5 Praktikum: Graphen

Literaturhinweise: Folien/Videos zu Kapitel 8 / Buch: Algorithmen und Datenstrukturen Robert Sedgewick

In dieser Aufgabe ist erneut Ihre Kompetenz als Systemarchitekt bei „Data Fuse“ gefragt. Die verschiedenen Standorte sollen mit neuen Routern verbunden werden und Sie sollen testen, ob die geplanten Router und deren Verbindungen ausreichen um alle Standorte zu verbinden. Um dies zu lösen, nutzen Sie Algorithmen der Graphentheorie.

In Abbildung 1 sehen Sie ein Beispiel, wie so ein Graph aufgebaut sein könnte, die Knoten stellen die Router an den Standorten dar, die Kanten die Verbindungen mit einem Gewicht als fiktive Entfernung. Hier lässt sich schnell erkennen, dass alle Standorte verbunden sind, mit der minimalen Entfernung wird es auch hier auf den ersten Blick schon schwieriger.

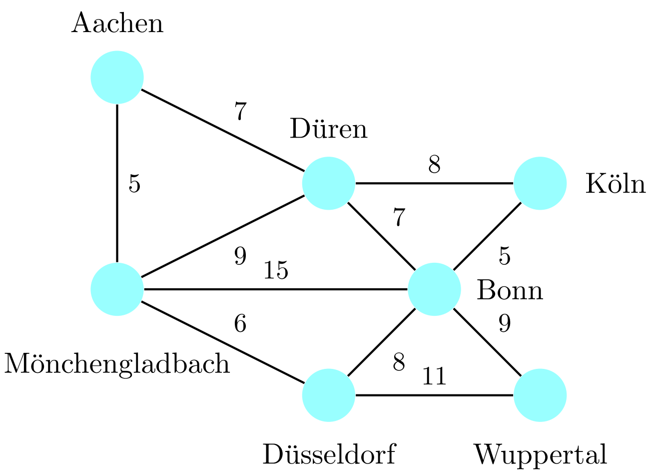


Abbildung 1: Beispiel Unternehmens Standorte mit Verbindungen

5.1 Aufgabenstellung

1. Implementieren Sie ein Programm, dass zur Verarbeitung von vorgegebenen Graphen verwendet werden soll. Legen Sie dazu eine Graphen- und Knotenklasse nach den unten aufgeführten Vorgaben (Klassen: `Edge, EdgeWeightedGraph, PrimMST, KruskalMST, DirectedEdge, EdgeWeightedDigraph und DijkstraSP`) an. Folgende Funktionalität müssen Sie hier selber implementieren:

   - Ausgabe des Graphen
   - Rekursive Tiefensuche
   - Iterative Breitensuche
   - Prim (Minimaler Spannbaum)
   - Kruskal (Minimaler Spannbaum)

- Kürzeste Wege nach Dijkstra

Es dürfen weitere Attribute und Methoden hinzugefügt werden und auch Übergabeparameter verändert werden. Begründen Sie diese Änderungen bei der Abgabe.

2. Implementieren Sie wie schon in den letzten Aufgaben ein Menü zur Auswahl für den Benutzer wie folgt:

```
Praktikum 5: Graphenalgorithmen:
1) Graph einlesen
2) Tiefensuche
3) Breitensuche
4) MST nach Prim (Eingabe: Startknoten)
5) MST nach Kruskal
6) Kürzeste Wege nach Dijkstra (Eingabe: Startknoten)
7) Ausgabe der Adjazenzliste
8) Programm beenden
Weiter mit beliebiger Eingabe ...
?>
```

Bei dem Einlesen des Graphs muss es möglich sein zwischen allen Beispielgraphen (Siehe Kapitel 5.2.1) zu wählen.

3. Für die Berechnung der Minimalen Spannbäume werden gewichtete Graphen benötigt. Verwenden Sie dazu die folgenden Klassen:

- API für eine gewichtete und ungerichtete Kante:

```
class Edge {
  private:
    int _either;     // ein Knoten der Kante
    int _other;      // der andere Knoten der Kante
    double _weight;  // Kantengewicht
    ...
  public:
    Edge(int v, int w, double weight);
    int either();                      // einer der beiden Knoten
    int other(int v);                  // der andere Knoten
    double weight();                   // Gewicht dieser Kante
    ...
};
```

- API für ungerichteten kantengewichteten Graphen

```
class EdgeWeightedGraph {
  private:
    int V;  // Anzahl Knoten von G
    int E;  // Anzahl Kanten von G
    ...
```

```
  public:
    EdgeWeightedGraph(int V);          // Leerern Graphen mit V Knoten erstellen
    EdgeWeightedGraph(std::string fn); // Graph einlesen aus Textdatei
    int getV();                        // liefert Anzahl Knoten
    int getE();                        // liefert Anzahl der Kanten
    void add(Edge e);                  // fügt Kante e dem Graphen hinzu
    std::vector<Edge> getAdj(int v);   // liefert Array der adjazenten Kanten zu v
    std::vector<Edge> edges();         // alle Kanten dieses Graphen
    bool del_Edge(Edge e);              // Löscht eine Kante, wenn Sie enthalten ist
    ...
};
```

- API für den Minimalen Spannbaum nach Prim:

```
class PrimMST {
  private:
    std::vector<bool> marked;     // MST-Knoten
    std::vector<Edge> mst;        // MST-Kanten
    std::priority_queue<Edge> pq; // Menge der Randkanten in PQ
    ...
  public:
    PrimMST(EdgeWeightedGraph G);
    void visit(EdgeWeightedGraph G, int v);
    std::vector<Edge> edges();    // liefert Kanten des MST
    double weight();              // berechnet die Gesamtkosten des MST
    ...
};
```

- API für den Minimalen Spannbaum nach Kruskal:

```
class KruskalMST {
  private:
    std::vector<Edge> mst;      // MST-Kanten
    std::vector<int> treeID;    // BaumId zu jedem Knoten
    ...
  public:
    KruskalMST(EdgeWeightedGraph G); // Konstruktor
    std::vector<Edge> edges();  // liefert Kanten des MST
    double weight();            // berechnet Gesamtkosten des MST
    ...
}
```

Die Grundgerüste dieser Klassen werden als Vorlage für Sie auf Illias bereit gestellt.

4. Implementieren Sie eine Methode zur **Ausgabe des Graphen auf der Konsole als Adjazenzliste**, wie im Beispiel unten gezeigt.

```
0  -> 1 [7]  -> 3 [5]
1  -> 0 [7]  -> 2 [8]  -> 3 [9]  -> 4 [7]
2  -> 1 [8]  -> 4 [5]
3  -> 0 [5]  -> 1 [9]  -> 4 [15]  -> 5 [6]
4  -> 1 [7]  -> 2 [5]  -> 3 [15]  -> 5 [8]  -> 6 [9]
5  -> 3 [6]  -> 4 [8]  -> 6 [11]
6  -> 4 [9]  -> 5 [11]
```

Hinweis: Beispiel erste Zeile: Knoten 0 hat eine Kante zu Knoten 1 mit Kosten 7 und zu Knoten 3 mit Kosten 5, Für die anderen Zeilen gilt entsprechendes. Die Gewichte stehen in den eckigen Klammern.

5. Überprüfen Sie mittels **Tiefen- oder Breitensuche**, ob alle Knoten verbunden sind, also alle Standorte miteinander kommunizieren könnten.

   Implementieren Sie dazu die modifizierte rekursive Tiefensuche nach Folie 54 und die iterative Breitensuche nach Folie 66 der Vorlesung Kapitel 8 Graphenalgorithmen Teil 1. Verwenden Sie `marked` und `edgeTo`, um festzustellen ob der Graph zusammenhängend ist und in welcher Reihenfolge die Knoten besucht wurden. Geben Sie das Resultat auf der Konsole wie im Beispiel unten aus.

```
Tiefensuche (Depth-First-Search (DFS)) - Startknoten: 0
Besuchsreihenfolge:
0 -> 1 -> 2 -> 4 -> 3 -> 5 -> 6

EdgeTo_Array:
0 -> -1 (Startknoten)
1 ->  0
2 ->  1
.....

Marked-Array:
0 -> true (Startknoten)
1 -> true
.....

Graph ist zusammenhängend

Breitensuche (Breadth-First-Search (BFS)) - Startknoten: 0
Besuchsreihenfolge:
0 -> 1 -> 3 -> 2 -> 4 -> 5 -> 6
```

```

EdgeTo_Array:
0 -> -1 (Startknoten)
1 ->  0
2 ->  0
.....

Marked-Array:
0 -> true (Startknoten)
1 -> true
.....

Graph ist zusammenhängend
```

6. Berechnen Sie mittels **Prim und Kruskal den minimalen Spannbaums**. Geben Sie alle Kanten des Minimalen Spannbaums und die Gesamtkosten für beide Algorithmen aus und vergleichen Sie diese. Prüfen Sie durch Zeichnen der Bäume, ob ihr Ergebnis stimmt (Zur Überprüfung eignet sich folgende Website: `https://graphonline.ru/en/`).

   Die Ausgabe sollte in Anlehnung an das unten stehende Beispiel erfolgen:

```
Minimaler Spannbaum (MST) nach Prim:
Gewicht: 39
Adjazenzliste:
 0 ->  3 [ 5] ->  1 [ 7]
 1 ->  4 [ 7]
 2 ->  4 [ 5]
 3 ->  5 [ 6]
 4 ->  6 [ 9]
```

   Hinweis: Da der Graph keine gerichteten Kanten hat, sind zusätzlich auch noch alle Kanten mit anderer Richtung zulässig.

7. Berechnen Sie mittels **Dijkstra** die **kürzesten Wege** zu einem von Ihnen definierten Startknoten. Berechnen Sie den kürzesten Weg und geben Sie den Pfad beginnend vom Startknoten auf der Konsole aus - dabei müssen auch die Kantengewichte ausgegeben werden. Verwenden Sie die unten vorgegebenen Klassen als Basis.

   - API für eine gerichtete Kante

```cpp
class DirectedEdge {
  private:
    double _weight; // Gewicht der Kante
    int _from;      // Index des Startknoten
    int _to;        // Index des Endknoten
    ...
```

```
  public:
    DirectedEdge(double weight, int from, int to);
    int from();      // liefert den Startknoten
    int to();        // liefert den Endknoten
    double weight(); // Gewicht der Kante
    ...
};
```

- API für einen gerichteten Digraphen

```
class EdgeWeightedDigraph {
  private:
    int V;  // Anzahl Knoten von G
    int E;  // Anzahl Kanten von G
    ...
  public:
    EdgeWeightedDigraph(int V);             // Leeren Digraphen mit V Knoten erstellen
    EdgeWeightedDigraph(std::string fn);    // Graph einlesen aus Textdatei
    void add(DirectedEdge e);               // gerichtete Kante hinzufügen
    int getV();                             // liefert Anzahl Knoten
    int getE();                             // liefert Anzahl der Kanten
    std::vector<DirectedEdge> getAdj(int v);// liefert Array der adjazenten Kanten zu v
    std::vector<DirectedEdge> edges();      // alle Kanten dieses Graphen
    bool del_Edge(DirectedEdge e);          // Löscht eine Kante, wenn Sie enthalten ist
};
```

- API für die Kürzesten Wege nach Dijkstra:

```
class DijkstraSP {
  private:
    std::map<int, DirectedEdge> edgeTo;        // Map mit Kanten für kürzeste Wege
    std::vector<double> distToVect;            // Distanzvektor
    Utils::PriorityQueue<int> pq;              // Prioritätswarteschlange (gegeben)
    void relax(EdgeWeightedDigraph G, int v);  // Relaxation
    ...
  public:
    DijkstraSP(EdgeWeightedDigraph G, int s);
    double distTo(int v);                     // Abstände vom Startvertex zu v
    bool hasPathTo(int v);                    // Überprüft die existens eines Pfades
    std::vector<DirectedEdge> pathTo(int v);  // Kanten des kürzsesten Weges
  ...
};
```

- Beispielausgabe für Dijkstra

```
Gewichtete Kanten des Graphen
0 ->  1 [ 7] ->  3 [ 5]
1 ->  2 [ 8] ->  3 [ 9] ->  4 [ 7]
2 ->  4 [ 5]
3 ->  4 [15] ->  5 [ 6]
4 ->  5 [ 8] ->  6 [ 9]
5 ->  6 [11]
6
```

```

Kürzester Weg (Dijkstra):
Start:  0
Ziel:   6
Pfad:   0 [5] -> 3 [6] -> 5 [11] -> 6
Kosten: 22
```

5.2 Lösungshinweise

In den nächsten Abschnitten folgen allgemeine Hinweise zu den Algorithmen, die Ihnen die Programmierung erleichtern sollen. Beachten Sie, dass hier nur Hinweise gegeben werden. Gegebene Beispiele müssen erweitert oder überarbeitet und können nicht so übernommen werden.

5.2.1 Beispielgraphen

Es stehen Ihnen drei Graphen als Beispiel zur Verfügung. Alle Graphen stehen als einfache Textdatei (graph*.txt) zur Verfügung. Sie finden den Graph dort als Kantenliste aufgebaut. Die erste Zeile gibt die Knotenzahl V und die zweite Zeile die Anzahl der Kanten E an. In allen folgenden Zeilen sind die Kanten als Tripel in folgender Form aufgeführt: Startknoten $\rightarrow$ Endknoten $\rightarrow$ Gewicht, wobei die Knoten mit $0 \dots (V-1)$ nummeriert sind. Die Graphen sind ungerichtet und gewichtet.
Aufgabe zum Praktikum: Zeichnen Sie die Graphen, die als Adjazenzlisten auf Illias vorgegeben werden auf, damit die Algorithmen am Praktikum-Termin so überprüft werden können.

5.2.2 Hinweise Unittests

In dieser Aufgabe helfen Ihnen die Unittests wieder bei der Lösung. Für Tiefen- und Breitensuche wird erwartet, dass Ihr Programm `false` zurückliefert, wenn von dem Startknoten nicht alle Knoten erreicht werden konnten. Es soll hingegen `true` liefern, wenn alle Knoten besucht wurden.

Bei Prim und Kruskal werden die Kosten und Kanten des minimalen Spannbaumes geprüft. Für Prim sind diese bei den ersten beiden Graphen mit beliebigen Startknoten immer gleich. Graph 3 ist nicht zusammenhängend und daher sollte Ihr Algorithmus abhängig vom Startknoten verschiedene Ergebnisse liefern. Sorgen Sie dafür, dass Ihr Programm trotzdem zum Ende kommt und die unvollständige Anzahl an Kanten im MST akzeptiert.

Für Kruskal gilt im Prinzip genau das gleiche, nur dass es hier keinen Startknoten gibt. Kruskal liefert bei den ersten beiden Graphen ein eindeutiges Ergebnis. Bei Graph 3, soll er ebenfalls zum Ende kommen und die Summe zurück geben, auch wenn damit nicht alle Knoten verbunden werden konnten.

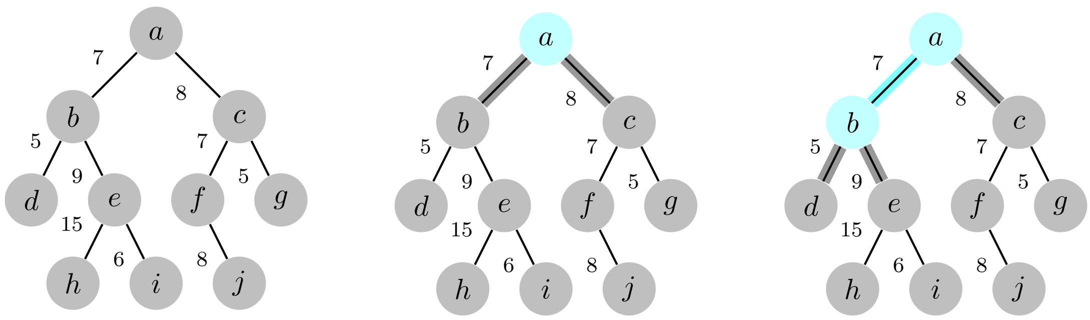
5.2.3 Tiefensuche

Die Tiefensuche wurde Ihnen in der Vorlesung vorgestellt. Sie sucht zunächst solange in die Tiefe, bis kein weiterer Knoten als Kind mehr folgt und geht dann wieder eine Ebene höher. Verwenden Sie den modifizierten Algorithmus 1 mit integrierter Pfadsuche für die rekursive Tiefensuche (s. Folie 54). Bei einer Tiefensuche werden alle erreichten Knoten als besucht markiert in dem `marked`-Array vom Datentyp *bool* und im `edgeTo`-Array werden alle Kanten eingetragen, die zur Traversierung beigetragen haben. Gibt es Knoten, die nicht durch eine einmalige Tiefensuche erreicht werden können, ist der Graph nicht zusammenhängend. Dies kann im `marked`-Array erkannt werden, falls Knoten nach einmaliger Tiefensuche noch nicht besucht wurden.

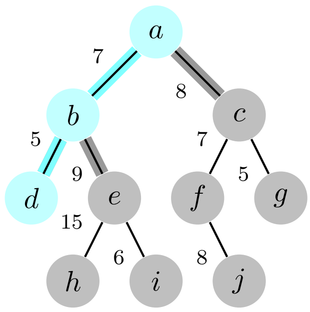
Algorithm 1: Modifizierte Rekursive Tiefensuche

$marked[0..|V|-1] = false$
`/* edgeTo speichert letzten Knoten auf Pfad zu diesem Knoten */`
$edgeTo[0..|V|-1] = -1$
$s = v_0$ `/* s ist Startknoten */`
DFS(G,s) `/* Starten der Pfadsuche */`
Function *DFS*(G, v)
    $marked[v] = true$
    for $\forall w \in G.adj(v)$ **do**
        if $marked(w) == false$ **then**
            $edgeTo[w] = v$
            $DFS(G, w)$
    end
end

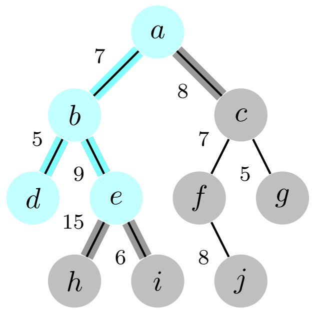
Die folgenden Grafiken zeigen die Tiefensuche exemplarisch an einem Baum, für einen Graphen funktioniert diese auf die gleiche Weise. Es muss nur berücksichtigt werden, ob ein Knoten bereits besucht wurde.



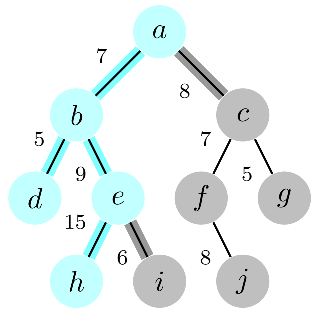
Original   Schritt 1   Schritt 2



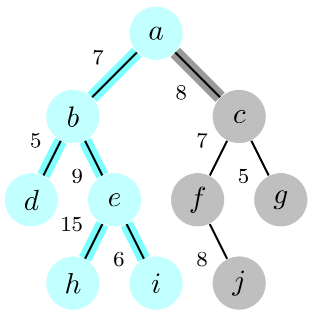
Schritt 3



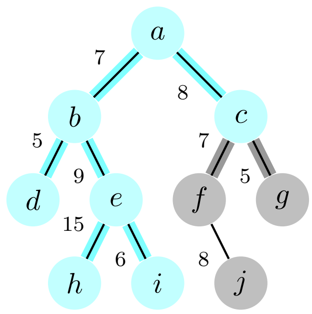
Schritt 4



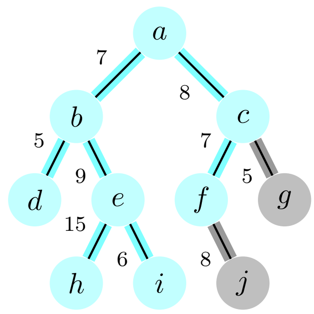
Schritt 5



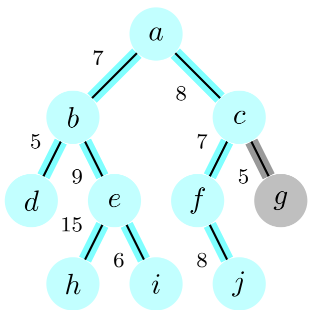
Schritt 6



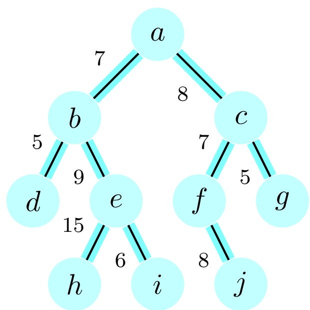
Schritt 7



Schritt 8



Schritt 9



Endergebnis

An dem Beispiel wird nun zusätzlich gezeigt, wie sich der Zustand des EdgeTo-Arrays und des Marked-Arrays mit jedem Schritt verändert. Das Marked-Array speichert, ob ein Knoten über einen anderen Knoten bereits besucht wurde. Die blaue Schrift gibt an, wann der jeweilige Knoten in welcher Iteration besucht wurde.

Knoten										
Schritt	a	b	c	d	e	f	g	h	i	j
0	*false*	*false*	*false*	*false*	*false*	*false*	*false*	*false*	*false*	*false*
1	*true*	*false*	*false*	*false*	*false*	*false*	*false*	*false*	*false*	*false*
2	*true*	*true*	*false*	*false*	*false*	*false*	*false*	*false*	*false*	*false*
3	*true*	*true*	*false*	*true*	*false*	*false*	*false*	*false*	*false*	*false*
4	*true*	*true*	*false*	*true*	*true*	*false*	*false*	*false*	*false*	*false*
5	*true*	*true*	*false*	*true*	*true*	*false*	*false*	*true*	*false*	*false*
6	*true*	*true*	*false*	*true*	*true*	*false*	*false*	*true*	*true*	*false*
7	*true*	*true*	*true*	*true*	*true*	*false*	*false*	*true*	*true*	*false*
8	*true*	*true*	*true*	*true*	*true*	*true*	*false*	*true*	*true*	*false*
9	*true*	*true*	*true*	*true*	*true*	*true*	*false*	*true*	*true*	*true*
10	*true*	*true*	*true*	*true*	*true*	*true*	*true*	*true*	*true*	*true*

Abbildung 2: Zustand des Marked-Arrays nach jeder Iteration

Zusätzlich wird nun ähnlich wie beim Marked-Array, das EdgeTo-Array angegeben. Das EdgeTo-Array gibt an, über welchen Knoten man zu einem anderen traversieren kann. Dazu wird zunächst eine Abbildung von den Knoten in die natürliche Zahlen definiert.

a	b	c	d	e	f	g	h	i	j
0	1	2	3	4	5	6	7	8	9

Abbildung 3: Abbildung Namen ⇔ natürliche Zahlen

Nun wird über den Algorithmus bestimmt, über welchen Knoten man zu einen anderen kommt. Da der Knoten *a(0)* der Startknoten ist, bleibt dieser auf dem Wert -1.

Zum Schluss muss die Abbildung wieder rückgängig gemacht werden:

5.2.4 Breitensuche

Im Gegensatz zur Tiefensuche wird hier zunächst in der Breite gesucht. Das bedeutet zu einem Knoten werden zunächst alle Kinder untersucht, bevor man die Kindes-Kinder betrachtet.

Auch hier können Sie sich wieder am Pseudocode aus der Vorlesung in Algorithmus 2 orientieren.

Knoten										
Schritt	a (0)	b (1)	c (2)	d (3)	e (4)	f (5)	g (6)	h (7)	i (8)	j (9)
0	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
1	-1	0	-1	-1	-1	-1	-1	-1	-1	-1
2	-1	0	-1	1	-1	-1	-1	-1	-1	-1
3	-1	0	-1	1	1	-1	-1	-1	-1	-1
4	-1	0	-1	1	1	-1	-1	4	-1	-1
5	-1	0	-1	1	1	-1	-1	4	4	-1
6	-1	0	0	1	1	-1	-1	4	4	-1
7	-1	0	0	1	1	2	-1	4	4	-1
8	-1	0	0	1	1	2	-1	4	4	5
9	-1	0	0	1	1	2	2	4	4	5

Abbildung 4: Zustand des EdgeTo-Arrays nach jeder Iteration

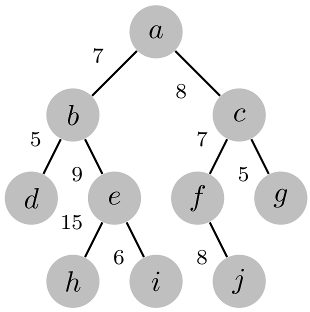
a (0)	b (1)	c (2)	d (3)	e (4)	f (5)	g (6)	h (7)	i (8)	j (9)
-1	*0*	*0*	*1*	*1*	*2*	*2*	*4*	*4*	*5*
(Start)	*a*	*a*	*b*	*b*	*c*	*c*	*e*	*e*	*f*

Abbildung 5: Zustand des EdgeTo-Arrays nach der Rekursiven Tiefensuche

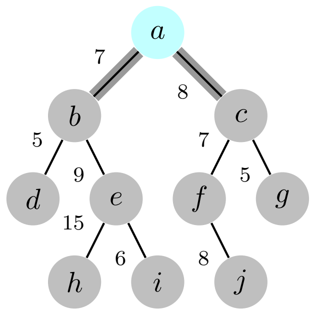
Algorithm 2: Iterative Breitensuche

```
marked[0..|V| − 1] = false
edgeTo[0..|V| − 1] = −1
Function BFS_it(G, s)
    queue q
    q.enqueue(s)
    while (!q.isEmpty()) do
        v = q.dequeue()
        if (marked[v]) then
            continue

        marked[v] = true
        for ∀w ∈ G.adj(v) do
            if (marked[w] == false) then
                edgeTo[w] = v
                q.enqueue(w)
        end
    end
end
```

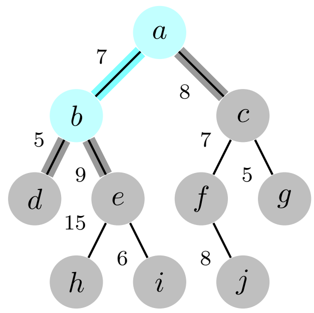
Analog zur Tiefensuche ist im folgenden auch die Breitensuche grafisch dargestellt. Auch hier kann das Prinzip einfach auf einen Graphen übertragen werden.



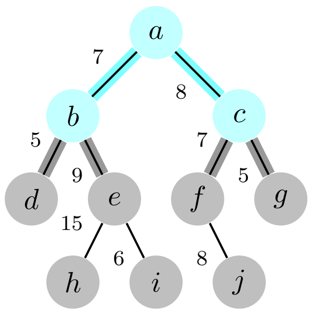
Original



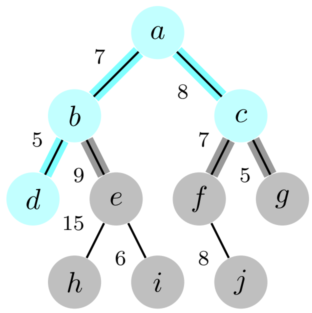
Schritt 1



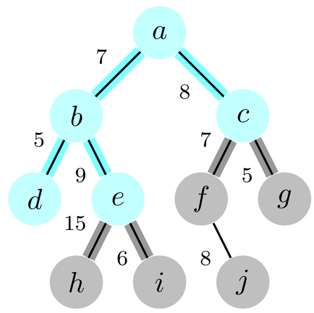
Schritt 2



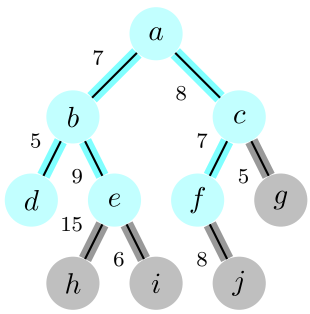
Schritt 3



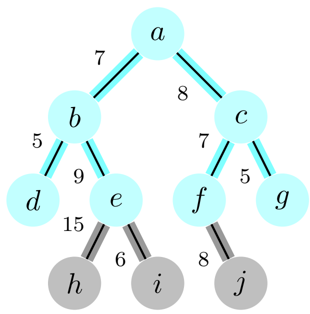
Schritt 4



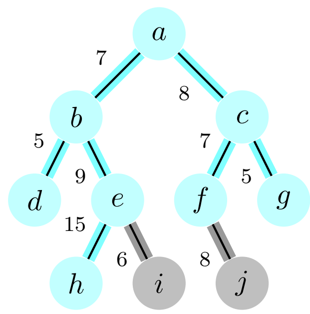
Schritt 5



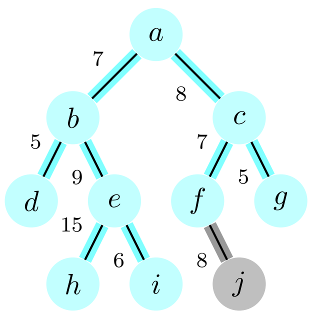
Schritt 6



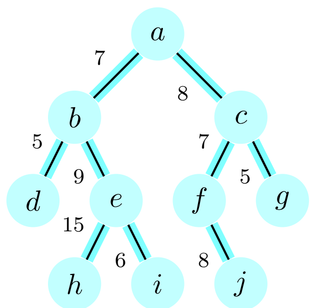
Schritt 7



Schritt 8



Schritt 9



Endergebnis

5.2.5 Prim

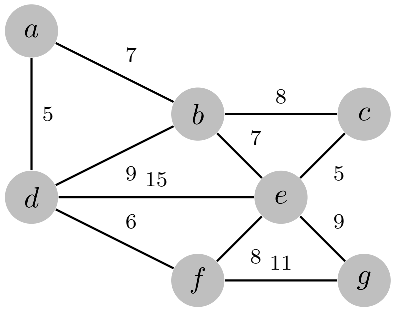
Der Prim Algorithmus dient der Erstellung eines Minimalen Spannbaumes (Minimal-Spanning-Tree MST). Algorithmus 3 beschreibt ihn als Pseudocode. Als alternative Ressource wird auf die Vorlesungsunterlagen Kapitel 8 Teil 2, Folie 14 bis 19 verwiesen.

Algorithm 3: Prim Algorithmus - Lazy-Version (s. Sedgewick S.662)

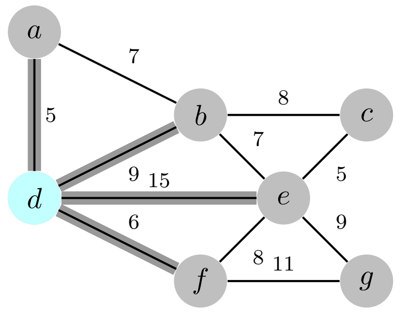
input : Graph G, Startknoten s
output : MST zu Graph G
parameter: $marked[0..|V|-1] = false$: markiert $\forall\ u \in G$ ob u bereits traversiert/besucht wurde
$Adj[e]$: Adjazenzliste von Kanten [s. Vorlesungsunterlagen Kapitel 8 Teil 2 Folie 13]
$wert[e]$: Wert der Kante e
$PQ \leftarrow$ Prioritätswarteschlange mit Kanten

```
Markiere den Startknoten s als besucht
marked[s] = true

Pushe alle adjazenten Kanten e zum Startknoten s in die PQ
for e ∈ Adj[s] do
    PQ.push(e)

while PQ nicht leer do
    Hole Kante e aus PQ mit min(wert[e])
    MST.enqueue(e) füge Kante e zum MST hinzu, wenn Ziel- oder Startknoten noch nicht
    besucht wurde.
    Sei v der nicht besuchte Knoten von der Kante e
    marked[v] = true markiere Knoten v als besucht

    Lege alle Kanten e von v, die zu unmarkierten Knoten führen in die PQ
    for e ∈ Adj[v] do
        u sei der nicht besuchte Knoten der Kante e = (u, v)
        if (!marked[u]) then
            PQ.push(e)
```

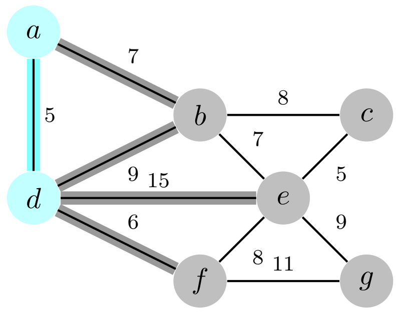
Ein Beispiel mit dem folgenden Graph beschreibt den Algorithmus wobei als Startknoten Knoten d verwendet wurde. Die grau eingefärbten Kanten sind die zur Auswahl stehenden und die mint gefärbten Kanten die dem MST hinzugefügten Kanten.



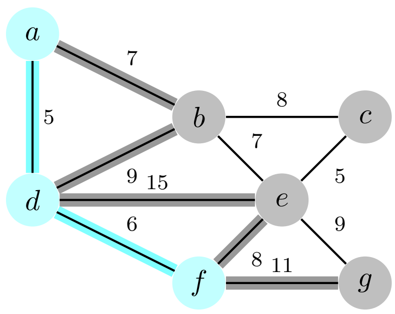
Originalgraph



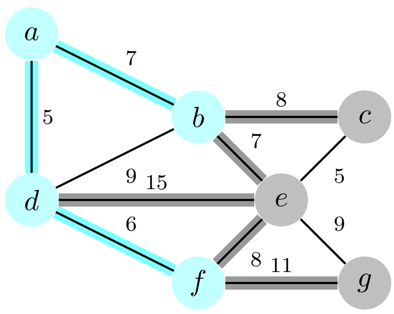
Startknoten d, erste Kantenauswahl



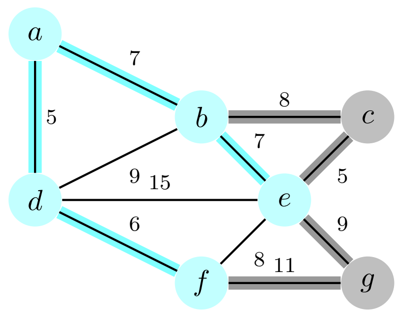
a hinzugefügt, neue Kantenauswahl



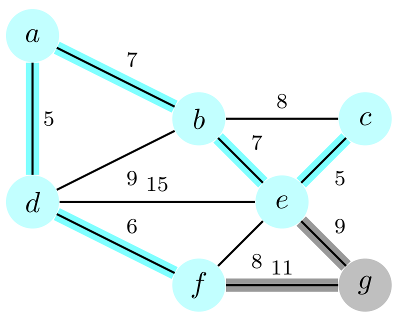
f hinzugefügt, neue Kantenauswahl



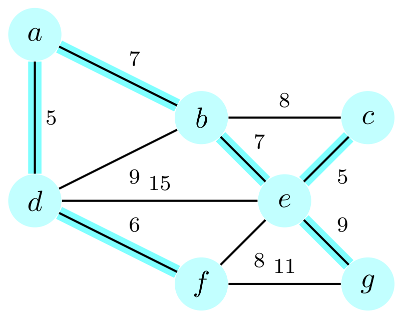
b hinzugefügt, neue Kantenauswahl



e hinzugefügt, neue Kantenauswahl



c hinzugefügt, neue Kantenauswahl



g hinzugefügt, MST vollständig

Der vollständige MST verbindet nun alle Knoten kostenminimal von Knoten d ausgehend. Die Kosten dieses MST betragen 39.

5.2.6 Kruskal

Dieser Algorithmus ist ebenfalls zum Finden eines MST, arbeitet aber etwas anders als Prim. In diesem Fall wird kein Startknoten gewählt, sondern alle Kanten zu Beginn der Größe nach sortiert und immer die (dem Kantengewicht nach) kleinste Kante als nächste gewählt, sofern mindestens einer der Knoten noch nicht mit dem MST verbunden wurde. Auch hierzu sehen Sie in Algorithmus 4 den Pseudocode, den Sie zur Implementierung verwenden können. Als alternative Ressource wird auf die Vorlesungsunterlagen Kapitel 8 Teil 2, Folie 21 bis 25 verwiesen.

Algorithm 4: Kruskal Algorithmus (s. Sedgewick S. 670)

input : Graph G
output : MST zu Graph G
parameter: $treeID[0..|V|-1]$ BaumID zu einem Knoten
$PQ \leftarrow$ Prioritätswarteschlange mit Kanten aufsteigend sortiert nach ihren Gewichten
$E \leftarrow$ alle Kanten des Graphen G
$|V| \leftarrow$ Anzahl der Knoten des Graphen G

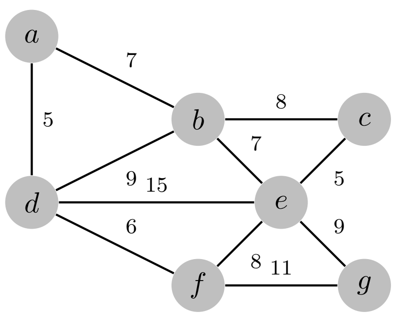
Füge alle Kanten $e \in E$ aufsteigend sortiert in die PQ ein:
for $e \in E$ **do**
  $PQ.enqueue(e)$

Markiere zu Beginn jeden Knoten als einzelnen Baum:
for $i \in [0, 1, \ldots, |V|-1]$ **do**
  $treeID[i] = i$

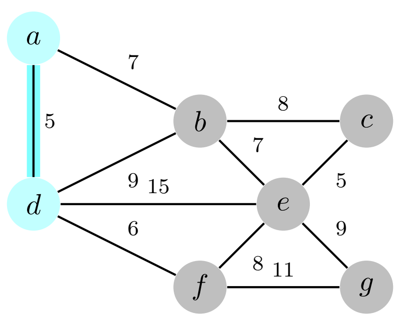
Entnehme Kante $e \in E$ aus PQ und füge diese zum MST hinzu, wenn die Kante zwei verschiedene Teilbäume miteinander verbindet:
while PQ *nicht leer und* $MST.size() < |V|-1$ **do**
  $e = PQ.dequeue()$ hole kürzeste Kante e=(u,v) aus PQ
  if $(treeID[u] \neq treeID[v])$ **then**
    Füge Kante $e = (u, v)$ zum MST hinzu, die 2 verschiedene Bäume verbindet:
    $MST.enqueue(e)$
    Vereinige die BaumID für beide Bäume:
    $treeID.union(u, v)$

Die folgenden Abbildungen zeigen am gleichen Beispiel wie bei Prim den Ablauf des Algorithmus. Die Kanten wurden initial wie folgt sortiert:

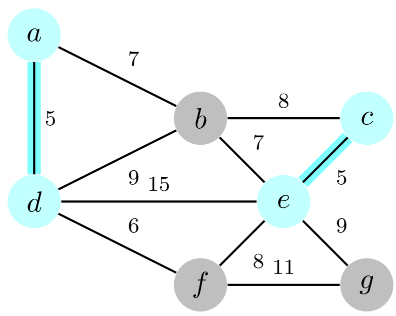
$(a, d, 5)$ $(c, e, 5)$ $(d, f, 6)$ $(a, b, 7)$ $(b, e, 7)$ $(b, c, 8)$ $(e, f, 8)$ $(b, d, 9)$ $(e, g, 9)$ $(f, g, 11)$ $(d, e, 15)$



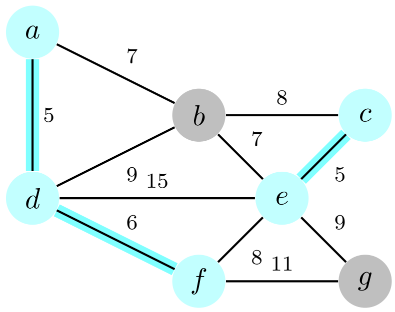
Originalgraph



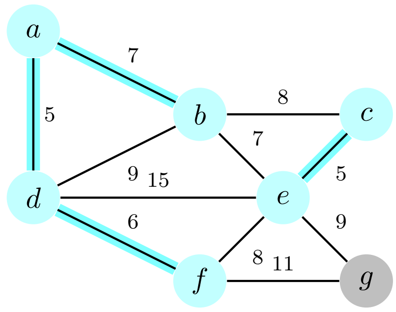
Kante $(a, d, 5)$ hinzugefügt



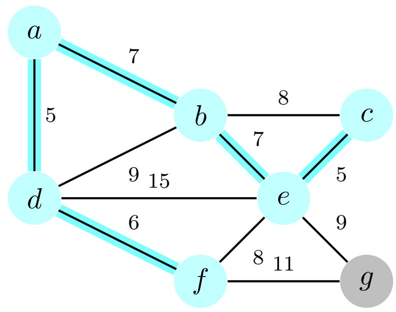
Kante $(c, e, 5)$ hinzugefügt



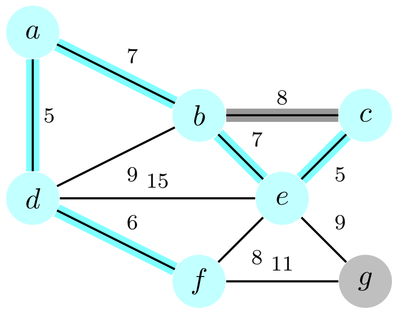
Kante $(d, f, 6)$ hinzugefügt



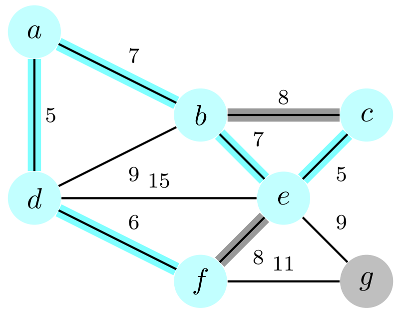
Kante $(a, b, 7)$ hinzugefügt



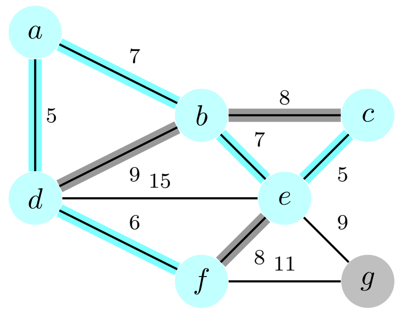
Kante $(b, e, 7)$ hinzugefügt



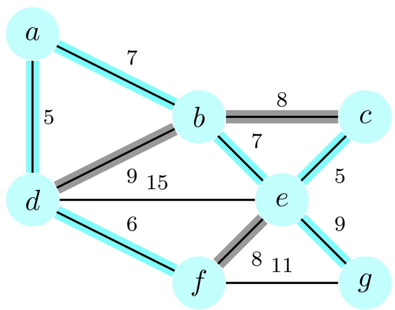
Kante $(b, c, 8)$ verwerfen



Kante $(e, f, 8)$ verwerfen



Kante $(b, d, 9)$ verwerfen



Kante $(e, g, 9)$ hinzufügen

Nachdem Knoten g hinzugefügt wurde ist der MST vollständig und es müssen keine weiteren Kanten mehr betrachtet werden. Die Gesamtkosten berechnen sich durch die Summe aller verwendeten Kanten des MST. In diesem Fall wurden exakt die gleichen Kanten wie beim Prim Algorithmus verwendet und somit beträgt die Summe des MST auch hier 39.

5.2.7 Dijkstra

Der Dijkstra-Algorithmus liefert zu einem Startknoten ausgehend die kürzesten Wege zu jedem Knoten in einem zusammenhängenden Digraph. Ein Digraph ist ein gerichteter und gewichteter Graph. Der in der Vorlesung vorgestellte Algorithmus basiert auf der Version von Sedgewick, Algorithmus 4.9, S. 697. In jedem Durchlauf wird diejenige Kante hinzugefügt, die von einem Baumknoten zu einem Nicht-Baumknoten verläuft und deren Ziel w dem Startknoten s am nächsten ist.

Folgende Datenstrukturen werden für die Implementierung benötigt:

- In einem Array `edgeTo` seien alle gerichteten Kanten des Graphen gespeichert.
- in einem Array `distTo` werden die Gesamtkosten der Wege vom Startknoten bis zu einem anderen Knoten des Graphen gespeichert.
- PQ sei eine Prioritätswarteschlage, die zum bestehenden Kürzeste-Pfade-Baum die Wegekosten zu adjazenten noch nicht besuchten Knoten speichert

Weitere Informationen zum Dijkstra-Algorithmus entnehmen Sie bitte aus den Vorlesungsunterlagen Kapitel 8 Teil 2, Folie 43 bis 52

Algorithm 5: Djikstra Algorithmus Pseudo-Code (s. abgewandelt in Sedgewick S. 697)

input : Graph G, Startknoten s

output : Kürzeste-Pfade Graph $G's$ zu Graph G und Startknoten s

parameter: $edgeTo[0..|V|-1]$ markiert zu einem Knoten s den Vorgänger Knoten s'

$distTo[0..|V|-1]$ markiert die Weg-Kosten vom Startknoten s zu jeden anderen Knoten

$PQ \leftarrow$ Prioritätswarteschlange mit Knoten aufsteigend nach Kosten sortiert

$E \leftarrow$ alle Kanten des Graphen G

$|V| \leftarrow$ Anzahl der Knoten des Graphen G

Initialisiere edgeTo-Array und distTo-Array, zunächst ist noch kein kürzester-Pfad bekannt, also ist die Entfernung unendlich groß und kein Knoten hat einen Vorgängerknoten:

for $v \in V$ **do**
    $edgeTo[v] = -1$
    $distTo[v] = \infty$

Distanz zum Startknoten gleich 0 setzten:

$distTo[s] = 0.0$

$PQ.enqueue(s, 0.0);$

while *PQ nicht leer* **do**
    $v = PQ.dequeue()$ hole den nächsten unbesuchten Knoten mit minimalen Kosten aus PQ
    for $\forall e = (u, v) \in E$ *mit Startknoten/Zielknoten* v **do**
        if $distTo[u] > distTo[v] + e.weight()$ **then**
            $distTo[w] = distTo[v] + e.weight()$
            $edgeTo[w] = v$
            if $PQ.contains(w)$ **then**
                $PQ.update(w, distTo[w])$
            else
                $PQ.enqueue(w, distTo[w])$