Algorithmen für gewichtete Graphen*

Hans Jürgen Ohlbach

23. März 2018

Keywords: Gewichtete Graphen, Kürzeste Wege-Problem, Dijkstra-Algorithmus, A*-Algorithmus, Spannbäume, Prim und Kruskal-Algorithmus, Network Flows, Dinic's Algorithmus, das Handlungsreisenden-Problem

Empfohlene Vorkenntnisse: Java ist hilfreich, aber nicht notwendig

Inhaltsverzeichnis

*Dieser Text ist Teil einer Sammlung von Miniskripten zur Einführung in die Informatik. Er ist in erster Linien für Nichtinformatiker gedacht, kann aber natürlich auch als erste Einführung für Informatiker nützlich sein.

1 Einleitung

Viele Probleme aus Anwendungen können mit gewichteten Graphen modelliert werden. Dazu gehören Wegesuche in Straßennetzen, kürzeste Leitungen für z.B. Telefon-, Stromnetze oder Wasserleitungen, Maximierung der Transportleistung durch Netzwerke, sogar auch die Erzeugung von Labyrinthen, und auch kürzeste Wege für Lötautomaten für Platinen. Für all diese Probleme eignen sich gewichtete Graphen, sowie Algorithmen, die auf diesen Graphen arbeiten. In diesem Miniskript werden gewichtete Graphen eingeführt, und dann fünf der prominentesten Problem, die mit diesen Graphen gelöst werden können, diskutiert. Es wird soweit wie möglich auf Formalismen verzichtet, und die wichtigen Ideen an Beispielen illustriert.

2 Gewichtete Graphen

Ein *Graph* besteht aus *Knoten* (engl. node oder vertex) und *Kanten* (engl. edge). Jede Kante verbindet genau zwei Knoten. Wenn die Kanten eine Richtung haben, spricht man von einem *gerichteten Graphen*. Haben die Kanten sog. Gewichte, spricht man von einem *gewichteten Graphen*..

Beispiele:

Das Straßennetz ist ein gewichteter Graph. Die Kreuzungen sind die Knoten, und die Straßen dazwischen sind die Kanten. Die Länge der Straßen zwischen den Kreuzungen sind die Gewichte. Betrachtet man jede Fahrbahn als eigene Kante, dann ist der Graph auch gerichtet. Die Fahrtrichtung bestimmt die Richtung der Kanten.

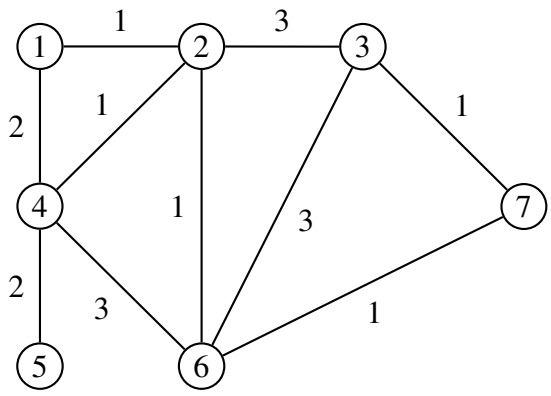
Das Internet ist auch ein gewichteter Graph. Die Router und Rechner sind die Knoten. Die Leitungen und Funkverbindungen sind die Kanten. Die Übertragungskapazität in Bits/Sekunde sind die Gewichte.

Das Telefonnetz bildet einen Graphen, mit den Vermittlungsstellen als Knoten, und den Leitungen als Kanten. Die Übertragungskapazität, d.h. wieviele Gespräche gleichzeitig übertragen werden können, bildet die Gewichte.

Das Schienennetz bildet einen Graphen, mit den Weichen als Knoten und den Schienen als Kanten. Gewichte können die Entfernungen zwischen den Weichen sein, oder auch die Transportkapazität bei Schienennetzen für den Güterverkehr.

Eine zu verlötende Platine bildet einen Graphen, mit den Lötstellen als Knoten, und den Wegen zwischen den Lötstellen als Kanten.

Das folgende Bild zeigt einen gewichteten Graphen.



Einschränkung: In diesem Miniskript gehen wir davon aus, dass die Kantengewichte *nicht negativ* sind. Anwendungen mit Graphen mit negativen Kantengewichten gibt es wohl auch, und die Algorithmen dazu wurden auch untersucht, werden hier aber nicht behandelt.

3 Kürzeste Wege in gewichteten Graphen

Das erste Problem, dem wir uns zuwenden, ist das Problem, zwischen zwei Knoten eines gewichteten Graphen einen kürzesten Weg zu finden. Die Weglänge ist dabei die Summe der Gewichte, entlang der Kanten auf diesem Weg.

Im obigen Graphen, z.B. läuft der kürzeste Weg zwischen Knoten 5 und Knoten 7 über die Knoten 5,4,2,6,7. Die Weglänge ist dabei 5. Der Weg über die Knoten 5,4,6,7 hat zwar weniger Kanten, aber mehr Gesamtgewicht, nämlich 6.

3.1 Der Dijkstra-Algorithmus

Der bekannteste Algorithmus, der dieses Problem löst, ist der nach seinem Erfinder Edsger W. Dijkstra benannte Dijkstra-Algorithmus.

Die Beschreibung dieses Algorithmus hängt etwas davon ab, ob man Zusatzinformationen direkt an Knoten und Kanten anbringen kann, oder nicht. Im folgenden nehmen wir an, dass das nicht geht.

Der Kern des Algorithmus funktioniert folgendermaßen:

- Erzeuge eine zunächst leere aufsteigend sortierte Liste (Prioritätenschlange, oder priority queue) von Knoten, wobei das Sortierkriterium die momentane Gesamtwegelänge vom Startknoten bis zum jeweiligen Knoten ist.

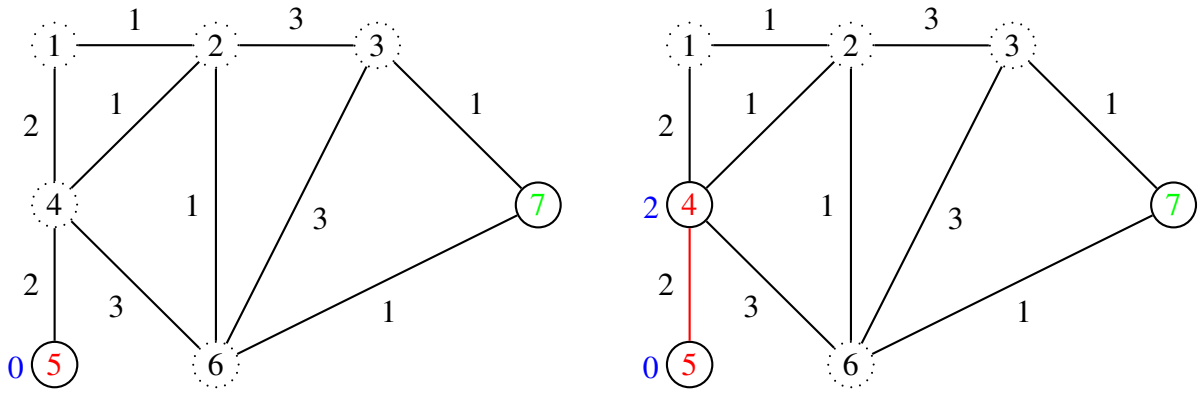
- Erzeuge eine zunächst leere Abbildung (Map), die Knoten auf Vorgängerknoten in ihrem zugehörigen Pfad abbildet (Vorgängerabbildung).
- Initialisiere die Liste mit dem Startknoten und der Gesamtwegelänge 0.
- Solange, bis der Zielknoten erreicht ist:
  - entnimm der Liste den ersten Knoten K mit bisheriger kleinsten Gesamtwegelänge L_K
  - für alle Nachbarknoten N von K, berechne die neue Gesamtwegelänge L_N für N aus L_K + Kantengewicht der Kante von K nach N.
  - füge den Knoten N mit der Gesamtwegelänge L_N ordnungserhaltend in die sortierte Knotenliste ein,
    und K als Vorgänger von N in die Vorgängerabbildung.

Es gibt jedoch einen wichtigen Fall, der eine Sonderbehandlung erfordert. Falls ein Knoten N erreicht wird, für den schon ein Pfad mit Gesamtwegelänge L'_N errechnet wurde, die neue Gesamtwegelänge L_N aber kleiner ist als L'_N, dann muss die größere durch die kleinere Gesamtwegelänge ersetzt werden und als Vorgängerknoten von N der Knoten K eingetragen werden. Das überschreibt den vorherigen Eintrag.

Dieser Algorithmus berechnet für alle Knoten, die er auf dem Weg zum Zielknoten besucht, den kürzesten Weg vom Startknoten. Konkret kann man die Wege aus der Vorgängerabbildung rekonstruieren.

Das folgende Beispiel illustriert die Vorgehensweise. Die Suche startet bei Knoten 5. Ziel ist Knoten 7. Links neben bzw. über den Knoten stehen die momentanen Mindestabstände von Knoten 5.

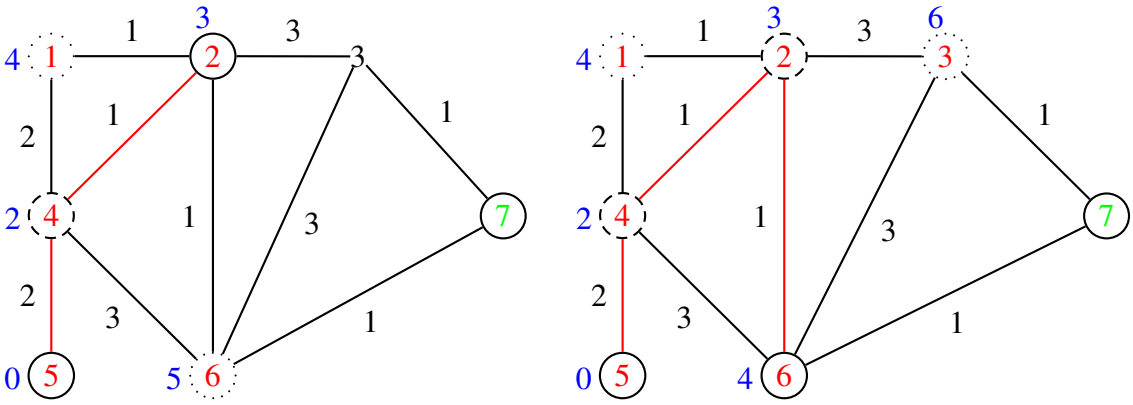
Der erste Nachbarknoten von Knoten 5 ist Knoten 4 mit Abstand 2.
Die Prioritätenschlange hat nur das Element (4).



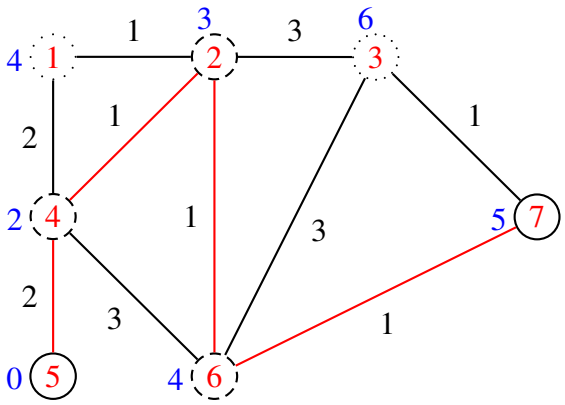
Knoten 4 wird expandiert.
Die Nachbarknoten sind 1, mit Abstand 4, 2 mit Abstand 3 und 6 mit Abstand 5.
Die Prioritätenschlange hat jetzt die Knoten (2,6,1), zusammen mit den jeweiligen aktuellen Abständen zum Startknoten.

Daher wird als nächstes Knoten 2 expandiert.
Die Nachbarknoten sind 1 mit Abstand 4, 6 mit Abstand 4 und 3 mit Abstand 6.
Knoten 1 hat schon Abstand 4, braucht also nicht verbessert zu werden.

Knoten 6 hatte bisher Abstand 5, wird also zu Abstand 4 verbessert.
Die Prioritätenschlange hat jetzt die Knoten (6,1,3).
Knoten 6 und 1 haben gleichen Abstand. Deshalb könnte die Prioritätenschlange auch sein: (1,6,3).
Knoten 1 wäre aber eine Sackgasse, und würde nicht weiter expandiert.



Knoten 6 wird expandiert.
Die Nachbarknoten sind 3, mit Abstand 7, und 7 mit Abstand 5.
Knoten 7 ist das Ziel. Daher stoppt der Algorithmus hier.



Der kürzeste Pfad ist daher 5,4,2,6,7, mit Abstand 5.

3.2 Eine Java Implementierung des Dijkstra-Algorithmus

Die Java-Implementierung unten geht davon aus, dass die Node- und Edge-Klassen beliebig sein können, und keine Vorkehrungen getroffen sind, um an die Knoten Zusatzinformation anzuheften. Diese Zusatzinformation wird daher in HashMaps gepackt. Kann man an die Node-Klasse Zusatzinformation anheften, dann braucht man keine HashMaps.

`Node` und `Edge` sind generische Klassennamen, die durch konkrete Klassen ersetzt werden. Die Methode `dijkstra` hat folgende Argumente:

start ist der Startknoten,
target ist der Zielknoten,

outgoing ist eine Funktion, die für einen Knoten die Collection der ausgehenden Kanten liefert. Sie kann leer sein, sollte aber nicht `null` liefern,
weights ist eine Funktion, die das Gewicht einer Kante liefert,
otherSide ist eine Funktion, die für einen Knoten und eine Kante den anderen Knoten der Kante liefert.

Es gibt folgende interne Speicher:

distances speichert für die bisherigen Knoten den momentanen Minimalabstand zum Startknoten. Zu Beginn ist der Abstand des Startknotens 0 (Zeile 15).

parentNodes speichert für die bisherigen Knoten den Vorgängerknoten im bisherigen kürzesten Pfad,

queue ist die Prioritätenschlange, die die Knoten nach dem bisherigen Minimalabstand sortiert, initialisiert mit dem Startknoten (Zeile 14).

Die Suchschleife läuft ab Zeile 17 und sucht so lange, bis der aktuelle Knoten (`node`) gleich dem Zielknoten ist. Wichtig ist die Unterscheidung, ob ein gerade untersuchter Knoten schon mal untersucht wurde, oder nicht (Zeilen 24 und 26). Wenn ja, und der neue Gesamtabstand kleiner ist als der alte, dann muss der Knoten aus der Schlange gelöscht werden. In den Zeilen 27-29 werden die neuen Werte eingetragen.

Die Methode liefert als Ergebnis ein Tupel [Mindestabstand zum Ziel, Vorgängerknoten-Map]. Aus der Vorgängerknoten-Map kann man den Pfad rekonstruieren.

```
1  static <Node,Edge> Object[] dijkstra(Node start, Node target,
2                       Function<Node,Collection<Edge>> outgoing,
3                       Function<Edge,Integer>          weights,
4                       BiFunction<Edge,Node,Node>      otherSide) {
5     if(start == target) {return new Object[]{(Integer)0,null};}
6     HashMap<Node,Integer> distances = new HashMap<Node,Integer>();
7     HashMap<Node,Node>  parentNodes = new HashMap<Node,Node>();
8     PriorityQueue<Node> queue = new PriorityQueue(
9             ((node1,node2) -> {          // Vergleichsfunktion
10                int distance1 = distances.get(node1);
11                int distance2 = distances.get(node2);
12                if(distance1 == distance2) {return 0;}
13                return (distance1 < distance2) ? -1 : 1;}));
14    queue.add(start);
15    distances.put(start,0);
16    Node node = null;
17    while(node != target) {
18        node = queue.poll(); // Knoten mit kleinstem bisherigen Abstand
19        int distance = distances.get(node);
20        for(Edge edge : outgoing.apply(node)) {
21            int newDistance = distance + weights.apply(edge);
22            Node otherNode = otherSide.apply(edge,node);
23            Integer oldDistance = distances.get(otherNode);
24            if(oldDistance != null && newDistance < oldDistance)
25               {queue.remove(otherNode);}       // muss neu einsortiert werden
26            if(oldDistance == null || newDistance < oldDistance) {
27                 distances.put(otherNode,newDistance);
28                 parentNodes.put(otherNode,node);
29                 queue.add(otherNode); }}}
30     return new Object[]{(Integer)distances.get(target),parentNodes};}
```

In Anhang A steht ein kleines Programm, mit dem man den Algorithmus testen kann.

3.3 Zeitkomplexität des Dijkstra-Algorithmus:

Der Algorithmus läuft entlang aller Kanten und berührt sie maximal einmal. Die Knoten werden in der Prioritätenschlange sortiert, was bei optimaler Implementierung mit $\mathcal{O}(n \log(n))$ Aufwand möglich ist. Der Gesamtaufwand bei optimaler Implementierung ist daher
$\mathcal{O}(\textit{Anzahl Knoten} \cdot \log(\textit{Anzahl Knoten}) + \textit{Anzahl Kanten})$.

3.4 Nachteile des Dijkstra-Algorithmus:

Der Dijkstra-Algorithmus nutzt ausschließlich die Graphstruktur und die Gewichte. Das bedeutet insbesondere, dass er das Ziel erst erkennen kann, wenn er es erreicht hat. Es gibt keinerlei Hinweise, wo das Ziel liegen könnte. Würde man ihn z.B. benutzen, um die kürzeste Straßenverbindung zwischen München und Hamburg zu berechnen, dann würde er von München aus Schicht für Schicht *in alle Richtungen* die Straßenkreuzungen untersuchen. Bis er in Hamburg angelangt wäre, dann hätte er vermutlich auch die Kreuzungen bis Wien, Prag, Rom und Paris untersucht.

4 Der A*-Algorithmus

Dieser Algorithmus ist eine Verfeinerung des Dijkstra-Algorithmus. Er ist dann einsetzbar, wenn man die Entfernung vom aktuellen Knoten bis zum Zielknoten wenigstens abschätzen kann. Für Wegeplanung im Straßennetz z.B. ergibt sich eine Abschätzung der Entfernung von der aktuellen Straßenkreuzung bis zum Ziel aus der *Entfernung in Luftlinie*.

Für die Implementierung braucht man daher eine zusätzliche Funktion
`BiFunction<Node,Node,Integer> estimatedDistance`, die für einen aktuellen Knoten und den Zielknoten die *geschätzte* Entfernung berechnet. Die Prioritätenschlange sortiert man dann nicht nach der aktuellen Entfernung vom Startknoten, sondern nach der aktuellen Entfernung vom Startknoten *plus* der geschätzten Entfernung zum Zielknoten. In der dijkstra-Methode oben würden sich nur die Zeilen 10 und 11 ändern zu:

```
int distance1 = distances.get(node1) + estimatedDistance.apply(node1,target);
int distance2 = distances.get(node2) + estimatedDistance.apply(node2,target);
```

Für die Wegeplanung im Straßennetz, z.B. von München nach Hamburg, macht das einen riesigen Unterschied. Alle Straßen nach Süden z.B. vergrößern die Luftlinie nach Hamburg, so dass sie in der Prioritätenschlange weiter nach hinten kommen. Nur die Straßen nach Norden verkleinern den Gesamtabstand. Der A*-Algorithmus würde also ziemlich gezielt Richtung Hamburg suchen.

Die Luftlinie unterschätzt i.A. den Abstand zum Zielknoten. Nur wenn die Straße schnurgerade zum Ziel läuft ist die Luftlinie gleich der tatsächlichen kürzesten Strecke. Sie ist auf keine Fall länger als die kürzeste Straßenverbindung. Das ist wichtig für den A*-Algorithmus. Nur wenn die

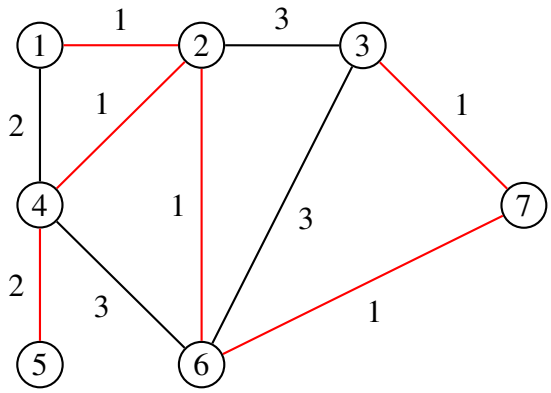
Schätzfunktion den Weg zum Ziel *nicht überschätzt*, ist garantiert, dass er *den kürzesten* Weg zum Ziel findet. Ansonsten findet er wohl auch einen Weg vom Start zum Ziel, aber nicht unbedingt den kürzesten. Gibt die `estimatedDistance`-Funktion immer 0 zurück, dann benimmt sich der A*-Algorithmus ganz genauso wie der Dijkstra-Algorithmus.

Geben die Kantengewichte im Straßennetz die echte Entfernung wieder, und die Schätzfunktion berechnet die Entfernung in Luftlinie, dann ist diese Bedingung automatisch eingehalten. Will man jedoch nicht die kürzeste Verbindung, sondern die schnellste Verbindung berechnen, dann braucht man als Kantengewichte die durchschnittliche Fahrzeit zwischen zwei Kreuzungen, abhängig vom Straßentyp. Als Schätzfunktion könnte man aus der Entfernung in Luftlinie eine geschätzte Fahrzeit berechnen. Diese so hinzubekommen, dass sie einerseits einigermaßen realistisch ist, aber trotzdem nie überschätzt, ist aber absolut nicht einfach.

5 Spannbäume (Spanning Trees)

Ein *Spannbaum* ist ein Teilgraph eines ungerichteten zusammenhängenden Graphen, der ein Baum ist, d.h. insbesondere keine Zyklen enthält, und der alle Knoten des Graphen enthält. Man stelle sich z.B. das Straßennetz einer Stadt vor, wo zu jedem Haus mindestens eine Straße führt. Jetzt möchte man in der Stadt neue Wasserleitungen verlegen, indem man nur die Straßen aufgräbt. Ein Spannbaum wäre jetzt ein Teil des Straßennetzes, so dass aber jedes Haus versorgt wird[1]. Berücksichtigt man die Länge der Straßen, und damit die Länge der Wasserleitungen, dann kann man einen *minimalen Spannbaum* definieren, nämlich ein Spannbaum, dessen Gesamtlänge minimal ist. Das minimiert natürlich die Kosten der Wasserleitungen.

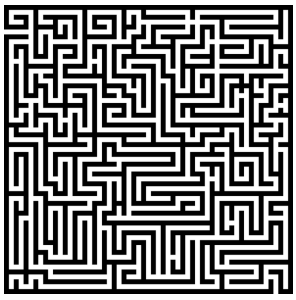
Für unseren schon bekannten Graphen bilden die rot markierten Kanten einen Spannbaum, und zwar einen mit minimalem Gesamtgewicht 7.



Spannbäume sind immer dann hilfreich, wenn man Leitungsnetze verlegen will, die jeden Knoten erreichen, aber mit minimalen Kosten, z.B. Stromleitungen, Telefonnetze, Wasserleitungen, Internet-Verbindungen usw.

[1]Das Beispiel ist nicht 100% passend, da ein Spannbaum alle Knoten des Graphen enthalten muss, die Wasserleitungen aber nicht unbedingt alle Straßenkreuzungen berühren müssen. Erst wenn man zusätzlich fordert, dass an jeder Straßenkreuzung ein Hydrant installiert wird, hat man ein „klassisches“ Spannbaum-Problem.

Aber auch bei ganz anderen Anwendungen spielen Spannbäume eine Rolle, z.B. bei der Erzeugung von Labyrinthen. Der Graph besteht z.B. aus dem Karomuster eines Papiers. Wenn man dafür einen Spannbaum berechnet, erhält man ein Labyrinth mit nur einem Weg hindurch.



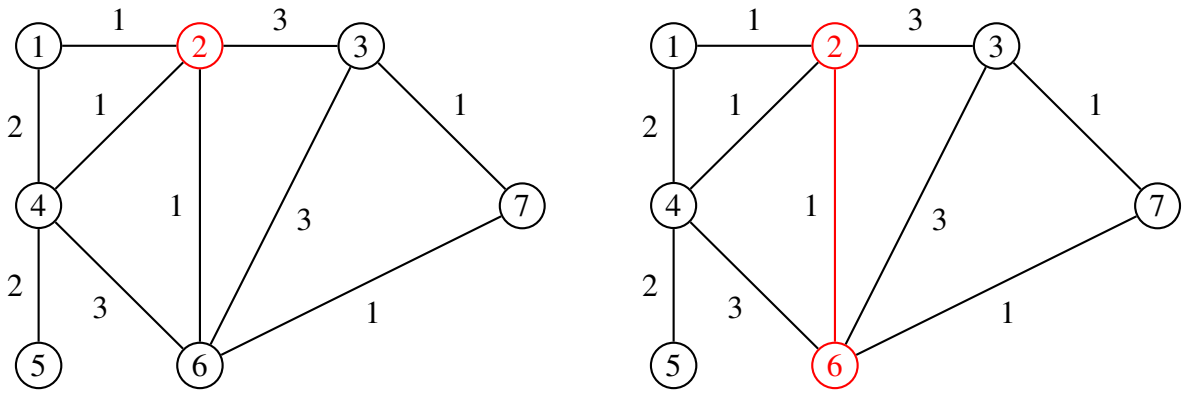
5.1 Die Algorithmen von Prim und Kruskal

Beide Algorithmen erzeugen in einem gewichteten Graphen einen minimalen Spannbaum. Sie unterscheiden sich darin, dass der Algorithmus von Prim die Knoten durchsucht, während der Algorithmus von Kruskal die Kanten durchsucht.

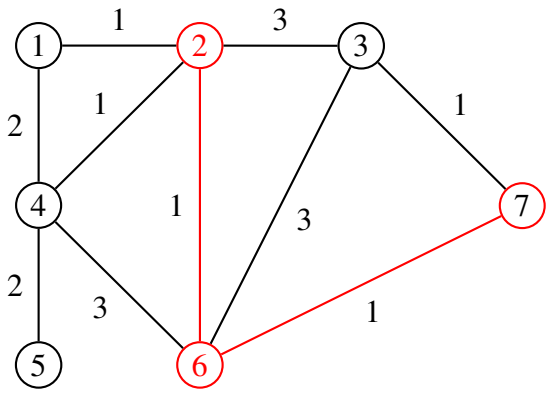
5.1.1 Der Algorithmus von Prim

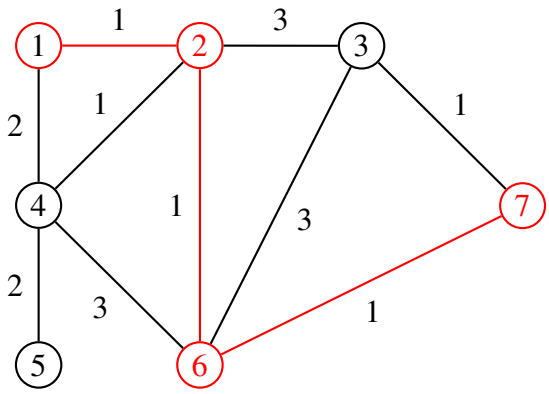
Dieser Algorithmus geht von einem beliebigen Knoten aus, und baut nacheinander in dem schon erzeugten Teilbaum denjenigen neuen Knoten an, dessen Kante zu einem Knoten des Teilbaums das kleinste Gewicht hat.

Im obigen Graphen könnten wir z.B. mit Knoten 2 starten. Als Nachbarknoten mit minimalem Gewicht kämen Knoten 1, 4 und 6 in Frage. Wir wählen Knoten 6.

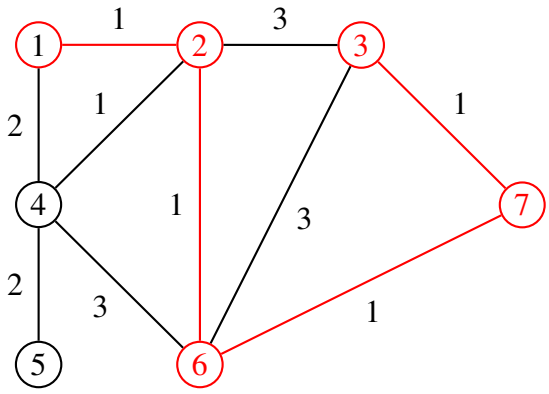


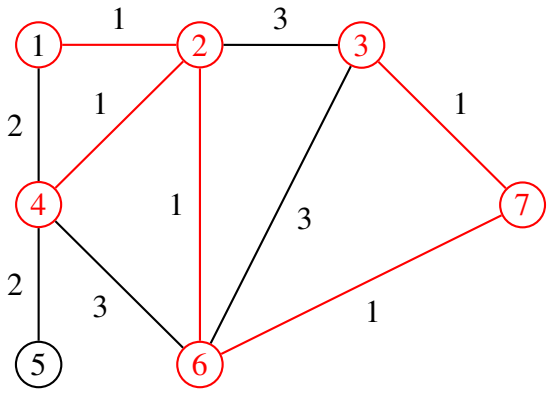
Nachbarknoten mit minimalem Gewicht wären Knoten 1,4 und 7, und gleich anschließend Knoten 1.





Jetzt bleiben noch Knoten 3 als Nachbarknoten von Knoten 7, und Knoten 4 als Nachbarknoten von 2. Sie haben beide das kleinste Gewicht 1. Wir wählen erst Knoten 3 und dann Knoten 4.

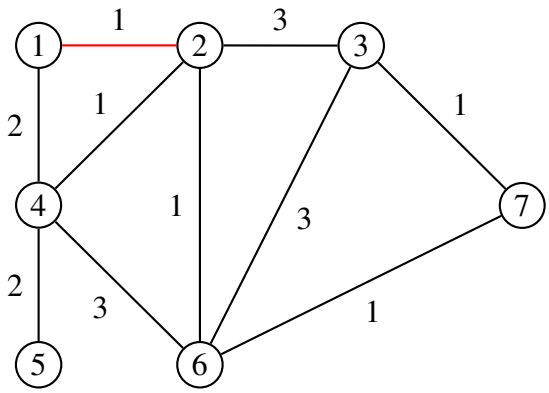


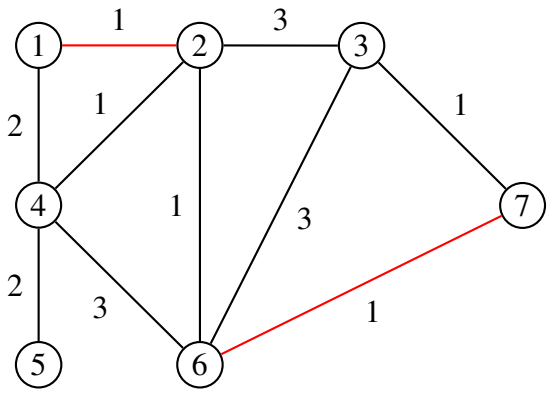


5.1.2 Der Algorithmus von Kruskal

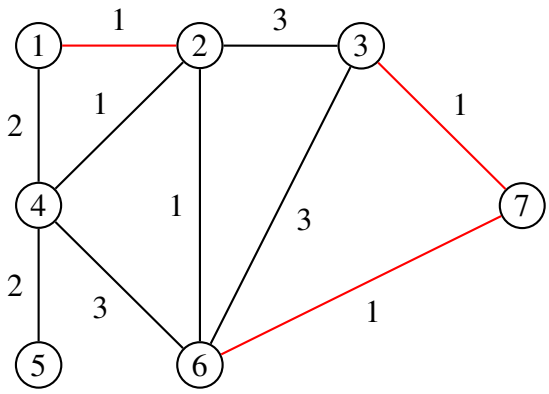
Dieser Algorithmus geht von einer beliebigen Kante *mit kleinstem Gewicht* aus. Er fügt nacheinander Kanten *mit kleinstem Gewicht* hinzu, die aber *keinen Zyklus bilden*. Daher bildet sich meist erst ganz zum Schluss ein Baum.

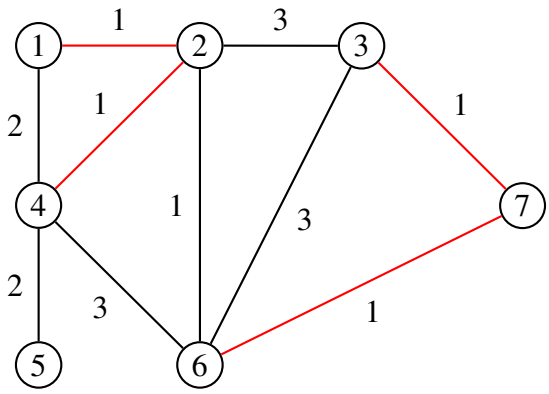
In unserem bekannten Beispielgraphen können wir z.B. mit der Kante 1-2 starten. Mit Gewicht 1 hat sie das kleinste Gewicht. Alternativen wären 2-4,2-6,6-7. Anschließend fügen wir 6-7 hinzu.



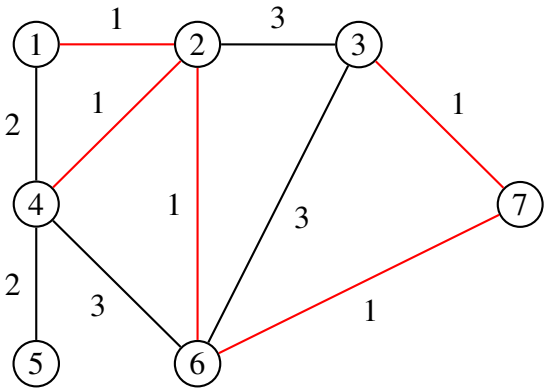


Als nächsten fügen wir 3-7 und 2-4 hinzu.





und schließlich 2-6



Falls die Kante 2-3 auch das Kantengewicht 1 hätte, würde sie trotzdem nicht hinzugefügt werden, weil das einen Zyklus erzeugen würde. Das muss verhindert werden.

In diesem Beispiel gibt es nur einen minimalen Spannbaum. Das ist aber eher selten. Insbesondere wenn alle Kantengewichte gleich groß sind, hat man sehr viele Alternativen. Damit lassen sich z.B. die unterschiedlichsten Labyrinthe erzeugen.

5.2 Eine Java-Implementierung des Kruskal-Algorithmus

Der Kruskal-Algorithmus ist etwas aufwendiger zu implementieren, weil er einen Zyklentest beinhaltet. Die Implementierung unten geht wiederum davon aus, dass die Node- und Edge-Klassen keine Vorkehrungen haben, um Zusatzinformation anzuheften. Das erschwert etwas die Implementierung, macht sie aber flexibler einsetzbar.

Die Methode `kruskal` hat folgende Argumente:

edges: eine Collection von Kanten (List, ArrayList etc.),
node1: eine Funktion, die für eine Kante einen der Endknoten liefert,
node2: eine Funktion, die für eine Kante den anderen Endknoten liefert,
weights: eine Funktion, die für eine Kante dessen Kantengewicht liefert.

Die Methode sortiert die Kanten nach Kantengewicht in einer PriorityQueue und fügt dann eine Kante nach der anderen, wenn sie keinen Zyklus erzeugt, in die ArrayList `spanning` ein. Am Ende liefert sie die Liste der Kanten des Spannbaums zurück.

```
static <Node,Edge> ArrayList<Edge> kruskal(Collection<Edge> edges,
                          Function<Edge,Node> node1,
                          Function<Edge,Node> node2,
                          Function<Edge,Integer> weights) {
   PriorityQueue<Edge> queue = new PriorityQueue<Edge>(
           ((edge1,edge2) -> {
               int weight1 = weights.apply(edge1);
               int weight2 = weights.apply(edge2);
               if(weight1 == weight2) {return 0;}
               return (weight1 < weight2) ? -1 : 1;}));
   queue.addAll(edges);  // automatisch sortiert
   ArrayList<Edge> spanning = new ArrayList();
   Edge edge = null;
   while((edge = queue.poll()) != null) {
       if(!isCyclic(edge,spanning, node1, node2)) {spanning.add(edge);}}
   return spanning;}
```

Während der Kern des Algorithmus extrem einfach zu implementieren ist, ist der Zyklustest etwas aufwendiger, insbesondere wenn man keine Information an die Knoten anheften kann. Man kann sich dann mit HashSets behelfen. Einen Knoten in einem HashSet einzufügen hat dann die gleiche Funktion, wie eine boolesche Marke an den Knoten anzuheften.

Um eine Kante daraufhin zu testen, ob sie einen Zyklus im bisherigen Teil-Spannbaum schließt, läuft die Methode von beiden Enden der Kante den Spannbaum entlang, markiert die getroffenen Knoten, und stoppt wenn sie auf einen schon markierten Knoten trifft.

Ein weiteres Problem ist, dass die Liste `spanning` nicht irgendwie sortiert ist. Man kann also nicht gezielt die Knoten des Teil-Spannbaums entlang laufen, sondern muss die Liste evtl. mehrfach durchsuchen. Es kann nämlich passieren, dass die Markierung der Endknoten eine Kanten *hinten* in der Liste bewirkt, dass man anschließend erst an einer Kante *vorne* in der Liste den Zyklus bemerkt.

```
static <Node,Edge> boolean isCyclic(Edge edge, ArrayList<Edge> spanning,
                        Function<Edge,Node> node1,
                        Function<Edge,Node> node2) {
   HashSet<Node> marks = new HashSet<Node>();
   HashSet<Edge> edgeMarks = new HashSet<Edge>();
   marks.add(node1.apply(edge)); marks.add(node2.apply(edge));
   boolean goon = true;
   while(goon) {
       goon = false;
       for(Edge edg: spanning) {
           if(edgeMarks.contains(edg)) {continue;} // Kante schon markiert
           Node nd1 = node1.apply(edg);
           Node nd2 = node2.apply(edg);
           boolean mark1 = marks.contains(nd1);
           boolean mark2 = marks.contains(nd2);
           if(mark1 && mark2) {return true;} // Zyklus gefunden
           if(mark1 || mark2) {
               edgeMarks.add(edg);
               marks.add(mark1 ? nd2 : nd1);
               goon = true; break;}}} // Nochmal von vorne suchen
   return false;}
```

In Anhang B gibt es ein kleines Programm zum Test der Implementierung.

5.3 Zeitkomplexität und Vergleiche

Der Algorithmus von Prim läuft einmal durch alle Knoten, muss aber die Kanten am Anfang sortieren. Das geht in $\mathcal{O}(n \log(n))$ Zeit. Insgesamt benötigt er daher einen Aufwand von $\mathcal{O}(\textit{Anzahl Knoten} + \textit{Anzahl Kanten} \cdot \log(\textit{Anzahl Kanten}))$.

Der Algorithmus von Kruskal läuft einmal durch die sortierte Liste von Kanten, muss aber den Zyklustest machen. Der oben angegebene Zyklustest ist nicht sehr effizient. Um ihn effizient zu machen, braucht man eine sog. Union-Find Struktur (die in einem anderen Miniskript eingeführt wird). Dann ist der Zyklustest noch effektiver implementierbar als das Sortieren, so dass faktisch ein Aufwand von $\mathcal{O}(\textit{Anzahl Kanten} + \textit{Anzahl Kanten} \cdot \log(\textit{Anzahl Kanten}))$ übrig bleibt.

Sind alle Kantengewichte gleich, z.B. für die Erzeugung eines Labyrinths, oder sind die Listen schon vorsortiert, dann macht sich der Zyklustest doch bemerkbar, wodurch der Algorithmus von Prim einen Vorteil hat.

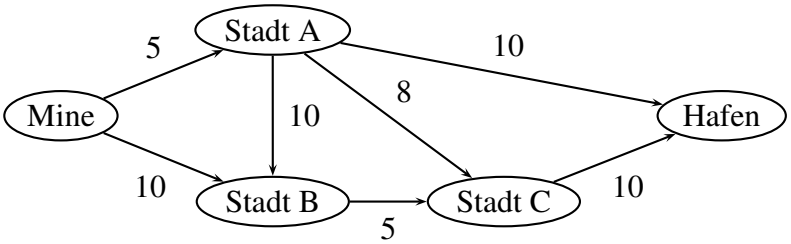
6 Flüsse in Netzwerken (Network Flows)

In diesem Kapitel wird ein weiteres Problem eingeführt, welches man mit gewichteten Graphen modellieren kann. Die Graphen beschreiben Transportnetze, und die Gewichte geben die Transportkapazität an. Ein Beispiel ist das Internet. Jede Verbindung im Internet hat eine Transportkapazität/Bandbreite, in Bits pro Sekunde. Will man eine große Menge Daten von einem Rechner zu einem anderen Rechner transportieren, dann werden die Daten in einzelne Pakete zerhackt, auf evtl. unterschiedlichen Wegen ans Ziel transportiert, und dort wieder zusammengesetzt.

Ein Frage ist nun: was ist die maximale Transportkapazität von Rechner A nach Rechner B, unter optimaler Ausnutzung aller parallelen Leitungen? Das ist das *Maximum Flow*-Problem.

Ein weiteres Beispiel, welches wir noch häufiger benutzen werden: Eine Kohlenmine produziert Kohle, die mit Eisenbahnzügen zu einem Hafen transportiert werden sollen. Es gibt verschiedene Zugstrecken mit verschiedenen Transportkapazitäten. Das Maximum Flow-Problem lautet dann: wieviel Kohle können maximal pro Tag von der Kohlenmine zum Hafen transportiert werden.

Ein Beispielgraph könnte so aussehen,

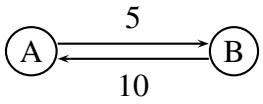


wobei die Kantengewichte die Transportkapazität in 1000 Tonnen Kohle pro Tag angeben. Es gibt immer zwei ausgezeichnete Knoten, die *Quelle* (source) und das *Ziel* (sink). Der „Fluss“ fließt von der Quelle zum Ziel. Im Beispiel wäre die Mine die Quelle und der Hafen das Ziel.

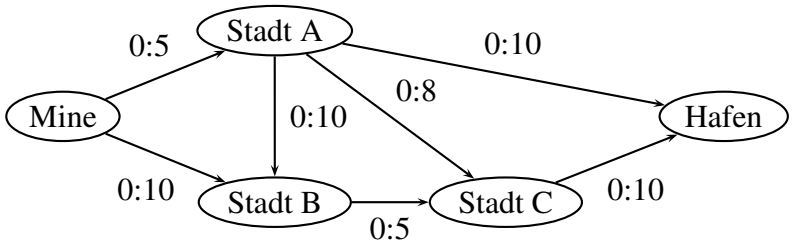
Es gibt etwa ein Dutzend verschiedene Algorithmen, um das Maximum Flow-Problem zu lösen. Etliche basieren auf Ideen des Algorithmus von Dinic.

6.1 Der Algorithmus von Dinic

Im folgenden gehen wir von *gerichteten* Graphen aus. Wenn sowohl von Knoten A nach Knoten B als auch umgekehrt, von Knoten B nach Knoten A Transportmöglichkeiten bestehen, dann werden zwei verschiedene Kanten eingeführt. Diese können sogar unterschiedliche Transportkapazitäten haben. (Das kennt man von der Internetanbindung zu Hause. Da ist der Uplink langsamer als der Downlink).



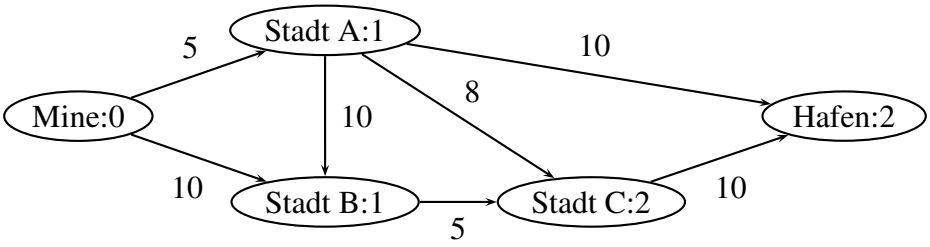
Der Algorithmus von Dinic berechnet in mehreren Schritten immer größere Flüsse. Zu Beginn ist der Fluss durch jeden Knoten 0. Im Graphen wird das folgendermaßen angezeigt.



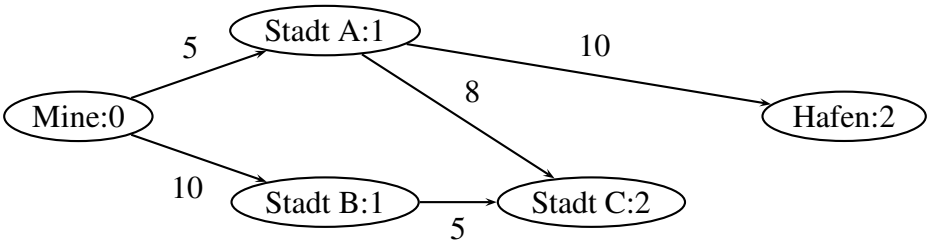
Um den Algorithmus zu verstehen, beginnen wir mit einer vereinfachten Variante. Die volle Version beinhaltet einen Mechanismus, um einmal gemachte ungünstige Entscheidungen in einem späteren Schritt wieder zu korrigieren.

Wir brauchen jedoch zunächst einige Konzepte, die im Algorithmus verwendet werden.

Schicht eines Knoten (engl. level):
Die *Schicht* eines Knoten K ist die *minimale* Anzahl von Kanten von der Quelle bis zu K. Im Bild unten sind die Schichten hinter jedem Knotennamen angegeben.

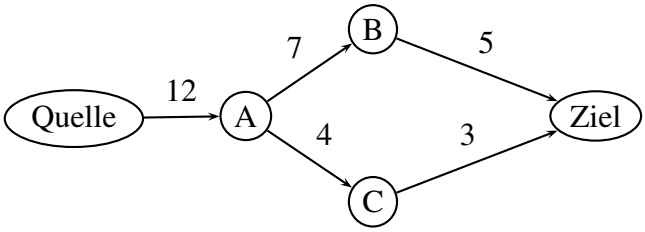


Schichtengraph (engl. level graph)
Der Schichtengraph eines Graphen ist der Teilgraph, dessen Kanten nur von einer Schicht zur direkt nächsten Schicht führen, also von Schicht 0 nach Schicht 1, von Schicht 1 nach Schicht 2 usw. Im obigen Graphen würden daher die Kanten von Stadt A nach Stadt B und von Stadt C zum Hafen wegfallen, da sie die Schichten nicht erhöhen.

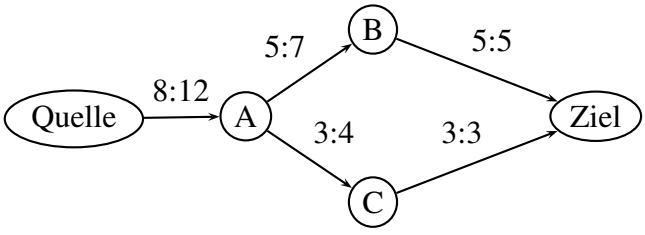


Der Schichtgraph kann auch Sackgassen enthalten, die nicht zum Ziel führen. Jeder Pfad von der Quelle bis zum Ziel führt von Knoten zu Knoten jeweils eine Schicht weiter.

Sperrfluss
Der *Sperrfluss* durch einen Graphen ist der Fluss, bei dem auf jedem Pfad von der Quelle zum Ziel mindestens eine Kante bis zur Kapazitätsgrenze ausgenutzt wird. Wir verdeutlichen das an einem Beispiel:

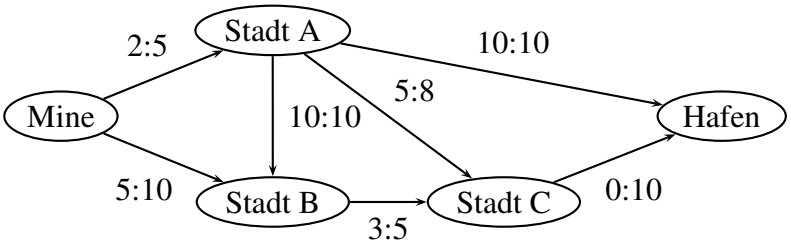


Auf dem Pfad Quelle-A-B-Ziel ist ein maximaler Fluss von 5 möglich.
Auf dem Pfad Quelle-A-C-Ziele ist noch zusätzlich ein maximaler Fluss von 3 möglich.
Der Sperrfluss sieht daher so aus:

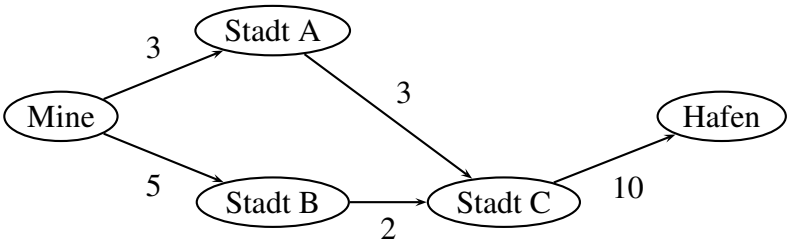


Residualgraph (Restgraph), vereinfachte Version
Für einen Graphen, bei dem sowohl die Kapazitäten als auch die aktuellen Flüsse angegeben sind, beschreibt der *Residualgraph* die Restkapazität. Ist die Restkapazität 0, dann wird die Kante weggelassen.

Für einen Graphen mit folgenden Flüssen

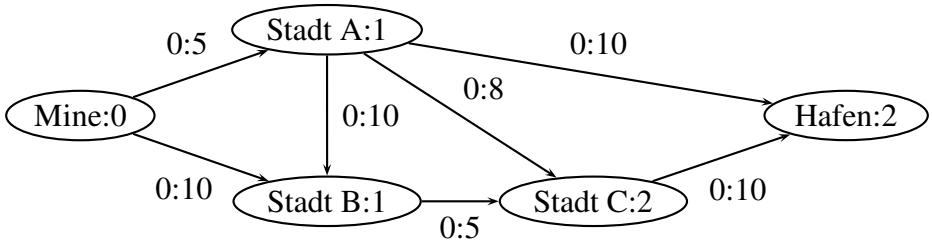


wäre der Residualgraph

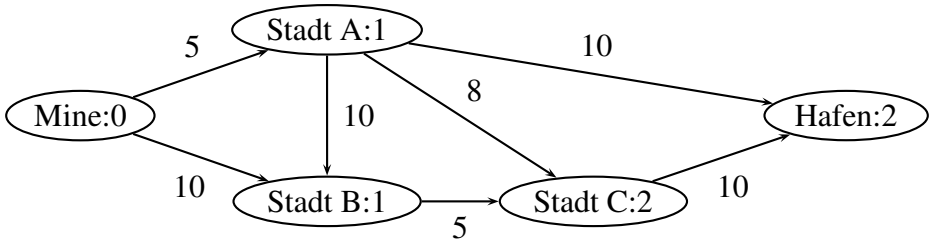


6.2 Dinic's Algorithmus, vereinfachte Version

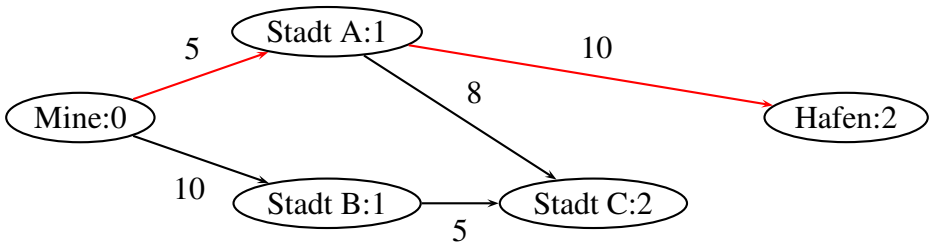
Wir illustrieren den Algorithmus an dem Kohlenbeispiel und starten mit dem leeren Fluss



Der Residualgraph ist zu Beginn trivial, da ja noch kein Fluss vorliegt:

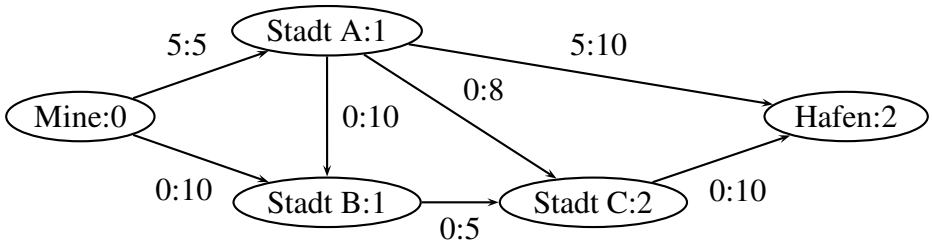


Hieraus wird der Schichtengraph bestimmt, wobei die Kanten von A nach B und von C nach dem Hafen gelöscht werden. Sie wären unnütze Umwege.



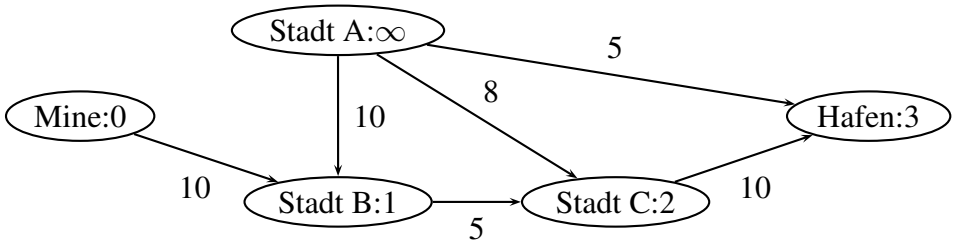
Jetzt gibt es nur einen Weg von der Mine zum Hafen, über die Stadt A. Der Sperrfluss ist daher 5. Mehr kann nicht transportiert werden.

Wir haben jetzt einen ersten Fluss gefunden. Im Ausgangsgraphen sieht das jetzt so aus:

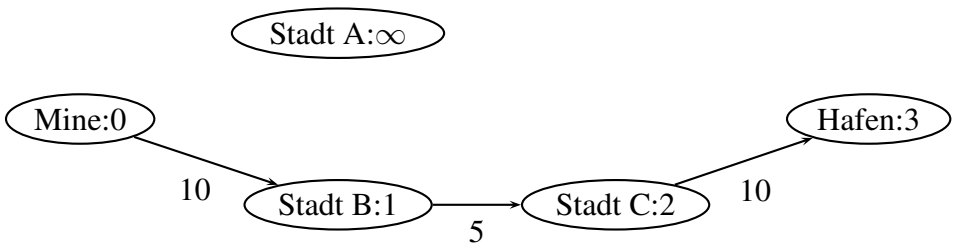


Jetzt beginnt die Schleife von vorne:
- Residualgraphen bestimmen
- Schichtengraphen bestimmen
- Sperrfluss berechnen
- Ausgangsgraphen mit dem Sperrfluss aktualisieren.

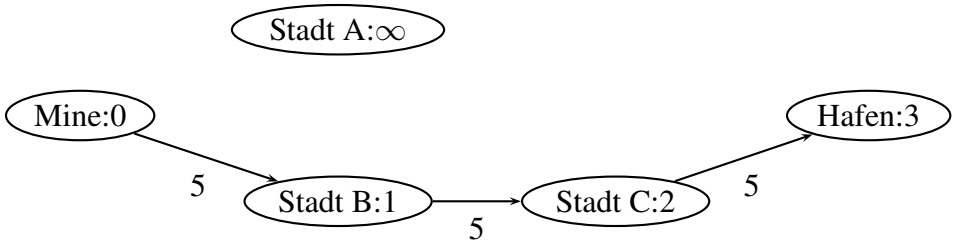
Dadurch, dass im Residualgraphen die voll ausgelasteten Kanten wegfallen, ändern sich die Schichten der Knoten:



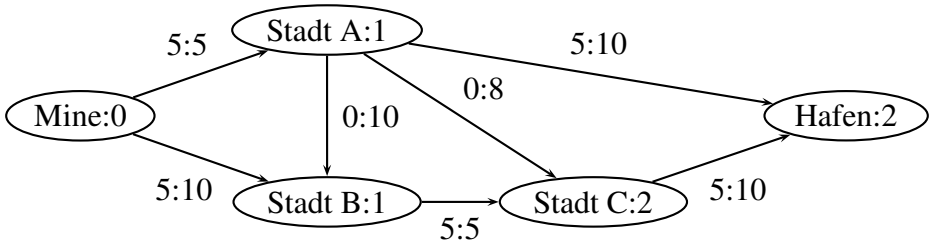
Der Schichtengraph vereinfacht sich dadurch wesentlich:



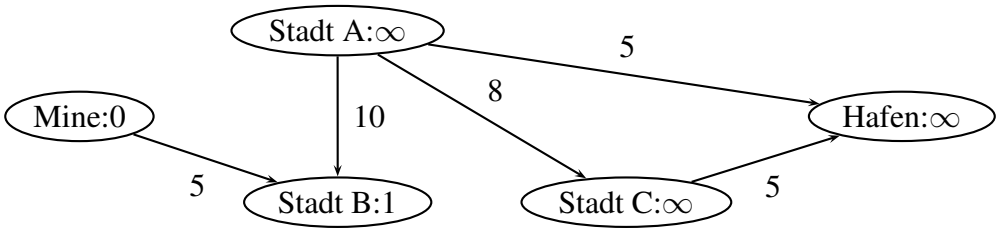
Hierfür ergibt sich ein Sperrfluss von 5:



Wir haben einen weiteren möglichen Fluss gefunden. Dieser wird in den Ausgangsgraphen eingetragen



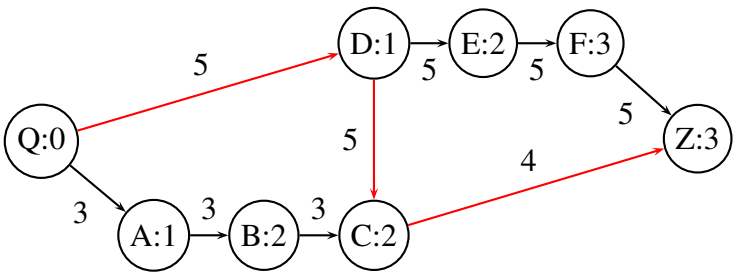
Und jetzt beginnt die Schleife wieder:
Residualgraph:



Da der Hafen nicht mehr zu erreiche ist, kann der Fluss nicht mehr verbessert werden. Pro Tag können also 10000 Tonnen von der Mine bis zum Hafen transportiert werden.

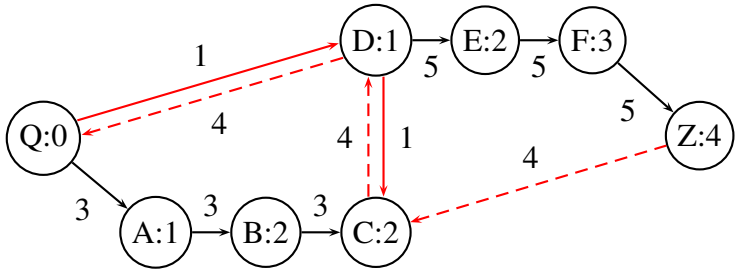
6.3 Dinic's Algorithmus, volle Version

Die bisher skizzierte Vorgehensweise kann zu unnötigen Engpässen führen, die nur überwunden werden können, wenn man frühere Entscheidungen korrigieren kann. Folgendes Beispiel illustriert das. Q ist die Quelle, Z das Ziel.

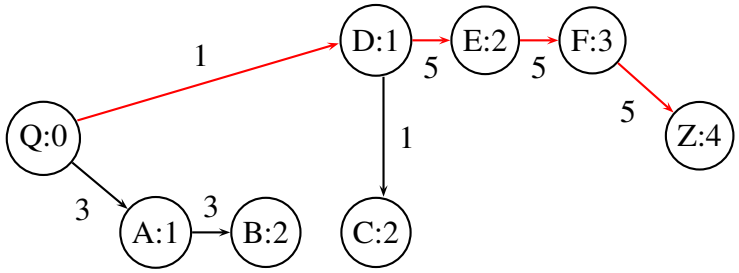


Im Schichtengraph wäre Q-D-C-Z die kürzeste Strecke, mit Sperrfluss 4. Das Problem ist jetzt, dass die Strecke C-Z schon ausgelastet ist. Es gäbe daher keine weitere Transportmöglichkeit über Q-A-B-C. Die Entscheidung, die kürzeste Strecke im Schichtengraph zu wählen, war daher fatal. Man muss die Entscheidung, von D nach C zu gehen, wenigstens teilweise rückgängig machen. Dazu wurde eine erweiterte Version des Residualgraphen definiert, bei dem die Möglichkeit eingebaut ist, Entscheidungen rückgängig zu machen. Er enthält für jede Transportkante auch eine Rückkante.

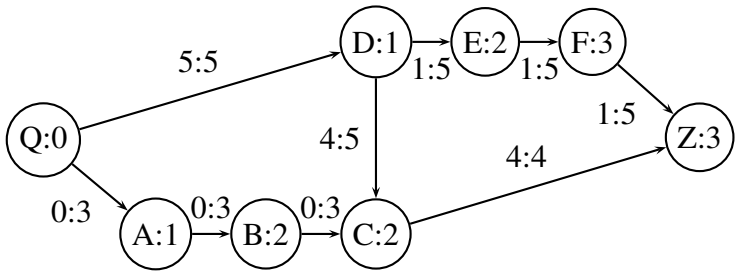
Der neue Residualgraph für das Beispiel sieht dann so aus:



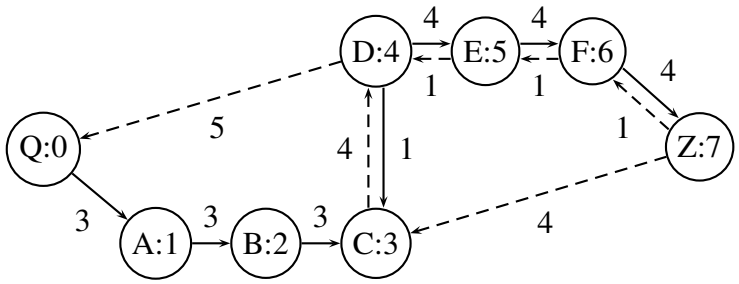
Interessant sind die Pfeile zwischen D und C. D→C mit Gewicht 1 bedeutet, dass noch 1 Einheit transportiert werden kann. C→D mit Gewicht 4 bedeutet, dass man evtl. 4 Einheiten Transport rückgängig machen kann. C→Z fehlt, da diese Route voll ausgelastet ist. Deshalb hat Z jetzt Schicht 4. Die Maßnahme mit den umgedrehten Pfeilen wirkt allerdings noch nicht sofort, denn der Schichtengraph sieht jetzt so aus.



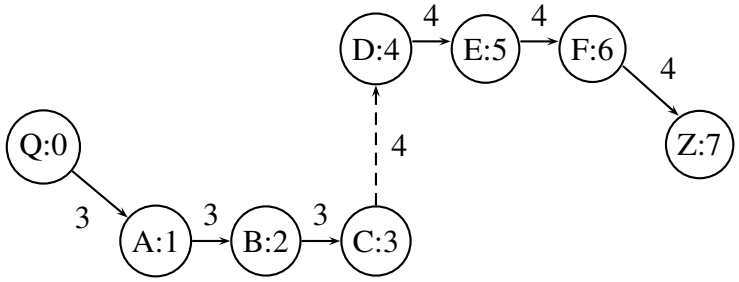
Eine Transporteinheit kann also noch über Q,D,E,F nach Z geschickt werden. Trägt man das in den Ausgangsgraphen ein, dann wird daraus:



Der neue Residualgraph ist dann:

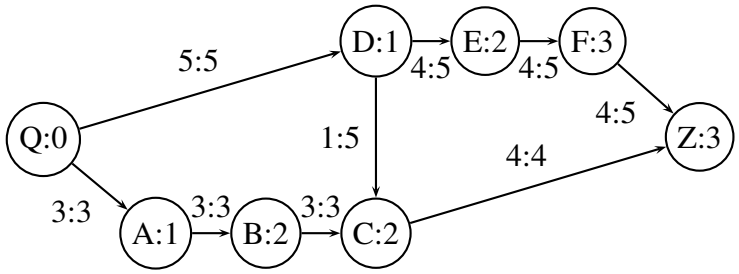


wobei die gestrichelten Rückpfeile die Änderungsmöglichkeiten andeuten. Der Wegfall von Q→D bedeutet insbesondere, dass sich die Schichten der hinteren Knoten stark ändern. Der Schichtgraph ist dann:



Es bleibt jetzt insbesondere die Kante C→D. Sie signalisiert, dass man den ursprünglich geplanten Transport rückgängig machen kann. Der Sperrfluss ist jetzt 3. D.h. 3 der ursprünglich geplanten 4 Einheiten von D nach C muss man umleiten. Der neue Pfad gibt sogar die Umleitung an, nämlich über E,F nach Z. Damit wird auch ein Teil der Strecke C nach Z wieder frei und kann anstelle des Teilpfades C,D,E,F nach Z genutzt werden.

Im Ausgangsgraphen sieht das dann so aus. 8 Einheiten können von Q nach Z transportiert werden.



Der Algorithmus ist mit diesem Beispiel nur skizziert. Die wesentlichen Ideen sollten aber verstanden sein.

Eine genau Komplexitätsanalyse ergab eine Zeitkomplexität von $\mathcal{O}(\textit{Anzahl Knoten}^2 \cdot \textit{Anzahl Kanten})$.

7 Das Problem des Handlungsreisenden (Travelling Salesman-Problem oder TSP)

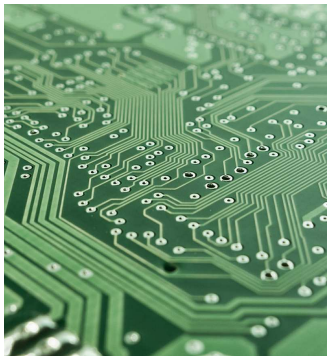
Man stelle sich einen Handlungsreisenden vor, der von München aus alle Städte Bayerns besuchen, und dann wieder nach München zurück kehren muss. Wie schnell kann er das schaffen? Das Problem gibt es in mehreren Formen:

Das Hamiltonkreis-Problem: Gibt es in einem gerichteten oder ungerichteten Graphen einen *geschlossenen* Pfad, der alle *Knoten genau einmal* durchläuft[2]? Beide Varianten gehören zu den sog. NP-vollständigen Problemen, für die es bisher keine effiziente Lösung gibt.

Optimierungsproblem: Was ist die *kürzeste Route*, die alle Städte (Knoten des *gewichteten Graphen*) berührt, und wieder zum Ausgangspunkt zurück führt? Hierbei dürfen u.U. auch Knoten/Städte mehrfach durchfahren werden.

Aufwandsproblem: Gibt es eine Route, die alle Knoten/Städte berührt, wieder zum Ausgangspunkt zurück führt, und einen vorgegebenen Aufwand nicht überschreitet? Das könnte eine Kilometerzahl sein, eine Tankfüllung seines Autos, oder eine Zeitvorgabe.

Solche Probleme treten insbesondere in der Logistik auf, wo man Transportrouten optimal planen möchte. Sogar ein Briefträger muss so ein Problem für sich lösen, wenn er früh genug fertig werden will.



Es gibt aber eine besondere Anwendung, wo eine Lösung des Optimierungsproblems extrem viel Geld sparen kann. Dabei geht es um die optimale Reihenfolge für Bestückungsautomaten. Man stelle sich z.B. eine Leiterplatine vor, an der Dutzende oder gar Hunderte von Lötpunkten anzubringen sind. Ein Lötautomat fährt mit seiner Lötspitze jede Lötstelle an, und bringt dort einen Lötpunkt an. Die Lötstellen sind die Knoten des Graphen. Der Lötautomat kann sich frei von jeder Lötstelle zu jeder anderen Lötstelle bewegen. Die Kanten des Graphen sind daher die Wege zwischen den Lötstellen. Die Kantengewichte sind die Weglängen zwischen den Lötstellen. Da jede Lötstelle von jeder anderen Lötstelle direkt angefahren werden kann, ist der Graph *vollständig*, d.h. *jeder Knoten ist mit jedem anderen Knoten verbunden*.

Das Problem besteht nun darin, für den Lötautomaten den kürzesten Weg durch alle Lötstellen zu finden. Je kürzer der Gesamtweg ist, desto schneller ist er fertig, und desto mehr Platinen kann er pro Stunde löten, und desto mehr Geld verdient die Firma.

7.1 Die perfekte Lösung: permutieren

Die allereinfachste Lösung des TSP besteht darin, *alle Permutationen* der Knoten daraufhin zu testen, ob sie einen zulässigen Pfad durch den Graphen darstellen, und für das eigentliche TSP die Gesamt-

[2] Es gibt auch das analoge *Eulerkreisproblem*, bei dem all *Kanten* genau einmal durchlaufen werden müssen.

distanz zu berechnen. Der Pfad mit der kleinsten Gesamtdistanz ist die Lösung.

Das Problem besteht in der Forderung „alle Permutationen“. Für n Knoten gibt es $n!$ Permutationen. Angenommen, für eine Permutation braucht der Rechner 1 Mikrosekunde. Dann ergeben sich folgende Zeiten:

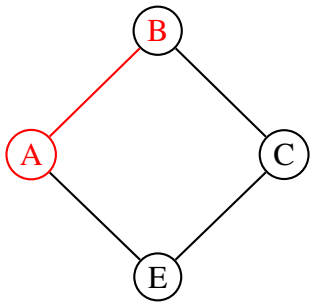
n	$n!$	Zeit
10	3628800	3.6 sec
20	$2.432902 \cdot 10^{18}$	77146 Jahre
100	$9.332622 \cdot 10^{157}$	$2.959355 \cdot 10^{146}$ Jahre[3]

Da müssen unsere Computer schon noch enorm viel schneller werden, um das zu schaffen.

7.2 TSP mit Dynamischem Programmieren

Held und Karp fanden 1962 eine deutlich bessere Methode als das Permutieren. Die Idee ist folgende: Man startet mit dem Startknoten S, und probiert nacheinander alle Nachbarknoten N_i von S, sortiert nach dem jeweiligen Kantengewicht. Jetzt braucht man einen kürzesten Pfad von N_i nach S, wo die Tour enden soll. Dafür probiert man für jedes N_i wiederum dessen Nachbarknoten usw. Die eigentliche Verbesserung besteht aber in der Beobachtung, dass auf diese Weise immer wiederkehrende Unterprobleme entstehen, die man nicht nochmal lösen muss, sondern deren Lösung man sich merken kann, und dann wiederverwenden.

Das folgende extrem vereinfachte Beispiel illustriert das.



Wir starten mit Knoten A und wählen als ersten Nachbarknoten Knoten B. Jetzt braucht man noch einen kürzesten Weg von Knoten B nach A, der C und E berührt. Das ist natürlich B-C-E-A. Wir merken uns diesen Pfad.

Dies muss aber nicht der kürzeste Weg sein. Daher muss Nachbarknoten E von A auch getestet werden. Von E aus muss man auch nach A kommen, und C und B berühren. Der vorher berechnete Pfad B-C-E-A muss nur etwas anders durchlaufen werden, nämlich rückwärts, E-C-B-A. Er kann also wiederverwendet werden.

Durch geschickt Implementierung der Methode, einmal berechnete Pfade wiederzuverwenden, kann man die Komplexität des Algorithmus auf $\mathcal{O}(n^2 \cdot 2^{n-1})$ reduzieren. Wieviel Zeit das immer noch ist, wenn ein Rechenschritt 1 Mikrosekunde braucht, zeigt die Tabelle:

n	$n^2 \cdot 2^{n-1}$	Zeit
10	51200	0.512 sec
20	209715200	209.715 sec
100	$6.338253 \cdot 10^{33}$	$2.0098468 \cdot 10^{26}$ Jahre

[3] Das Alter des Universums wird auf $1.4 \cdot 10^{10}$ Jahre geschätzt.

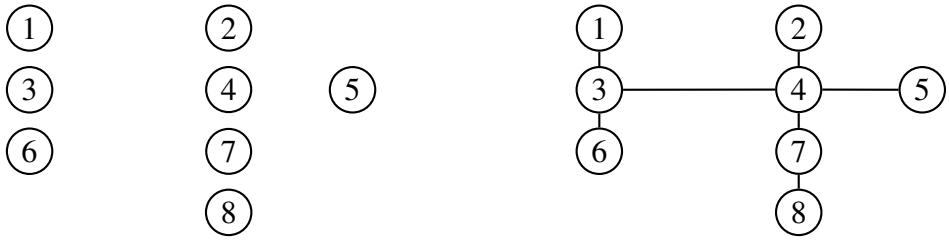
Das ist zwar deutlich besser als die Permutationsmethode, aber immer noch nicht praktikabel. Es ist bisher leider kein Algorithmus bekannt, der das TSP für größere Knotenmengen in akzeptabler Zeit perfekt löst[4].

7.3 Näherungsverfahren

Da das TSP sehr viel praktische Anwendungen hat, wurden viele Versuche unternommen, verbesserte Algorithmen zu finden. Dafür kann man entweder für Algorithmen, die die optimale Lösung finden, heuristische Verbesserungen einbauen, um z.B. unnötige Schritte zu vermeiden. Oder man verzichtet darauf, die optimale Lösung zu finden, und begnügt sich mit Lösungen, die gut genug sind, um praktisch brauchbar zu sein. Dazu hat man das TSP in ein ganz anderes Problem transformiert, nämlich in sog. ganzzahlige lineare Programme. Damit lassen sich einigermaßen effizient suboptimale Lösungen finden, für die man sogar abschätzen kann, wie nahe sie an der optimalen Lösung liegen. Die Technik ist komplex und kann daher in diesem Miniskript nicht behandelt werden.

Es gibt jedoch eine Methode, die leicht zu erklären ist, und für den wichtigen Spezialfall der Bestückung von Platinen funktioniert. Für Lötstellen auf Platinen, und Automaten, die quer über die Platine fahren können, gilt nämlich die *Dreiecksungleichung*. Das bedeutet, dass ein direkter Weg immer kürzer ist, als ein Umweg. In diesem Fall kann man nämlich Umwege durch direkte Wege abkürzen, und damit Pfade durch einen Graphen durch Abkürzungen optimieren. Wie das funktioniert zeigt folgendes Beispiel.

Der linke Graph, wo nur die Knoten gezeichnet sind, soll vollständig verbunden sein, d.h. es gibt Kanten von jedem Knoten zu jedem anderen Knoten. Für diesen Graphen berechnen wir einen minimalen Spannbaum. Den zeigt der rechte Graph.

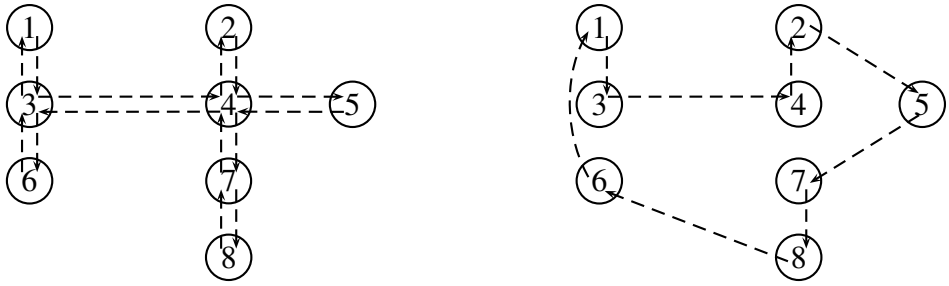


Ausgangsgraph Spannbaum

Jetzt konstruieren wir einen Pfad, der im Uhrzeigersinn, einmal rund um den Spannbaum läuft. Im linken Graphen unten starten wir z.B. bei Knoten 1.

Dieser Pfad wäre ein möglicher Weg für den Lötautomaten, fährt aber mehrfach zu denselben Lötstellen. Dies kann er vermeiden, indem er Lötstellen, die er schon mal besucht hat, überspringt. Der rechte Graph stellt daher das Endergebnis des Verfahrens dar.

[4]Wenn jemand so einen Algorithmus finden würde, wäre P = NP, und derjenige bekäme 1 Million $.



Zusammengefasst geht das Verfahren in folgenden Schritten vor:

- berechne einen Spannbaum,
- erzeuge einen Rundweg, der den Spannbaum im Uhrzeigersinn abläuft,
- vermeide, Knoten doppelt zu besuchen, indem dieser Weg wiederum abgelaufen wird, und schon einmal besuchte Knoten übersprungen werden.

Das Verfahren funktioniert für vollständige Graphen, in denen die Dreiecksungleichung gilt. Man kann zeigen, dass die Weglänge, die dieses Verfahren erzeugt, höchstens zweimal so lang ist, wie der optimale Weg. Mit einer weiteren Verbesserung (von Christophides 1976), erhält man sogar einen Pfad, der höchstens 1.5 mal so lang ist wie der optimale Weg.

Das Problem des Handlungsreisenden ist immer noch ein aktives Forschungsgebiet, wo neue Ideen zu verbesserten Algorithmen führen können.

A Anhang: Dijkstra Test

Mit diesem Java Programm kann man den Dijkstra Algorithmus an einem konkreten Beispiel testen. Das Beispiel ist das aus dem entsprechenden Kapitel. Knoten und Kanten werden dabei als Zahlen repräsentiert. Die Kante mit der Nummer 12, z.B. läuft von Knoten 1 nach Knoten 2.

```
static HashMap<Integer,Collection<Integer>> adjacencyList = new HashMap();
 static HashMap<Integer,Integer> weightsMap = new HashMap();
 static {
     adjacencyList.put(1, Arrays.asList(2, 4));
     adjacencyList.put(2, Arrays.asList(1, 3, 4, 6));
     adjacencyList.put(3, Arrays.asList(2, 6, 7));
     adjacencyList.put(4, Arrays.asList(1,2,5,6));
     adjacencyList.put(5, Arrays.asList(4));
     adjacencyList.put(6, Arrays.asList(4,2,3,7));
     adjacencyList.put(7, Arrays.asList(3,6));

     weightsMap.put(12,1);
     weightsMap.put(14,2);
     weightsMap.put(23,3);
     weightsMap.put(24,1);
     weightsMap.put(26,1);
     weightsMap.put(36,3);
     weightsMap.put(37,1);
     weightsMap.put(45,2);
     weightsMap.put(46,3);
     weightsMap.put(67,1);
 }

 static Integer revertEdge(Integer edge) {return (edge % 10)*10+edge/10;}


 public static void main(String[] args) {
     Function<Integer,Integer> weights =  (edge-> {
             Integer w = weightsMap.get(edge);
             return (w == null) ? weightsMap.get(revertEdge(edge)) : w;});

     Function<Integer,Collection<Integer>> outgoing = (node -> {
             ArrayList<Integer> outg = new ArrayList();
             for(Integer i : adjacencyList.get(node)) {outg.add(node*10+i);}
             return outg;});

     BiFunction<Integer,Integer,Integer> otherSide = ((edge,node) -> {
         int node1 = edge/10;
         int node2 = edge % 10;
         return (node == node1) ? node2 : node1;});

     Object[] result = dijekstra(5,7,outgoing,weights,otherSide);
     System.out.println(result[0]);
     System.out.println(result[1]);
 }
```

B Anhang: Kruskal Test

Mit diesem Programm kann der Kruskal-Algorithmus an dem gleichen Beispiel getestet werden.

```
public static void main(String[] args) {
        Function<Integer,Integer> weights =  (edge-> {
            Integer w = weightsMap.get(edge);
            return (w == null) ? weightsMap.get(revertEdge(edge)) : w;});
        Function<Integer,Integer> node1 = (edge -> edge % 10);
        Function<Integer,Integer> node2 = (edge -> edge / 10);
        List<Integer> edges = Arrays.asList(12,14,23,24,26,36,37,45,46,67);
        System.out.println(kruskal(edges,node1,node2,weights));
    }
```

Stichwortverzeichnis