Problem des minimalen Versorgungsnetzes

Beschreibung des Problems

Zu mehreren Orten, die untereinander mit unterschiedlich langen Kanten verbunden sind, soll ein Versorgungsnetz mit minimalem Aufwand generiert werden.

Modellierung des Problems

Das Problem wird mit einem Graphen beschrieben, dessen Knoten für die zu versorgenden Orte stehen und deren Kanten die Wege zwischen ihnen darstellen. Die Kanten sind mit einer Zahl bewertet, die beispielsweise für den jeweiligen Herstellungsaufwand steht.

Gesucht ist zu diesem Graphen ein minimaler aufspannender Baum [minimal spanning tree].

Entwicklung des Algorithmus

Hierfür gibt es zwei Ansätze:

Der Algorithmus von Prim

- Beginne mit einem Baum, der allein aus einem Knoten besteht.
- Ordne alle Kanten von diesem Knoten aus nach ihren Kosten. (→ Prioritätswarteschlange nach den Bewertungen)
- Füge nun jeweils immer aus dieser jeweils die nächste Kante mit den geringsten Kosten ein, die keinen Zyklus erzeugt. Dann füge alle von dem freien Knoten ausgehenden zulässigen Kanten in die Prioritätswarteschlange ein.
- Brich damit ab, wenn alle Knoten des Graphen zum Baum gehören.

Der Algorithmus von Kruskal

- Ordne alle Kanten nach ihren Kosten.
- Beginne mit einer Liste von Teilbäumen, die zunächst jeweils allein aus den isolierten Punkten des Graphen besteht.
- Verbinde dann jeweils immer mit der nächsten Kante mit den geringsten Kosten, die keinen Zyklus erzeugt[1], zwei Teilbäume zu einem Teilbaum.
- Brich damit ab, wenn die Liste aller Teilbäume nur noch ein Element enthält.

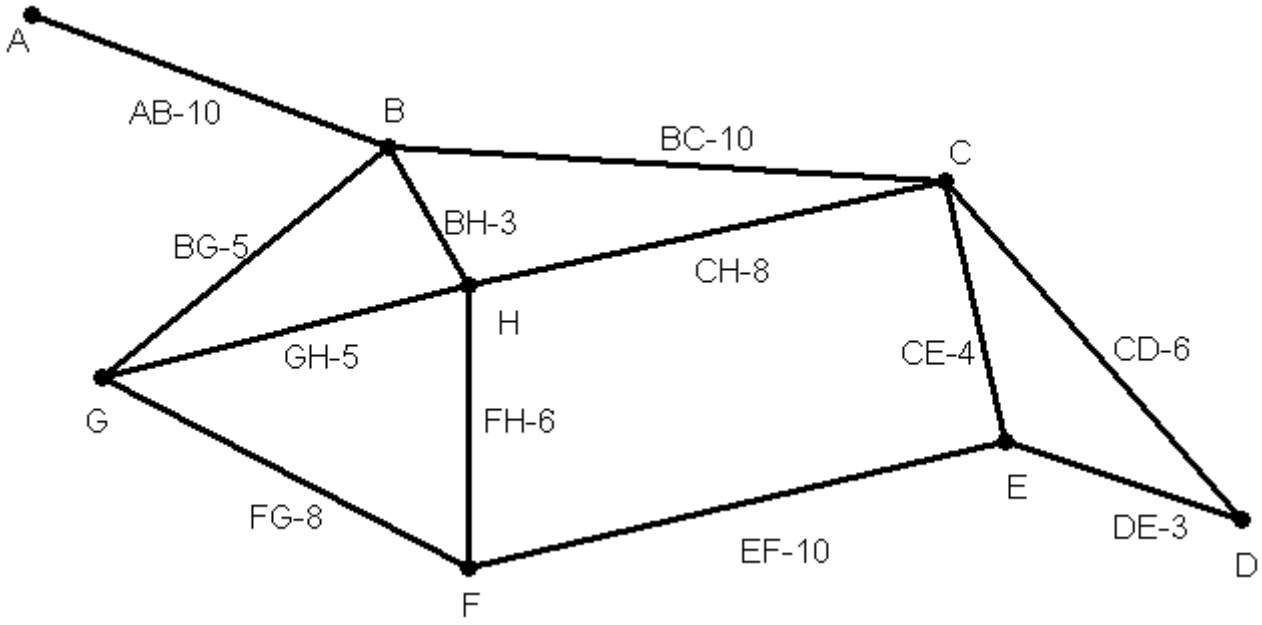
1 Das ist dann der Fall, wenn beide Knoten der Kante zum selben Teilgraphen gehören.

Datenmodellierung mit Knotenliste

Scheme-typisch ist eher eine Knotenliste, die eine Assoziationsliste darstellt.

Dies ist eine Liste mit Teilisten, die jeweils zu dem Knoten nachfolgend eine zweielementige Liste aus dem Zielknoten und der Bewertung angeben:

```
(define
  knotenliste
  '((A (B 10))
    (B (A 10) (C 10) (G 5) (H 3))
    (C (B 10) (D 6) (E 4) (H 8))
    (D (C 6) (E 3))
    (E (C 4) (D 3) (F 10))
    (F (E 10) (G 8) (H 6))
    (G (B 5) (F 8) (H 5))
    (H (B 3) (C 8) (F 6) (G 5))
    )
  )
```



Datenmodellierung mit Kantenliste

```
(define
  kanten
  '((A B 10)
    (B A 10) (B C 10) (B G 5) (B H 3)
    (C B 10) (C D 6) (C E 4) (C H 8)
    (D C 6) (D E 3)
    (E C 4) (E D 3) (E F 10)
    (F E 10) (F G 8) (F H 6)
    (G B 5) (G F 8) (G H 5)
    (H B 3) (H C 8) (H F 6) (H G 5)
    )
  )
```

Kanten sind dreielementige Listen mit dem Ausgangsknoten, dem Zielknoten und der Bewertung der Kante.

Zugriffsfunktionen

Selbstverständlich müsse nsich die Zugriffsfunktionen unterscheiden. Bei der Knotenliste ist die Bestimmung der Nachfolgekanten zu einem Knoten sehr einfach, da Schem dazu die Standardfunktion **`assoc`** bereitstellt.

`(assoc 'C knotenliste)` liefert **`(C (B 10) (D 6) (E 4) (H 8))`**.

Bei der Kantenliste erstellt man dazu (eine) Hilfsfunktion(en):

```
; ===== Hilfsfunktion ist-nachfolge-kante? =====
(define
  (ist-nachfolge-kante? aktuelle-stadt kante)
  (equal? aktuelle-stadt (first kante)))

; ===== nachfolge-kanten endrekursiv mit Aufrufhuelle =====
(define
  (nachfolge-kanten aktuelle-stadt kanten)
  (define
    (intern aktuelle-stadt kanten akku)
    (cond
      ((null? kanten) (reverse akku))
      ((ist-nachfolge-kante? aktuelle-stadt (first kanten))
       (intern
        aktuelle-stadt
        (rest kanten)
        (cons (first kanten) akku)))
      (else (intern aktuelle-stadt (rest kanten) akku))
      )
    )
  (intern aktuelle-stadt kanten '())
  )
```

Der Algorithmus von Prim

Die Aufruf-Funktion bei Prim

Die Aufruf-Funktion muss den Graphen übergeben bekommen. Rückgabewert ist der minimal spanning tree.

```
(define
  (minimal-spanning-tree start-knoten kanten)
```

Diese Funktion hat nur die Funktion einer Aufrufhülle und ruft die eigentliche Suchfunktion (siehe unten) mit den notwendigen generierten Startparametern (hier zur Kantenliste) auf.

```
  (prim-intern
   (list start-knoten)                          ; in die Besuchtliste
   (anzahl-knoten kanten)                       ; extern berechnen
   '()                                          ; noch kein Weg
   (fuege-alle-ein                              ; Start-PrioWS
    (nachfolge-kanten start-knoten kanten)
    vor?
    '())
```

Der letzte übergebene Parameter baut aus den Nachfolgekanten den ersten Zustand der verwalteten Prioritätswarteschlange auf. Dazu wird auf die schon erarbeitete(n) Funktion(en) zur Verwaltung einer Prioritätswarteschlange zugegriffen (**`fuege-alle-ein`**) und auf die Zugriffsfunktion auf die Daten für den Graphen (**`nachfolge-kanten knoten`**).

Startet man mit H, erhält man also zunächst alle Nachfolgekanten des Knoten H:

```
((H G 5) (H F 6) (H C 8) (H B 3))
```

Diese werden in die [noch leere] Prioritätswarteschlange eingefügt, funktional wird der Aufruf von **`nachfolge-kanten`** in die Funktion **`fuege-alle-kanten-ein`** geschachtelt.

Die Funktion prim-intern

```
(define
    (prim-intern besucht anzahl kuerzeste priows)
    (cond
      ((= (length besucht) anzahl)              ; alle Knoten enthalten
       kuerzeste)
      ((null? priows)                           ; nichts geht mehr
       #f)
      ((zyklus? (first priows) besucht)         ; freier Knoten in besucht?
       (prim-intern besucht anzahl kuerzeste (rest priows)))
      (else
       (prim-intern
        (aktualisiere-besuchtliste (second (first priows)) besucht)
        anzahl
        (cons (first priows) kuerzeste)
        (aktualisiere-priows priows)))
      )) ; Ende interne Funktion
```

In **`prim-intern`** selbst muss einmal der Erfolgsfall getestet werden. Die Suche war erfolgreich, wenn alle Knoten des Graphen im Teilbaum enthalten sind. Rückgabe ist Liste **`kuerzeste`**, die alle Kanten des Baums enthält.

Der Misserfolgsfall wird an einer leeren Prioritätswarteschlange erkannt.

Bevor eine Kante weiter verfolgt wird, muss zunächst getestet werden, ob durch sie ein Zyklus entstehen würde. Das ist dann der Fall, wenn der Zielknoten bereits zum Teilbaum gehört. Diese wird aus der Prioritätswarteschlange ohne weitere Bearbeitung entfernt.

```
;;; ===== zyklus? =====
; prüft, ob der freie Knoten der Kante in der Besuchtliste enthalten ist
(define (zyklus? kante besucht)
  (member (second kante) besucht))
```

Der eigentliche Bearbeitungsschritt im else-Fall

Der else-Fall von **prim-intern** stellt den eigentlichen Bearbeitungsschritt dar. Es wird einerseits der freie Knoten in die Besuchtliste und andererseits die Kante in den Teilbaum aufgenommen und die Prioritätswarteschlange aktualisiert werden (erste raus, Nachfolge-kanten rein).

```
;;; ===== Hilfsfunktion aktualisiere-besuchtliste =====
(define
  (aktualisiere-besuchtliste knoten besucht)
  (cond
   ((member knoten besucht)
    besucht)
   (else
    (cons knoten besucht))
   ))

;; ===== Hilfsfunktion aktualisiere-priows =====
; Zur ersten Kante aus der Prioritaetswarteschlange werden alle weiter
fuehrenden
; Kanten bestimmt und in den Rest der priows eingefuegt
(define
  (aktualisiere-priows priows)
  (fuege-alle-ein
   (nachfolge-kanten (second (first priows)) kanten)
   vor?
   (rest priows)))
```

Eine Lösung

Eine Beispiellösung zu dem angegebenen Startknoten **H** ist

((B A 10) (E D 3) (C E 4) (H C 8) (H F 6) (H G 5) (H B 3)).

Das Interessante ist, dass die Lösung prinzipiell unabhängig vom gewählten Startknoten ist. Startet man beispielsweise mit der Knoten **A** , erhält man dieselben Kanten mit anderer Reihenfolge. Abweichende Lösungen ergeben sich nur, wenn eine Kante durch eine andere mit gleicher Kantenbewertung ersetzt werden kann.

Algorithmus von Kruskal

Die Aufruf-Funktion bei Kruskal

Die Aufruf-Funktion muss bei Kruskal allein die Kantenliste übergeben bekommen. Rückgabewert ist der minimal spanning tree.

```
(define
  (minimal-spanning-tree kanten)
```

Diese Funktion hat ebenfalls wieder nur die Funktion einer Aufrufhülle, welche die eigentliche Suchfunktion aufruft:

```
  (kruskal-intern
    (fuege-alle-ein kanten vor? '())      ; einmal Prioritaetswarteschlange[1]
    (erzeuge-teilbaeume kanten))          ; alle Teilbäume nur ein Knoten
```

Die Übergabe für den ersten Parameter nutzt die Möglichkeit, mit den Funktionen der Prioritätswarteschlange die Kantenliste nach den Bewertungen zu sortieren. Der zweite Parameter muss zu jedem Knoten des Graphen einen leeren Teilbaum erzeugen.

Datenstruktur der Teilbäume

In den Daten der Teilbäume werden einmal alle enthaltenen Knoten in der ersten Teilliste gehalten, in der zweiten alle enthaltenen Kanten in einer Teilliste. Am Beginn hat diese Liste daher die Form **`'( ((A) ( )) ((B) ( )) ((C) ( )) ...)`**.

Die Funktion kruskal-intern

```
(define
  (kruskal-intern kanten teilbaeume)
  (cond
    ((null? (rest teilbaeume))                ; Baum ist fertig
     (first teilbaeume))
    ((null? kanten)                           ; Fehler !
     "Der Graph ist nicht zusammenhaengend!")
    ((zyklus? (first kanten) teilbaeume)      ; beide in einem
     (kruskal-intern (rest kanten) teilbaeume))
    (else
     (kruskal-intern
       (rest kanten)
       (verbinde-teilbaeume-mit-kante (first kanten) teilbaeume)))
  ))
```

In **`kruskal-intern`** muss einmal der Erfolgsfall getestet werden. Die Suche war erfolgreich, wenn es nur noch genau einen Teilbaum gibt. Rückgabe ist Liste **`(first teilbaeume)`**, die in der ersten Teilliste die Liste aller Knoten und in der zweiten alle Kanten des Baums enthält[2].

Der Misserfolgsfall wird an einer leeren Kantenliste erkannt. Das wird insbesondere dann auftreten, wenn der Ausggangsgraph nicht zusammenhaengend ist.

Da immer vom Kopf der sortierten Datenliste abgebaut wird, kann hier eine Kante stehen, die zwei Knoten eines bereits vorhandenen Teilbaums verbindet. Dadurch würde ein Zyklus entstehen. In dem Fall wird ohne die Kante weiter gearbeitet.
Die Funktion:

1 vor? ist das Prädikat, das die beiden Funktionen zur Verwaltung einer Prioritaetswarteschlange benötigen. Man braucht bei Kruskal aber nur einmal eine Sortierung am Beginn.

2 Man könnte sich auch auf die zweite Teilliste als Rückgabe beschränken.

```
;;; ===== zyklus? =====
; prüft, ob beide Knoten im selben teilbaum enthalten sind
(define
  (zyklus? kante teilbaeume)
  (cond
    ((null? teilbaeume)
     #f)
    ((beide-knoten-im-teilbaum? kante (first teilbaeume))
     #t)
    (else
     (zyklus? kante (rest teilbaeume)))))
```

Der eigentliche Bearbeitungsschritt im else-Fall

Wenn die beiden Knoten der Kante zu verschiedenen Teilbäumen gehören, müssen diese beiden Teilbäume verbunden werden. Man erleichter sich das, wenn man diese beiden Teilbäume nach vorn in die Liste der Teilbäume holt. Der Programmabschnitt zu beiden Aufgaben:

```
; ----- beide-nach-vorn -----
; bringt die beiden teilbaeume zur Kante nach vorn in die Liste der teilbaeume
(define
  (beide-nach-vorn kante teilbaeume)
  (define
    (beide-intern beide bearbeitet teilbaeume)
    ;(ausgabe beide bearbeitet teilbaeume)
    (cond
      ((= 2 (length beide))
       (append beide bearbeitet teilbaeume))
      ((knoten-im-teilbaum? (first kante) (first teilbaeume))
       (beide-intern
        (cons (first teilbaeume) beide)
        bearbeitet
        (rest teilbaeume)))
      ((knoten-im-teilbaum? (second kante) (first teilbaeume))
       (beide-intern
        (cons (first teilbaeume) beide)
        bearbeitet
        (rest teilbaeume)))
      (else
        (beide-intern
         beide
         (cons (first teilbaeume) bearbeitet)
         (rest teilbaeume)))
      ))
  (beide-intern '() '() teilbaeume))

; ----- verbinde-zwei -----
; verbindet die beiden Teilbaeume mit der Kante zu einem
(define
  (verbinde-zwei kante ersten zweiten)
  (list
   ; knoten
   (append (first ersten) (first zweiten))
   ;kanten
   (cons kante (append (second ersten) (second zweiten)))))
```

```
; ----- verbinde-teilbaeume-mit-kante -----
; verbindet die beiden teilbaeume mit der kante
(define
  (verbinde-teilbaeume-mit-kante kante teilbaeume)
  (define
    (verbinde-intern kante teilbaeume)
    (cons
     (verbinde-zwei kante (first teilbaeume) (second teilbaeume))
     (rest (rest teilbaeume))))
  (verbinde-intern kante (beide-nach-vorn kante teilbaeume)))
```

Eine Lösung

Eine Beispiellösung zum gegebenen Graphen:

((A C E D F G B H) ((A B 10) (C H 8) (C E 4) (D E 3) (F H 6) (B G 5) (B H 3))).