Netzwerkalgorithmen
SoSe 20

Benedikt Lüken-Winkels

21. Oktober 2020

Inhaltsverzeichnis

1 Bellman/Ford - Allgemeiner Graph

Geben Sie den Bellman/Ford Algorithmus an und analysieren Sie seine Laufzeit.

Algorithmus 1 : Bellman/Ford

```
input : Graph G(V, E)
        Knoten s
        Kostenfunktion c
        Knotenarray DIST
        Knotenarray PRED
Queue Q;                                    //Kandidatenmenge
Nodearray count ← (G, 0)
for v ∈ G do
    DIST[v] ← ∞
    PRED[v] ← NULL
DIST[s] ← 0
Q ← Q ∪ s
while Q ≠ ∅ do
    u ← Q.pop()
    count[u] + +
    if count[u] > |V| then
        return negativer Zyklus gefunden
    for e ∈ u.out_edges() do
        v ← e.target()
        if DIST[u] + c(e) < DIST[v] then
            DIST[v] = DIST[u] + c(e)
            PRED[v] = e
            Q ← Q ∪ v
return kein negativer Zyklus gefunden
```

$PRED$ wird dazu verwendet, um nachher den kürzesten Weg zu einem Knoten nachverfolgen zu könnten. $PRED$-Verweise ändern sich daher, wenn sich der kürzeste Weg ändert.

Laufzeit

$$n \cdot \underbrace{(\text{Iterationen über alle Knoten}|\text{ausgehende Kanten})}_{O(\underbrace{\sum_{v\in V}(1+outdeg(v))}_{n+m})}$$

$\Rightarrow O(n(n+m))$ als Gesamtlaufzeit. Ist der Graph zusammenhängend, so gilt $m \geq n-1$ und die Laufzeit wird von $m \cdot n$ dominiert $\Rightarrow O(nm)$

2 SSSP-Algorithmus ohne neagtive Zyklen - Dijkstra

Geben Sie einen SSSP-Algorithmus an, der O(n + m) als Laufzeit hat und auf nicht-negativen Netzwerken funktioniert.

Algorithmus 2 : dist - verfeinert

```
for v ∈ G do
    DIST[v] ← ∞
    PRED[v] ← NULL
DIST[s] ← 0
Priority Queue PQ.insert(s, 0)
while PQ ≠ ∅ do
    wähle Knoten u ∈ PQ mit DIST[u] minimal: PQ.del_min();
    for v ∈ V mit e = (u,v) ∈ E do
        d ← DIST[u] + c(e)
        if DIST[v] > d then
            if DIST[v] = ∞ then
                PQ.insert(v, d)
            else
                PQ.decrease_p(v, d)
            DIST[v] ← d
            PRED[v] ← e
```

Priority = Distanz

Laufzeit $O(n+m)$ durch perfekte Wahl.

- Operationen auf dem Graphen: $O(n+m)$
- Priority Queue

$\Rightarrow$ Gesamtlaufzeit $O(n \cdot (T_{insert}(n) + T_{delmin}(n) + T_{empty}(n)) + m \cdot T_{decrease}(n))$
m ist der Flaschenhals des Algorithmus.

Realisierung der Datenstruktur

- binärer Min-Heap
- balancierter Baum

$\Rightarrow$ $O((n+m)\log n)$

Mit Fibonacci Heap: Insert $O(1)$, Delmin $(\log n)$) $\Rightarrow O(n\log(n)+m)$ (Decrease $O(1)$,

3 SSSP-Algorithmus azyklische Netzwerke

Geben Sie einen SSSP-Algorithmus an, der O(n + m) als Laufzeit hat und auf azyklischen Graphen funktioniert.

Azyklische Graphen besitzen eine topologische Sortierung.

Algorithmus 3 : SSSP-Alogrithmus mit Topsort

```
Knotenarray INDEG(G, 0)
for v inV do
    INDEG[v] ← indeg(v)
    if INDEG[v] = 0 then
        zero.append(v)
    DIST[v] ← ∞
DIST[s] ← 0
while zero is not empty do
    u ← zero.pop()
    for (u, v) ∈ E do
        INDEG[v] ← INDEG[v] − 1
        if INDEG[v] = 0 then
            zero.append(v)
        d ← DIST[u] + c((u, v))
        if DIST[v] > d then
            DIST[v] ← d
```

Laufzeit $O(n + m)$ durch perfekte Wahl mit topologischem Sortieren

4 SSSP-Algorithmus mit negativen Zyklen

Geben Sie einen SSSP-Algorithmus an, der negative Zyklen erkennt und ihn ausgibt. Wie ist die Laufzeit?

Bellman/Ford

5 Maxflow Labeling

Geben Sie einen Labeling-Algorithmus zur Lösung des Maxflow-Problems in Pseudo-Code an und analysieren Sie die Laufzeit.

Bemerkung Der Wert des maximalen Flusses ist gleich der Kapazität eines minimalen (s,t)-Schnittes

Laufzeit In jeder Iteration von Augment erhöht der Algorithmus den Flusswert um min 1. Daher können maximal $F_{max} \leq n \cdot U$ Iterationen benötigt werden. (U = maximale Kapazität). Eine Labeling Iteration kostet $O(n + m) \Rightarrow O(n^2 \cdot U + m \cdot n \cdot U)$.
Ist G zusammenhängend, so git $m \geq n - 1 \Rightarrow$ **Insgesamt** $O(n \cdot m \cdot U)$

Worst Case: U kann sehr groß sein, daher keine polynomielle Laufzeit in m und n

Algorithmus 4 : Maxflow Labeling

```
input : Graph G = (V, E), Knoten s, t, Kapazitätsfunktion u, Flussfunktion x
Liste S ← [ ]
Knotenarray PRED(G, NULL)
Kantenarray labelled(G)
for v ∈ V do
    labelled[v] ← false
while true do
    labelled[s] ← true
    S.append(s)
    while !S.empty() do
        v ← S.pop()
        for e = (v, w) ∈ E, do                               // Forwärtskanten
            if x(e) = u(e) or labelled[w] then
                continue          // Restkapazität ist = 0 oder schon besucht
            labelled[w] ← true
            PRED[w] ← e
            S.append(w)
        for e = (w, v) ∈ E, do                                   // Rückkanten
            if x(e) = 0 or labelled[w] then
                continue                  // Fluss ist = 0 oder schon besucht
            labelled[w] ← true
            PRED[w] ← e
            S.append(w)
        if labelled[t] then
            S.clear()
    if labelled[t] then
        AUGMENT(G, s, t, PRED, u, x)
    else
        break
```

Algorithmus 5 : Augment - Pfaderhöhung

input : Graph $G = (V, E)$, Knoten s, t, Kapazitätsfunktion u, Flussfunktion x

$\delta \leftarrow \infty$ `// Restkapazität des Pfades P`
$v \leftarrow t$
while $v \neq s$ **do**
    $e \leftarrow PRED[v]$
    if $e = (v, w)$ **then** `// Rückkante`
        $r \leftarrow x(e)$
        $v \leftarrow w$
    else `//` $e = (w, v)$ `Forwärtskante`
        $r \leftarrow u(e) - x(e)$
        $v \leftarrow w$
    if $r < delta$ **then**
        $\delta \leftarrow r$ `// neues minimum`
$v \leftarrow t$
while $v \neq s$ **do**
    $e \leftarrow PRED[v]$
    if $e = (v, w)$ **then** `// Rückkante`
        $x(e) \leftarrow x(e) - \delta$
        $v \leftarrow w$
    else `//` $e = (w, v)$ `Forwärtskante`
        $x(e) \leftarrow x(e) + \delta$
        $v \leftarrow w$

6 Capacity-Scaling

Geben Sie CAPACITY-SCALING unter Verwendungen des Maxflow Labeling Algorithmus an. Begründen Sie die Laufzeit.

Bemerkung Maxflow mit möglichst hoher Restkapazität im Pfad P oder kürzesten erhöhenden Pfad. Letzte Δ-Phase entspricht dem normalen Labeling Alogrithmus $\rightarrow$ Korrektheit.

Algorithmus 6 : Capacity-Scaling

input : Graph $G = (V, E)$, Knoten s, t, Kapazitätsfunktion u, Flussfunktion x, Maximale Kapazität U

$\Delta \leftarrow 2^{\log U}$
while $\Delta > 1$ **do**
  $MFLabeling(G, s, t, PRED, u, x)$
  $\Delta \leftarrow \frac{\Delta}{2}$

Laufzeit Jede Δ-Phase führt maximal $2m$ Erhöhungen aus. Die Restkapazität eines (s,t)-Schnittes $r[S, \overline{S}] \leq m \cdot \Delta$, da jede Kante zwischen S und $\overline{S}$ höchstens Restkapazität Δ hat. Die nächste $\frac{\Delta}{2}$-Phase führt dann maximal $\frac{m\Delta}{\Delta/2} \leq 2m$ Erhöhungen aus. Die Laufzeit einer Δ-Phase ist daher $O(2m \cdot m) \Rightarrow O(m^2)$

Insgesamt ergibt sich die Laufzeit $O(m^2 \log U)$. Polynomiell, aber nicht streng polynomiell.

Algorithmus 7 : Maxflow Labeling mit Δ

```
input : Graph G = (V, E), Knoten s, t, Kapazitätsfunktion u, Flussfunktion x,
        Restkapazitätsgrenzwert Δ
Liste S ← [ ]
Knotenarray PRED(G, NULL)
Kantenarray labelled(G)
for v ∈ V do
    labelled[v] ← false
while true do
    labelled[s] ← true
    S.append(s)
    while !S.empty() do
        v ← S.pop()
        for e = (v, w) ∈ E, do                          // Forwärtskanten
            if u(e) − x(e) < Δ or labelled[w] then
                continue     // Restkapazität ist < Δ oder schon besucht
            labelled[w] ← true
            PRED[w] ← e
            S.append(w)
        for e = (w, v) ∈ E, do                          // Rückkanten
            if x(e) < Δ or labelled[w] then
                continue               // Fluss ist < Δ oder schon besucht
            labelled[w] ← true
            PRED[w] ← e
            S.append(w)
        if labelled[t] then
            S.clear()
    if labelled[t] then
        AUGMENT(G, s, t, PRED, u, x)
    else
        break
```

7 Preflow Push

Admissible Distanz-Funktion von t d mit $d(t) = 0$. $d(i) \leq d(j) + 1$ für alle Kanten $(i, j) \in G(x)$. Gilt für eine Kante (i, j), dass $d(i) = d(j) + 1$, so ist sie *admissible*

7.1 Generisch

Laufzeit

- Relabel: $O(2n^2)$. $d(i)$ ist $< 2n$ und da jedes mal nur um 1 erhöht wird ergibt $n \cdot 2n$
- Saturierende Push-Operationen: $O(nm)$. Zwischen 2 SatPush-Operationen derselben Kante muss d und 2 erhöht worden sein. Da das $d(i) < 2n$, geht das für jede Kante n mal.
- Nicht-Saturierende Push-Operationen: $O(n^2m)$. Potential (Zustand des Netzwerks als numerischer Wert) Φ als Summe aller Distanz-Labels aktiver Knoten $i \in I$ (also $e(i) > 0$)
  1. Anfang: $\Phi \leq 2n^2$
  2. Bei Termination: $\Phi = 0$ und keine aktiven Knoten mehr.

  Nun werden 2 Fälle unterschieden:
  Fall 1: keine admissible Edge für i: $d(i)$ wird um mindestens 1 und höchstens $2n$ erhöht. Eine Erhöhung von $2n \cdot n = 2n^2$ ist also möglich.

  Fall 2.1: für die admissible Edge wird ein saturierendes Push ausgeführt: Die Anzahl der aktiven Knoten kann um 1 erhöht und das Potential um $2n$, also $2n \cdot nm$.

  Fall 2.2: für die admissible Edge wird ein nicht-saturierendes Push ausgeführt. Bei einem nicht saturierenden Push wird Φ um mindestens 1 verringert, da falls ein neuer Knoten j aktiviert wird $d(i) = d(j) + 1$ gilt (i ist nun deaktiviert).

Das Potential kann also auf $2n^2 + 2n^2 + 2n^2m$ (Anfangswert + Fall 1 + Fall 2.1) wachsen. Die nicht-saturierenden Pushes verringern um min 1, also $O(n^2m)$ nicht saturierende Pushes. Die Laufzeit wird durch durch die Anzahl der nicht-saturierenden Pushes dominiert $\Rightarrow O(n^2m)$. Die Kapazität spielt hier keine Rolle mehr für die Laufzeit.

Algorithmus 8 : Generischer Preflow-Push Algorithmus

```
for i ∈ V do
    d(i) ← 0
    e(i) ← 0
for (i, j) ∈ E do
    x_ij ← 0
for j ∈ V mit (s, j) ∈ E do    // Saturiere alle aus s ausgehenden Kanten
    x_sj ← u_sj
    e(s) ← e(s) − u_sj
    e(j) ← e(j) + u_sj
d(s) ← n                        // Hebe s auf auf Level n
while ∃i ∈ V mit e(i) > 0 do
    Wähle i
    PUSH/RELABEL(i)
```

Algorithmus 9 : Push/Relabel

```
if i hat eine admissible Kante (i,j) ∈ G(x) then          // Push
    Wähle Kante (i,j)
    δ ← min{e(i), r_ij}
    x_ij ← x_ij + δ
    e(i) ← e(i) − δ
    e(j) ← e(j) + δ
else                                                      // Relabel
    d(i) ← min{d(j)|(i,j) ∈ G(x)} + 1
```

7.2 FIFO Preflow-Push

Aktive Knoten sind in einer Queue. Folge von Operationen auf einem Knoten sind möglich.

Laufzeit Der Ablauf wird in Phasen eingeteilt. i-te Phase ist die Behandlung aller Knoten, die sich nach Phase $i-1$ in Q befinden. Anzahl der Phasen ist $\leq 2n^2$. Pro Phase wird jeder Knoten maximal einmal behandelt und führt für jeden Knoten maximal ein nicht-saturierendes Push. $\Rightarrow n \cdot 2n^2 = O(n^3)$

Algorithmus 10 : FIFO-Queue Preflow-Push Algorithmus

```
Q ← empty Queue                              // Alle aktiven Knoten mit e > 0
for i ∈ V do
    d(i) ← 0
    e(i) ← 0
for (i, j) ∈ E do
    x_ij ← 0
for j ∈ V mit (s, j) ∈ E do      // Saturiere alle aus s ausgehenden Kanten
    x_sj ← u_sj
    e(s) ← e(s) − u_sj
    e(j) ← e(j) + u_sj
    Q.append(j)
d(s) ← n                                     // Hebe s auf auf Level n
while Q is not empty do
    i ← Q.pop()
    for (i, j) ∈ G(x) do
        if d(i) = d(j) + 1 then              // Wähle eine admissible Kante
            δ ← min{e(i), r_ij}                                  // Push
            x_ij ← x_ij + δ
            e(i) ← e(i) − δ
            e(j) ← e(j) + δ
            if e(j) ≠ 0 then
                Q.append(j)
        if e(i) = 0 then
            break
    if e(i) > 0 then                // Keine admissible Kante verfügbar
        d(i) ← min{d(j) | (i, j) ∈ G(x)} + 1                     // Relabel
        Q.append(i)
```

7.3 Highest-Label Preflow-Push

Wähle einen Knoten mit maximalen Dist-Label in Priority-Queue. Da sich die Dist-Labels auf das Intervall $\{0, \ldots, 2n\}$ beschränken sind Buckets möglich.

Realisierung der Priority Queue Distanzen sind aus einem beschränkten Intervall $\{0, \ldots, 2n\} \Rightarrow$ Feld von Listen (Bucket-Array). Pointer auf das maximale Bucket

Laufzeit

- Q.insert: füge i in das entsprechende Bucket und überprüfe max-Pointer $\Rightarrow O(1)$
- Q.delmax: Entferne vorderstes Element eines Buckets und aktualisiere max-Pointer $\Rightarrow O(n)$

$O(n^2\sqrt{m})$

Algorithmus 11 : Priority Queue Preflow-Push (Highest Label)

```
Q ← empty Queue                                  // Alle aktiven Knoten mit e > 0
for i ∈ V do
    d(i) ← 0
    e(i) ← 0
for (i, j) ∈ E do
    x_ij ← 0
for j ∈ V mit (s, j) ∈ E do      // Saturiere alle aus s ausgehenden Kanten
    x_sj ← u_sj
    e(s) ← e(s) − u_sj
    e(j) ← e(j) + u_sj
    Q.insert(j, d(j))
d(s) ← n                                         // Hebe s auf auf Level n
while Q is not empty do
    i ← Q.delmax()
    for (i, j) ∈ G(x) do
        if d(i) = d(j) + 1 then                  // Wähle eine admissible Kante
            δ ← min{e(i), r_ij}                  // Push
            x_ij ← x_ij + δ
            e(i) ← e(i) − δ
            e(j) ← e(j) + δ
            if e(j) ≠ 0 then
                Q.insert(j, d(j))
        if e(i) = 0 then
            break
    if e(i) > 0 then                  // Keine admissible Kante verfügbar
        d(i) ← min{d(j)|(i, j) ∈ G(x)} + 1       // Relabel
        Q.insert(i, d(i))
```

8 Min-Cost-Flow-Problem (MCF)

Wie lautet die reduzierte Kosten-Optimalität für das Min-Cost-Flow-Problem? Folgern Sie aus ihrer Gültigkeit, dass im Restnetzwerk keine negativen Zyklen existieren.

Eingaben Kostenfunktion c, Supplyfunktion b, Flussfunktion x, Kapazitätsfunktion u

Feasible Ein Fluss x, für den alle Kanten Kapazitätsbedingung und Massebalance gelten, ist feasible.

Optimalitätsbedingung (Negative Cycle Optimality) Wenn im Restnetzwerk $G(x)$ kein negativer Zyklus enthalten ist.

Reduzierte Kosten Potentialfunktion π. $c_{ij}^{\pi} = c_{ij} + \underbrace{\pi(j) - \pi(i)}_{\Delta\pi}$

Reduzierte Kostenoptimalität x optimal $\Leftrightarrow \exists$ Potential $\pi : \forall(i,j) \in G(x) : c_{ij}^{\pi} \geq 0$ also alle Kosten mit addiertem Potential (zB negative Distanz)

Complementary Slackness Optimality arbeitet auf G.

Laufzeit $U = \max u_{ij}$, $C = \max |c_{ij}|$.

- Pro Iteration wird um mindestens 1 reduziert.
- Die Spanne der Kosten geht von $-mCU$ bis mCU, also $\leq mCU$ Iterationen
- Eine Iteration kostet $O(nm)$ (Bellman/Ford)

$\Rightarrow O(nm^2CU)$

Algorithmus 12 : Negative Cycle Canceling - Eleminierung von negativen Zyklen

---

Erstelle einen Maxflow in G
$W \leftarrow Bellman/Ford/negCycle(G)$
while $W \neq NULL$ **do**
    $\delta \leftarrow \min\{r_{ij} : (i,j) \in W\}$
    $augment(W, \delta)$
