

Inhaltsverzeichnis

1	Grundlagen	2
1.1	Graphen und Digraphen	2
1.2	Bäume und Wälder	2
1.3	Handshaking-Lemma	2
1.4	Planare Graphen	2
1.5	Topologische Sortierung	2
2	Suche in Graphen	3
2.1	Breitensuche (BFS)	3
2.2	Tiefensuche (DFS)	3
3	Minimal aufspannende Bäume	4
3.1	Kruskal	4
3.2	Prim	4
4	Kürzeste Wege	4
4.1	Dijkstra	4
4.2	Moore-Bellman-Ford	4
4.3	A*	5
4.4	Dynamische Programmierung - Rucksackproblem (KSP)	5
5	Flüsse in Netzwerken	6
5.1	Max-Flow-Min-Cut-Theorem	6
5.2	Edmonds-Karp	6
5.3	Push-Relabel	7
6	Matchings	7
6.1	Blossom-Shrinking	8
7	Euler- & Hamiltonkreise	8
7.1	Eulertour & -kreis	8
7.2	Chinesisches Postboten-Problem (CPP)	8
7.3	Hamiltonkreis	8
7.4	Travelling-Salesman-Problem (TSP)	8
8	Färbung von Graphen	8
8.1	Färbung planarer Graphen	8
8.2	Heuristiken zur Graphenfärbung	8

1 Grundlagen

1.1 Graphen und Digraphen

Graph G

$$G = (V, E)$$

wobei V eine endliche, nichtleere Menge an Knoten oder Ecken (Vertices) und E eine Menge Kanten (Edges), gegeben durch ungeordnete Paare von Knoten (v, w) , $v, w \in V$, ist.

Digraph D

$$D = (V, A)$$

wobei V eine endliche, nichtleere Menge an Knoten oder Ecken (Vertices) und A eine Menge von gerichteten Bögen (Arcs) $A \subseteq V \times V$ mit Gewichten $c \in \mathbb{R}$. Ein Digraph heißt konservativ, wenn er keinen Kreis mit negativem Gesamtgewicht enthält.

Grad eines Knotens v

$\deg(v) :=$ Grad von v - Anzahl zu Knoten v inzidenter (benachbarter) Kanten in einem Graphen

$\deg_+(v), \deg_{in}(v) :=$ Eingangsgrad von v - Anzahl aller auf v zeigenden Bögen in einem Digraphen

$\deg_-(v), \deg_{out}(v) :=$ Ausgangsgrad von v - Anzahl aller von v ausgehenden Bögen in einem Digraphen

Kantenzug/Kette/Pfad P

$$P = (v_0, e_1, v_1, e_2, v_2, \dots, e_k, v_k), v_i \in V, e_i \in E$$

ist eine Sequenz zusammenhängender Kanten.

Weg

ist ein Kantenzug, bei dem alle Kanten (paarweise) verschieden sind, also keine wiederholt wird.

Kreis/Zyklus

ist ein geschlossener Weg mit identischem Start und Zielknoten.

vollständiger Graph K_n

ist der Graph aus n Knoten, der jeden Knoten mit jedem anderen Knoten verbindet.

bipartiter Graph

Die Knotenmenge V kann in zwei disjunkte Teilmengen V_1 und V_2 aufgeteilt werden, sodass jede Kante einen Endpunkt in V_1 und V_2 hat.

Satz von König

Ein Graph ist genau dann bipartit, wenn er keinen Kreis ungerader Länge enthält.

1.2 Bäume und Wälder

Wald F

ist ein Graph, der keinen Kreis enthält.

Baum T in einem Graphen G

ist ein zusammenhängender Wald. T heißt aufspannend, wenn er alle Knoten von G enthält.

1.3 Handshaking-Lemma

In einem Graphen $G = (V, E)$ gilt:

$$\sum_{v \in V} \deg(v) = 2|E|$$

Daraus folgt implizit, dass die Anzahl der Knoten mit ungeradem Grad gerade ist.

1.4 Planare Graphen

Ein planarer Graph kann auf einer zweidimensionalen Fläche kreuzungsfrei gezeichnet werden. Jeder Kreis, jeder Baum und der vollständige Graph K_4 sind planar. Der vollständige Graph K_5 und der vollständige bipartite Graph $K_{3,3}$ sind nicht planar.

Satz von Kuratowski

ein Graph ist genau dann nicht planar, wenn er durch Kontraktion von Kanten in den K_5 oder den $K_{3,3}$ überführt werden kann.

Eulersche Polyederformel

$$n - m + f = 2$$

wobei n die Anzahl der Knoten, m die Anzahl der Kanten und f die Anzahl der Flächen in einem zusammenhängenden planaren Graphen ist.

Für nicht zusammenhängende Graphen gilt

$$n - m + f = k + 1$$

wobei k die Anzahl der Zusammenhangskomponenten ist.

Aus der Polyederformel folgt unmittelbar für planare Graphen aus 3 oder mehr Knoten:

$$m \leq 3n - 6$$

$$f \leq 2n - 4$$

Ist der Graph außerdem noch bipartit gilt für die Anzahl der Kanten:

$$m \leq 2n - 4$$

Insbesondere hat jeder planare Graph mindestens einen Knoten von Grad höchstens 5.

1.5 Topologische Sortierung

Eine topologische Sortierung eines Digraphen $D = (V, A)$ ist eine injektive Abbildung $f : V \Rightarrow \mathbb{N}$, sodass gilt:

$$(v, w) \in A \Rightarrow f(v) < f(w)$$

Ein Digraph hat genau dann eine topologische Sortierung, wenn er keinen gerichteten Kreis enthält.

TopSort

input: ein beliebiger Digraph
 output: eine topologische Sortierung, gegeben durch die
 Knoten-Index-Beziehung oder die Aussage, dass ein Kreis vorliegt

```

index = 0
sort()

while( es ist ein Knoten v mit Eingangsgrad > 0 vorhanden )
    entferne v und aktualisiere die Eingangsgrade seiner Nachbarn
    sort(v) = index
    index++

if( es sind noch Knoten übrig )
    return false, es liegt ein Kreis vor
else
    return sort
  
```

2 Suche in Graphen

Suchverfahren für Graphen traversieren alle von einem Startknoten s aus erreichbaren Knoten eines Graphen $G = (V, E)$ nach einem rekursiven Schema. Jedem traversierten Knoten wird sein Index (Reihenfolge der Traversierung), sein Vorgänger und sein Abstand zum Startknoten s zugeordnet. Aus den Vorgänger-Beziehungen lässt sich ein aufspannender Baum von G mit Wurzel s rekonstruieren.

Außerdem gilt für die Breitensuche: ein zusammenhängender Graph ist genau dann bipartit, wenn es bei der Breitensuche mit beliebigem Startknoten keine Nicht-Baumkante e gibt, die eine Querverbindung im Spannbaum darstellt (e verbindet zwei Baumknoten auf gleicher Höhe des Baumes).

2.1 Breitensuche (BFS)

input: ein zusammenhängender Graph G ,
 ein Startknoten s
 output: Index der Traversierung, Vorgänger und Abstand von s für
 jeden von s aus erreichbaren Knoten

```

counter = 1
index()
distance()
predecessor()
queue

foreach( Knoten v von G )
    if( v = s )
        index(v) = 1
        distance(v) = 0
        predecessor(v) = undefined
        füge v in queue ein
  
```

```

else
    index(v) = undefined
    distance(v) = undefined
    predecessor(v) = undefined

while( queue ist nicht leer )
    v = head(queue)
    if( v hat einen noch nicht untersuchten Nachbarn w )
        markiere w als untersucht
        füge w am Ende der queue ein
        index(w) = ++counter
        predecessor(w) = v
        distance(w) = distance(v) + 1
    else
        entferne v aus der queue

return index, predecessor, distance
  
```

2.2 Tiefensuche (DFS)

input: ein zusammenhängender Graph G , ein Startknoten s
 output: Index der Traversierung, Vorgänger und Abstand von s für
 jeden von s aus erreichbaren Knoten

```

counter = 1
index()
distance()
predecessor()
stack

foreach( Knoten v von G )
    if( v = s )
        index(v) = 1
        distance(v) = 0
        predecessor(v) = undefined
        lege v auf stack
    else
        index(v) = undefined
        distance(v) = undefined
        predecessor(v) = undefined

while( stack ist nicht leer )
    v = peek(stack)
    if( v hat einen noch nicht untersuchten Nachbarn w )
        markiere w als untersucht
        lege w auf stack
        index(w) = ++counter
        predecessor(w) = v
        distance(w) = distance(v) + 1
    else
  
```

```
nimm v von stack

return index, predecessor, distance
```

3 Minimal aufspannende Bäume

Ein minimal aufspannender Baum, ist der aufspannende Baum, mit der kleinsten Gesamtsumme der Kantengewichte. Jeder aufspannende Baum mit n Knoten hat genau $n-1$ Kanten. Entfernt man eine Kante e , ist der aufspannende Baum nicht mehr zusammenhängend, sondern in zwei Zusammenhangskomponenten zerlegt - den zu e gehörigen Fundamentalschnitt. Fügt man dem aufspannenden Baum eine Kante k hinzu, enthält der Baum nun einen Kreis - den zu k gehörigen Fundamentalkreis.

3.1 Kruskal

input: ein zusammenhängender Graph mit Kantengewichten
output: ein minimal aufspannender Baum

```
sortiere alle Kanten aufsteigend nach ihren Gewichten
counter = 0
tree
```

initialisiere für jeden Knoten v eine Zusammenhangskomponente V ,
die nur v enthält

```
foreach( Kante e der aufsteigend sortierten Kanten )
    if( e verbindet zwei unterschiedliche Komponenten V und W )
        verschmilz V und W
        füge e zu tree hinzu
        counter++
    if( counter = Anzahl aller Knoten - 1 )
        return tree
```

3.2 Prim

input: ein zusammenhängender Graph mit Kantengewichten,
ein Startknoten s
output: ein minimal aufspannender Baum gegeben durch eine
Vorgänger-Beziehung

```
predecessor()
distance()
```

```
foreach( Knoten v des Graphen )
    predecessor(v) = undefined
    if( v = s )
        distance(v) = 0
    else
        distance(v) = infinity
```

```
while( noch nicht alle Knoten erreichbar )
    v = noch nicht erreichbarer Knoten mit minimalem dist(v)
    foreach( noch nicht erreichbarer Nachbar w von v )
        if( distance(w) > Kantengewicht(v,w) )
            distance(w) = Kantengewicht(v,w)
            predecessor(w) = v
    markiere v als erreichbar
```

4 Kürzeste Wege

4.1 Dijkstra

input: ein gewichteter Digraph mit nichtnegativen Kantengewichten,
ein Startknoten s
output: die kürzesten Wege von s zu jedem anderen erreichbaren Knoten,
gegeben durch eine Vorgänger Beziehung,
die absoluten Distanzen von s zu jedem anderen Knoten

```
predecessor()
distance()
```

```
foreach( Knoten v des Graphen )
    predecessor(v) = undefined
    if( v = s )
        distance(v) = 0
    else
        distance(v) = infinity
```

```
while( es gibt nicht markierte Knoten v mit distance(v) < infinity )
    v = nicht markierter Knoten mit minimalem distance(v) Wert
    markiere v
    foreach( unmarkierter Nachbar w von v )
        if( distance(w) > distance(v) + Kantengewicht(v,w) )
            distance(w) = distance(v) + Kantengewicht(v,w)
            predecessor(w) = v
```

4.2 Moore-Bellman-Ford

input: ein konservativer Digraph,
ein Startknoten s
output: die kürzesten Wege von s zu jedem anderen erreichbaren Knoten,
gegeben durch eine Vorgänger Beziehung,
die absoluten Distanzen von s zu jedem anderen Knoten

```
predecessor()
distance()
```

```
foreach( Knoten v des Graphen )
    predecessor(v) = undefined
```

```

if( v = s )
    distance(v) = 0
else
    distance(v) = infinity

repeat( Anzahl Knoten - 1 ) times
    foreach( Kante e = (v,w) des Graphen )
        if( distance(w) > distance(v) + Kantengewicht(v,w) )
            distance(w) = distance(v) + Kantengewicht(v,w)
            predecessor(w) = v

```

4.3 A*

Der A*-Algorithmus ist eine optimierte Variante des Algorithmus von Dijkstra. Die Breitensuche, über die Dijkstra sich den Graphen erschließt, ist für Graphen mit einer großen Anzahl an Knoten speicherplatzkritisch. Der A*-Algorithmus löst dieses Problem, indem er nicht, wie Dijkstra, immer den nächsten Knoten als Kandidaten wählt, sondern den nächsten Knoten, der außerdem die geschätzt kürzeste Distanz zu einem gewünschten Zielknoten hat. Diese Schätzung erfolgt über eine heuristische Schätzfunktion $h: V \rightarrow \mathbb{R}^+$. Damit der A*-Algorithmus korrekt arbeitet, muss h folgende Bedingungen erfüllen:

$$h(t) = 0$$

$$h(v) \leq c(v, w) + h(w), \quad \forall (v, w) \in A$$

wobei t den Zielknoten, c das Kantengewicht von Knoten v zu Knoten w und A die Menge aller Bögen im Graphen beschreibt.

input: ein zusammenhängender, gewichteter Digraph
mit nichtnegativen Kantengewichten,
eine heuristische Schätzfunktion heuristic,
ein Startknoten s ,
ein Zielknoten t

output: die kürzesten Wege von s zu jedem anderen erreichbaren Knoten,
gegeben durch eine Vorgänger Beziehung,
die absoluten Distanzen von s zu jedem anderen Knoten

```

predecessor()
distance()

```

```

foreach( Knoten v des Graphen )
    predecessor(v) = undefined
    if( v = s )
        distance(v) = 0
    else
        distance(v) = infinity

```

```

while( es gibt einen nicht markierten Knoten v )
    v = nicht markierter Knoten mit minimalem distance(v) + heuristic(v)

```

```

markiere v
if( t ungleich v )
    foreach( nicht markierten Nachbarn w von v )
        if( distance(w) > distance(v) + Kantengewicht(v,w) )
            distance(w) = distance(v) + Kantengewicht(v,w)
            predecessor(w) = v

```

4.4 Dynamische Programmierung - Rucksackproblem (KSP)

Das Rucksackproblem ist ein doppeltes Optimierungsproblem. Es gibt $n \in \mathbb{N}$ Elemente, die je über einen gewissen Profit $c_k \in \mathbb{N}$ und ein gewisses Gewicht $a_k \in \mathbb{N}$ verfügen. Zusätzlich ist ein Maximalgewicht $b \in \mathbb{N}$, das nicht überschritten werden darf, definiert. Es gilt nun die Elemente so auszuwählen, dass der meiste Nutzen entsteht.

Beim binären Rucksackproblem können Elemente nur ausgewählt oder weggelassen werden. Dagegen können beim allgemeinen Rucksackproblem auch mehrere Elemente von einem Typ ausgewählt werden.

Das binäre Rucksackproblem lässt sich mithilfe dynamischer Programmierung lösen. Dabei wird ein mehrdimensionales Optimierungsproblem in leichter zu lösende Teilprobleme zerlegt und deren Lösungen gespeichert, um diese später beim Zusammensetzen der Einzellösungen wiederverwenden zu können. Dem Modell unterliegt ein kreisfreier, lexikographisch sortierter Digraph, der alle möglichen Kombinationen der Elemente darstellt. Ein Knoten entspricht einem Tupel (k, β) , wobei k zwischen 0 und n , β zwischen 0 und b liegt. Außerdem wird jedem Knoten der bisher akkumulierte Nutzen z_k zu gewiesen (Eintragungen in der Tabelle). Es gibt nun drei mögliche Arten von Kanten in der Tabelle:

- Kante 1 nach rechts ($\beta \rightarrow \beta + 1$, k bleibt gleich) - es wird kein weiteres Element ausgewählt und der akkumulierte Nutzen bleibt gleich.
- Kante 1 nach unten (β bleibt gleich, $k \rightarrow k + 1$) - ein neues Element $_{k+1}$ wird betrachtet aber nicht ausgewählt und der akkumulierte Nutzen bleibt gleich.
- Kante a_{k+1} nach rechts, 1 nach unten ($\beta \rightarrow \beta + a_{k+1}$, $k \rightarrow k + 1$) - ein neues Element $_{k+1}$ wird betrachtet, ausgewählt und der akkumulierte Nutzen erhöht sich.

Der akkumulierte Nutzen z_k ist das Maximum aus zwei möglichen Fällen: entweder der Wert $z_{k-1}(\beta)$, also der z -Wert der Zellen 1 über der zu bestimmenden Zelle, oder der Wert $z_{k-1}(\beta - a_{k-1})$ (, wenn er überhaupt existiert), also der Wert der Zelle, von der eine Diagonalkante zu der zu bestimmenden Zelle führt.

c_k	a_k	$k \backslash \beta$	1	2	3	4	5	6	7	8	9	10	11	12	13
-	-	0	0	0	0	0	0	0	0	0	0	0	0	0	0
10	1	1	10	10	10	10	10	10	10	10	10	10	10	10	10
80	9	2	10	10	10	10	10	10	10	10	80	90	90	90	90
40	5	3	10	10	10	10	40	50	50	50	80	90	90	90	90
30	4	4	10	10	10	30	40	50	50	50	80	90	90	90	110
22	3	5	10	10	22	32	40	50	52	62	80	90	90	102	112

5 Flüsse in Netzwerken

Ein (Fluss-)Netzwerk $N = (D, c, s, t)$ ist ein Digraph $D = (V, A)$ mit einer Kapazitätsfunktion $c : A \mapsto \mathbb{R}_+$ und zwei ausgezeichneten Knoten s , genannt Quelle, und t , genannt Senke. N simuliert einen Fluss $f : A \mapsto \mathbb{R}_+$ (Wasser, Elektrizität, Verkehr, etc.) von der Quelle s zur Senke t .

Kapazitätsbedingung

Für alle Kanten aus A gilt: der Fluss $f(e)$ über eine Kante $e \in A$ ist nach oben beschränkt durch die Kapazität $c(e)$ der Kante.

Flusserhaltungsbedingung

Für alle Knoten aus V außer s und t gilt: die Summe des Zuflusses eines Knotens $v \in V$ (Summe der Flüsse aller in v eingehenden Kanten) muss gleich der Summe aller Ausflüsse (Summer der Flüsse aller aus v ausgehenden Kanten) sein.

Gesamtfluss $|f|$

Der Gesamtfluss $|f|$ ist die Differenz aller Ausflüsse und Einflüsse der Quelle s (alternativ die Differenz aller Einflüsse und Ausflüsse der Senke t).

s-t-Schnitt

Ein s-t-Schnitt ist eine Teilmenge S der Knotenmenge V , sodass S die Quelle s , nicht aber die Senke t enthält. Die Kapazität des Schnitts ergibt sich aus der Summe der Kapazitäten aller Kanten, die aus S herausragen - also Knoten aus S mit Knoten verbinden, die nicht in S enthalten sind.

augmentierender Weg

Ein augmentierender Weg ist ein Weg von s nach t , auf dem die Kapazität keiner Kante voll ausgeschöpft ist, also für jede im Weg enthaltene Kante e gilt $f(e) < c(e)$. Der Fluss entlang des Weges ist also noch nicht maximal und kann daher auf jeder Kante um die kleinste $c(e) - f(e)$ Differenz des ganzen Weges erhöht werden.

Residualgraph D_f

Der Residualgraph D_f zu einem Netzwerk N und einem Fluss f beschreibt durch Vorwärts- und Rückwärtskanten, um wie viel der Fluss jeder Kante von N erhöht und erniedrigt werden kann, sodass die Kapazitäts- und Flussbedingung erhalten bleiben. Für jede Kante in N , auf der der Fluss um Wert x erhöht werden kann, wird in D_f eine korrespondierende Vorwärtskante mit Gewicht x eingetragen. Für jeden auf einer Kante in N bereits existierenden Fluss mit Wert y wird eine korrespondierende Rückwärtskante mit Gewicht y eingetragt (der Fluss kann hier um y reduziert werden).

5.1 Max-Flow-Min-Cut-Theorem

Ein s-t-Fluss in einem Netzwerk ist genau dann maximal, wenn es keinen augmentierenden Weg mehr gibt. Der Wert eines maximalen s-t-Flusses stimmt mit dem Wert der minimalen Kapazität eines s-t-Schnitts überein.

5.2 Edmonds-Karp

Der Algorithmus von Edmonds und Karp erschließt sich systematisch kürzeste augmentierende Wege durch Breitensuche (`find_augmenting_path`) und erhöht dann den Fluss entlang dieser Wege so weit wie möglich (`augment_flow`). Wenn kein augmentierender Weg mehr gefunden werden kann, ist der Fluss maximal.

`find_augmenting_path()`

```
input:   ein Flussnetzwerk N,
        ein Fluss flow
output:  ein augmentierender Weg gegeben über die Vorgänger-Beziehung,
        ein minimaler Wert, um den der Fluss entlang des augmentierenden
        Weges erhöht werden kann,
        eine Orietierungsfunktion, die Knoten als über Vorwärts- oder
        Rückwärtskanten erreichbar markiert,
        ein korrespondierender s-t-Schnitt

predecessor
flow_delta()
orientation()
cut

markiere s, indem s cut hinzugefügt wird
foreach( Knoten v )
    flow_delta(v) = infinity

führe von s Breitensuche aus mit:
    if( neuer Knoten w ist über Vorwärtskante erreichbar )
        orientation(w) = +
        flow_delta(w) = Minimum aus flow_delta(predecessor(w)) und Wert,
                        um den der Zufluss von w erhöht werden kann
    else
        orientation(w) = -
```

```

        flow_delta(w) = Minimum aus flow_delta(predecessor(w)) und Zufluss
                        von w
    predecessor(w) = v
    markiere w, indem w cut hinzugefügt wird

```

```

return (predecessor, flow_delta(t), orientation, cut)

```

augment_flow()

```

input:  ein Flussnetzwerk,
        ein Fluss flow,
        ein augmentierender Weg gegeben über eine Vorgänger-Beziehung
        predecessor,
        ein minimaler Wert flow_delta, um den der Fluss entlang des
        augmentierenden Weges erhöht werden kann,
        eine Orientierungsfunktion, die Knoten als über Vorwärts- oder
        Rückwärtskanten erreichbar markiert
output: ein um flow_delta augmentierter Fluss

```

```

v = t
w = undefined

```

```

while( w ungleich s )
    w = predecessor(v)
    if( orientation(v) = + )
        flow(w,v) = flow(w,v) + flow_delta
    else
        flow(w,v) = flow(w,v) - flow_delta
    v = w

```

edmonds_karp()

```

input:  ein Flussnetzwerk N
output: ein maximaler Fluss,
        ein minimaler Schnitt

```

```

flow()
cut

```

```

foreach( Kante e des Netzwerkes )
    flow(e) = 0
cut = alle Knoten des Netzwerkes

```

```

while( Senke ist noch in cut enthalten )
    (path, flow_delta, orientation, new_cut) = find_augmenting_path(N, flow)

```

```

cut = new_cut
if( Senke ist noch in cut enthalten )
    flow = augment_flow(flow, path, flow_delta, orientation)

return flow, cut

```

5.3 Push-Relabel

Der Push-Relabel-Algorithmus erzeugt - im Gegensatz zum Algorithmus von Edmonds und Karp - nicht in jedem Schritt einen korrekten Fluss, sondern arbeitet mit sogenannten Präflüssen, einem "Arbeitsfluss", der noch zum maximalen Fluss ausgearbeitet werden muss. Im Präfluss darf der Zufluss eines Knotens größer als sein Abfluss sein, jedoch darf der Präfluss die Kapazitätsgrenzen der Kanten nicht übersteigen. Der Überschuss, der in einem Knoten v durch einen höheren Zu- als Abfluss entstehen kann, wird Exzess $e(v)$ genannt. Ein Knoten, dessen Exzess größer 0 ist, wird aktiv genannt. Ein Präfluss ist genau dann ein gültiger Fluss, wenn es keine aktiven Knoten gibt.

Idee des Push-Relabel-Algorithmus ist, von der Quelle den größtmöglichen Präfluss ins Netzwerk ausströmen zu lassen und alle (ungültigen) Exzesse sukzessive "zurückzuschieben", sodass ein maximaler gültiger Fluss entsteht. Damit nicht im Kreis zurückgeschoben wird, werden Knoten, die einen Exzess ausgeglichen haben, angehoben auf eine Höhe d . Die Höhe der Senke $d(t)$ ist 0.

Hierzu werden zu Beginn alle Abflüsse der Quelle auf deren Maxima (= Kapazitäten) und die Flüsse über alle anderen Kanten auf 0 gesetzt - somit sind alle Nachbarn der Quelle aktiv. Zudem wird die Quelle auf die Höhe $d(s) = |V|$ Anzahl der Knoten im Netzwerk angehoben. Alle anderen Knoten haben Höhe 0. Man wählt nun sukzessive (z.B. nach lexikographischer Ordnung) einen aktiven Knoten v aus bis es keinen aktiven Knoten mehr gibt und führt je eine der folgenden zwei Operationen aus:

Relabel , wenn alle Nachbarn, auf die v schieben kann, höher als v selbst liegen.

Push , wenn zumindest ein Nachbarn, auf den v schieben kann, mindestens ein Level niedriger liegt.

Relabel(v): hebe v ein Level höher als den niedrigsten all seiner höheren beschiebbaren Nachbarn.

Push(v,w): γ = Minimum aus Exzess $e(v)$ und schiebbarem Betrag zum niedrigsten Nachbarn w aller niedrigeren beschiebbaren Nachbarn von v . Schiebe γ -viel von v auf w und passe den Exzess von v und die Restkapazität der Kante (v, w) entsprechend an.

6 Matchings

Matching M

In einem Graphen $G = (V, E)$ nennt man eine Kantenteilmenge $M \subseteq E$ Matching, wenn keine zwei Kanten aus M einen gemeinsamen Endknoten haben. Ist ein Knoten v Endpunkt einer in M enthaltenen Kante, nennt man v M -gesättigt, ansonsten M -ungesättigt oder exponiert.

Ein Matching M heißt perfekt, wenn jeder Knoten in G M -gesättigt ist.

Ein Matching M heißt maximal, wenn es kein Matching M' in G gibt, das mehr Kanten enthält als M .

Alternierender Pfad P

In einem Graphen $G = (V, E)$, der ein Matching M enthält, ist ein alternierender Pfad ein Pfad, der nur aus Kanten besteht, die abwechselnd zu M und den Rest-Kanten $E \setminus M$ gehören. Ist in einem alternierenden Pfad der erste und der Letzte Knoten M -ungesättigt (hierzu muss der Pfad gerade Länge haben), nennt man den Pfad M -erweiternd oder M -augmentierend. Tauscht man in einem M -augmentierenden Pfad alle Kanten (flip) so, dass genau die Kanten zum Matching gehören, die davor zu den Rest-Kanten gezählt haben, und umgekehrt, erhält man einen alternierenden Pfad, dessen Enden gesättigt sind - man hat das Matching um die beiden vormals ungesättigten Endknoten des Pfades erweitert:

$$1 - 2 = 3 - 4 = 5 - 6 \quad \rightarrow \quad 1 = 2 - 3 = 4 - 5 = 6$$

Ein Matching M ist genau dann maximal, wenn es keinen M -augmentierenden Pfad mehr gibt.

Symmetrische Differenz

Seien M_1 und M_2 Matchings im Graphen $G = (V, E)$, dann ist die symmetrische Differenz von zwei Matchings definiert als:

$$M_1 \triangle M_2 = (M_1 \setminus M_2) \cup (M_2 \setminus M_1)$$

Für jede Zusammenhangskomponente K des Untergraphen $G_\triangle = (V, M_1 \triangle M_2)$ von G gilt eine der folgenden Eigenschaften:

- K ist ein Zyklus gerader Länge mit Kanten abwechselnd in M_1 und M_2 .
- K ist ein Weg, dessen Kanten abwechselnd in M_1 und M_2 liegen und dessen Endknoten in einem der beiden Matchings ungesättigt sind.

Heiratssatz von Hall

In einem bipartiten Graphen $G = (V = V_1 \uplus V_2, E)$ mit Bipartition V_1/V_2 gibt es genau dann ein Matching, dass jeden Knoten in V_1 sättigt, wenn für jede Teilmenge $U \subseteq V_1$ für die Menge der mit einem Element aus U adjazenten Knoten $N(U)$ gilt:

$$|U| \leq |N(U)|$$

6.1 Blossom-Shrinking

Der Blossom-Shrinking-Algorithmus findet ein maximales Matching M in einem Graphen $G = (V, E)$. Hierzu findet er durch Breitensuche M -augmentierende Pfade und tauscht (flip) deren Kanten wie oben beschrieben, um M um die Endknoten des jeweiligen Pfades zu erweitern.

Die Breitensuche startet immer vom lexikographisch kleinsten exponierten Knoten. Der entstehende Breitensuchbaum ist ein alternierender Baum, dessen Kantenebenen abwechselnd nur Matching-Kanten oder nur Rest-Kanten enthalten.

Wird ein ungerader Kreis, erkennbar als Nicht-Matching-Querkante oder -Rückwärtskante im alternierenden Baum, gefunden, liegt ein sogenanntes Blossom vor. Ein Blossom wird durch einen neuen Knoten v_* (lexikographisch 1 größer als der bisherige lexikographisch größte Knoten) ersetzt. v_* erhält alle in das Blossom eingehenden Kanten. Hierbei können Doppelungen auftreten - in diesem Fall kann jeweils irgendeine der doppelten Kanten ausgewählt werden.

Ziel ist nun, erst den reduzierten Blossomknoten v_* in eine maximale Matching-Verkantung einzubetten, dann das Innere des Blossoms auf die schon maximale umgebende Verkantung abzustimmen und ggf. letztere noch korrektiv anzupassen. Dazu führt man den bisherigen Prozess des Findens und Erweiterns augmentierender Pfade fort, bis man einen Pfad findet, der den Knoten v_* enthält, der das Blossom ersetzt hat. Je nach Einbindung von v_* in den erweiterten Weg nach dem Flip kann das Innere des Blossoms entsprechend auf einen längeren alternierenden Pfad angepasst werden, der dann das komplette Blossom enthält. Kann kein weiterer augmentierender Pfad mehr gefunden werden, ist das Matching maximal und man ist fertig.

7 Euler- & Hamiltonkreise

7.1 Eulertour & -kreis

7.2 Chinesisches Postboten-Problem (CPP)

7.3 Hamiltonkreis

7.4 Travelling-Salesman-Problem (TSP)

8 Färbung von Graphen

8.1 Färbung planarer Graphen

8.2 Heuristiken zur Graphenfärbung