# Algorithmen und Datenstrukturen 1

Ausgearbeitetes Übungsblatt 5

Datum: 2. Juni 2005
Erstellt mit LaTeX

## Aufgabe 5.1

**Aufgabenstellung:**

Gegeben sei ein ungerichteter Graph $G = (V, E)$ mit Knotenmenge $V$ und Kantenmenge $E$. Erstellen Sie einen Algorithmus in detailliertem Pseudocode, der für einen gegebenen Knoten $u \in V$ feststellt, ob ein Kreis in $G$ existiert, der $u$ beinhaltet. Gibt es einen solchen Kreis, so sollen alle darauf liegenden Knoten ausgegeben werden.

**Lösung:**

```
Kreis(G, u) {
  markiert[1...|V|]=0;
  return DFS(G, u, u);
}

DFS(G, v, u) {
  markiert[v] = 1;
  for all(w in N[v]) {
    if((markiert[w] == 1) && (w == u)) {
      return 1;
    }
    else if(markiert[w] == 0) {
      if(DFS(G, w, u)==1) {
        Ausgabe(v);
        return 1;
      }
    }
  }
  return 0;
}
```

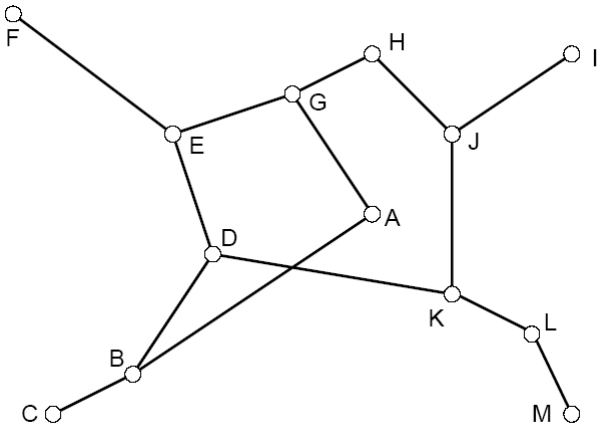
Der Algorithmus durchsucht den Graphen auf den Umstand, ob der Knoten $u$ auf einem Kreis liegt oder nicht. Dazu startet die Tiefensuche beim Knoten $u$. Wenn die Tiefensuche wieder bei $u$ ankommt, „marschiert“ sie den ganzen Weg, den sie gekommen ist, wieder zurück, und gibt alle auf diesem Weg liegenden Knoten aus.
Der Algorithmus **Kreis** gibt 1 zurück, falls $u$ auf einem Kreis liegt, ansonsten 0.

---

## Aufgabe 5.2

**Aufgabenstellung:**

Schreiben Sie einen effizienten Algorithmus in Pseudocode, der den kürzesten Kantenzug zwischen zwei Knoten in einem *ungewichteten* Graphen ermittelt.
Verwenden Sie das Prinzip der *Breitensuche*: Im Gegensatz zur Tiefensuche werden hier zunächst alle Nachbarn des aktuellen Knotens besucht, dann rekursiv die Nachbarn der Nachbarn.
Hinweis: Ersetzen Sie den bei der Tiefensuche implizit oder explizit eingesetzten Stack durch eine Queue; die Queue-funktionen *put*, *get* und *empty* können Sie als gegeben voraussetzen.
Demonstrieren Sie die Funktionsweise Ihres Algorithmus, indem Sie den kürzesten Weg zwischen $A$ und $K$ im hier angegebenen Graphen ermitteln:




**Lösung:**

```
Pfad(G, u, v) {
  Vorgaenger[1...|V|] = -1;

  put(u);
  Vorgaenger(u)=u;

  while(!empty()) {
    w = get();
    for all(t in N(w)) {
      if(Vorgaenger[t] == -1) {
        Vorgaenger[t] = v ;
        put(t);
        if(t == v) {
          w = t;
          break;
        }
      }
    }
    if(w == v) {
      break;
    }
  }

  t = v;
  while(t != u) {
    Ausgabe(t);
    t = Vorgaenger[t];
  }
  Ausgabe(t);
}
```

Angewandt auf die Problemstellung, den kürzesten Weg von $A$ nach $K$ zu finden, werden die folgenden Aktionen gesetzt (**get(x)** steht für: Nimm **x** aus der Queue):

- Initialisierung (alle Vorgänger werden auf -1 gesetzt)
- **put**(A); // Queue: A
- **get**(A); // Queue: leer
- **put**(B); **put**(G); // Queue: B, G
- **Vorgaenger**(B) = A; **Vorgaenger**(G) = A;
- **get**(B); // Queue: G
- **put**(C); **put**(D); // Queue: G, C, D

- **Vorgaenger**(C) = B; **Vorgaenger**(D)=B;

- **get**(G); // Queue: C, D
- **put**(E); **put**(H); // Queue: C, D, E, H
- **Vorgaenger**(E) = G; **Vorgaenger**(H)=G;

- **get**(C); // Queue: D, E, H

- **get**(D); // Queue: E, H
- **put**(K); // Queue: E, H, K
- Vorgaenger(K) = D;

- **Ausgabe**(K);
- **Ausgabe**(D);
- **Ausgabe**(B);
- **Ausgabe**(A);

---

# Aufgabe 5.3

**Aufgabenstellung:**

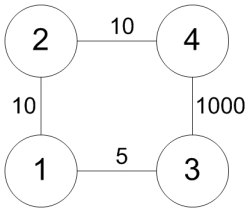
Gegeben sei ein Greedy-Algorithmus, der zur Suche des längsten Pfades zwischen zwei Knoten eines ungerichteten, gewichteten Graphen gedacht ist. Ausgehend vom Startknoten $S$ wählt dieser Algorithmus bei jedem Knoten jeweils immer die längste noch nicht verwendete Kante, um dem Zielknoten $Z$ auf einem zusammenhängenden Pfad näher zu kommen.
Schreiben Sie den Algorithmus in Pseudocode und verwenden Sie dabei eine Adjazenzmatrix. Liefert dieser Algorithmus immer den längsten Pfad? - Warum (nicht)?
Geben Sie einen Graphen mit mindestens sieben Knoten und entsprechende Start- und Zielknoten an, sodass der Algorithmus funktioniert. Wenn er nicht funktioniert, geben Sie zusätzlich ein möglichst einfaches Gegenbeispiel an.

**Lösung:**

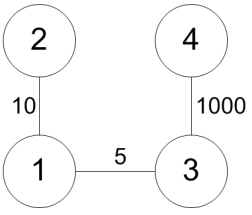
```
Pfad(G, u, v) {
  if(u == v) {
    Ausgabe(v);
    return 1;
  }

  maxpos = -1;
  for(i=1; i<=|V|; i++) {
    if(maxpos < 0 || A[u,i] > A[u, maxpos] {
      maxpos = i;
    }
  }

  if(A[u, maxpos] == 0 || Vorgaenger[A[u,maxpos]] >= 0) {
    return 0;
  }
  else {
    Vorgaenger[A[u, maxpos]] = u;
    if(Pfad(G, A[u, maxpos], v) == 1) {
      Ausgabe(u);
      return 1;
    }
    else {
```

```
        return 0;
      }
    }
}
```

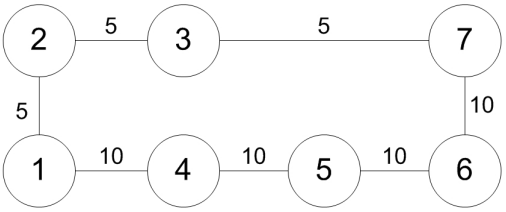
Dass eine Kante, welche an den bereits gefundenen Pfad anschließt, ein geringes Gewicht hat, sagt nichts über das Gewicht jener Kanten aus, welche an diese Kante bzw. an eine auf Grund zu hohen Gewichts nicht angefügte Kante anschließen. Man betrachte den folgenden Graphen (Startknoten ist 1, Endknoten ist 4):




Der Algorithmus wählt die Kanten $(1, 2)$ und $(2, 4)$, um von 1 nach 4 zu kommen. Das Gesamtgewicht dieser beiden Kanten beträgt 20. Der Pfad über die Kanten $(1, 3)$ und $(3, 4)$ hat allerdings ein Gewicht von 1005. In diesem (extremen) Beispiel liefert der Algorithmus also kein korrektes Ergebnis.
Auch dieser Graph führt zu Problemen (Startknoten ist wieder 1, Zielknoten wieder 4):




Hier verfolgt der Algorithmus die Kante $(1, 3)$ und bleibt dann beim Knoten 3 stehen, weil es von dort keine weiterführende Kante gibt.
Mit folgendem Graphen gibt es jedoch beispielsweise keine Probleme (1 ist Startknoten, 7 ist Endknoten):




Der Algorithmus findet einwandfrei den Kantenzug „1-4-5-6-7“ mit einem Kantengewicht von 40, während der Kantenzug „1-2-3-7“ nur ein Gewicht von 15 hat.

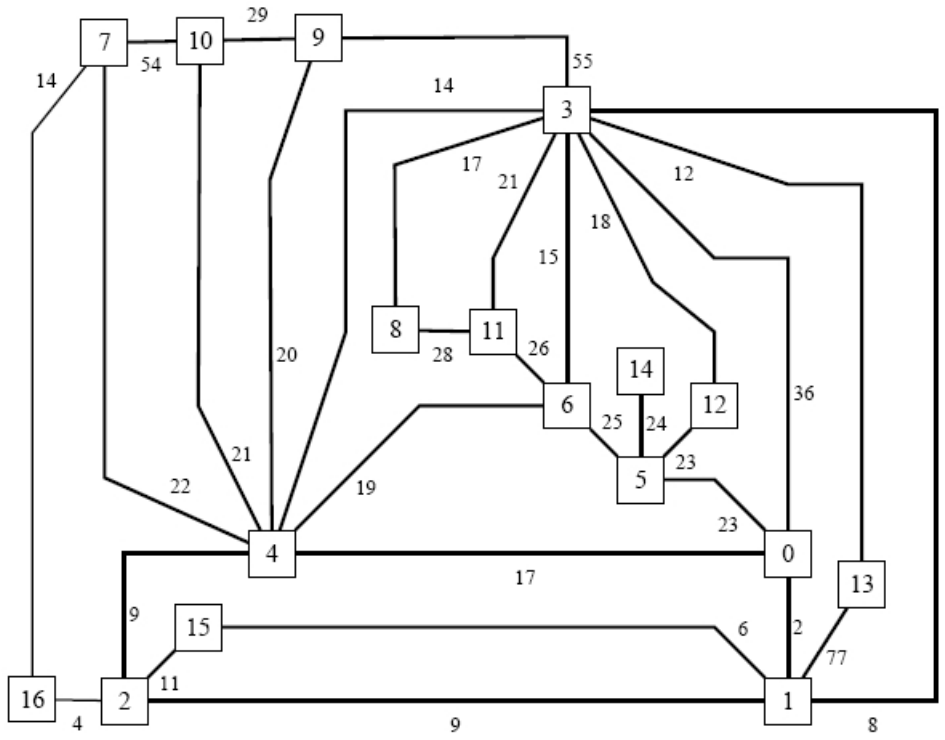
Anzumerken ist hier jedoch, dass meiner Meinung nach die Aufgabenstellung nicht eindeutig ist. Es geht nämlich aus der Angabe nicht eindeutig hervor, ob der Algorithmus bei der Auswahl einer Kante darauf achten muss, dass überhaupt ein Pfad vom Start- zum Zielknoten existiert, welcher diese Kante enthält.

---

# Aufgabe 5.4

**Aufgabenstellung:**

In einem zusammenhängenden, gewichteten, ungerichteten Graphen soll eine kreisfreie Teilmenge von Kanten gefunden werden, für welche die Summe der Gewichte maximal ist.
Welchen Algorithmus aus der Vorlesung können Sie für dieses Problem modifizieren? Wie muss er verändert werden? Geben Sie den neuen Pseudocode an.




Zeigen Sie den Ablauf Ihres Algorithmus am angegebenen Graphen. Markieren Sie die Kanten der Lösungsmenge und nummerieren Sie diese in der Reihenfolge, in der sie durch Ihren Algorithmus gefunden werden. Geben Sie die Laufzeit des Algorithmus in Θ-Notation an und begründen Sie Ihre Behauptung.

**Lösung:**

Hier bieten sich die Algorithmen von Prim und Kruskal an. Verändert werden muss jeweils nur das Auswahlkriterium, d. h. es wird jeweils nicht aus den für das Hinzufügen zum Spannbaum zur Verfügung stehenden Kanten jene mit dem geringsten Gewicht ausgewählt und hinzugefügt, sondern jene mit dem größten Gewicht.
Im Folgenden eine entsprechend modifizierte Variante des Algorithmus von Prim, sodass dieser den maximalen Spannbaum eines Graphen ermittelt:

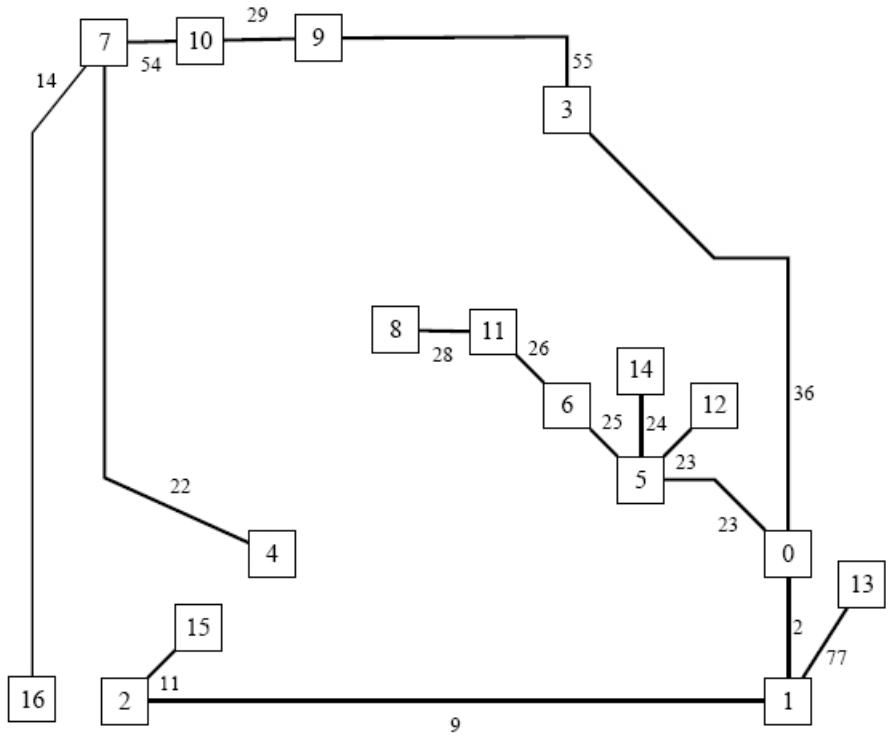
```
Anti-Prim (G) {
  Wähle beliebigen Startknoten s aus V;
  C = {s};
  T = {};
  while(|C| != |V|) {
    Ermittle eine Kante e=(u,v) mit u aus C, v aus V\C mit maximalem Gewicht;
    T = T + {e};
    C = C + {v};
  }
  return T;
}
```

Dabei steht + für die Bildung der Vereinigungsmenge („a + b“ steht also statt $a \cup b$).
Der vorliegende Algorithmus hat eine Laufzeit von $O(|V|)^2)$, und zwar aus denselben Gründen wie der Algorithmus von Prim im Skriptum (S. 140f).
Die Reihenfolge, in welcher die Kanten zum Spannbaum hinzugefügt werden, ist abhängig von der Wahl des Startknotens. In jedem Fall ist nach Ablauf des Algorithmus

$$T = \{(0,1),(0,3),(0,5),(1,2),(1,13),(2,15),(3,9),(4,7),(5,6),(5,12),(5,14),(6,11),(7,10),(7,16),(8,11),(9,10)\}$$

Die Summe über alle Kantengewichte des Spannbaums beträgt in jedem Fall 468. Wird beispielsweise 0 als Startknoten angenommen, so werden die Kanten in der folgenden Reihenfolge aufgenommen: $(0,3)$, $(3,9)$, $(9,10)$, $(7,10)$, $(0,5)$, $(5,6)$, $(6,11)$, $(8,11)$, $(5,14)$, $(5,12)$, $(7,4)$, $(7,16)$, $(3,13)$, $(1,13)$, $(1,2)$, $(2,15)$. Wird hingegen 13 als Startknoten angenommen, so lautet die Reihenfolge wie folgt: $(1,13)$, $(3,13)$, $(3,9)$, $(0,3)$, $(9,10)$, $(7,10)$, $(0,5)$, $(5,6)$, $(6,11)$, $(8,11)$, $(5,14)$, $(5,12)$, $(4,7)$, $(7,16)$, $(1,2)$ und $(2,15)$.

Der entstehende Maximum Spanning Tree sieht wie folgt aus:




---

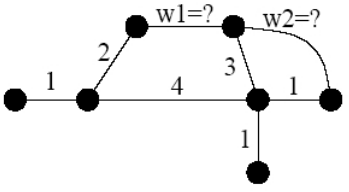
# Aufgabe 5.5

**Aufgabenstellung:**

Gegeben sei ein vollständiger, ungerichteter Graph $G = (V, E)$. Jeder Kante $e \in E$ sei ein Gewicht $w_e \geq 0$ zugeordnet. Ein *Grad-3-beschränkter Spannbaum* von G ist ein Spannbaum von $G$, für den gilt, dass der Grad jedes Knotens maximal drei ist. Ein *minimaler* Grad-3-beschränkter Spannbaum $T \subseteq E$ ist ein Grad-3-beschränkter Spannbaum mit minimalem Gesamtgewicht $w(T) = \sum_{e \in T} w_e$.

(a) Schreiben Sie in einfachem Pseudocode (ohne auf eventuelle Datenstrukturen genauer einzugehen) einen Algorithmus, der auf *Kruskals* Algorithmus zum Finden eines minimalen Spannbaums basiert und immer einen gültigen Grad-3-beschränkten Spannbaum mit möglichst geringem, aber nicht unbedingt minimalem, Gewicht zurückliefert.

(b) Da das Finden eines Grad-3-beschränkten Spannbaums im Allgemeinen NP-schwierig ist, können Sie davon ausgehen, dass Ihr Algorithmus nicht immer einen minimalen Grad-3-beschränkten Spannbaum zurückliefert. Das sollen Sie nun mit einem Beispiel beweisen.
Nennen Sie für den folgenden nicht-euklidischen Graphen konkrete Beispielwerte für die zwei fehlenden Kantengewichte $w1$ und $w2$, sodass Ihr Algorithmus sicher einen gültigen, aber nicht minimalen Grad-3-beschränkten Spannbaum zurückliefert. Erklären Sie die Situation auch mit einem Satz. (Um entsprechend der Definition von einem vollständigen Graphen auszugehen, nehmen Sie einfach an, dass alle nicht eingezeichneten Kanten das Gewicht $\infty$ haben.)




**Lösung:**

```
Spannbaum (G) {
  Sortiere E nach Gewichten;
  T = {};
  for all(e in E) {
    if(kreisfrei(T + {e}) && |N(start(e))|<3 && |N(end(e))|<3) {
      T = T + {e};
    }
  }
}
```

Die verwendeten Methoden erfüllen die folgenden Funktionen:

- **kreisfrei** liefert logisch *wahr* zurück, wenn die übergebene Menge an Kanten einen Kreis enthält, ansonsten logisch *falsch*.
- **start** gibt den Startknoten der angegebenen Kante zurück.
- **end** gibt den Endknoten der angegebenen Kante zurück.

Da der MST eines Graphen nicht Grad-3-beschränkt sein muss, muss es nicht für jeden Graphen einen minimalen Grad-3-beschränkten Spannbaum geben. Das ist hier beispielsweise der Fall, wenn man $w_1 = 5$ und $w_2 > 3$ annimmt.

---

# Aufgabe 5.6

**Aufgabenstellung:**

Sei $T$ ein *Radix Trie* für 5-Bit-Zahlen. Es hat also jedes in $T$ gespeicherte Element die Form $(b_1, b_2, b_3, b_4, b_5) \in 0, 1^5$. In der Wurzel wird nach Bit $b_1$ unterschieden, in der zweiten Ebene nach Bit $b_2$, und so weiter.

(a) Zeichnen Sie einen Radix Trie, der die folgenden Elemente enthält:

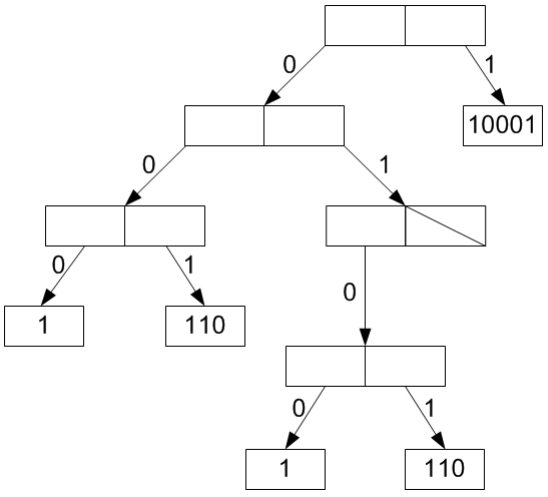
$$1001, 110, 1010, 1, 10001$$

(Hinweis: Auch bei Binärzahlen ist es üblich, führende Nullen beim Anschreiben wegzulassen. Ergänzen Sie diese, wo es Ihnen nötig erscheint!)

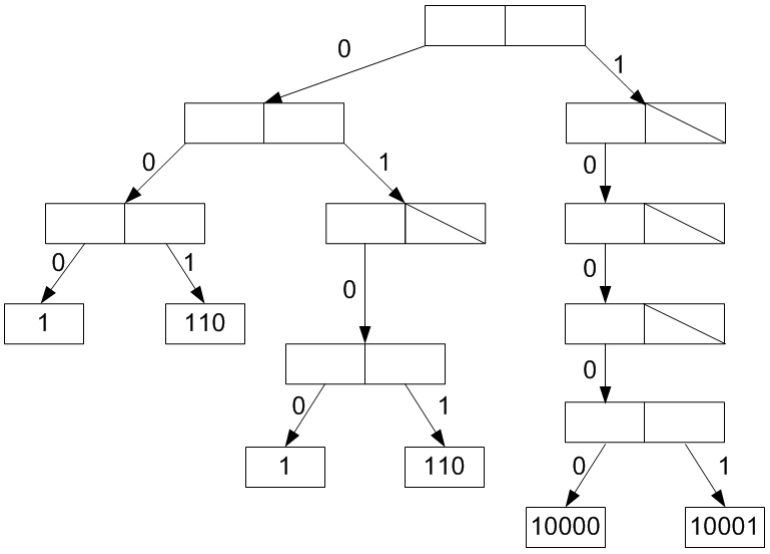
(b) Fügen Sie anschließend das Element 10000 in den Radix Trie ein und zeichnen Sie diesen neu.

**Lösung:**

Radix Trie vor dem Einfügen von 10000:




Radix Trie nach dem Einfügen von 10000:




---

# Aufgabe 5.7

**Aufgabenstellung:**

Gegeben sei ein Alphabet $\Sigma = \{'A', 'B', 'C', 'D', 'E'\}$.

(a) Zeichnen Sie einen *Indexed Trie*, der die Worte

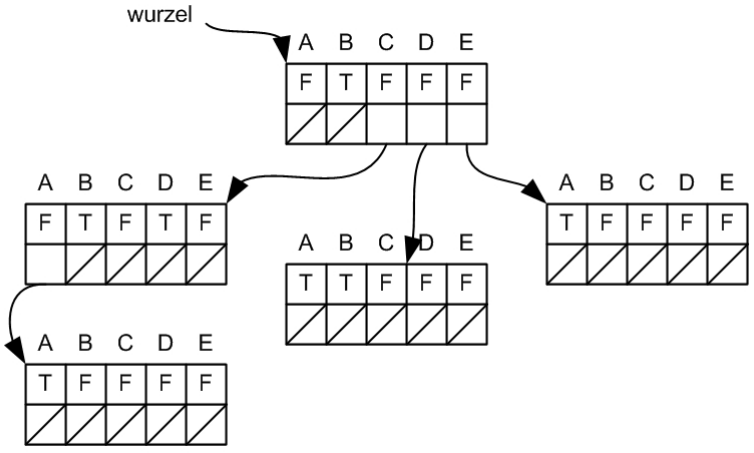
$$\{'B', 'DA', 'CAA', 'CB', 'EA', 'CD', 'DB'\}$$

beinhaltet.

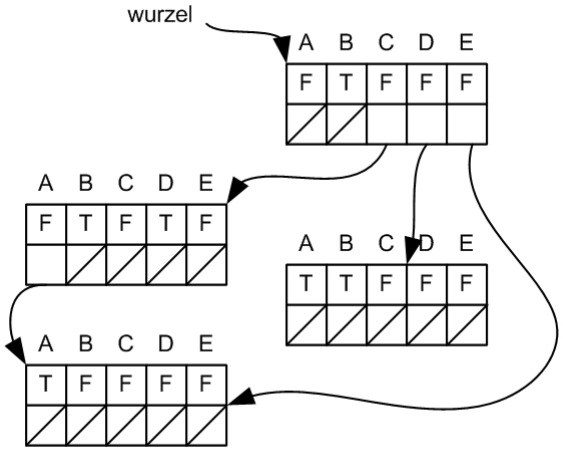
(b) Führen Sie anschließend Suffix Compression in dem von Ihnen gezeichneten Indexed Trie durch. Kennzeichnen Sie die Änderungen deutlich!

**Lösung:**

Indexed Trie:




Indexed Trie mit Suffix Compression:




---

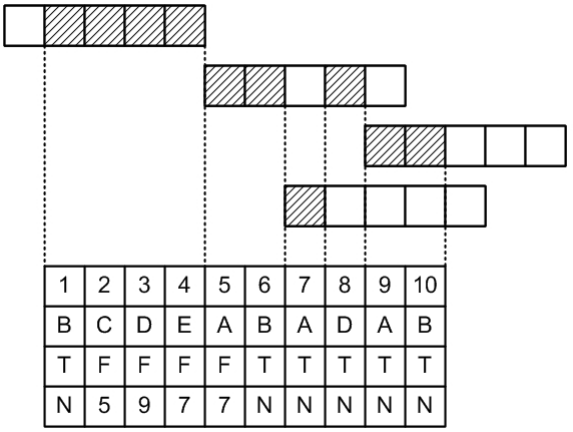
# Aufgabe 5.8

**Aufgabenstellung:**

Aus dem resultierenden Indexed Trie mit Suffix Compression aus Aufgabe 5.7 soll nun ein *Packed Trie* erstellt werden. Verwenden Sie dazu die Greedy-Heuristik aus der Vorlesung bzw. aus dem Skriptum.
Zeigen Sie dabei mit Hilfe einer kleinen Graphik (wie in der Vorlesung bzw. im Skriptum), auf welche Weise die Knoten gepackt werden, und zeichnen Sie den Packed Trie.

**Lösung:**

Die First Fit-Heuristik sortiert die Knoten nach der Anzahl der belegten Elemente und fügt diese Knoten anschließend wie folgt zusammen:



| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|
| B | C | D | E | A | B | A | D | A | B |
| T | F | F | F | F | T | T | T | T | T |
| N | 5 | 9 | 7 | 7 | N | N | N | N | N |

---

# Aufgabe 5.9

**Aufgabenstellung:**

Benutzen Sie die *Union-Find-Datenstruktur*, die Sie in der Vorlesung bzw. im Skriptum kennengelernt haben, um einen Algorithmus in kommentiertem Pseudocode anzugeben, der die Zusammenhangskomponenten eines ungerichteten Graphen $G = (V, E)$ identifiziert. Dabei soll jedem Knoten eine Zahl zugewiesen werden, wobei Knoten in der gleichen Komponente die gleiche Zahl haben müssen.

(a) Wie oft ruft Ihr Algorithmus die Funktionen *findset* und *union* für einen Graphen mit $k$ Komponenten auf?

(b) Wie ist die Laufzeit Ihres Algorithmus in $\Theta$-Notation, abhängig von $|V|$, $|E|$ und $k$?

(c) Wie verhält sich die Laufzeit Ihres Algorithmus zu der Realisierung, die DFS benutzt?

**Lösung:**

```
Komponentennummern(G) {
  for all(v in V) {
    makeset(v);
    markier[v] = 0;
  }
  for all(e(v, w) in E) {
    union(findset[v], findset[w]);
  }

  compnum = 1;
  for all(v in V) {
    if(markier[findset[v]] == 0) {
      markiert[findset[v]] = compnum;
      markiert[v] = compnum;
      compnum++;
    }
    else {
      markiert[v] = markiert[findset[v]];
    }
  }
}
```

(a) Der Algorithmus ruft die Funktion *makeset* insgesamt $2 \cdot |E| + 2 \cdot |V|$-mal auf, die Funktion *union* $|E|$-mal.

(b) Dir Laufzeit des Algorithmus kommt bei Optimierung der Union Find-Operationen mit Pfadverkürzung auf $\Theta(|E| + |V|)$.

(c) Da auch die Laufzeit des DFS-Algorithmus zur Lösung desselben Problems in $\Theta(|E| + |V|)$ liegt, sind die Laufzeiten der beiden Algorithmen praktisch identisch.

---

# Aufgabe 5.10

**Aufgabenstellung:**

Sei $G = (V, E)$ ein *gewichteter* Graph, sei $s \in V$ ein frei gewählter Startknoten. Beschreiben Sie einen möglichst effizienten Algorithmus, der die jeweils kürzesten Kantenzüge von $s$ zu allen anderen Knoten in $V$ ermittelt.
Hinweis: Suchen Sie in einschlägiger Literatur oder im Internet nach dem Algorithmus von *Dijkstra* zur Ermittlung der kürzesten Pfade (bzw. eventuell auch nach *shortest path*).
Demonstrieren Sie die Funktionsweise Ihres Algorithmus anhand des Graphen in Aufgabe 5.4 mit den dort angegebenen Kantengewichten.

**Lösung:**

```
Dijkstra(G,s) {
  for all(v in V) {
    Distance[v] = infinite;
    Predecessor[v] = NULL;
  }

  Distance[s]=0;
  Predecessor[s]=s;
  U = V;

  while(U != {}) {
    choose u from U with minimal Distance[u];
    U = U - {u};
    for all((u,v) in E with v in U) {
      if(Distance[v] + weight(u,v) < Distance[v]) {
        Distance[v] = Distance[u] + weight(u,v);
```

```
          Predecessor[v] = u;
        }
      }
    }
}
```

Nach Ablauf des Algorithmus kann für jeden Knoten bestimmt werden, welcher dessen Vorgänger auf dem kürzesten Pfad vom Startknoten zu eben diesem Knoten ist. Diese Information wird im Array **Predecessor** gespeichert, die Länge des kürzesten Pfades vom Startknoten zu jedem Knoten im Array **Distance**. Voraussetzung für das erfolgreiche Funktionieren des Algorithmus ist der Umstand, dass das Nichtexistieren einer Kante mit dem Gewicht $\infty$ bewertet wird.
Angewandt auf den Graphen aus Aufgabe 5.4 mit Startknoten 0 ergibt das die folgenden Schritte:

- **U**=**V**={0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16}
- Startknoten: 0
- Initialisierung der Arrays **Predecessor** sowie **Distance**
- Auswahl: Knoten 0, **U**={1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16}
  - **Distance**[1]=2;
    **Predecessor**[1]=0;
  - **Distance**[3]=36;
    **Predecessor**[3]=0;
  - **Distance**[4]=17;
    **Predecessor**[4]=0;
  - **Distance**[5]=23;
    **Predecessor**[5]=0;
- Auswahl: Knoten 1, U={2,3,4,5,6,7,8,9,10,11,12,13,14,15,16}
  - **Distance**[2]=11;
    **Predecessor**[2]=1;
  - **Distance**[3]=10;
    **Predecessor**[3]=1;
  - **Distance**[13]=79;
    **Predecessor**[13]=1;
  - **Distance**[15]=8;
    **Predecessor**[15]=1;
- Auswahl: Knoten 15, U={2,3,4,5,6,7,8,9,10,11,12,13,14,16}
- Auswahl: Knoten 3, U={2,4,5,6,7,8,9,10,11,12,13,14,16}
  - **Distance**[6]=25;
    **Predecessor**[6]=3;
  - **Distance**[8]=27;
    **Predecessor**[8]=3;
  - **Distance**[9]=65;
    **Predecessor**[9]=3;
  - **Distance**[11]=31;
    **Predecessor**[11]=3;
  - **Distance**[12]=28;
    **Predecessor**[12]=3;
  - **Distance**[13]=22;
    **Predecessor**[13]=3;
- Auswahl: Knoten 2, U={4,5,6,7,8,9,10,11,12,13,14,16}
  - **Distance**[16]=15;
    **Predecessor**[16]=2;

- Auswahl: Knoten 16, U={4,5,6,7,8,9,10,11,12,13,14}
  - **Distance**[7]=29;
    **Predecessor**[29]=16;
- Auswahl: Knoten 4, U={5,6,7,8,9,10,11,12,13,14}
  - **Distance**[9]=37;
    **Predecessor**[9]=4;
  - **Distance**[10]=38;
    **Predecessor**[10]=4;
- Auswahl: Knoten 13, U={5,6,7,8,9,10,11,12,14}
- Auswahl: Knoten 5, U={6,7,8,9,10,11,12,14}
  - **Distance**[14]=47;
    **Predecessor**[14]=5;
- Auswahl: Knoten 6, U={7,8,9,10,11,12,14}
- Auswahl: Knoten 8, U={7,9,10,11,12,14}
- Auswahl: Knoten 12, U={7,9,10,11,14}
- Auswahl: Knoten 7, U={9,10,11,14}
- Auswahl: Knoten 11, U={9,10,14}
- Auswahl: Knoten 9, U={10,14}
- Auswahl: Knoten 10, U={14}
- Auswahl: Knoten 14, U={}
- Ende des Algorithmus

Mit Hilfe der in den beiden Arrays **Predecessor** und **Distance** gespeicherten Informationen kann man nun zu allen Knoten die kürzesten Pfade sowie deren Gewicht rekonstruieren:

- Knoten 0: keine Kante, Gewicht: 0
- Knoten 1: 0-1, Gewicht: 2
- Knoten 2: 0-1-2, Gewicht: 11
- Knoten 3: 0-1-3, Gewicht: 10
- Knoten 4: 0-4, Gewicht: 17
- Knoten 5: 0-5, Gewicht: 23
- Knoten 6: 0-1-3-6, Gewicht 25
- Knoten 7: 0-1-2-16-14, Gewicht: 29
- Knoten 8: 0-1-3-8, Gewicht: 27
- Knoten 9: 0-4-9, Gewicht: 37
- Knoten 10: 0-4-10, Gewicht: 38
- Knoten 11: 0-1-3-11, Gewicht: 31
- Knoten 12: 0-1-3-12, Gewicht: 28
- Knoten 13: 0-1-3-13, Gewicht: 22
- Knoten 14: 0-5-14, Gewicht: 47
- Knoten 15: 0-1-6, Gewicht: 8
- Knoten 16: 0-1-2-16, Gewicht: 15