Algorithmus von Kruskal

Algorithmus von Kruskal

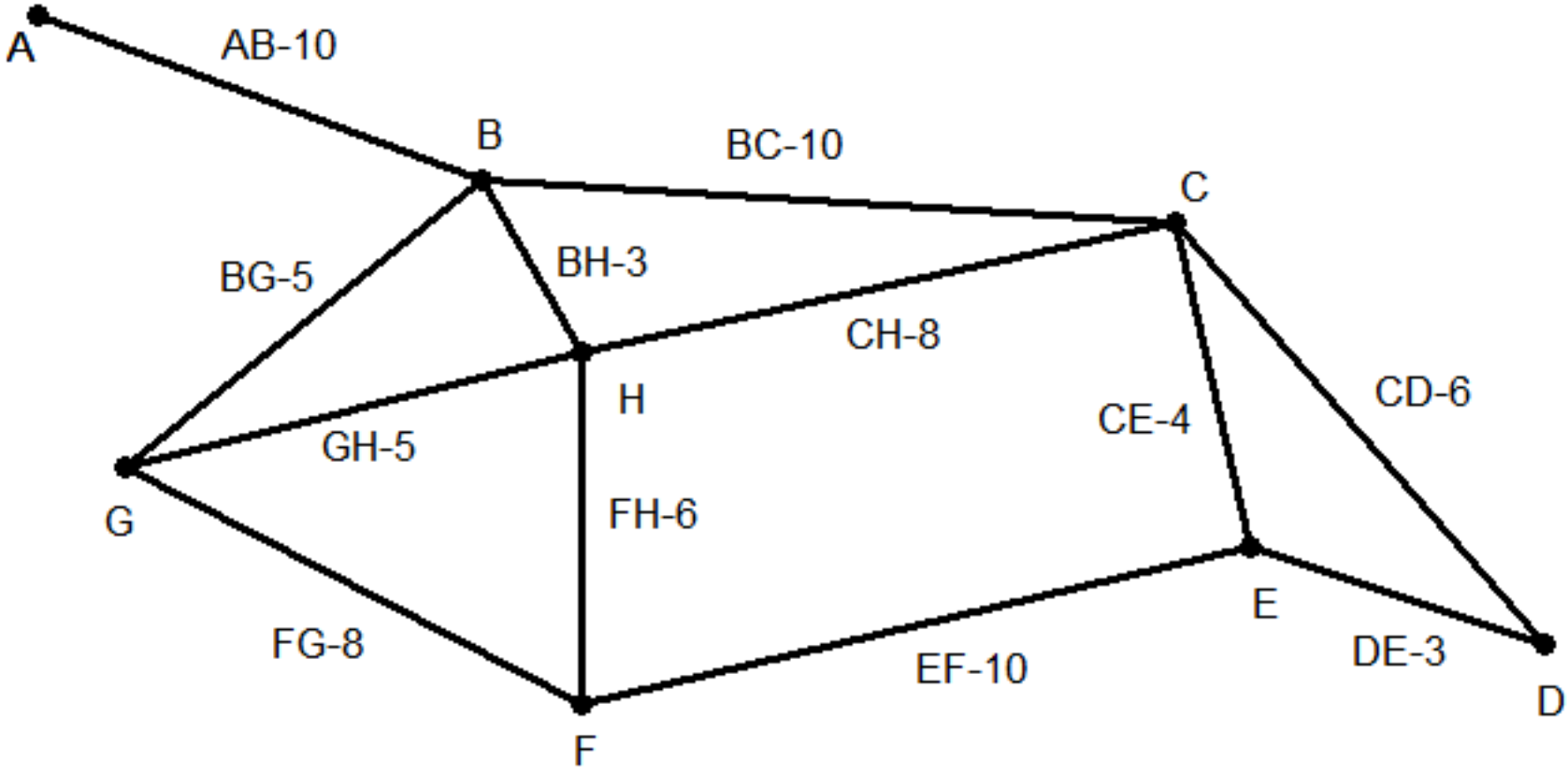
ein Algorithmus zur Bestimmung eines minimalen aufspannenden Baums

(minimal spanning tree]

© Claus Albowski

Algorithmus von Kruskal

Ein Graph



© Claus Albowski

Algorithmus von Kruskal

Beschreibung des Algorithmus

- Ordne alle Kanten nach ihren Kosten.
- Beginne mit einer Liste der Teilbäume, die von allen Knoten jeweils genau einen enthalten.
- Füge dann jeweils immer die nächste Kante mit den geringsten Kosten hinzu, die genau zwei Teilbäume verbindet.
- Lasse Kanten weg, die Zyklen bilden.
- Beende die Suche, wenn es nur noch einen Teilbaum gibt.

Voraussetzung: Der Ausgangsgraph muss zusammenhängend sein.

© Claus Albowski

Algorithmus von Kruskal

Aufgaben:

- Vor dem Start: Ordnen der Kanten nach ihren Bewertungen
- Prüfen, ob beide Knoten einer neuen Kante im selben Teilbaum enthalten sind (→ *Zyklus)*
- Verbinden zweier Teilbäume mit der neuen Kante
- Die eigentliche Steuerung des Algorithmus

© Claus Albowski

Algorithmus von Kruskal

Ordnen der Kanten

wird durch das Einbinden
der Funktionen für die
Prioritätswarteschlange
realisiert

*(PrioWS: Hier wird allein die Sortierung genutzt,
sie wird bei Kruskal also nur einmalig aufgebaut]*

© Claus Albowski

Algorithmus von Kruskal

Hinweise

- Die ***Prioritätswarteschlange*** selbst wird in der funktionalen Modellierung allein vom zugreifenden Programm als Liste gehalten.
- Die Eigenschaft dieser Liste, Prioritäts-warteschlange zu sein, ergibt sich aus den Zugriffsfunktionen, die aber identisch sind mit den Funktionen für die Aufgabe *„Sortieren durch Einfügen“.*
- *Objektorientiert stellt das (einzige) erzeugte Prioritätswarteschlangen-Objekt die Zugriffsfunktionen bereit.*

© Claus Albowski

Algorithmus von Kruskal

- Die Prioritätswarteschlange benötigt das Prädikat zum Einfügen, die Funktion **`vor`**, konkret **`kuerzer`**

```
def kuerzer( kante_1 , kante_2 )
    return kante_1[2] < kante_2[2]
```

Die Kanten sind also gespeichert in der Form:

`[<Knoten-1>, <Knoten-2>, <Bewertung>]`

oder

`(<Knoten-1>, <Knoten-2>, <Bewertung>)`

© Claus Albowski

Algorithmus von Kruskal

- Ohne OO beispielsweise:

```
# ----- ordneKanten -----
# iteriert über alle Kanten
def ordneKanten(kanten):
    sortierte=[]
    for kante in kanten:
        sortierte=ordneEin(kante, sortierte)
    return sortierte
```

© Claus Albowski

Algorithmus von Kruskal

- mit:

```
# ----- ordneEin -----
# sortiert durch Einfuegen
def ordneEin(kante, sortierte):
    for index in range(len(sortierte)]:
        if kuerzer(kante, sortierte[index]):
            sortierte[index:index]=[kante]
            return sortierte
    sortierte[len(sortierte):len(sortierte)]=[kante]
    return sortierte
```

© Claus Albowski

Algorithmus von Kruskal

Erzeugen der Teilbäume

© Claus Albowski

Algorithmus von Kruskal

- Datenstruktur der Teilbäume
  (es gäbe auch Alternativen]
- Eine Liste von zwei Teillisten aus
  - den enthaltenen Knoten
  - den enthaltenen Kanten
- Anfangszustand (*noch keine Kanten eingebaut*) zum Beispielgraph:

```
[ [['A'],[]],  [['B'],[]],  [['C'],[]],
  [['D'],[]],  [['E'],[]],  [['F'],[]],
  [['G'],[]],  [['H'],[]] ]
```

© Claus Albowski

Algorithmus von Kruskal

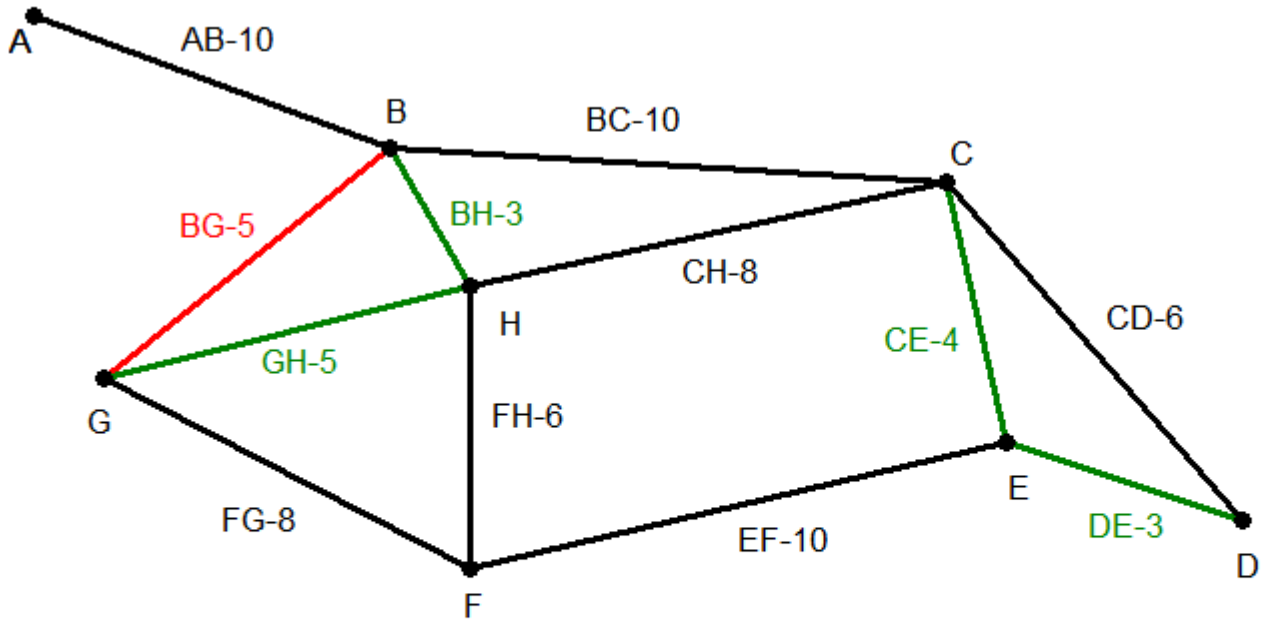
- Teilbaeume initialisieren

```
# erzeugt den Anfangszustand aus der Kantenliste
def erzeugeTeilbaeume(kanten):
    alleKnoten=gibAlleKnoten(kanten)
    teilbaeume=[]
    for knoten in alleKnoten:
        teilbaeume.append([[knoten],[]])
    return teilbaeume
```

© Claus Albowski

Algorithmus von Kruskal

Prüfen,
ob beide Knoten einer Kante
im selben Teilbaum enthalten sind
(→ Zyklus]

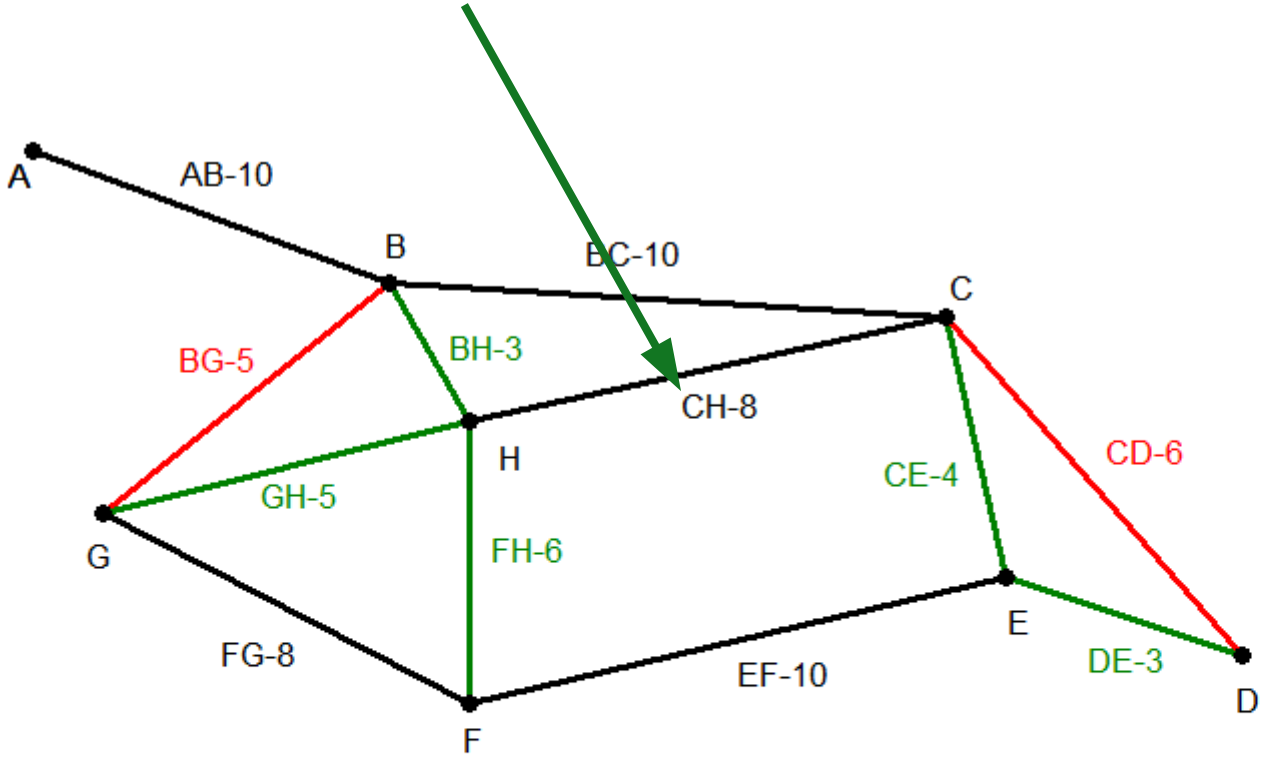


© Claus Albowski

Algorithmus von Kruskal

Verbinden zweier Teilbäume

- Bei dieser Funktion werden zwei Teilbäume durch die neue Kante verbunden.



© Claus Albowski

Algorithmus von Kruskal

- Verbinden zweier Teilbäume

```
# verbindet die beiden Teilbaeume durch die Kante.
def verbinde(kante, erster, zweiter, alleTeilbaeume):

tb1,tb2=alleTeilbaeume[erster],alleTeilbaeume[zweiter]
    neu=[ tb1[0]+tb2[0], [kante]+tb1[1]+tb2[1] ]
    alleTeilbaeume.remove(tb1)
    alleTeilbaeume.remove(tb2)
    alleTeilbaeume.append(neu)
    return alleTeilbaeume
```

© Claus Albowski

Algorithmus von Kruskal

- Die eigentliche Suchfunktion

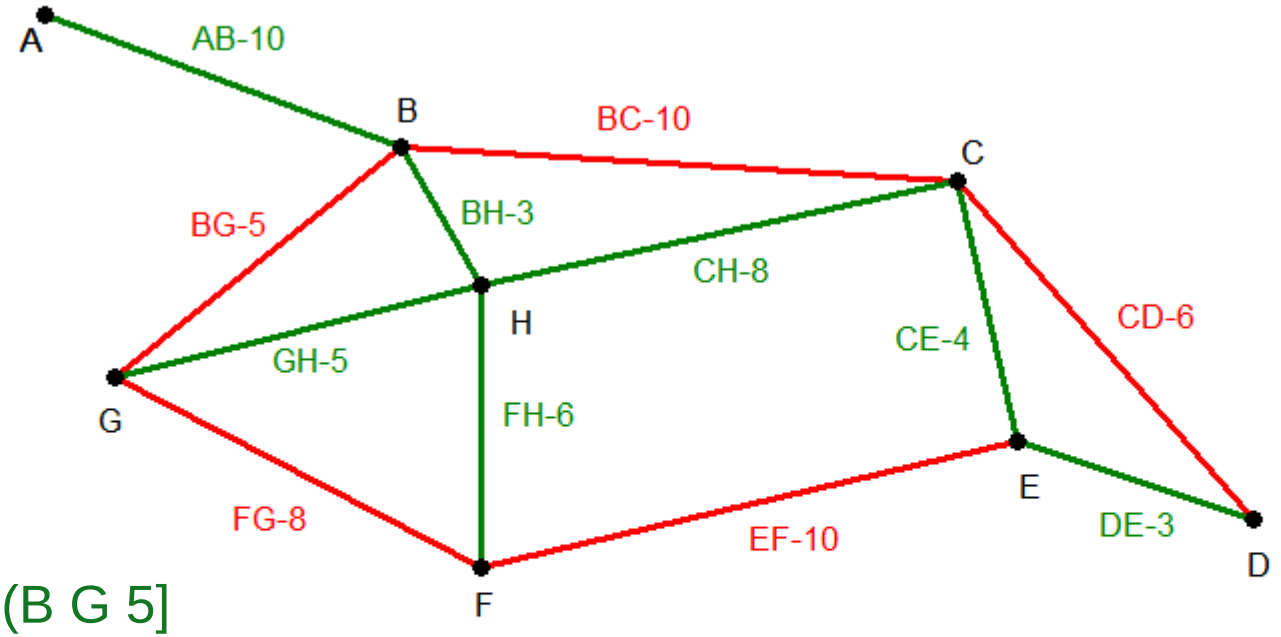
```
def kruskal(kanten]:
    kanten=ordneKanten(kanten)
    teilbaeume=erzeugeTeilbaeume(kanten)
    for kante in kanten:
        erster=posTeilbaumZuKnoten(kante[0],
                                   teilbaeume)
        zweiter=posTeilbaumZuKnoten(kante[1],
                                   teilbaeume)
        if erster!=zweiter:
            tbaeume=verbinde(kante, erster,
                             zweiter, teilbaeume)
            if len(teilbaeume]==1:
                return teilbaeume[0][1]
    return "Fehler"
```

© Claus Albowski

Algorithmus von Kruskal

Ein Aufruf für den angegebenen Graphen ergibt

```
[[A,C,E,D,F,G,B,H],                  # die Knoten
  [[A,B,10] [C,H,8],[C,E,4],[D,E,3],
   [F,H,6],[G,H,5],[B,H,3]] # die Kanten
]
```



*] alternativ (B G 5]

© Claus Albowski