Max-Flow-Min-Cut Algorithmen

Projektunterricht

Verfasser: Nandor Kovacs

https://github.com/NandorKovacs/flow

MNG Rämibühl

May 20, 2023

1 Problemstellung

Das Max-Flow-Min-Cut Theorem besagt, dass der maximale Fluss auf einem Flussnetzwerk, und der minimale Schnitt des Netzwerks equivalent sind. Ein Flussnetzwerk ist ein Graph mit einer Quelle und einer Senke. Das kann man sich so vorstellen, dass aus der Quelle unendlich viel Wasser fliessen kann und die Senke unendlich viel Wasser schlucken kann.

Diese Werte sind aus vielen Gründen sehr interessant für einen Graphenn und kommen auch in der realen Welt oft zur Anwendung. Hat man Beispielsweise ein Netz aus Röhren und Schläuchen und möchte wissen wie schnell man dadurch Wasser pumpen kann, ist der maximale Fluss dieses Flussnetzwerks die Lösung. Erweiterungen dieser Algorithmen werden genutzt um die Wasserzufuhr gewisser Gebiete optimal zu konstruieren. Will man neue Wohngebiete oder Industrieanlagen bauen, brauchen diese genug Wasser. Mit solchen Algorithmen kann man bestimmen, wo es sinnvoll wäre die bestehende Infrastruktur auszubauen.

Eine der ersten Anwendungen, die zur Erforschung dieser Algorithmen führte, war eine taktische Kriegsfrage. Denn hat man ein Strassennetzwerk, auf den Soldaten losgeschickt werden, will man diese möglichst effizient aufhalten. Wo es am optimalsten ist Barrikaden aufzubauen lässt sich auf die Frage nach dem minimalen Schnitt reduzieren. Dafür wurde der Ford-Fulkerson Algorithmus entwickelt, eine der ersten, und heute noch einer der meistbekanntesten Lösungen für das Max-Flow-Min-Cut Problem.

Ungefähr Zeitgleich versuchte man ebenfalls mit hilfe von computergestützten Rechnungen, die Logistik der Nahrungsverteilung zwischen Städten und Dörfern zu verbessern. Es war überraschend, dass der selbe Algorithmus funktionierte. Dies schuf das Max-Flow-Min-Cut Theorem, das einfach beweisbar ist. Es besagt das der maximale Fluss, und der minimale Schnitt in einem Flussnetzwerk gleich sind. Das der maximale Fluss kleiner oder gleich dem minimalen Schnitt ist, ist eindeutig. Es kann nicht mehr Wasser über dem minimalen Schnitt fliessen wie der minimale Schnitt selber. Und die Algorithmen die ich hier vorstelle, finden beweisbar auf allen Flussnetzwerken einen Fluss der gleich dem minimalen Schnitt ist.

Noch wichtiger ist die Version des Problems, wo das Flussnetzwerk auch noch Kantengewichte hat, die die jeweiligen Kosten für einen bestimmten Weg darstellen. Diese Version wird für die oben beschriebene Wasserinfrastruktur verwendet, da es in so einem Fall wichtig ist, wie viel Geld es kostet bestimmte Stellen auszubauen. So ein Problem kann man beispiel-

sweise mithilfe von linearem Programmieren lösen.

In diesem Bericht werde ich zwei Algorithmen vorstellen, die ich gelernt und selbst implementiert habe.

2 Lösung mit naiven Ansätzen

2.1 Was wäre der naheliegendste Ansatz an das Problem?

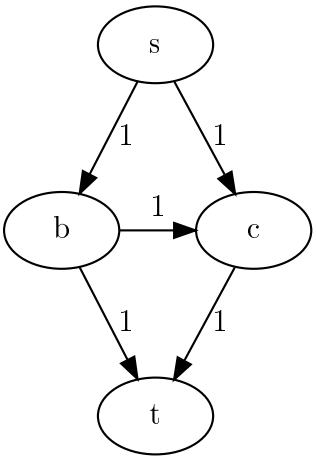
Meine erste Idee ist, einen beliebigen Pfad von der Quelle zur Senke zu finden, beispielsweise mit dem bekannten Dijkstra Algorithmus. Danach schaue ich, wie viel Wasser ich entlang dieses Pfades schicken kann. Das entspricht dem minimum der Kantengewichte entlang des Pfades. Jetzt aktualisiere ich alle Kantengewichte entlang des Pfades, indem ich so viel abziehe wie ich Wasser entlanggeschickt habe. Alle Kanten die dabei 0 erreichen entferne ich.

```
g ← Flussnetzwerk;
resultat ← 0;
while pfad ← dijkstra(g) do
    wassermenge ← ∞;
    foreach kante ∈ pfad do
        wassermenge ← min(wassermenge, kante.kapazitaet)
    end
    result ← result + wassermenge;
    foreach kante ∈ pfad do
        kante.gewicht ← kante.gewicht − wassermenge;
        if kante.gewicht = 0 then
            g.loesche(kante);
        end
    end
end
```

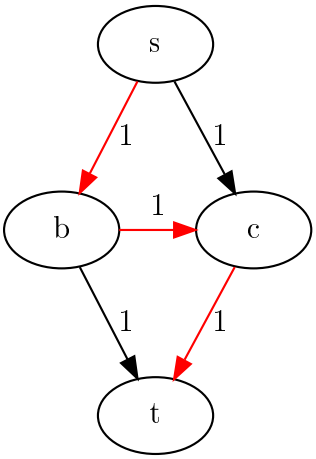
Algorithm 1: Naiver algorithmus mit Dijkstra

2.2 Wieso funktioniert das nicht?

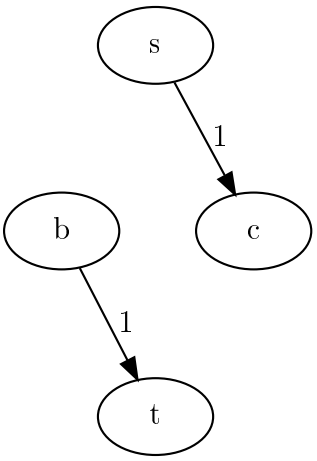
Nehmen wir folgendes Flussdiagramm als Beispiel:



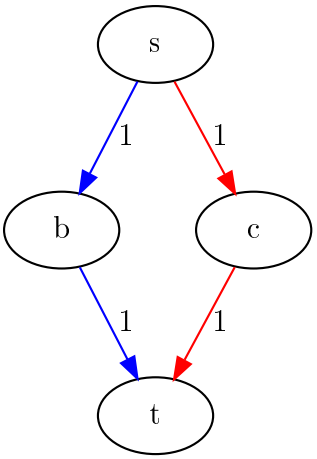
Zuerst wählen wir einen Zufälligen Pfad von s nach t aus:



Jetzt nehmen wir von jeder Kante 1 Gewicht Pfad, da wir 1 Wasser entlang des Pfades schicken. Somit verfallen die Kanten auf dem roten Pfad, da alle nur ein Kantengewicht von 1 haben.

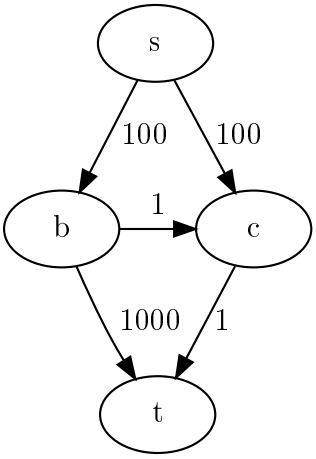


Jetzt hat es keinen Pfad mer von s nach t. Somit kämen wir zur Lösung, dass der maximale Fluss auf diesem Netzwerk 1 ist. Wir haben hier aber eine Beispiel, wo wir 2 Wasser über das Netzwerk schicken können:



2.3 Wieso hat das nicht funktioniert?

Nur weil wir einen Pfad gefunden haben, heisst dass noch lange nicht das wir diesen auch nutzen können, um zur optimalen Lösung zu kommen. Bei folgendem Beispiel ist es ziemlich klar, dass es keinen Sinn ergibt die Kannte (b, c) zu nutzen. Wir haben eine sehr hohe Kapazität auf (b, t), und nur eine niedrige auf (c, t). Wir möchten alles Wasser, das bei b ankommt sofort durch (b, t) schicken, damit möglichst viel Kapazität wie möglich auf (c, t) übrig bleibt für Wasser, das bei c ankommt.



Natürlich könnten wir Glück haben, so dass wir zufälligerweise genau die richtige Pfade in der richtigen Reihenfolge auswählen. Aber wenn wir das nicht garantieren können, ist unser Algorithmus nicht korrekt.

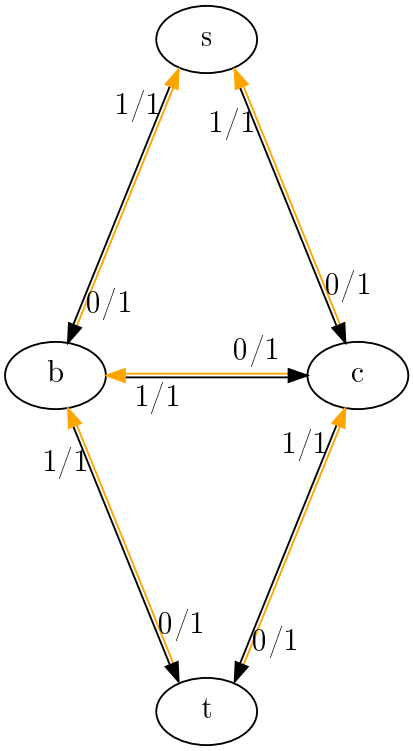
3 Die Ford-Fulkerson Methode

3.1 Die Idee

Das Prinzip der Ford-Fulkerson Methode ist das gleiche wie das Prinzip vom beschriebenen naiven Algorithmus. Wir suchen einen Pfad von s nach t, und lassen so viel Wasser wie es geht durchfliessen.

Der einzige Unterschied ist, das wir beim Ford-Fulkerson für jede Kante eine Kante einfügen, die in die entgegengesetzte Richtung geht, die gleiche Kapazität hat, und immer umgekehrt so viel Wasser darauf fliesst, wie auf der Partnerkante. Anstatt Kanten allmählich zu entfernen, verändern wir unseren Dijkstra so, dass es Kanten auf denen gleich viel Wasser ist wie ihre Kapazität nicht in betracht zieht.

Die jetzt eingefügte Rückkanten bilden einen sogenannten Restnetzwerk.



Die Intuition dahinter ist, dass bei einer schlechten Pfadwahl, die unser Resultat wie zuvor kaputt macht, der schlechte Teil zurückgefahren werden kann. Anfangs sind die Kantengewichte des Restnetzwerks gleich gross wie dessen Kapazitäte, also können wir sie nicht nutzen. Sobald wir eine Kante aber genutzt haben, können wir dessen Restkante nutzen.

3.2 Beweis

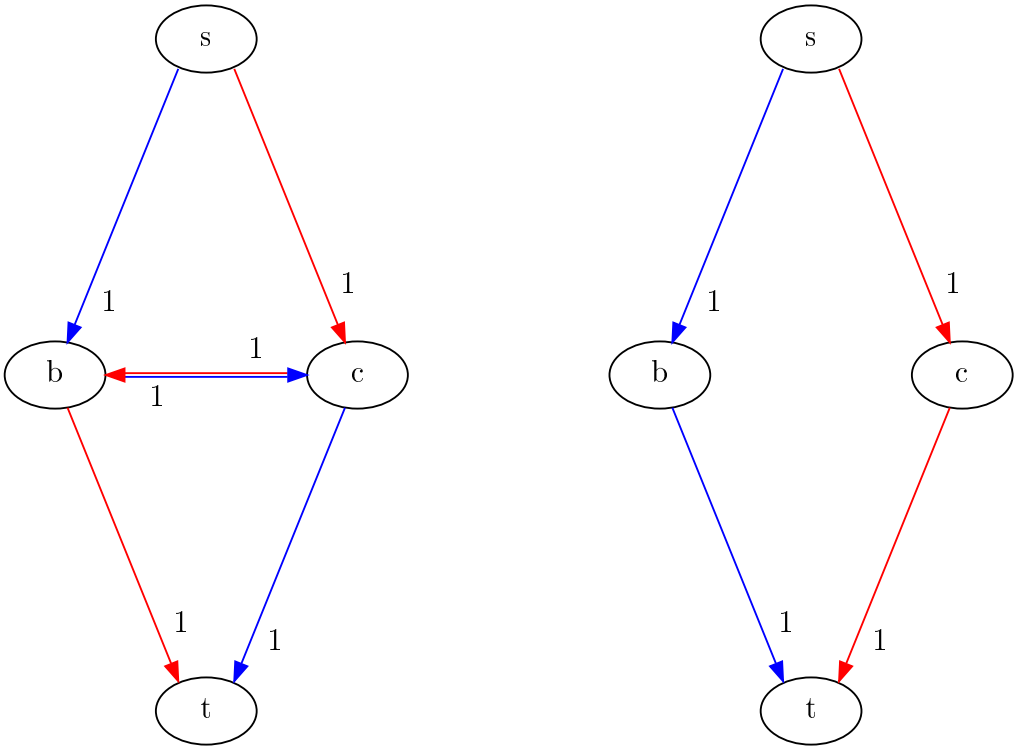
3.2.1 Der Algorithmus findet keine unmögliche Lösungen

Das unser naiver Algorithmus keine unmögliche Lösungen gibt, ist trivial zu sehen. Alle Pfade können nacheinander ausgeführt werden, und gehen von der Quelle bis zur Senke.

Haben wir nun eine Kante, dessen Restkante wir benutzen, können wir das immer so umstellen, dass wir dies nichtmehr tun. Benutzen wir die Kante (a, b) mit Gewicht x, und rückwärts mit Gewicht y, wissen wir das $x \geq y$ aufgrund der Regel das Kanten im Restnetzwerk immer so viel Platz haben, wie Platz auf der Kante im Ausgangsflussnetzwerk bereits verbraucht wurde.

Wir wissen nun, dass man vom Vertex b mindestens x Wasser zu t befördern können, und vom Vertex a mindestens y Wasser zu t befördern kann. Grund dafür ist, dass wir Pfade, die in t enden, gefunden haben die durch (a, b) gehen mit der Gewichtssumme x und Pfade, die durch (b, a) gehen mit der Gewichtssumme y.

Jetzt verändern wir die Pfade so, dass wir die Restkante nicht benutzen. Wir schicken nur noch $x-y$ Wasser durch (a, b) und kein Wasser durch (b, a). Die Menge an Wasser, die von a abfliessen muss, ändert sich nicht, da von b nun y weniger kommt, aber auch y weniger nach b weggeschickt wird. Genau gleich muss nicht mehr Wasser von b abfliessen, da jetzt zwar y weniger nach a geschickt wird, aber auch y weniger von a kommt.



3.2.2 Der Algorithmus findet die optimale Lösung

Das können wir mit enem Beweis durch Wiederspruch belegen. Wir geben eine Lösung aus, sobald wir keinen Pfad mehr finden können von s nach t im Gesamtnetzwerk (Ausgangsnetzwerk + Restnetzwerk).

Die Menge aller Punkte, die wir von s erreichen können, nennen wir C.

Wir haben keine Kanten oder Restkanten, die aus C hinausführen, und noch Kapazität haben. Hätten wir so eine Kante, könnten wir den Zielpunkt von s aus erreichen, und somit wäre der Punkt auch in C -> die Kante führt nicht aus C hinaus.

Was bedeutet das genau? Alle Restkanten, die aus C hinausführen, sind voll, somit sind alle Ausgangskanten, die in C hineinführen, leer. Alle Ausgangskanten, die aus C hinausführen, sind voll, und ihre respektive Restkanten deshalb leer. Es fliesst kein Wasser in C hinein, es fliesst nur Wasser raus (Volle Restkanten heissen nicht das Wasser auf den Restkannten fliesst, siehe Ausgangssituation!). Somit ist C eine Menge von Punkten, die einen Schnitt bilden.

C ist ausserdem minimal. Wäre C nicht minimal, wäre es unmöglich den Fluss von s nach t aufzubauen, den wir aufgebaut haben. Schon vorher hätten wir keinen Pfad mehr finden müssen. Das wir keine unmögliche Lösungen finden, ist schon bewiesen.

3.3 Edmond-Karp Algorithmus

Die Ford-Fulkerson Methode gibt keinen Algorithmus vor, einen Weg von s nach t zu finden. Der Edmond-Karp Algorithmus identisch zum Ford-Fulkerson, es findet aber den kürzesten Weg von s nach t mit einer Breitensuche.

```
Data: g ist das Gesamtflussnetzwerk. Dabei ist g[i] eine Liste von
      Kanten und Restkanten, die aus dem Vertex i führen. g hat n
      Vertex, und 2m Kanten. m Normale, und m Restkanten. s
      Quelle t Senke resultat ← 0
while true do
    q ;                    /* Eine Queue für die Tiefensuche */
    prev ;      /* Ein Array der Länge n, die beinhaltet mit
     welcher Kante wir zu diesem Vertex gekommen sind */
    q.push(s);
    while q.size() > 0 do
        x ← q.pop();
        foreach Kante k ∈ g[x] do
            if prev[k.ziel] = null ∧ k.ziel ≠ s ∧ k.kapazitaet >
             k.wasser then
                prev[k.ziel] ← k;
                q.push(k.ziel);
            end
        end
    end
    if prev[t] = null then
        break;
    end
    /* Wir haben einen Pfad gefunden                          */
    wassermenge ← ∞;
    for Kante k ← prev[t]; k ≠ null; k ← prev[k.start] do
        wassermenge ← min(wassermenge, k.kapazitaet − k.wasser);
    end
    for Kante k ← prev[t]; k ≠ null; k ← prev[k.start] do
        k.wasser ← k.wasser + wassermenge;
        k.rest.wasser ← k.rest.wasser − wassermenge;
    end
    resultat ← resultat + wassermenge;
end
```

Algorithm 2: Edmond-Karp Algorithmus

4 Push-relabel Algorithmus

Dieser Algorithmus ist ein sehr spezieller. Es ist eines der schnellsten Algorithmen für Max-Flow-Min-Cut Probleme, und funktioniert ganz anderst wie die Ford-Fulkerson Methode. Während der Ford-Fulkerson in jedem Schritt einen möglichen Fluss erstellt, sind in den Zwischenschritten vom Push-relabel Algorithmus keine Flüsse von s nach t zu finden. In den Zwischenschritten sind sogenannte Preflow's. Das heisst, das in die Vertex mehr Wasser reinfliesst wie hinaus. Und in den Schritten wird es neubalanciert, so das sich das ausgleicht.

4.1 Die Idee

Wir stellen uns den Flussnetzwerk g mit $2n$ (wobei n die Anzahl Knoten ist) Rängen r vor. s startet auf dem $n$ten Rang, t auf dem nullten Rang. Diese zwei Vertex bleiben auf diesen Rängen. Alle andere Vertex starten auch vom nullten Rang, und können sich aber im Verlauf des Algorithmus bis auf den $2n$ten Rang bewegen.

Folgende zwei Eigenschaften werden in jedem Schritt immer zutreffen:

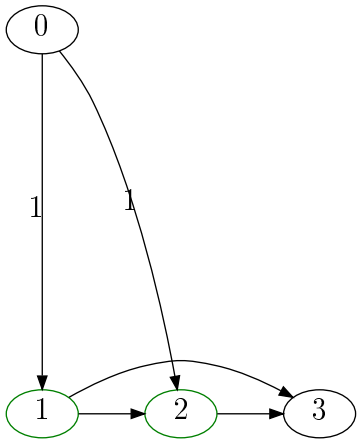
1. Wenn auf einer Kante $k = (a, b)$ Wasser fliesst ($k.w > 0$), ist $r[k.a]+1 \geq r[k.b]$, das heisst, eine Kante mit Wasser kann höchstens einen Rang nach oben gehen. Nach unten darf es beliebig Weit. Ist $r[k.a] + 1 = r[k.b]$, wird ausserdem k als enge Kante bezeichnet.

2. Wenn auf einer Kante $k = (a, b)$ weniger Wasser fliesst wie es Kapazität hat ($k.w < k.c$), ist $r[k.a] \leq r[k.b] + 1$, das heisst, die Kante darf höchstens einen Rang nach unten gehen. Gegen oben darf es beliebig Weit. Ist $r[k.a] = r[k.b] + 1$, wird ausserdem k als enge Kante bezeichnet.

Auf enge Kanten müssen wir gut achten im Verlauf des Algorithmus, da man sonst versehentlich eines der zwei Eigenschaften verletzen könnte.

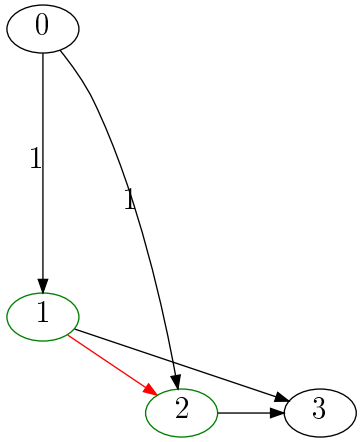
Wir sagen einem Vertex v aktiv, falls mehr Wasser reinfliesst wie raus. In einer Liste uf halten wir fest, wie viel der Überfluss in einem Vertex ist.

Am Anfang wird entlang allen Kanten $k \in g[s]$ die Menge $k.c$ geschickt.



Auf der Abbildung ist die Ausgangssituation zu sehen, die aktiven Vertex sind mit grün markiert.

In einem Schritt wählen wir eines der aktiven Vertex aus, die sich am höchsten befinden. Wenn wir dessen Rang um eins erhöhen können, ohne die zwei Eigenschaften zu brechen, erhöhen wir es um eins.



Jetzt ist eine enge Kante entstanden, dies markieren wir mit rot. Der aktive Vertex mit dem grössten Rang ist jetzt die 1.

Wir schauen uns die Kanten die in v enden, und die Kanten die aus v stammen der Reihe nach an. Finden wir eine enge Kante k (wie in (1, 2) in unserem Fall), dann dann ändern wir den Fluss entlang der engen Kante folgenderweise:

- Auf der engen Kante $k = (q, v)$ fliesst Wasser ($k.w > 0$), und $r[q] + 1 = r[v]$ trifft zu (Eigenschaft 1).

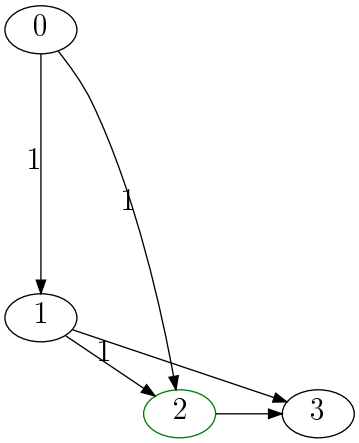
  Auf dieser Kante reduizeren wir die Wassermenge um $\min(k.w, uf[v])$. Entweder nehmen wir alles Wasser von der Kante weg, oder so viel Wasser, das der Überfluss in v gleich null wird. Im ersten Fall ist die Kante k nicht mehr eng, und wir schauen die Kanten von v der Reihe nach weiter an. Im zweiten Fall ist v nicht mehr ein aktiver Vertex (es hat keinen Überfluss mehr). Wir hören auf die Kanten von v zu aktualisieren, und wir beenden diesen Schritt.

- Auf der engen Kante $k = (v, q)$ hat es noch Platz($k.w < k.c$), und $r[k.a] = r[k.b] + 1$ trifft zu (Eigenschaft 2).

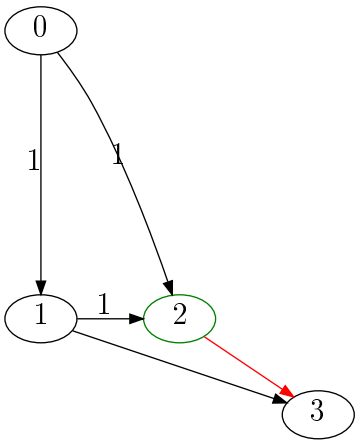
  Auf dieser Kante heben wir die Wassermenge an um $\min(k.c - k.w, uf[v])$. Entweder schicken wir so viel Wasser auf die Kante, wie es noch Kapazität übrig hat. In diesem Fall ist sie nicht mehr eng, und wir schauen die nächste Kante von v an. Im zweiten Fall schicken wir den ganzen Überfluss in v entlang k. Somit verliert v den Aktivitätsstatus, und wir beenden diesen Schritt.

In allen Fällen wird q aktiv, da es neues Wasser bekommt, oder Wasser das es weggeschickt hat zurückerhaltet.

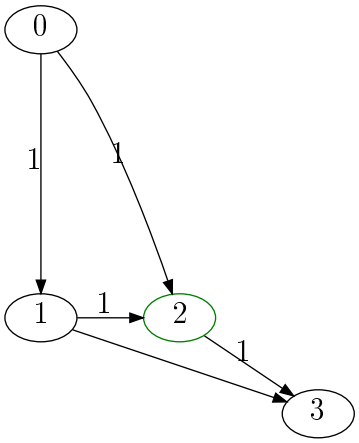
Schauen wir uns die Schritte auf dem Beispiel an. Wir nehmen die Kante (1, 2) aus der Vorherigen abbildung. Wir schicken den Überfluss von 1 entlang der Kante. So verliert 1 den Aktivitätsstatus, und die Kante ist nicht mehr eng. Der Vertex 2 bleibt aktiv.



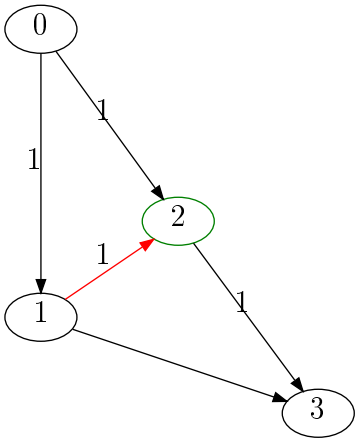
Wir machen den Schritt nochmals. Wir nehmen einen aktiven Vertex (2). Es hat keine enge Kanten die in 2 gehen oder von 2 kommen, also heben wir es an. Die Kante (2, 3) wird eng.



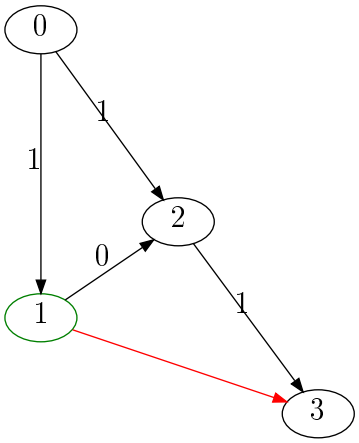
Wir erhöhen den Fluss durch die enge Kante (2, 3) um 1. Die 2 bleibt aktiv.



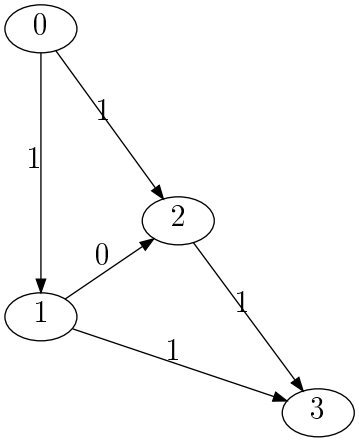
Jetzt kann der Rang vom Vertex 2 erhöht werden. Somit wird die Kante (1, 2) eng.



Wir schauen uns die enge Kante nun an. Wir reduzieren den Fluss entlang der Kante. Der Vertex 2 ist nun nicht mehr aktiv, aber die 1 wird aktiv.



Wir schauen jetzt die Kanten aus 1 an. Die Kante (1, 3) ist eng. Wir erhöhen dessen Fluss um 1.



Wir haben keine aktive Punkte mehr, also kein Überfluss. Im Vertex 3 kommt der maximale Fluss an.

Der Algorithmus bleibt stehen, sobald wir kein aktive Punkte mehr haben. In diesem Fall haben wir einen maximalen Fluss von s nach t.

4.2 Beweise

4.2.1 Der Algorithmus findet keine unmögliche Lösungen

Es fliesst in alle Punkte gleich viel Wasser hinein wie hinaus, ausser in s und in t da sie die Quelle und Senke sind. Es fliesst in keinen Punkt mehr Wasser

rein wie hinaus, da es sonst ein aktiver Punkt wäre und der Algorithmus noch nicht gehalten hätte. Und es fliesst nie weniger Wasser in einen Punkt wie hinaus, das so eine Situation nur mit dem Punkt s möglich ist.

4.2.2 Der Algorithmus findet die optimale Lösung

Für die optimalität zeigen wir wie vorher eine dazugehörigen minimalen Schnitt.

Da s auf dem $n$ten Rang ist, und t auf sich auf dem nullten befindet, hat es $n-1$ unbenutze Ränge dazwischen. Ausserdem hat es noch $n-2$ Punkte. Deshalb haben wir einen Rang i, auf dem sich kein Vertex befindet. Dieser Rang definiert den minimalen Schnitt.

Eine Kante, die von oben nach unten geht, und nicht voll ist, könnte nur einen Rang gegen unten gehen. Das heisst, das alle Kanten die diesen leeren Rang überqueren nach unten, voll sind. Eine Kante wo Wasser hat, dürfte nur einen Rang gegen oben gehen. Deshalb ist jede Kante die den leeren Rang nach oben überquert leer. Deshalb ist es ein Schnitt. Schneidet man die Kanten nach unten, ist das Netzwerk auseinandergelegt.

Gleichzeitig ist der Wert dieses Schnittes gleich dem Wert des gefundnen Flusses. Das kann kommt nur mit dem minimalen Schnitt und dem maximalen Fluss vor.

4.2.3 Der Algorithmus hält in allen Fällen

Wenn alle Kanten aus s gegen oben gehen, ausser die letze, die vom obersten Punkt nach t geht, hat es maximal $n(RangvonderQuelles) + (n-2)(AnzahlPunkteaussersundt) = 2n-2$ Ränge. Im Verlauf des Algorithmus erhält ein Punkt nie einen kleineren Rang wie vorher. Der Algorithmus kann weniger wie $(n-2)(2n-2)$ Hebeschritte tätigen. Zwischen zwei Hebeschritten werden alle Kanten maximal einmal angeschaut. Das führt uns zu einer Laufzeit von maximal $O(n^2 \cdot m)$, wo n die Anzahl Knoten, und m die Anzahl Kanten sind. Dies bewiest ein halten des Programmes. Mit etwas schwierigeren Methoden kann man auch eine Maximallaufzeit von $O(n^3)$ zeigen.

4.3 Implementation

–

```
Data: g ← Flussnetzwerk. Dabei ist g[i] eine Liste von Kanten, die
      aus dem Vertex i führen, oder in den Vertex i führen. g hat n
      Vertex.
      s ← Quelle
      t ← Senke
      ueberfluss ← array(laenge = n, startwert = 0)
      rang ← array(laenge = n, startwert = 0)
      aktiv ← priorityqueue
      /* Diese priority queue wird alle aktive Knoten
      beinhalten, und sie nach dem Rang sortieren          */
/* Hier initializieren wir alle Kanten aus s, damit sie
      voll sind                                            */
foreach Kante k ∈ g[s] do
    if k.ziel = s then
        continue;
    end
    if k.ziel ≠ t then
        aktiv.push(k.ziel);
    end
    ueberfluss[k.ziel] ← ueberfluss[e.ziel] + k.c;
    k.w ← k.c + k.w;
end
```

```
/* Hier machen wir die Schritte, bis nur noch t ein
   aktiver Knoten ist                                          */
while true do
    v ← aktiv.pop();
    if = then aktiv.size(
     | 0
    end
    ∧ v == t)  break;
    doublebreak ← false;          /* Damit wir aus der inneren
     Schlaufe weitergehen können */
    foreach Kante k ∈ g[v] do
        ausgehend ← k.start = v;
        q ← ausgehend ? k.ziel | k.start;
        if rang[q] >= rang[v] then
            /* Diese Kante ist nicht eng, da q > v          */
            continue;
        end
        if ausgehend then
            if k.w = k.c then
                /* Wenn die Kante voll und ausgehend ist, ist
                   sie nicht eng                             */
                continue;
            end
            /* Wenn q noch nicht aktiv ist, aktiv setzen    */
            if ueberfluss[q] = 0 ∧ q ≠ s ∧ q ≠ t then
                aktiv.push(q);
            end
            restc ← k.c - k.w;
            if ueberfluss[v] ≤ restc then
                k.w ← k.w + ueberfluss[v];
                ueberfluss[q] ← ueberfluss[q] + ueberfluss[v];
                ueberfluss[v] ← 0;
                break; /* Wir schicken den ganzen Überfluss
                   entlang der Kante                         */
            end
            ueberfluss[v] ← ueberfluss[v] - resc;
            k.w ← k.c;
            ueberfluss[q] ← ueberfluss[q] + resc;
            /* Wir schicken so viel Wasser entlang der Kante
               bis es voll ist                               */
        end
```

```
    if k.w = 0 then
        continue; /* Wenn die Kante keinen Fluss hat ist
           sie nicht eng                                    */
    end
    /* Wenn q noch nicht aktiv ist, aktiv setzen          */
    if ueberfluss[q] = 0 ∧ q ≠ s ∧ q ≠ t then
        aktiv.push(q);
    end
    if ueberfluss[v] ≤ k.w then
        ueberfluss[q] ← ueberfluss[q] + ueberfluss[v];
        k.w ← k.w - ueberfluss[v];
        ueberfluss[v] ← 0;
        doublebreak ← true;
        continue;
        /* Wir schicken den ganzen Überfluss entlang der
           Kante zurück                                     */
    end
    ueberfluss[q] ← ueberfluss[q] + k.w;
    ueberfluss[v] ← ueberfluss[v] - k.w;
    k.w ← 0;
    /* Wir schicken so viel Wasser entlang der Kante
       zurück bis es leer ist                             */
  end
  if doublebreak then
    continue;
  end
  ++rang[v];
  aktiv.push(v)
end
resultat ← ueberfluss[t];
```

Algorithm 3: Push-relabel Algorithmus

5 Zusammenfassung

Ich finde, dass Beispiel des maximum Fluss Problems in einem Flussnetzwerk zeigt gut wie man Algorithmen entwickelt. Man versucht sich eine Lösung zu überlegen. Findet man eine vermeintliche, versucht man es zu Beweisen. Widerlegt man sie, hat man einen guten Anhaltspunkt dafür, wie man es verbessern kann.

Es ist faszinierend das es trotzdem so unterschiedliche Lösungen gibt. Es zeigt aber trotzdem zwei naheliegende Ansatzweisen. Der Ford-Fulkerson sucht einen Fluss von der Quelle zur Senke, und verbessert sie nach und nach. Der Push-relabel fängt mit einem maximalen Fluss an von s, jedoch nicht bis nach t, sondern nur zu den Nachbarsknoten. Von dort aus weitet es den maximalen Fluss allmählich aus, bis es t erreicht.