Fachhochschule
Dortmund
University of Applied Sciences and Arts
Fachbereich Informatik

Algorithmen und Datenstrukturen

VL13 - Graphen 2

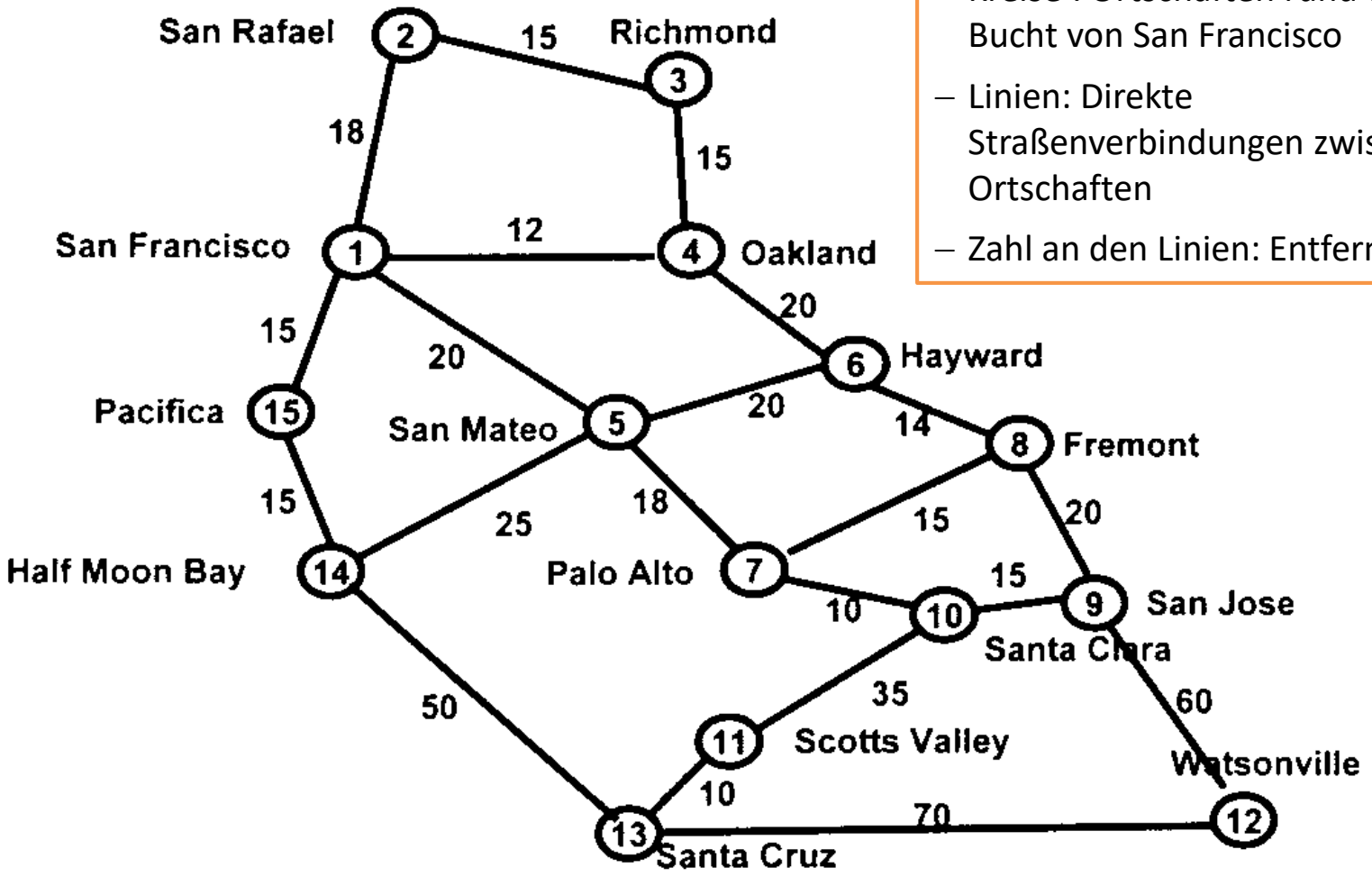
Inhalt

- Gewichtete Graphen
- Minimale Spannbäume
- Kürzeste Wege

Gewichtete Graphen

Gewichtete Graphen

Graph für Verkehrsverbindungen

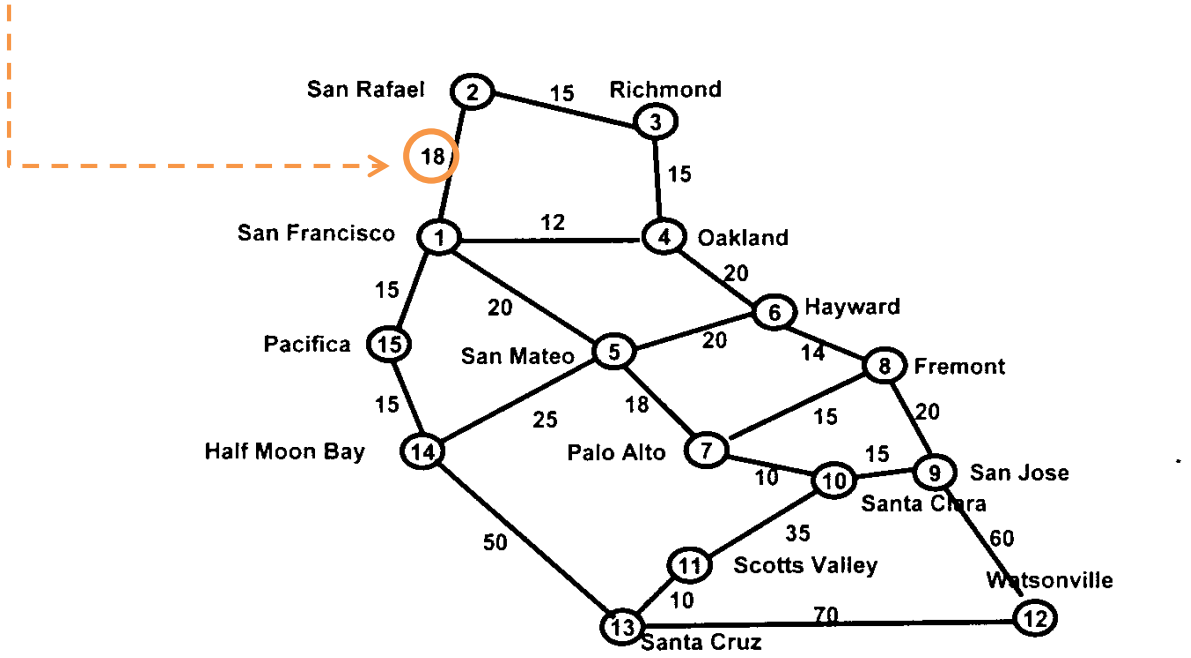


- Kreise : Ortschaften rund um Bucht von San Francisco
- Linien: Direkte Straßenverbindungen zwischen Ortschaften
- Zahl an den Linien: Entfernung

Gewichtete Graphen

Gewichteter / bewerteter Graph

- Ein **gewichteter** oder **bewerteter Graph** ist ein Paar G=(G',d), bestehend aus einem (gerichteten oder ungerichteten) Graphen G'=(V,E), für den zusätzlich jeder Kante e ein Wert **d(e)** zugeordnet ist. Dieser Wert kann ganzzahlig oder reell sein:

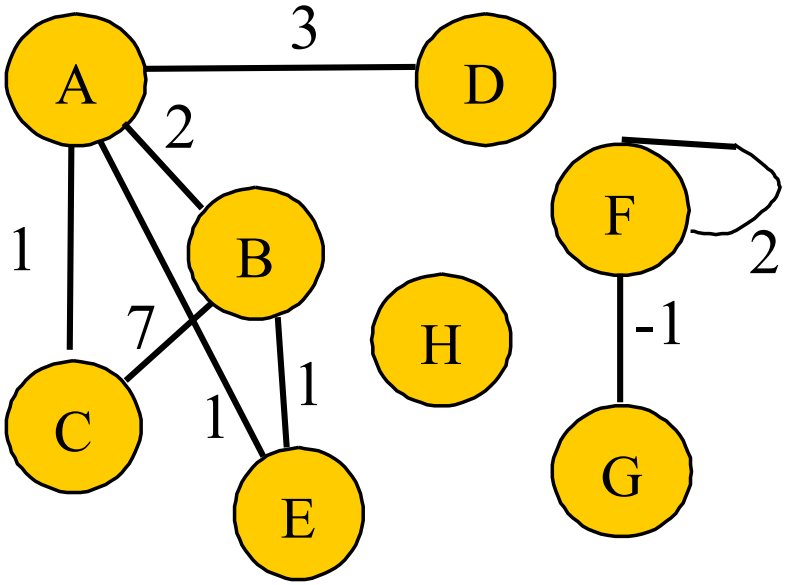


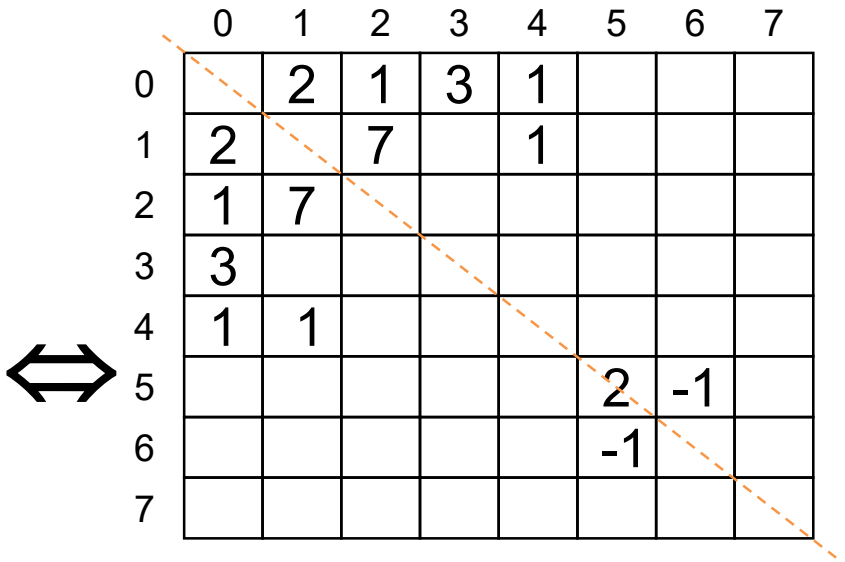
Datenstrukturen für gewichtete Graphen

Adjazenzmatrix

Die Adjazenzmatrix für einen gewichteten Graphen enthält ganze oder reelle Zahlen:

```
int[][] kanten
```





	0	1	2	3	4	5	6	7
0		2	1	3	1			
1	2		7		1			
2	1	7						
3	3							
4	1	1						
5						2	-1	
6						-1		
7								

	0	1	2	3	4	5	6	7
0	A	B	C	D	E	F	G	H

Datenstrukturen für gewichtete Graphen

Adjazenzmatrix

Beispiel: Adjazenzmatrix für den Graphen auf Folie 4:

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
0	0	18		12	20										15
1	18	0	15												
2		15	0	15											
3	12		15	0		20									
4	20				0	20	18							25	
5				20	20	0		14							
6					18		0	15		10					
7						14	15	0	20						
8								20	0	15		60			
9							10		15	0	35				
10										35	0		10		
11									60			0	70		
12											10	70	0	50	
13					25								50	0	15
14	15													15	0

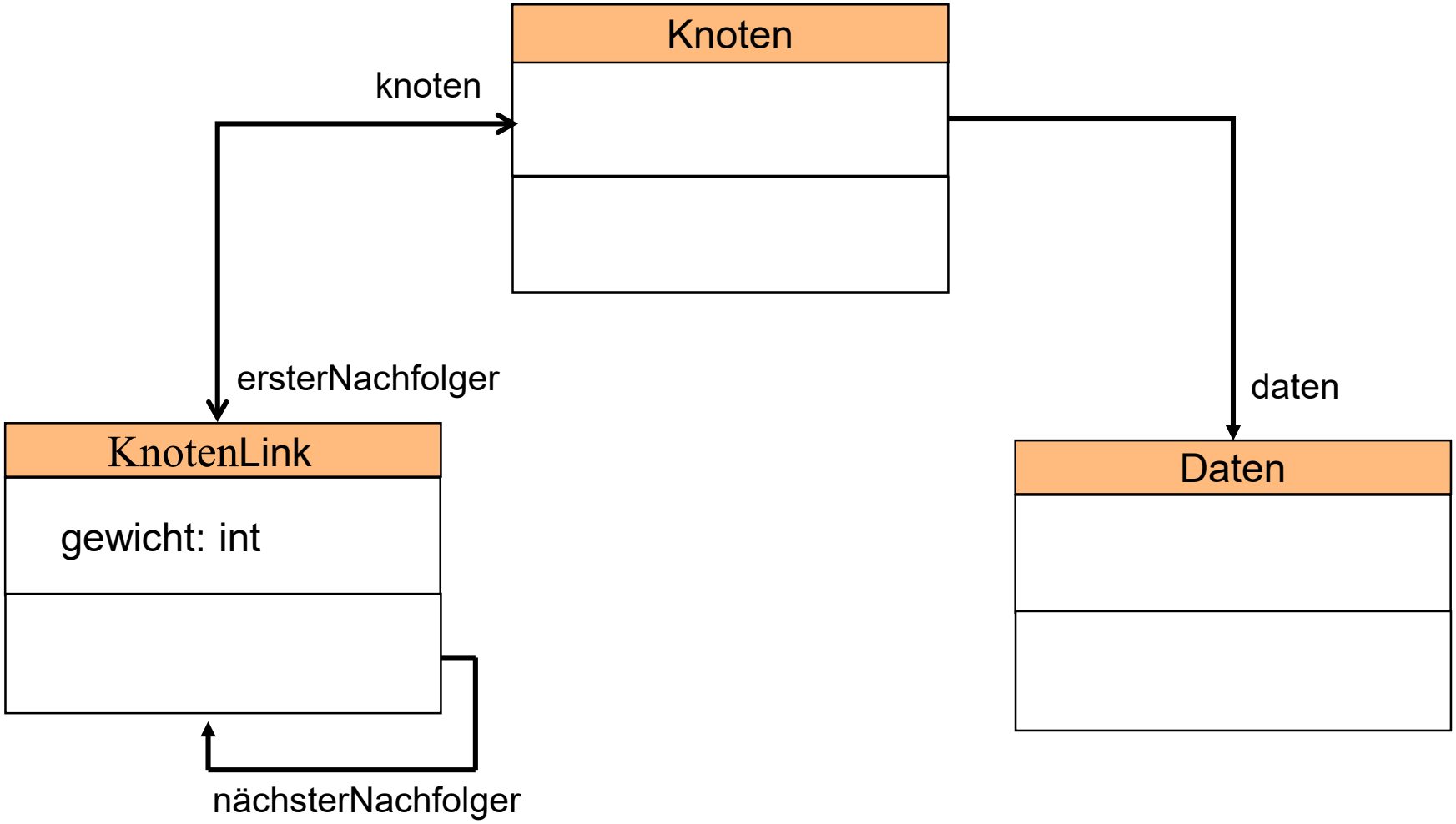
Datenstrukturen für gewichtete Graphen

alternative Darstellung: Adjazenzlisten

Datenstrukturen für gewichtete Graphen

Adjazenzlisten

- Die von einem Knoten abgehenden Kanten eines Graph kann man durch eine lineare Liste darstellen:

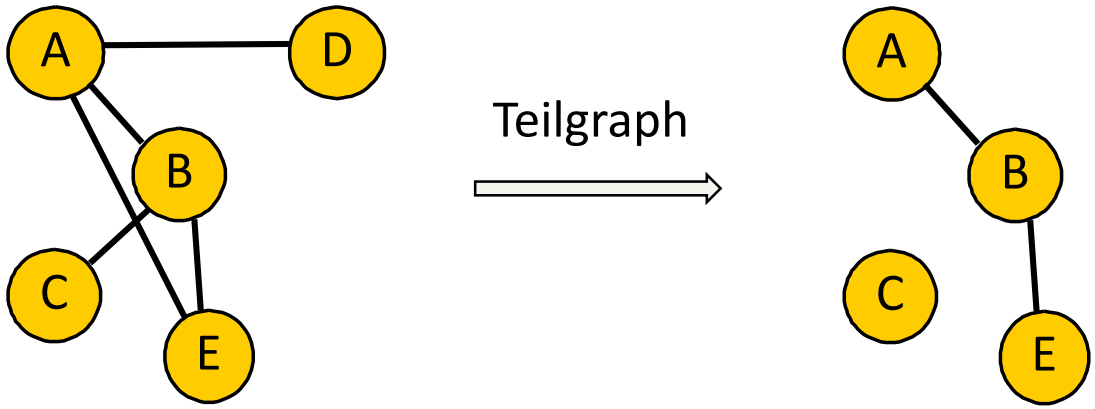


Minimale Spannbäume

Spannbäume

Teilgraph und Spannbaum

- Ein Graph G' = (V', E') ist **Teilgraph** eines Graphen G = (V, E), wenn gilt: $V' \subseteq V$ und $E' \subseteq E$.



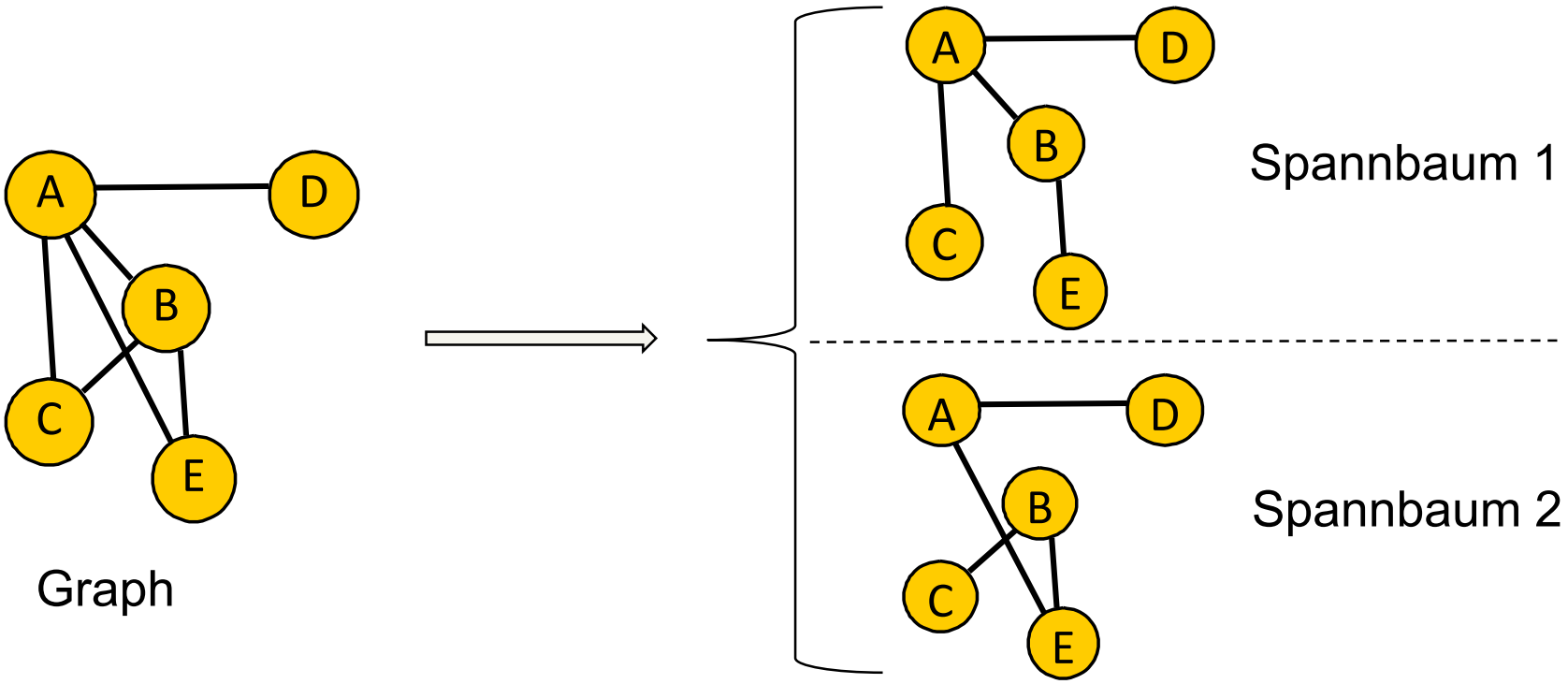
- Sei G ein ungerichteter, zusammenhängender Graph und G' ein (ungerichteter) Baum, der Teilgraph von G ist. Dann heißt G' ein **Spannbaum** (oder **erzeugender Baum**) **von G**, wenn er **alle Knoten von G enthält**, d.h. V' = V.

Spannbäume

Konstruktion eines Spannbaums

Jeder zusammenhängende, ungerichtete Graph besitzt einen Spannbaum. Der einfachste Algorithmus zur Konstruktion ist:

– Solange es einen Zyklus gibt, entferne eine Kante aus diesem Zyklus



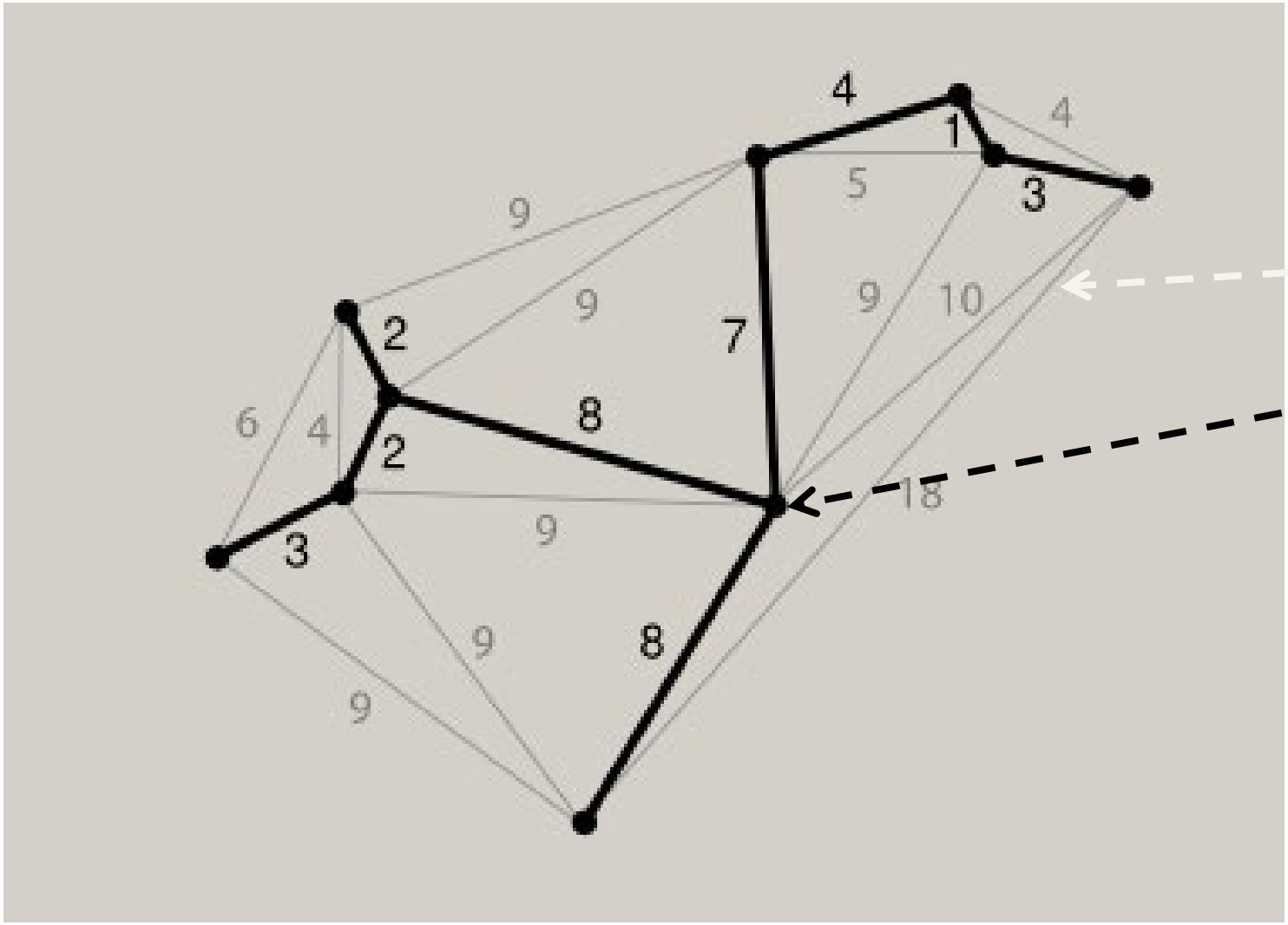
Spannbäume

Beispiel Einsatzgebiete für Spannbäume

- Spannbäume kann man z.B. bei der Planung der Verlegung von Glasfaserkabeln oder Wasserleitungen benutzen, um festzustellen, wo Kabel bzw. Rohre am besten verlegt werden sollten.
  - Dabei gilt bei solchen Planungen:
    Alle Orte müssen versorgt werden, es soll aber so wenig Material wie möglich verwendet werden bzw. die anfallenden Kosten minimiert werden.
  - Dafür werden in der Regel bewertete Graphen verwendet, wobei die Kanten mit Zahlen versehen werden, die die "Kosten" repräsentieren, z.B. in Form der Entfernung zwischen den Orten, der Menge an benötigtem Material oder bereits den Materialkosten, die zwischen 2 Orten anfallen.
- Gesucht ist dann ein **minimaler Spannbaum**, also ein Spannbaum, bei dem die Summe der Kosten der einzelnen Kanten minimal ist (Algorithmus von Kruskal 1956)
  - Voraussetzung: Der betrachtete Graph ist bewertet und ungerichtet.
  - **Greedy-Algorithmus** („Gieriger Algorithmus“), der in jedem Teilschritt so viel wie möglich erreicht

Spannbäume

Beispiel Spannbaum



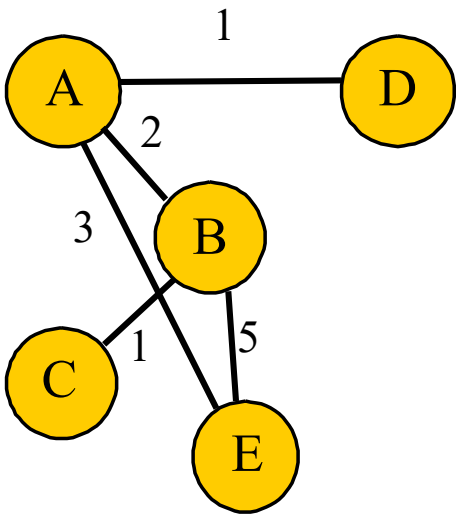
Graph G

Teilgraph G'
ist Spannbaum

Minimale Spannbäume

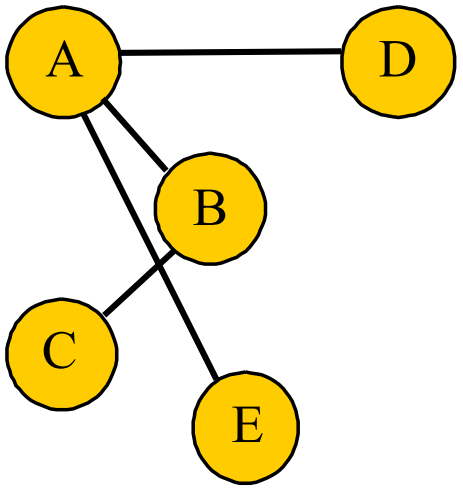
Algorithmus von Kruskal: Grundidee

- Zunächst werden die Kanten von G nach aufsteigendem Gewicht sortiert
- Danach werden die Kanten der Reihe nach (aufsteigendem Gewicht) betrachtet:
  - beginnend mit dem Teilgraph (V,{}) wird die betrachtete Kante dem Teilgraph hinzugefügt, falls der Teilgraph mit der hinzugefügten Kante zyklenfrei bleibt
- Der nach Betrachtung aller Kanten entstandene Graph ist dann ein **minimaler Spannbaum** von G



1. Schritt: {A,D}, {B,C}, {A,B}, {A,E}, {B,E}

2. Schritt:



Minimale Spannbäume

Algorithmus von Kruskal: Grundidee

- **Problem:** Wie testet man effizient, ob bei Hinzunahme einer Kante der entstehende Graph immer noch zyklenfrei ist?

- **Lösung:** Speichere die Zusammenhangskomponenten des Teilgraphen in einer **Union-Find-Datenstruktur**. Der Test auf Zyklenfreiheit ist dann ein Test, ob die Knoten der Kante unterschiedlichen Zusammenhangs-komponenten angehören.

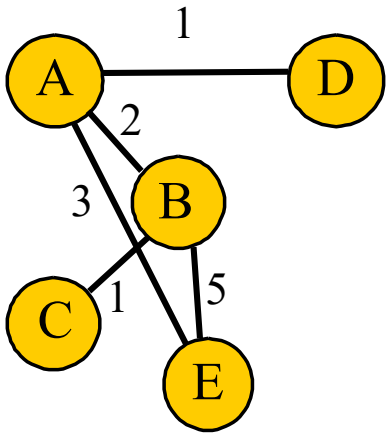
Minimale Spannbäume

Algorithmus von Kruskal: Union-Find-Datenstruktur

- Die Knoten einer Zusammenhangskomponente des bisher erzeugten Teilgraphen werden in einer Baumstruktur gespeichert. Anders als in den bisherigen Bäumen werden aber diesmal pro Baumknoten ein Zeiger auf den Elternknoten gespeichert
- Anfangs bildet jeder Knoten seinen eigenen Baum
- Um zu testen, ob zwei Knoten der selben Zusammenhangskomponente angehören, werden die Wurzeln der zu den Knoten gehörenden Bäume verglichen: Sind sie identisch, so sind beide Knoten in der selben Zusammenhangskomponente.
- Wird eine Kante in den Graphen aufgenommen, so müssen die beiden zu den Knoten der Kante gehörenden Zusammenhangskomponenten vereinigt werden. Dies geschieht, indem die kleinere Zusammenhangs-komponente als Teilbaum der Wurzel der größeren Zusammenhangs-komponente hinzugefügt wird.
- Um die größere Zusammenhangskomponente zu bestimmen, wird die Anzahl der Knoten in einer Zusammenhangskomponente in der Wurzel des entsprechenden Baumes gespeichert

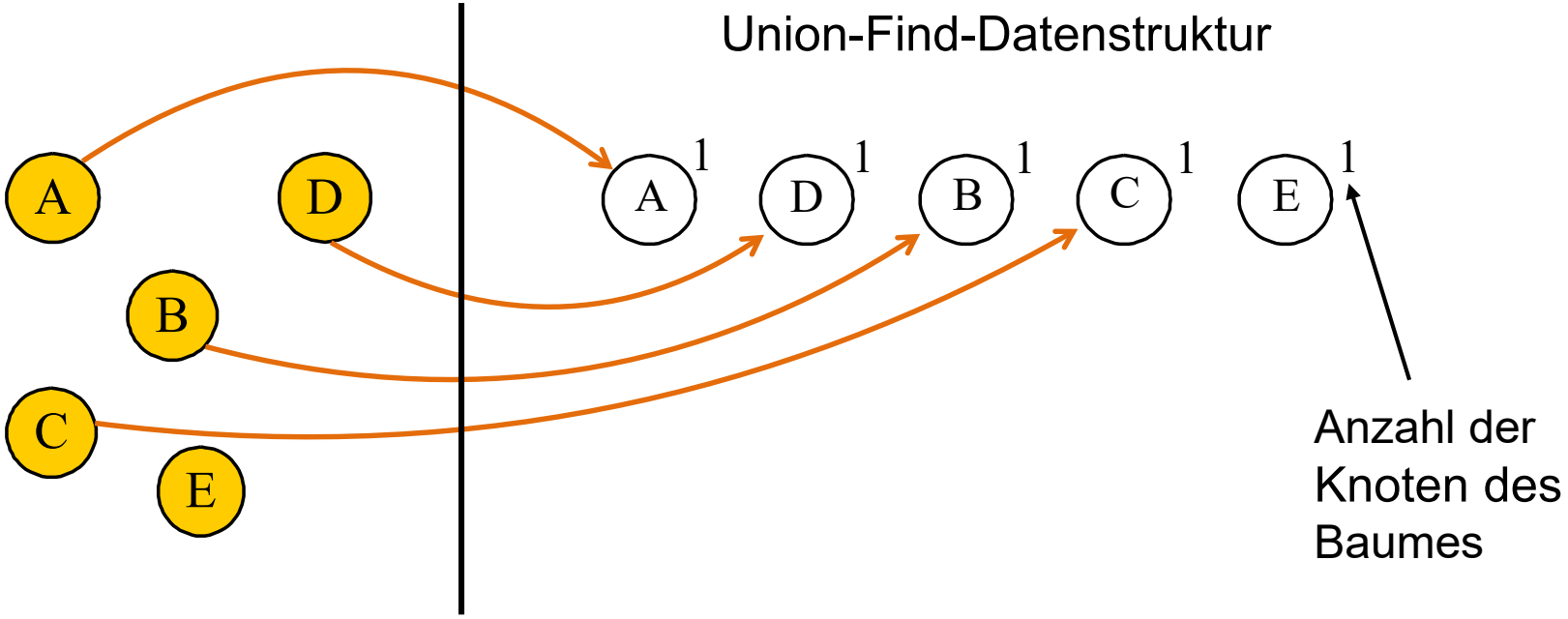
Minimale Spannbäume

Algorithmus von Kruskal



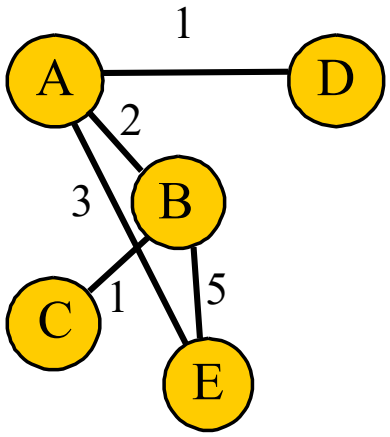
1. Schritt: {A,D}, {B,C}, {A,B}, {A,E}, {B,E}

2. Schritt:



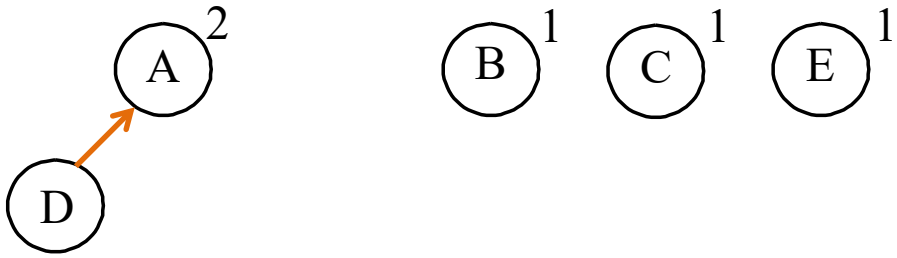
Minimale Spannbäume

Algorithmus von Kruskal



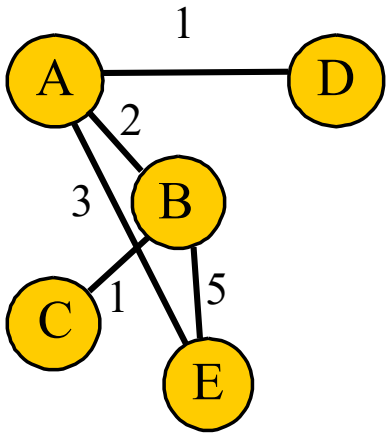
2. Schritt: {A,D}, {B,C}, {A,B}, {A,E}, {B,E}

Union-Find-Datenstruktur



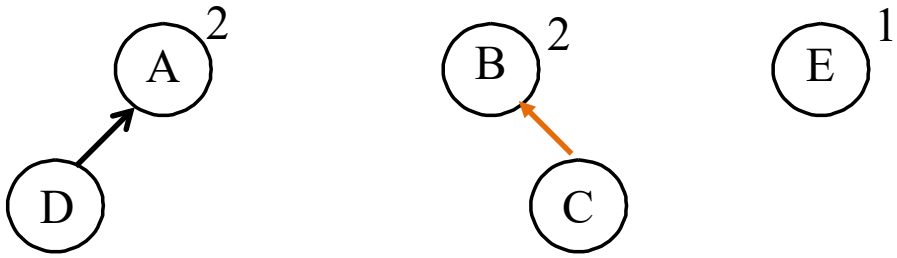
Minimale Spannbäume

Algorithmus von Kruskal



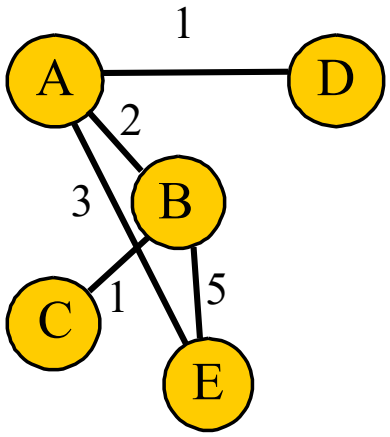
2. Schritt: {A,D}, {B,C}, {A,B}, {A,E}, {B,E}

Union-Find-Datenstruktur



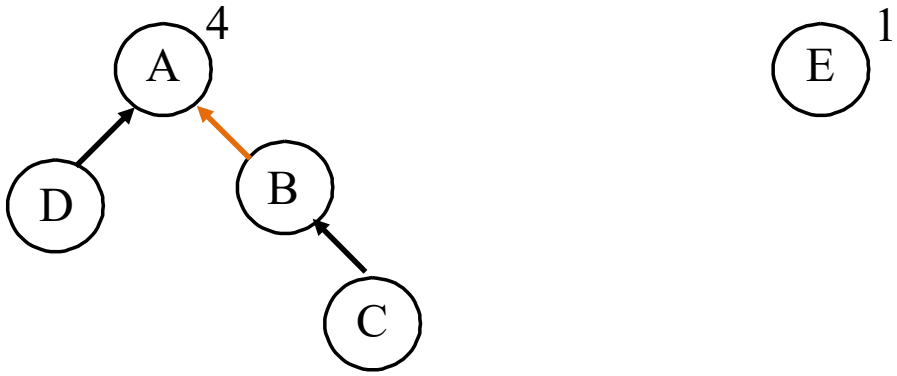
Minimale Spannbäume

Algorithmus von Kruskal



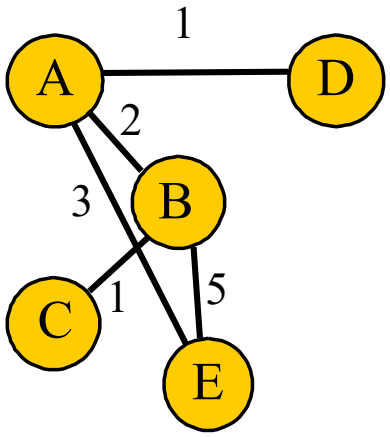
2. Schritt: {A,D}, {B,C}, {A,B}, {A,E}, {B,E}

Union-Find-Datenstruktur



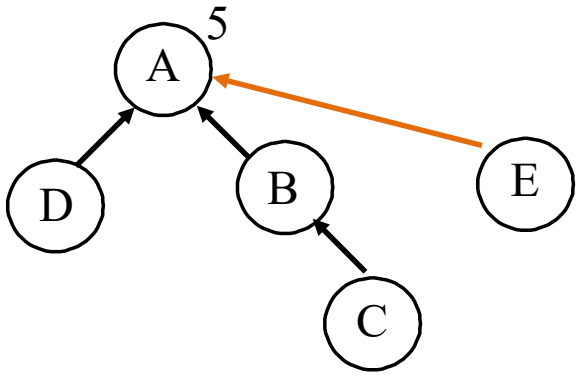
Minimale Spannbäume

Algorithmus von Kruskal



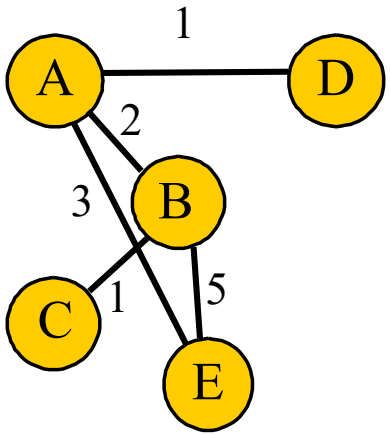
2. Schritt: {A,D}, {B,C}, {A,B}, {A,E}, {B,E}

Union-Find-Datenstruktur



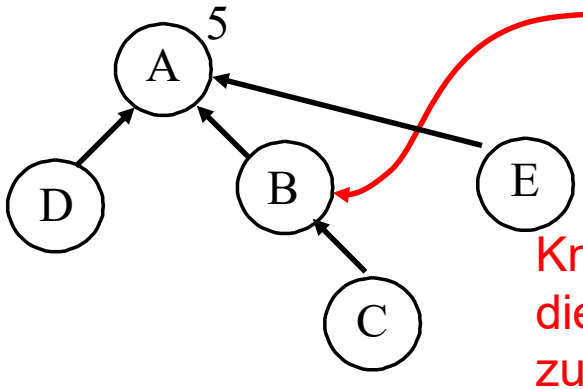
Minimale Spannbäume

Algorithmus von Kruskal



2. Schritt: {A,D}, {B,C}, {A,B}, {A,E}, {B,E}

Union-Find-Datenstruktur

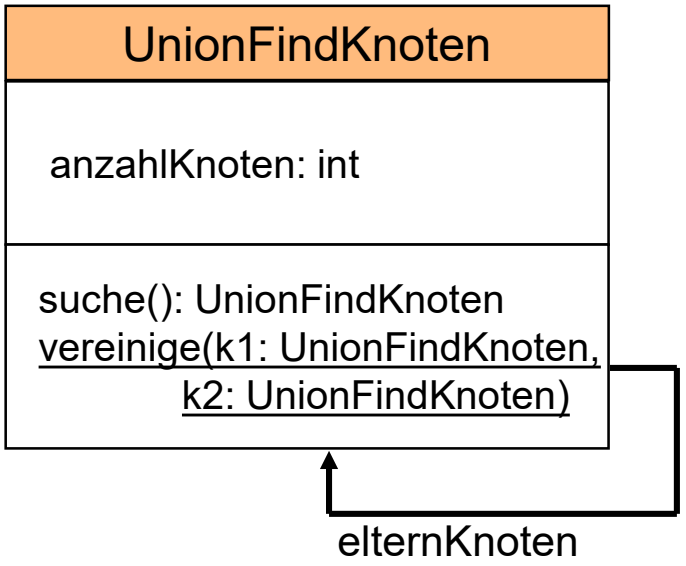


Knoten B und E besitzen dieselbe Wurzel, gehören damit zur selben Zusammenhangs-komponente
Kante {B,E} würde einen Zyklus erzeugen!

Minimale Spannbäume

Algorithmus von Kruskal: Union-Find-Datenstruktur

- Die Knoten werden in einem Baum gespeichert
- Im Gegensatz zu den Bäumen, die wir bisher kennengelernt haben, wird nicht die Eltern-Kind-Relation sondern die Kind-Eltern-Relation gespeichert
- Die Methode `suche()` gibt die Wurzel des Baumes wieder.
  Die Methode benötigt O(t) Schritte auf einem Baum der Tiefe t
- Die Methode `vereinige(k1,k2)` hängt den kleineren Baum an die Wurzel des größeren Baumes.
  Die Methode benötigt O(1) Schritte



- Für die Tiefe eines UnionFind-Baumes mit n Knoten gilt stets: t=O(log n)

Minimale Spannbäume

Algorithmus von Kruskal

```
minimalerSpannbaum(G)
   initialisiere zu jedem Knoten v in G einen UnionFindKnoten Uv
   K ← Ø     // Menge der Kanten, die zur Lösung gehören
   Sortiere die Kanten in G nach aufsteigenden Gewichten in eine Folge S
   while S != Ø do
     {u,v} ← erste Kante in S // Kante mit kleinstem Gewicht
     Streiche {u,v} aus S
     if Uu.suche() != Uv.suche() then // es gibt keinen Pfad von u nach v
       UnionFindKnoten.vereinige(Uu.suche(), Uv.suche())
       K ← K U { {u,v} }
     end if
   end while
   return K
```

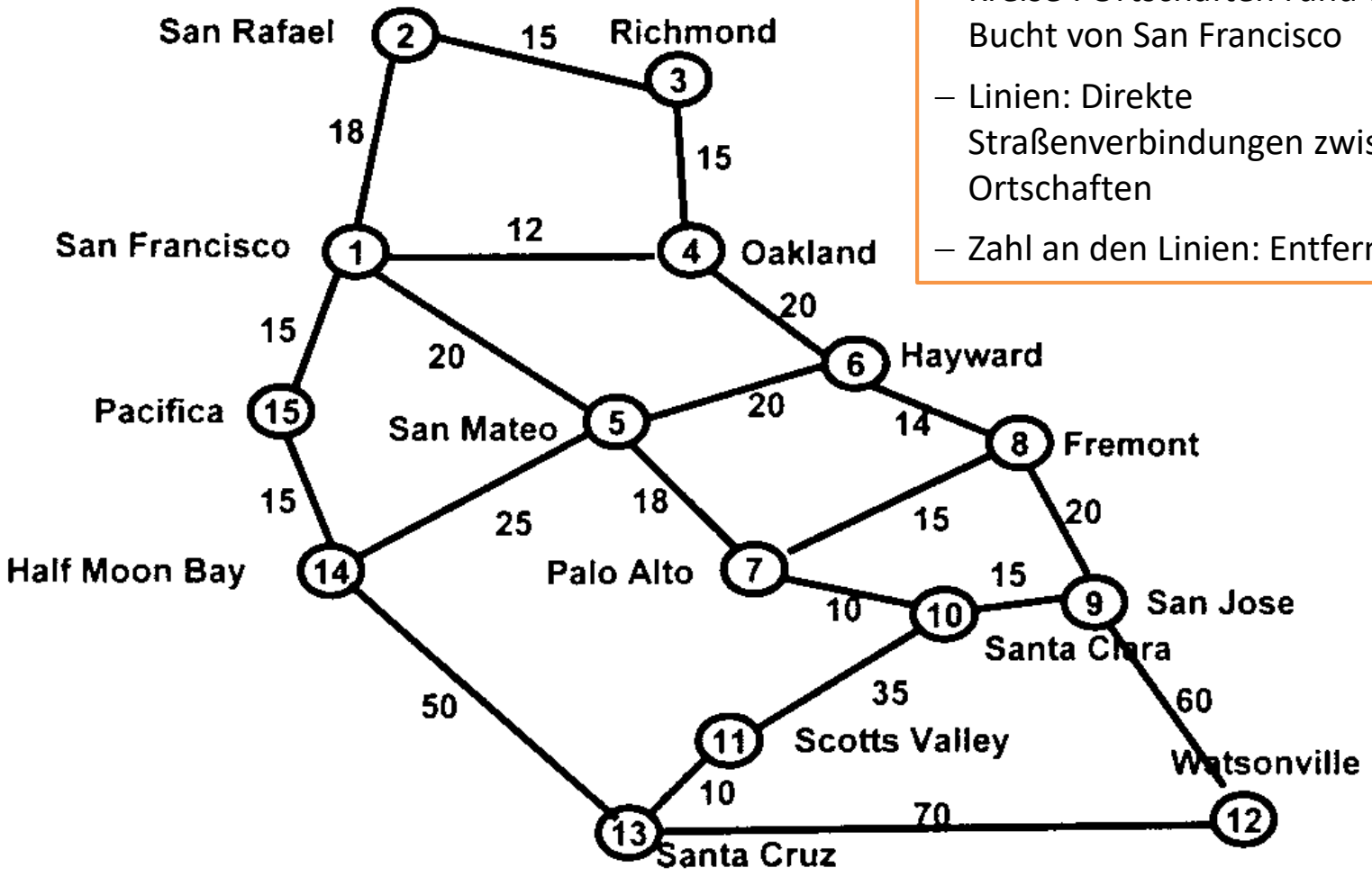
- Das Sortieren der Kantenmenge dominiert die Laufzeit des Algorithmus. Nimmt man z.B. MergeSort oder HeapSort so erhält man eine Laufzeit von

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

Kürzeste Wege

Kürzeste Wege

Graph für Verkehrsverbindungen



- Kreise : Ortschaften rund um Bucht von San Francisco
- Linien: Direkte Straßenverbindungen zwischen Ortschaften
- Zahl an den Linien: Entfernung

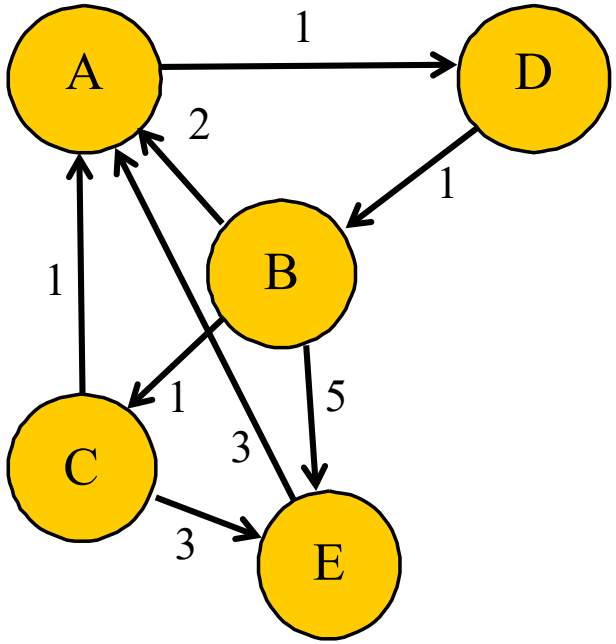
Kürzeste Wege

Kürzeste Wege im Ortsnetz

- Stellt man verschiedene Orte als Knoten eines Graphen dar und die Straßen zwischen ihnen als Kanten, so ist es nicht nur interessant zu wissen, ob es mindestens einen Weg zwischen zwei Orten gibt, sondern auch, welcher von mehreren Wegen der kürzeste ist
- Hierzu wird jeder Kante eine Zahl für ihre Länge (Gewicht) zugeordnet
- Die Summe dieser Zahlen auf einem Weg im Graphen wird (gewichtete) Länge des Weges genannt
- Gesucht ist dann ein Algorithmus, der von allen möglichen Wegen zwischen zwei Knoten des Graphen einen findet, der eine minimale gewichtete Länge besitzt; Diese Wege heißen kürzeste Wege zwischen diesen beiden Knoten; Der Abstand zwischen zwei Knoten in einem Graphen ist die gewichtete Länge eines kürzesten Weges zwischen diesen beiden Knoten
- Die Berechnung der kürzesten Wege von einem Knoten zu allen anderen Knoten in einem Graphen ist mit einem Aufwand von $O(|V|^2)$ möglich (Dijkstra, 1959)

Kürzeste Wege

Kürzeste Wege: Beispiel



Mögliche Wege von A nach E:

- A, D, B, E – Gesamtgewicht: 1 + 1 + 5 = 7
- A, D, B, C, E – Gesamtgewicht: 1 + 1 + 1 + 3 = 6
- A, D, B, A, D, B, C, E – Gesamtgewicht: 1 + 1 + 2 + 1 + 1 + 1 + 3 = 10
- …

Kürzeste Wege

Algorithmus von Dijkstra: Grundidee

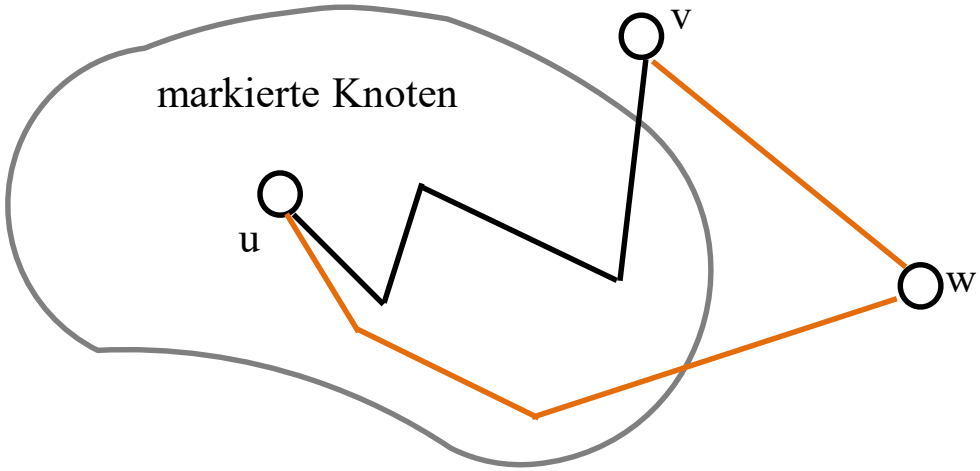
- **Ziel:** Zu gegebenem Knoten u sollen kürzeste Wege von u zu jedem anderen Knoten in einem **gerichteten**, **bewerteten** Graphen gesucht werden
- Hierzu werden Knotenmarkierungen und für jeden Knoten v die Werte **length(v)** und **pred(v)** bestimmt:
  - Es werden die Knoten markiert, für die length(v) bereits der Länge des kürzesten (gewichteten) Weges von u nach v entspricht
  - length(v) entspricht der Länge des kürzesten Weges von u nach v, der mit Ausnahme von v selbst nur markierte Knoten enthält
  - pred(v) enthält jeweils den Vorgängerknoten von v auf einem solchen kürzesten Weg
- Anfänglich ist nur der Knoten u markiert, length(v) = d(u,v) und pred(v) = u (falls die Kante (u,v) nicht existiert, so setzen wir length(v) = ∞)

Kürzeste Wege

Algorithmus von Dijkstra: Grundidee

- Jeder Schritt des Algorithmus besteht aus 2 Teilschritten:
  - Zunächst wird ein unmarkierter Knoten v gesucht, für den length(v) minimal unter den unmarkierten Knoten ist
  - Dieser Knoten v wird dann markiert und für jede Kante (v,w) von v zu einem unmarkierten Knoten w wird length(w) auf das Minimum des bisherigen Wertes von length(w) und length(v)+d(v,w) gesetzt. Ist length(v)+d(v,w) kleiner als der bisherige Wert von length(w) so wird zusätzlich pred(w) = v gesetzt

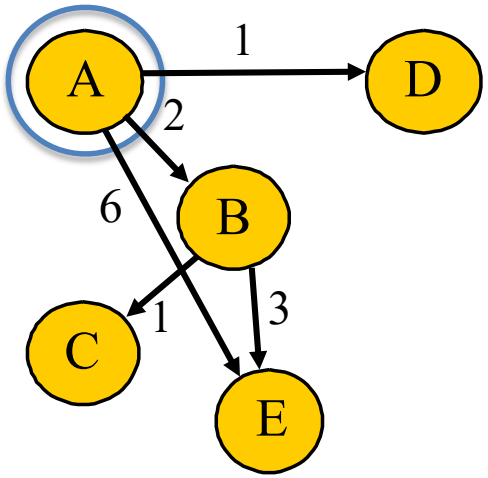
v habe den kürzesten Weg (schwarz) von u nach v, der mit Ausnahme von v nur markierte Knoten enthält. Dann ist der orange Weg zu v länger als der schwarze Weg. **Warum?**



Kürzeste Wege

Algorithmus von Dijkstra: Grundidee

Beispiel:

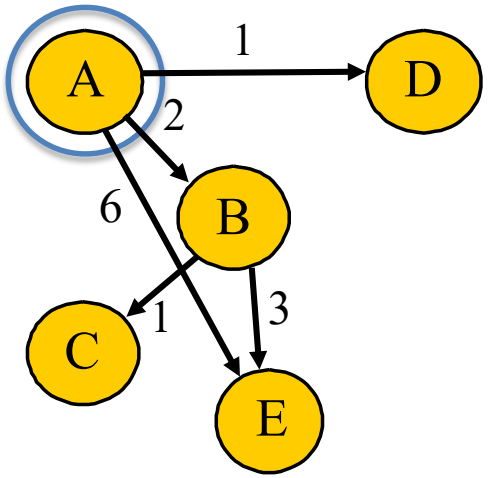


Neu markierter Knoten	*length(A), pred(A)*	*length(B), pred(B)*	*length(C), pred(C)*	*length(D), pred(D)*	*length(E), pred(E)*
A	*0, A*	*2, A*	*∞, A*	*1, A*	*6, A*

Kürzeste Wege

Algorithmus von Dijkstra: Grundidee

Beispiel:

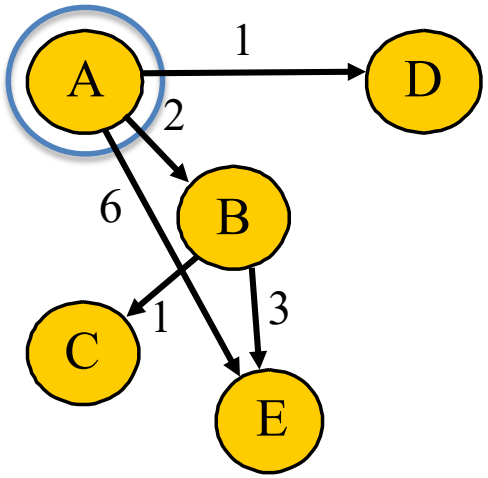


Neu markierter Knoten	*length(A), pred(A)*	*length(B), pred(B)*	*length(C), pred(C)*	*length(D), pred(D)*	*length(E), pred(E)*
A	*0, A*	*2, A*	∞, A	*1, A*	*6, A*
D	*0, A*	*2, A*	∞, A	*1, A*	*6, A*

Kürzeste Wege

Algorithmus von Dijkstra: Grundidee

Beispiel:

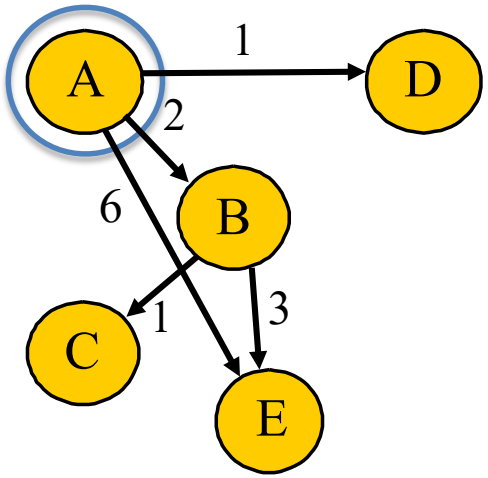


Neu markierter Knoten	*length(A), pred(A)*	*length(B), pred(B)*	*length(C), pred(C)*	*length(D), pred(D)*	*length(E), pred(E)*
A	*0, A*	*2, A*	∞, A	*1, A*	*6, A*
D	*0, A*	*2, A*	∞, A	*1, A*	*6, A*
B	*0, A*	*2, A*	*3, B*	*1, A*	*5, B*

Kürzeste Wege

Algorithmus von Dijkstra: Grundidee

Beispiel:

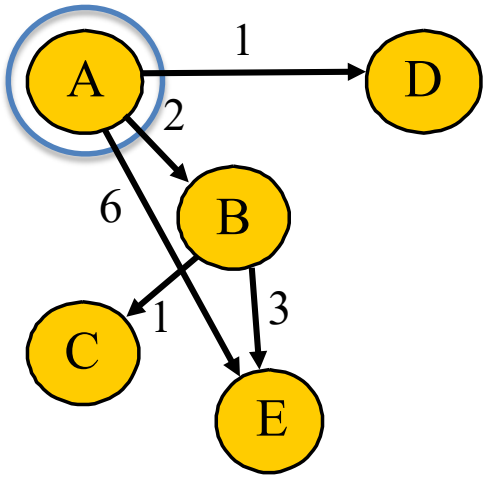


Neu markierter Knoten	*length(A), pred(A)*	*length(B), pred(B)*	*length(C), pred(C)*	*length(D), pred(D)*	*length(E), pred(E)*
A	*0, A*	*2, A*	∞, A	*1, A*	*6, A*
D	*0, A*	*2, A*	∞, A	*1, A*	*6, A*
B	*0, A*	*2, A*	*3, B*	*1, A*	*5, B*
C	*0, A*	*2, A*	*3, B*	*1, A*	*5, B*

Kürzeste Wege

Algorithmus von Dijkstra: Grundidee

Beispiel:



Neu markierter Knoten	*length(A), pred(A)*	*length(B), pred(B)*	*length(C), pred(C)*	*length(D), pred(D)*	*length(E), pred(E)*
A	*0, A*	*2, A*	∞, A	*1, A*	*6, A*
D	*0, A*	*2, A*	∞, A	*1, A*	*6, A*
B	*0, A*	*2, A*	*3, B*	*1, A*	*5, B*
C	*0, A*	*2, A*	*3, B*	*1, A*	*5, B*
E	*0, A*	*2, A*	*3, B*	*1, A*	*5, B*

Kürzeste Wege

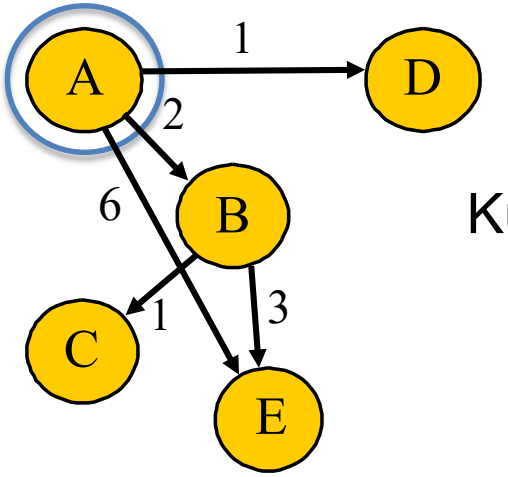
Algorithmus von Dijkstra: Verfeinerung

- Die unmarkierten Knoten w können in einer verketteten Liste verwaltet werden
- Um einen unmarkierten Knoten v mit minimalem Wert length(v) zu finden, wird diese Liste sequentiell durchsucht
- Wenn v markiert wird, so wird für jede Kante (v,w) im Graph
  - das Gewicht g = d(v,w) der Kante ermittelt,
  - und length(w) durch das Minimum aus length(v)+g und length(w) ersetzt und im ersten Fall pred(w) auf v gesetzt
- Einen kürzesten Weg (in umgekehrter Reihenfolge) von u zu einem markierten Knoten v erhält man dann, indem man von v ausgehend jeweils die Vorgänger bestimmt, bis man den Knoten u erreicht.
- Zeitaufwand: $O(|V|^2)$

Kürzeste Wege

Algorithmus von Dijkstra: Beispiel

Beispiel:



Kürzester Weg von A nach C:

C ⟵ pred(C) = B ⟵ pred(B) = A

Neu markierter Knoten	*length(A), pred(A)*	*length(B), pred(B)*	*length(C), pred(C)*	*length(D), pred(D)*	*length(E), pred(E)*
A	*0, A*	*2, A*	∞, A	*1, A*	*6, A*
D	*0, A*	*2, A*	∞, A	*1, A*	*6, A*
B	*0, A*	*2, A*	*3, B*	*1, A*	*5, B*
C	*0, A*	*2, A*	*3, B*	*1, A*	*5, B*
E	*0, A*	*2, A*	*3, B*	*1, A*	*5, B*

Kürzeste Wege

Algorithmus von Dijkstra

Wir setzen voraus, dass der gerichtete, bewertete Graph G = (V,E,d) in Adjazenzlistendarstellung gegeben ist

```
kürzesteWege(G, u)
   initialisiere Liste L mit allen von u verschiedenen Knoten
   length(u) ← 0
   for v ∈ V mit u≠v do
     pred(v) = u
     if (u,v) ∈ E then length(v) ← d(u,v)
     else length(v) ← ∞
     end if
   end for
   while L nicht leer do
     v ← ein Knoten in L mit minimalem length(v)
     Streiche v aus L
     nachfolger ← v.ersterNachfolger
     while nachfolger existiert do
        if length(v)+nachfolger.gewicht<length(nachfolger.knoten) then
          length(nachfolger.knoten) ← length(v) + nachfolger.gewicht
          pred(nachfolger.knoten) ← v
        end if
        nachfolger ← nachfolger.nächsterNachfolger
     end while
   end while
```

Lernziele

- Sie können die verschiedenen Darstellungsformen von gewichteten Graphen erläutern
- Sie können die Algorithmen von Kruskal und Dijkstra an Beispielen erläutern und kennen den Zeitaufwand dieser Algorithmen

Literatur

Quellen

- Günther Saake: Algorithmen und Datenstrukturen
  - Kapitel 16 Graphen
- Karsten Weicker: Algorithmen und Datenstrukturen
  - Kapitel 1 (Ein Anwendungsbeispiel),
    4.5 (Strukturierte Verarbeitung von Graphen)

Vertiefend

- **[Güting_2018]**
  Güting, Ralf Hartmut, Dieker, Stefan : Datenstrukturen und Algorithmen, Springer Vieweg, 4. Auflage, 2018 (https://link.springer.com/book/10.1007%2F978-3-658-04676-7)
- **[Ottmann_Widmayer_2017]**
  Ottmann, Thomas, Widmayer, Peter: Algorithmen und Datenstrukturen, Springer Vieweg, 6. Auflage, 2017 (https://link.springer.com/book/10.1007%2F978-3-662-55650-4)