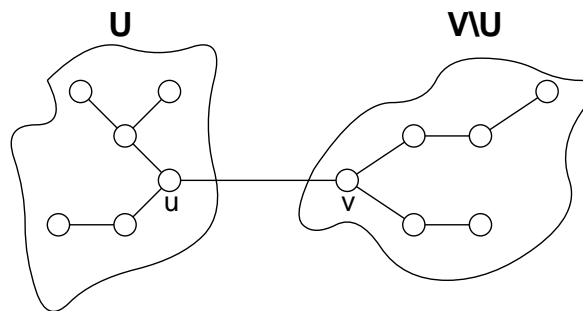
$$(u, v) = \arg \min \{c[u', v'] : u' \in U, v' \in V \setminus U\}$$

dann gibt es einen minimalen Spannbaum, der die Kante (u, v) enthält.

Beweisidee:

durch Widerspruch (ähnlich Optimalitätsprinzip; wiederum ist die Additivität der Kostenfunktion wesentlich):

Gälte die Aussage nicht, dann enthielte der minimale Spannbaum eine Kante (u', v') , die die beiden disjunkten Mengen verbindet und die „schlechter“ ist als (u, v) . Ersetzt man diese Kante entsprechend, so erhält man einen „besseren“ Spannbaum. Das führt zu einem Widerspruch zu der Aussage, daß der erste Spannbaum schon minimal ist.



Eigenschaft eines minimalen Spannbaums

Die *MST property* besagt also, daß alle paarweise disjunkten Teilmengen eines minimalen Spannbaumes über eine „minimale“ Kante verbunden sind.

4.5.3 Prim-Algorithmus (1957)

Sei $V = \{1, \dots, n\}$, T die Menge der Kanten des zu konstruierenden Minimalbaums und U die Menge seiner Knoten. Dann kann man den Prim-Algorithmus folgendermaßen herleiten:

Kriterium: Sei T die Menge der Kanten eines Spannbaums. Dann lautet das Optimierungskriterium:

$$\min_T \sum_{(u,v) \in T} c[u, v]$$

Beachte: Die Reihenfolge der Kanten in der Summe ist beliebig. Deswegen können wir den Startknoten v_0 beliebig wählen.

Ansatz: rekursive Konstruktion wachsender minimaler Spannbäume (ähnlich wie dynamische Programmierung) mit den Definitionen:

$U_k := \{v_0, \dots, v_k\}$
 = Menge der bisher erfaßten Knoten
 (beachte: $|U_k| = k$)
 $T_k := \{(u_0, v_0), \dots, (u_k, v_k)\}$
 = der zu U_k gehörige minimale Spannbaum

Wir bauen diese Mengen U_k und T_k rekursiv auf. Wir erweitern (U_{k-1}, T_{k-1}) um *eine* neue Kante (u_k, v_k) , $u_k \in U_{k-1}, v_k \in V \setminus U_{k-1}$:

$$\begin{aligned}
 U_k &:= U_{k-1} \cup \{v_k\} \\
 T_k &:= T_{k-1} \cup \{(u_k, v_k)\}
 \end{aligned}$$

Dann ergibt sich aus der Additivität des Optimierungskriteriums:

$$\min_{T_k} \sum_{(u,v) \in T_k} c[u, v] = \min_{T_{k-1}} \sum_{(u,v) \in T_{k-1}} c[u, v] + c[u_k, v_k]$$

mit $(u_k, v_k) = \arg \min\{c[u, v] : u \in U_{k-1}, v \in V \setminus U_{k-1}\}$

Procedure Prim

Initialization: $U_0 = \{v_0\};$

$T_0 = \emptyset;$

For each rank $k = 1, \dots, |V| - 1$ **DO**

$(u_k, v_k) := \arg \min\{c[u, v] : u \in U_{k-1}, v \in V \setminus U_{k-1}\}$

$U_k := U_{k-1} \cup \{v_k\}$

$T_k := T_{k-1} \cup \{(u_k, v_k)\}$

Der Index k wird letztlich nicht benötigt, so daß die abgearbeiteten Knoten v_k in *einer* Menge U (ohne Index) verwaltet werden können. Analog können die abgearbeiteten Kanten in *einer* Menge T verwaltet werden.

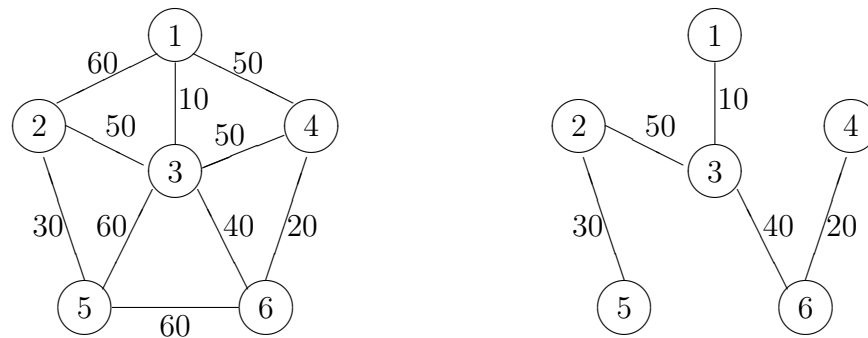
Initialisierung von $T := \{\}$ **und** $U := \{v_0\};$

WHILE $(U \neq V)$ **DO BEGIN**

Wähle beste Kante (u, v) , **die** U **und** $V \setminus U$ **verbindet** $(u \in U, v \in V \setminus U);$

$U := U \cup \{v\}; T := T \cup \{(u, v)\}$

END:

Beispiel

Um die beste Kante zwischen U und $V \setminus U$ zu finden, werden zwei Arrays definiert, die bei jedem Durchgang aktualisiert werden. Für $i \in V \setminus U$ gilt dann:

- $CLOSEST[i] := \arg \min \{c[u, i] : u \in U\}$
- $LOWCOST[i] := c[i, CLOSEST[i]]$

Der vorgestellte Code basiert auf dem Prim-Algorithmus aus [Aho et al. 83].

```

PROCEDURE Prim (C:ARRAY [1..n,1..n] OF REAL);
  VAR LOWCOST : ARRAY [1..n] OF REAL;
      CLOSEST : ARRAY [1..n] OF INTEGER;
      REACHED : ARRAY [1..n] OF BOOLEAN;
      i, j, k: INTEGER;
      min : REAL;
  { i and j are indices. During a scan of the LOWCOST array, k is the
    index of the closest vertex found so far, and min = LOWCOST[k] }
BEGIN
  FOR i:= 2 TO n DO BEGIN
    { initialize with only vertex 1 in the set U }
    LOWCOST[i] := C[1,i];
    CLOSEST[i] := 1;
    REACHED[i] := FALSE;
  END;
  FOR i:= 2 TO n DO BEGIN
    { find the closest vertex k outside of U to some vertex in U }
    min := ∞;
    FOR j:= 2 TO n DO      {*}
      IF (REACHED[j]=FALSE) AND (LOWCOST[j] < min) THEN BEGIN
        min := LOWCOST[j];
      END;
    END;
  END;

```

```

    k := j;
  END;
  writeln (k, CLOSEST[k])      { print edge }
  REACHED[k] := TRUE;        { k is added to U }
  FOR j := 2 TO n DO
    IF (REACHED[j]=FALSE) AND (C[k,j] < LOWCOST[j]) THEN BEGIN
      LOWCOST[j] := C[k,j];
      CLOSEST[j] := k;
    END
  END
END { Prim }

```

Komplexität

Der Code besteht im wesentlichen aus zwei verschachtelten Schleifen der Komplexität $O(|V|)$. Damit ergibt sich eine Zeitkomplexität von

$$O(|V|^2)$$

für den Prim-Algorithmus.

4.6 Zusammenfassung

Wir haben gesehen wie und in welcher Komplexität grundlegende Graph-Algorithmen arbeiten:

Single-Source Best Path : Dijkstra-Algorithmus $O(|V|^2)$

All-Pairs Best Path : Floyd/Warshall-Algorithmus $O(|V|^3)$

Minimaler Spannbaum (MST) : Prim-Algorithmus $O(|V|^2)$

Nicht untersucht haben wir das **Traveling Salesman Problem**. Hier sei nur erwähnt, daß man die Rundreise-Probleme mit einer Komplexität $O(n!)$ (bzw. $O((n-1)!)$) lösen kann. Mittels dynamischer Programmierung läßt sich diese Komplexität auf $O(n^2 \cdot 2^n)$ reduzieren. [Aigner]

Generell ist die Frage noch offen, ob das Traveling Salesman Problem auch in polynomieller Zeit (also in $O(n^k)$) gelöst werden kann.