

# **Simulation eines Spanning Tree Protokolls**

**Projekt im 4. Semester**

**F. Stockmayer**

# 1 Graphalgorithmen

Graphen sind zur Beschreibung struktureller Zusammenhänge in vielen Anwendungsbereichen der Informatik sehr hilfreich. Meistens geht es um die Darstellung von Verkehrs- oder Kommunikationsnetzen verbunden mit der Suche eines optimalen Weges von einem Ausgangspunkt (Knoten des Graphen) über mehrere Zwischenstationen (weitere Knoten) auf vorgegebenen Verbindungen der Knoten (Kanten). Die Kanten können gerichtet oder ungerichtet sein. Jede Kante bekommt dabei einen Zahlenwert als sog. Gewicht zugeordnet. Das Gewicht einer Kante könnte z.B. eine Entfernungsangabe, Kosten oder Wartezeit sein, die es gilt, über das gesamte System zu minimieren.

Die Lösung der Optimierung bildet einen sog. Spannbaum (Untermenge des Graphen) mit den Eigenschaften:

- Alle Knoten sind miteinander verbunden
- Der Spannbaum enthält keine Schleifen

Daraus folgt, dass der Spannbaum mit  $n$ -Knoten  $(n-1)$ -Kanten besitzt.

## 1.1 Bekannte Algorithmen

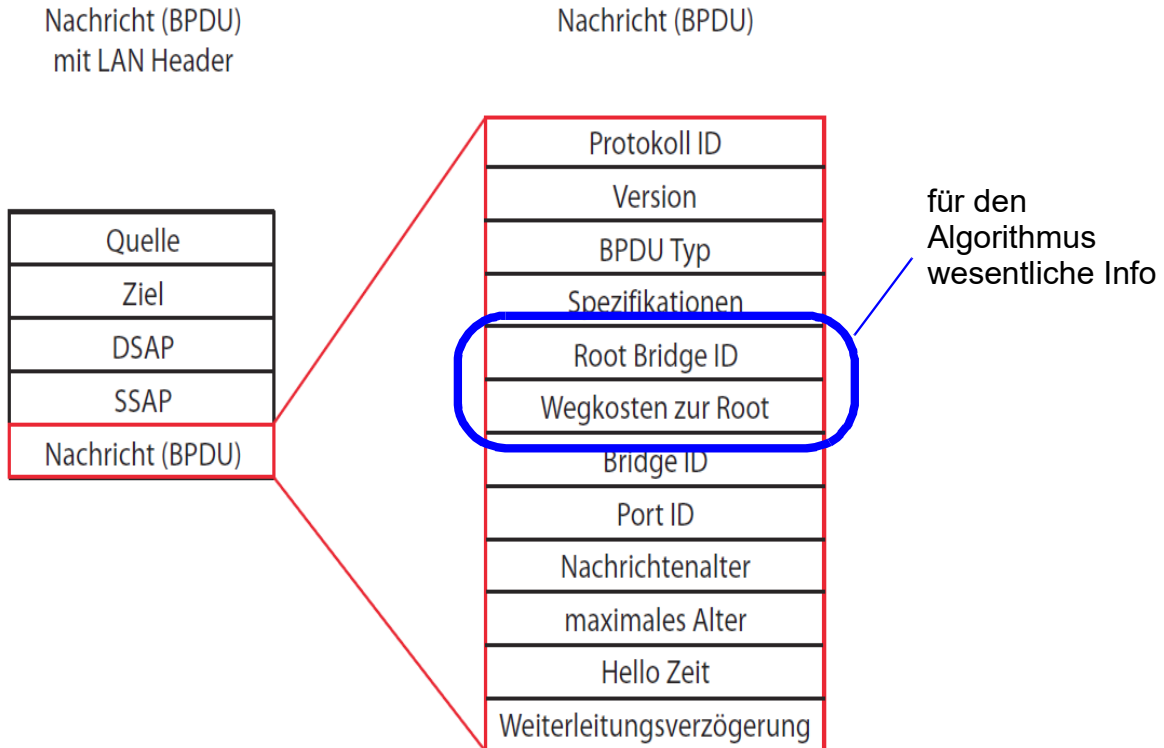
- Algorithmus von **Prim**: berechnet einen minimalen Spannbaum auf einem ungerichteten Graphen
- **Bellman-Ford**: Berechnung der kürzesten Wege in kantengewichteten Graphen
- **Dijkstra** - Algorithmus: berechnet den kürzesten Weg zwischen zwei Knoten

## 1.2 Spanning Tree in einem Layer-2 Netz

Sind in einem Kommunikationsnetz mehrere Switches miteinander verbunden, ist zur Verhinderung von Broadcast-Stürmen ein Spanning Tree für die Schleifenfreiheit verantwortlich. Wobei hier noch eine weitere Randbedingung hinzukommt: Jeder Knoten erhält ein Knotengewicht, mit dessen Hilfe die sog. Root bestimmt werden kann. Der Knoten mit dem geringsten Gewicht soll im Spanning Tree die Root bilden, die z.B. die Anbindung an das WAN herstellt. Mit einem geeigneten Algorithmus soll jeder Knoten am Ende den günstigsten Pfad zur Root kennen.

**Problem:** In einem Kommunikationsnetz kann der Spannbaum nicht an einer Stelle zentral berechnet werden. Jeder Knoten ist hier selbst verantwortlich, im Laufe der Zeit den Rootanschluss durch Informationsaustausch mit benachbarten Knoten zu erlernen. Dies führt zu einem verteilten Ansatz wofür ein Layer-2 Protokoll eingeführt wurde. Bridge-Protocol-Data-Units (BPDU) enthalten im wesentlichen folgende Informationen:

### 1.3 Bridge Protocol Data Unit (BPDU) im Layer 2 - Netz



Jeder Knoten nimmt zunächst an, er sei selbst Root und versendet die eigene Knoten-ID mit den Wegekosten 0 per Broadcast an alle direkt an den Knoten angeschlossene Nachbarn. Die Nachricht kann etwa so interpretiert werden: „Ich kenne einen Weg zur angenommenen Root mit der Knoten-ID id und den Wegekosten 0“.

Jeder Knoten entscheidet anhand der eingehenden BPDUs und Vergleich mit der eigenen ID, ob eines der Angebote angenommen werden kann. Falls ja, werden die Wegekosten zum ausgewählten Nachbarn dazuaddiert und diese Info als neues Angebot weitergegeben.

Liegen mehrere Angebote zur Root vor, aber mit unterschiedlichen Wegekosten, entscheiden die geringeren Kosten. Haben alle Knoten irgendwann dieselbe Root-ID als kleinste Knoten-ID im Netz gelernt, ändern sich die Angebote nicht mehr. Es sei denn, dass administrativ eine ID geändert wird, oder ein neuer Knoten hinzukommt.

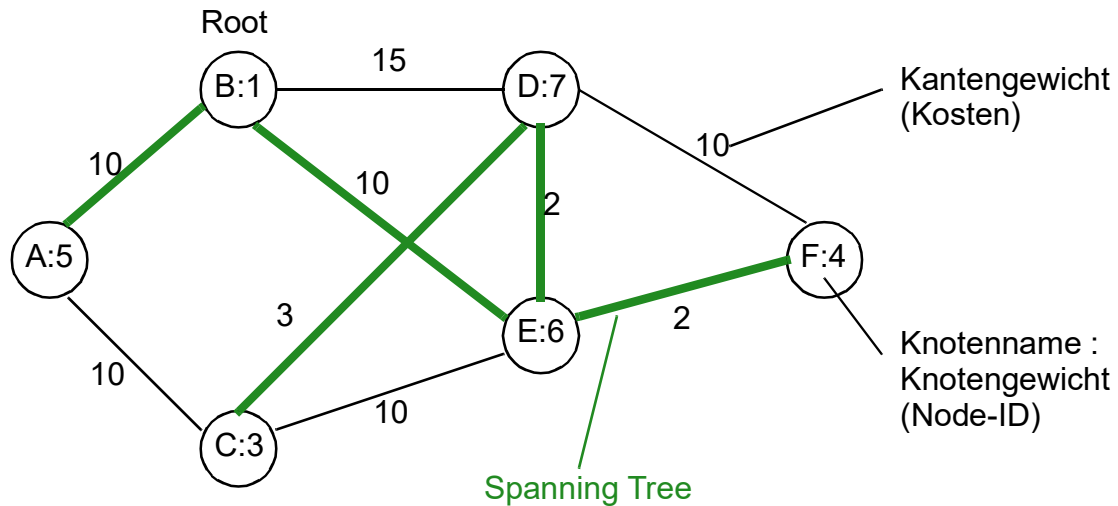
Die Konvergenz des Spanning Trees soll in dieser Laboraufgabe nachgewiesen werden.

Zur Simulation des Verhaltens wird der Berechnungsalgorithmus in C-Funktionen implementiert, die für jeden Knoten aufgerufen werden können. Prinzipiell bestünde die Möglichkeit einer nebenläufigen Ausführung innerhalb von Threads. Man müsste jedoch über mutex dafür sorgen, dass der Nachrichtenaustausch konsistent bleibt. Eine Alternative wäre eine zufallsgesteuerte sequentielle Ausführung, was der Realität ebenfalls sehr nahe kommt.

## 2 Simulation

### 2.1 Textuelle Beschreibung des Graphen

Bild 1: Beispielgraph



Inputdatei, Beschreibung des Graphen

```
Graph mygraph {  
  // Node-IDs  
  A = 5;  
  B = 1;  
  C = 3;  
  D = 7;  
  E = 6;  
  F = 4;  
  
  // Links und zugeh. Kosten  
  A - B : 10;  
  A - C : 10;  
  B - D : 15;  
  B - E : 10;  
  C - D : 3;  
  C - E : 10;  
  D - E : 2;  
  D - F : 10;  
  E - F : 2;  
}
```

Ausgabe des Spanning Tree

```
Spanning-Tree of mygraph {  
  
  Root: B;  
  A - B;  
  C - D;  
  D - E;  
  E - B;  
  F - E;  
}
```

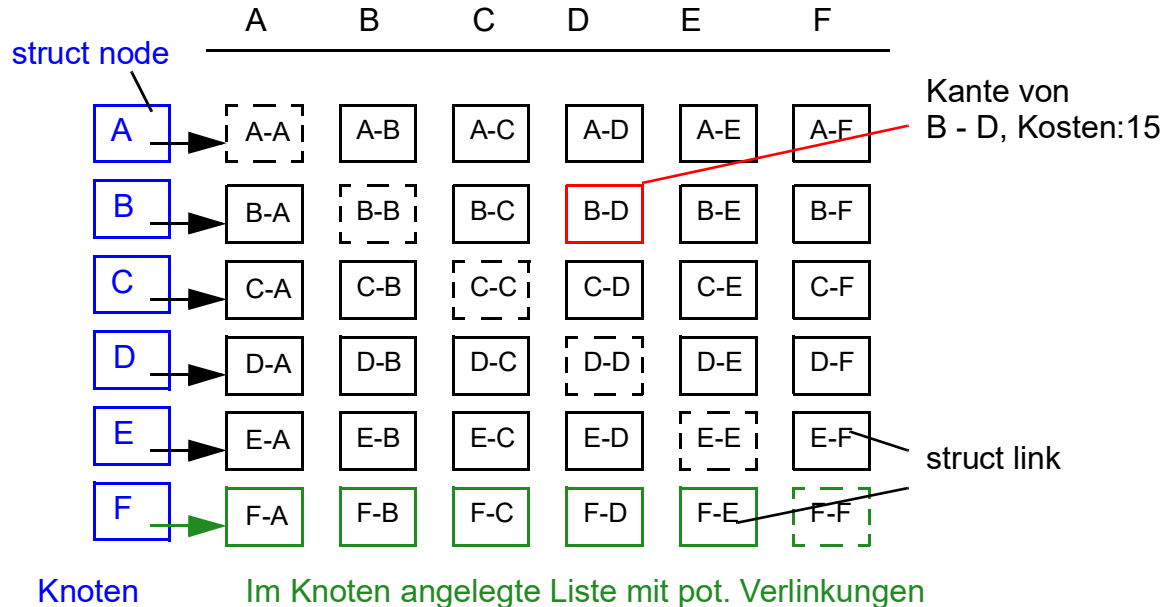
## 2.2 Vereinbarungen

- Die Graph- bzw. Knotenbezeichner beginnen mit einem Buchstaben gefolgt von alphanumerischen Zeichen bis zu einer über die Präprozessorkonstante MAX\_IDENT vorgegebenen Maximallänge.
- Kommentarzeilen beginnen mit „//“.
- Leerzeilen, führende Tabs oder Blanks werden ignoriert.
- Zeilen mit Definition der Knotengewichte und Zeilen mit Definition einer Kante können bel. gemischt werden.
- Die Eingabedatei kann max. MAX\_ITEMS Zeilen enthalten.
- Wertebereich der Kantengewichte (Kosten): 0 ... MAX\_KOSTEN, wobei der Wert 0 als „Kante nicht vorhanden“ interpretiert wird.
- Wertebereich der Knotengewichte (Node ID): 1 ... MAX\_NODE\_ID, wobei die kleinste Node-ID exklusiv im Graph vorkommen darf und damit den Root-Knoten kennzeichnet.
- Jede Kante darf nur einmal definiert werden
- Kanten, die einen Knoten mit sich selbst verbinden, sind ausgeschlossen.
- Jeder Knoten muss über mindestens eine Kante mit dem Graph verbunden sein.



## 2.3 Datenmodell

Ein wichtiger Teil der Aufgabe besteht in der Übertragung des Graphen in ein geeignetes Datenmodell zur weiteren Verarbeitung. Die Topologie des Graphen kann z.B. in einer zweidimensionalen Matrix abgebildet werden.



Im hier gewählten Datenmodell wird jeder Knoten durch eine Struktur **node** beschrieben mit den zum jew. Knoten zugehörigen Informationen:

```
typedef struct {  
    char name[MAX_IDENT+1]; // Bezeichner des Knotens  
    int nodeID;              // Knoten ID > 0  
    link *plink;             // Liste aller pot. Nachbarknoten  
    int nextHop;             // Berechneter Link zum nächsten Knoten in  
                             // Richtung Root  
    int msgCnt;              // Zählt mit, wie oft der Knoten bei der  
                             // Bearbeitung des Algorithmus aufgerufen wird  
} node;
```

Jeder Kreuzungspunkt der Matrix stellt eine potentielle Verlinkung zwischen zwei Knoten dar, angezeigt durch die initialen Kosten (0: nicht verlinkt, >0: verlinkt).

Gleichzeitig kann diese Stelle, beschrieben durch die Struktur **link**, auch zum Austausch der Nachrichten zwischen den Knoten dienen:

Ein Knoten sendet Nachrichten an die in der Zeile verlinkten Nachbarknoten

Ein Knoten empfängt Nachrichten von allen in der Spalte verlinkten Nachbarknoten.

Die Hauptdiagonalelemente spielen keine Rolle, da sie eine Verlinkung oder Nachrichtenaustausch des Knotens mit sich selbst bedeuten.

Die Struktur **link** könnte dann so aussehen:

```
typedef struct {  
    // Linkkosten von Node_i -> Node_k  
    // kosten=0: kein Link vorhanden  
    // Entspricht ursprüngliche Initialisierung des eingelesenen Graphen  
    int kosten;  
  
    // Über diesen Link erhaltene Nachricht der Nachbarknoten  
    // mit Vorschlag der Root incl. Gesamtkosten zur Root  
    int rootID;  
    int summeKosten;  
} link;
```

Hinweis: Es gibt sicher noch viele weitere Ansätze oder Alternativen zum hier gewählten Datenmodell. Die selbst entwickelten Ansätze sind natürlich immer die besten ;-)

So stellt man z.B. bei genauer Betrachtung fest, dass die beiden Integer-Variable `rootID` und `summeKosten` identisch an alle Nachbarknoten verteilt wird, um einem Broadcast-Verhalten des Layer-2 Netzes zu entsprechen. Die Daten liegen dann in allen Nachbarknoten redundant vor. Man könnte daher die beiden Variablen in die `struct node` verschieben, um sie nur einmal im Knoten vorzuhalten. Dadurch ändert sich jedoch der Charakter der Kommunikation: Jeder Knoten müsste dann bei seinem Nachbarn einen Pfad zur Root erfragen (wodurch sich das prinzipielle Verhalten des Programms jedoch nicht verändert).

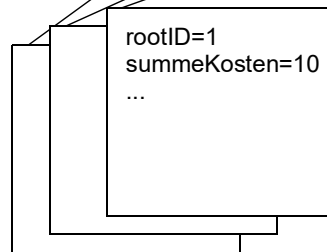
## 2.4 Nachrichtenaustausch

	A	B	C	D	E	F
A	[A-A]	[A-B]	[A-C]	[A-D]	[A-E]	[A-F]
B	[B-A]	[B-B]	[B-C]	[B-D]	[B-E]	[B-F]
C	[C-A]	[C-B]	[C-C]	[C-D]	[C-E]	[C-F]
D	[D-A]	[D-B]	[D-C]	[D-D]	[D-E]	[D-F]
E	[E-A]	[E-B]	[E-C]	[E-D]	[E-E]	[E-F]
F	[F-A]	[F-B]	[F-C]	[F-D]	[F-E]	[F-F]

Knoten C sendet Nachricht an alle  
verlinkte Knoten (Kosten > 0)

	A	B	C	D	E	F
A	[A-A]	[A-B]	[A-C]	[A-D]	[A-E]	[A-F]
B	[B-A]	[B-B]	[B-C]	[B-D]	[B-E]	[B-F]
C	[C-A]	[C-B]	[C-C]	[C-D]	[C-E]	[C-F]
D	[D-A]	[D-B]	[D-C]	[D-D]	[D-E]	[D-F]
E	[E-A]	[E-B]	[E-C]	[E-D]	[E-E]	[E-F]
F	[F-A]	[F-B]	[F-C]	[F-D]	[F-E]	[F-F]

Knoten C empfängt Nachrichten von allen  
verlinkten Knoten



Aussage: E hat C einen Pfad zur Root mit der ID 1 und  
Pfadkosten = 10 angeboten.

	A	B	C	D	E	F
A	[A-A]	A-B	A-C	A-D	A-E	A-F
B	B-A	[B-B]	B-C	B-D	B-E	B-F
C	C-A	C-B	[C-C]	C-D	C-E	C-F
D	D-A	D-B	D-C	[D-D]	D-E	D-F
E	E-A	E-B	E-C	E-D	[E-E]	E-F
F	F-A	F-B	F-C	F-D	F-E	[F-F]

Knoten C sendet Nachricht an alle  
verlinkte Knoten (Kosten > 0)

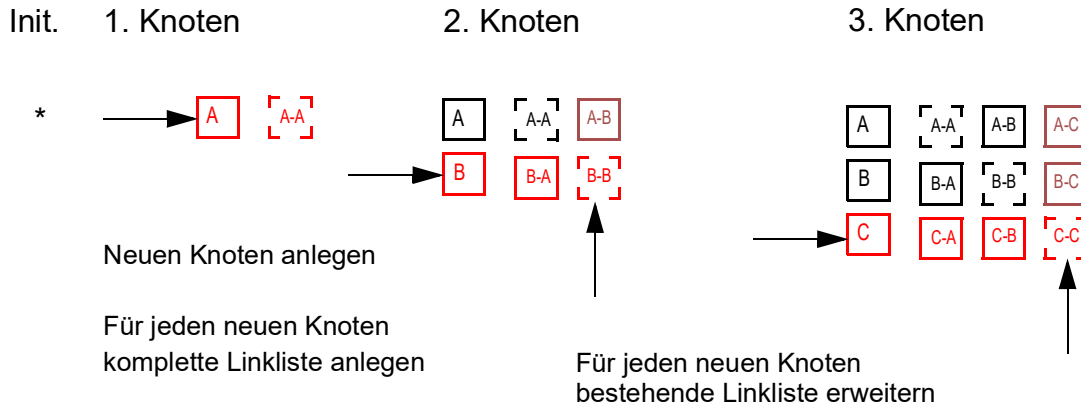
rootID = 1  
summeKosten = 15  
...

Aussage:  
Knoten C bietet seinen Nachbarn Knoten A, D, E  
einen Pfad zur Root mit der ID 1 und den Gesamtkosten 15 an

	A	B	C	D	E	F
A	[A-A]	A-B	A-C	A-D	A-E	A-F
B	B-A	[B-B]	B-C	B-D	B-E	B-F
C	C-A	C-B	[C-C]	C-D	C-E	C-F
D	D-A	D-B	D-C	[D-D]	D-E	D-F
E	E-A	E-B	E-C	E-D	[E-E]	E-F
F	F-A	F-B	F-C	F-D	F-E	[F-F]

Knoten C empfängt Nachrichten von allen  
verlinkten Knoten

## 2.5 Einlesen des Graphen und Aufbau der Matrix



Beim Einlesen des Graphen können an drei Stellen neue Knoten auftauchen:

1. Bei der Definition des Knotengewichtes, z.B.:  
    **Node1** = 15;
2. Bei der Definition einer neuen Kante, Bezeichner auf der linken Seite:  
    **Node2** - Node3 : 5;
3. Bei der Definition einer neuen Kante, Bezeichner auf der rechten Seite:  
    Node2 - **Node3** :5;

Beim Einlesen des Graphen ist die Anzahl der Knoten noch nicht bekannt. Der Matrixaufbau könnte deshalb mit Hilfe einer dynamischen Speicherverwaltung, z.B. mit `realloc()` einfach gelöst werden.

Zur Vereinfachung der Schnittstellen ist es (ausnahmsweise) erlaubt, die Matrix global zu deklarieren:

```
node *pnode = NULL;
```

Um beispielsweise die rootIDs der Matrix so zu initialisieren, damit jeder Knoten annimmt, er selbst sei Root:

```
for(i=0; i < nodeCnt; i++){
    pnode[i].nextHop = i;
    for(k=0; k < nodeCnt; k++){
        if(pnode[i].plink[k].kosten == 0) continue;
        pnode[i].plink[k].rootID = pnode[k].nodeID;
    }
}
```

Oder um alle Kanten mit den jew. Kosten auszugeben:

```
for(i=0; i < nodeCnt; i++){
    for(k=0; k < nodeCnt; k++){
        printf("%d ", pnode[i].plink[k].kosten);
    }
    printf("\n");
}
```

## 2.6 Vorschlag für einzelne C-Funktionen

### 2.6.1 `int getGraph(FILE *fp);`

**Übergabeparameter:** Filepointer auf geöffnete Graphen-Datei

**Returnwert:** Anzahl eingelesener Knoten

**Beschreibung:** Liest Datei zeilenweise, prüft eingelesene Daten, Aufbau der Matrix.

### 2.6.2 `int checkline(char *line);`

**Übergabeparameter:** Pointer auf eingelesene Zeile der Graphen-Datei

**Returnwert:** 1 falls Kommentarzeile, Leerzeile oder Zeile aus Blanks und Tabs

**Beschreibung:** Prüft eingelesene Textzeile

### 2.6.3 `int isValid(char *string);`

**Übergabeparameter:** Pointer auf Identifier (Name des Knoten oder Graphen)

**Returnwert:** 1 OK, 0 nicht OK

**Beschreibung:** Prüft eingelesenen Identifier auf ersten Buchstabe und weitere alpha-numerische Zeichen.



### 2.6.4 int getIndex(char \*name, int nodeCnt);

**Übergabeparameter:** Pointer auf Knotenname, Anzahl vorhandener Knoten

**Returnwert:** Index des gesuchten Knotens: 0 ... (nodeCnt-1)

**Beschreibung:** Findet den Index des Knotens über seinen Namen

### 2.6.5 int appendNode(char \*name, int nodeCnt);

**Übergabeparameter:** Pointer auf neuen Knotenname, Anzahl bisheriger Knoten

**Returnwert:** Anzahl aktuell vorhandener Knoten

**Beschreibung:** Fügt einen neuen Knoten der Matrix hinzu

### 2.6.6 void appendLink(int nodeCnt);

**Übergabeparameter:** Anzahl bisher vorhandener Knoten

**Returnwert:**

**Beschreibung:** Fügt in allen bisherigen Knoten einen neuen Link hinzu. Fügt im neuen Knoten eine komplette Linkliste hinzu

## 2.6.7 void sptree(int index, int nodeCnt);

**Übergabeparameter:** Index des Knotens, der bearbeitet werden soll, Anzahl Knoten

**Returnwert:**

**Beschreibung:** Berechnet im ausgewählten Knoten den günstigsten Pfad zur Root und gibt die Info an alle Nachbarn weiter

Der Aufruf erfolgt z.B. aus main und einer zufälligen Auswahl des Knotens.

Generierung einer Zufallszahl im Bereich 0 .... nodeCnt-1:

```
// Initialisierung des Zufallgenerators mit der aktuellen Systemzeit
time_t t;
time(&t);
srand((unsigned int)t);

...


// Zufallszahl generieren
node = rand() % nodeCnt;
sptree(node, nodeCnt);
```

### 3 Implementierung

1. Algorithmus verstehen, evtl. anhand kleiner Beispiele mit Bleistift und Papier!
2. Wie funktioniert die Speicherverwaltung (z.B. mit realloc)?
3. Anlegen einer Liste, Funktionen zum Bearbeiten der Liste  
z.B. einfaches Anhängen neuer Knoten
4. Einlesen des Graphen und Abbildung der Topologie z.B. in einer Matrix
5. Test durch Ausgabe aller Kanten mit jew. Kosten
6. Initialisieren aller Spalten der Matrix mit entsprechender Root-ID, gleichzeitig kann der Eintrag des nextHop im Knoten initialisiert werden (auf sich selbst).
7. Anwendung diverser Tests auf den Graphen, z.B.:
  - Alle Node-Ids > 0?
  - Gibt es nur eine Root-ID?
  - Ist der Graph verbunden?
  - Gibt es Knoten, die mit sich selbst verbunden sind?
  - Ausgabe Gesamtanzahl Knoten und Kanten
8. Generierung einer Zufallszahl
9. Implementierung der Pfadberechnung
10. Test über eine genügend hohe Anzahl Iterationszyklen  
Abbruchkriterium t.b.d., z.B. jeder Knoten wurde nodeCnt-mal besucht.
11. Beobachtung der Matrixveränderungen und Ausgabe des Ergebnisses

## Manuelle Verfolgung der Schritte bei sequentieller Bearbeitung

Initialisierung	A	B	C	D	E	F
A 5 0	0:0 1:0	3:0	0:0	0:0	0:0	0:0
B 1 1	5:0	0:0	0:0	7:0	6:0	0:0
C 3 2	5:0	0:0	0:0	7:0	6:0	0:0
D 7 3	0:0	1:0	3:0	0:0	6:0	4:0
E 6 4	0:0	1:0	3:0	7:0	0:0	4:0
F 4 5	0:0	0:0	0:0	7:0	6:0	0:0

nodeID      nextHop            rootID      summeKosten

Die Spalten werden so initialisiert, dass jeder Knoten bei Auswertung der Spalte annehmen muss, er selbst sei Root. Dies gelingt dann, wenn in allen Einträgen der Spalte die Node-ID des Knotens eingetragen ist, sowie einer summeKosten von 0. Daraus ergibt sich nachfolgende Simulation, wobei die Zwischenergebnisse abhängig von der Bearbeitung der einzelnen Knoten durchaus unterschiedlich sein können, am Ende jedoch immer gegen einen stabilen Endzustand konvergieren.

## 1. Schritt (A)

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

## 2. Schritt (B)

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

## 3. Schritt (C)

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

## 4. Schritt (D)

5 0 0:0 5:0 5:0 0:0 0:0 0:0  
 1 1 1:0 0:0 0:0 1:0 1:0 0:0  
 3 2 3:0 0:0 0:0 3:0 3:0 0:0  
 7 1 0:0 1:15 1:15 0:0 1:15 1:15  
 6 4 0:0 1:0 3:0 7:0 0:0 4:0  
 4 5 0:0 0:0 0:0 7:0 6:0 0:0

## 5. Schritt (E)

5 0 0:0 5:0 5:0 0:0 0:0 0:0  
 1 1 1:0 0:0 0:0 1:0 1:0 0:0  
 3 2 3:0 0:0 0:0 3:0 3:0 0:0  
 7 1 0:0 1:15 1:15 0:0 1:15 1:15  
 6 1 0:0 1:10 1:10 1:10 0:0 1:10  
 4 5 0:0 0:0 0:0 7:0 6:0 0:0

## 6. Schritt (F)

5 0 0:0 5:0 5:0 0:0 0:0 0:0  
 1 1 1:0 0:0 0:0 1:0 1:0 0:0  
 3 2 3:0 0:0 0:0 3:0 3:0 0:0  
 7 1 0:0 1:15 1:15 0:0 1:15 1:15  
 6 1 0:0 1:10 1:10 1:10 0:0 1:10  
 4 4 0:0 0:0 0:0 1:12 1:12 0:0

## 7. Schritt (A)

5 1 0:0 1:10 1:10 0:0 0:0 0:0  
 1 1 1:0 0:0 0:0 1:0 1:0 0:0  
 3 2 3:0 0:0 0:0 3:0 3:0 0:0  
 7 1 0:0 1:15 1:15 0:0 1:15 1:15  
 6 1 0:0 1:10 1:10 1:10 0:0 1:10  
 4 4 0:0 0:0 0:0 1:12 1:12 0:0

## 8. Schritt (B)

5 1 0:0 1:10 1:10 0:0 0:0 0:0  
 1 1 1:0 0:0 0:0 1:0 1:0 0:0  
 3 3 1:18 0:0 0:0 1:18 1:18 0:0  
 7 1 0:0 1:15 1:15 0:0 1:15 1:15  
 6 1 0:0 1:10 1:10 1:10 0:0 1:10  
 4 4 0:0 0:0 0:0 1:12 1:12 0:0

## 9. Schritt (C)

5 1 0:0 1:10 1:10 0:0 0:0 0:0  
 1 1 1:0 0:0 0:0 1:0 1:0 0:0  
 3 3 1:18 0:0 0:0 1:18 1:18 0:0  
 7 4 0:0 1:12 1:12 0:0 1:12 1:12  
 6 1 0:0 1:10 1:10 1:10 0:0 1:10  
 4 4 0:0 0:0 0:0 1:12 1:12 0:0

....

## 15. Schritt

5 **1** 0:0 1:10 1:10 0:0 0:0 0:0  
 1 **1** 1:0 0:0 0:0 1:0 1:0 0:0  
 3 **3** 1:15 0:0 0:0 1:15 1:15 0:0  
 7 **4** 0:0 1:12 1:12 0:0 1:12 1:12  
 6 **1** 0:0 1:10 1:10 1:10 0:0 1:10  
 4 **4** 0:0 0:0 0:0 1:12 1:12 0:0

root-ID überall 1

**Ergebnis:** Interpretation nextHop:A - B (Index **1**)B - B (Index **1**)C - D (Index **3**)D - E (Index **4**)E - B (Index **1**)F - E (Index **4**)

## 4 Erweiterung

### 4.1 Dijkstra

Das entwickelte Programm zur Berechnung eines Spanning Tree kann als Grundlage für eine Erweiterung zur Implementierung des Dijkstra-Algorithmus herangezogen werden.

Spanning Tree: Finde den günstigsten Pfad von jedem Knoten zur Root

Dijkstra: Finde den günstigsten Pfad von einem Startknoten bzw. von jedem Knoten zu allen anderen Knoten.

#### Aufgaben:

- Anpassung des Datenmodells
- Implementierung des Algorithmus  
void dijkstra(int index, int nodeCnt);
- Anpassung des Abbruchkriteriums t.b.d., z.B.  
jeder Knoten mind.  $2 \cdot \text{nodeCnt}$ -mal besucht. oder einstellbar über Kommandozeile
- Ausgabe/Darstellung des Ergebnisses

**Beispiel:** Ergebnis des Graphen aus Bild 1

Dijkstra of mygraph {

A B C D **E** F

**A**: - B C C **C** C

B: A - E E E E

C: A D - D **D** D

D: C E C - **E** E

E: D B D D - F

F: E E E E E -

}

**Günstigster Pfad von A nach E ?**

Next Hop von **A** in Richtung **E** ist **C**

Next Hop von C in Richtung E ist **D**

Next Hop von D in Richtung E ist **E**

Günstigster Pfad von A nach E:  
**A - C - D - E (Kosten = 15)**