Max Wisniewski , Alexander Steen

Tutor: Lena Schlipf

Aufgabe 1 Unabhängige Mengen in Pfaden

Sei $G = (V, E)$ ein *Pfad*, mit den Knoten $v_1, v_2, ..., v_n$.
Dazu gehört eine Funktion $w : V \longrightarrow \mathbb{N}^*$, die einem Knoten ein positives ganzzahliges Gewicht zuordnet, im Folgenden mit $w(v_i) = w_i$ bezeichnet.

Zur Notation:
Wir werden im Folgenden einen Pfad wie folgt darstellen:
$[w_0, w_1, w_2, w_3, ..., w_{n-1}]$.
Dabei sind zwei Knoten benachbart, wenn diese nebeneinander in der Liste stehen. Die Knoten sind ihrer Nummerierung entsprechend aufsteigend angeordnet und in der Liste steht nur der Wert, die die Funktion w ihm zuordnet.

(a) Geben Sie ein Beispiel, dass der Algorithmus (siehe Aufgabenblatt) keine unabhängige Menge maximalen Gewichts liefert.

Lösung:
Betrachten wir den *Pfad* $[1, 9, 10, 9, 1]$.

Nach dem Algorithmus nehmen wir den größten Wert, hier Knoten 2 mit dem Wert 10, und entfernen ihn und seine Nachbern (Knoten 1 und 3). Es verbleibt der Folgende Graph: $[1], [1]$. Der restliche Graph ist *unabhängig*, bei weiterer Ausführung werden die beiden 1en genommen und nicht mehr gelöscht als diese beiden in zwei Schritten, da sie keine Nachbarn mehr besitzen.

Der Algorithmus liefert uns also die Knoten $0, 2, 4$ mit dem gesammt Gewicht 12.
Wie man an diesem Beispiel aber leicht sieht, wäre die maximale unabängige Menge allerdings $1, 3$ mit dem Gewicht 18. Diese beiden Knoten wurden aber als Nachbarn eines wenig kleinere Knotens im 1 Schritt allerdings verworfen.

Da wir ein Gegenbeispiel gefunden haben, kann der Algorithmus nicht korrekt sein. □

(b) Geben Sie ein Beispiel, bei dem der Algorithmus (vgl. Aufgabenzettel) keine unabhängige Menge maximalen Gewichts bestimmt.

Lösung:
Der Algorithmus zerlegt die Menge in zwei unabängeige Menge, die eine hat gerade Indizes, die andere ungerade.
Wir wollen uns eine Menge konstruieren, in der die maximal unabhängige Menge sowohl gerade, als auch ungerade Indizes enthält.

Wir untersuchen $[1, 9, 1, 1, 9, 1]$.
Die Menge der Werte mit geraden Indizes ist hier $\{9, 1, 1\}$ und die mit ungeraden ist $\{1, 1, 9\}$. Beide ergeben ein Gesamtgewicht von 11.
Auch hier sehen wir schnell, dass Element $1, 4$ ein Gesamtgewicht von 18 hat und somit mehr Gewicht hat, als die beiden Menge, deren Maximum den größten Wert unter den unabhängigen Menge haben soll.

Da dies wie an diesem Beispiel gezeigt nicht immer der Fall ist, bestimmt der Algorithmus leider nicht die unabhängige Menge mit maximalen Gewicht. □

(c) Geben Sie einen effizienten Algorithmus an, der eine unabhängige Menge maximalen Gewichtes in einem Pfad bestimmt. Analysieren Sie Laufzeit und Platzbedarf.

Lösung:

Teillösungen:
Als Teillösung wählen wir $E[j]$:= Maximum der *unabhängigen Teilmengen*, wenn wir nur die ersten j Knoten betrachten.

Rekursion:

$$\begin{aligned} E[0] &= w(0) \\ E[1] &= max\{w(0), w(1)\} \\ \forall 2 \leq i \leq n : \quad E[i] &= \max\{E[j-2] + w(j),\ E[j-1]\} \end{aligned}$$

Sollten wir nur 1 Element haben nehmen wir dies einfach. Haben wir 2, nehmen wir das Maximum von beiden. Nehmen wir das Maximum von erstens, wir nehmen das j - te Element müssen dafür aber das daneben verwerfen, oder zweitens wir nehmen es nicht, aber können das daneben wählen.

Ziel ist es $E[n-1]$ zu bestimmen, weil wir dort alle Knoten betrachtet haben.

Korrektheit

Beh.: $\forall j \leq n$: $E[j]$ ist maximal.

I.A. Gilt für $j = 0$ und $j = 1$ trivialerweise.

I.S. $r \longrightarrow r + 1$

Wir wissen, dass die letzten beiden Mengen ohne j und ohne $j, j-1$ maximal sind. Wenn wir nun $j+1$ dazunehmen, wissen wir, dass wir den Eintrag j nicht betrachten können, da dieser ein Nachber von $j+1$ ist. Wollen wir $j+1$ dazunehmen, können wir das nur tun, wenn wir $E[j-1]$ nehmen. Der maximale Wert von $E[j+1]$ ist also das Optimum von $E[j]$ und $E[j-1] + w(j+1)$.

Damit ist unser berechneter Wert von $E[j+1]$ maximal. □

Algorithmus

```
if n = 1 then
  Erg := [0]; // Die optimale Menge enthaelt nur 0
  return w(0); // und hat deren Wert
Erg1 := [0];  // Sonst speichern wir den Wert
w1 := w(0); //  in der ersten Laufvariable
if(w(0) > w(1) then
  Erg2 := [0];  // Und den andern Anker
  w2 := w(0); // in der zweiten
else
  Erg2 := [1];
  w2 := w(1);

//
// Wir gehen wie bei Fibonacci mit
// Akkumulatoren durch, dabei schieben
// wir immer das neure Element (2) in das
// aeltere (1) und errechnen das neue.
//
for i := 2 to n-1 do
  if w1 + w(i) > w2 then
    save := w1 +w(i);
    w1 := w2;
    w2 := save;
    ergSave := Erg2;
    Erg2 := Erg1 ++ [i];
    Erg1 := Erg2;
  else
    save := w1;
    w1 := w2;
    w2 := save;
    ergSave := Erg2;
    Erg2 := Erg1;
    Erg1 := ergSave;
Erg := Erg2;
return w2;
```

Da wir immer nur den letzte Wert (w2) und den vorletzten Wert (w1) für den nächsten Schirtt benötigen, müssen wir auch nur diese beiden speichern. Danach gehen wir wie beim dynamischen Programmieren unserere Parameter von klein nach groß durch.

Laufzeit & Platz

Platzbedarf: $\Theta(1)$, da wir nur 2 Werte und swap speichern müssen. Dies wird durch die Speicherung der liste allerdings relativiert, da wir nun noch maximal $\frac{n}{2}$ Elemente in Erg1, Erg2 mitführen.

Laufzeit: Wir betrachten jeden Knoten von V genau einmal $\Rightarrow \Theta(n)$.

Aufgabe 2 Vorlesungsplanung

In der Vorlesung haben wir einen *Greedy - Algorithmus* gesehen, mit der wir aus einer Menge von Vorlesungen, eine Maximale Menge von Vorlesungen finden können, die alle paarweise *verträglich* sind. Untersuchen Sie, ob der Algorithmus mit einer der folgenden Ordnungen im Sortierschritt immer noch funktionert. Geben Sie Pseudocode an und begründen Sie seine Korrektheit und Laufzeit.

(a) Sortiere aufsteigend nach den Anfangszeiten

Dies ist kein valide Lösung. Nehmen wir das Beispiel
$R = \{I_1, I_2, I_3\}$, wobei $I_1 = (1, 100),\ I_2 = (2, 10),\ I_3 = (12, 20)$.

Wie wir schnell sehen, ist das beste Ergebnis I_2, I_3, diese beiden sind *verträglich*, da sie nach einander liegen. Der Algorithmus würde uns allerdings I_1 liefern, weil es am frühesten anfängt.
Danach würden alle unverträglichen verworfen und damit auch I_2, I_3.

Da wir ein Gegenbeispiel gefunden haben, kann der Algorithmus mit dieser Ordnung nicht korrekt sein.

(b) Sortiere absteigend nach den Anfangszeiten.

Diese Sortierung gibt zusammen mit dem Algorithmus

```
A := ∅
while R ≠ ∅ do
  i := max a(i)
  A := A ∪ {I_i}
  R := R \{x | e(x) > a(i)}
```

Beweis:
Sei S eine optimale Teilmenge von R. Wir wollen zeigen, dass $|A| = |S|$ gilt.

Seien $i_1, i_2, ..., i_k$ und $j_1, j_2, ..., j_m$ die Indizes der Intervalle von A bzw. von S, zeitlich (absteigend) geordneter Folgen.

Wir induzieren über die Länge der beiden Folgen.

Beh.: $\forall r \leq m\ :\ a(i_r) \geq a(j_r)$

I.A. $a(i_1) \geq a(j_1)$ ist trivialerweise erfüllt, da der Algorithmus im ersten Schritt das i_1 wählt, mit dem größten $a(i_1)$.

I.S. $r \longrightarrow r + 1$
Nach dem Algorithmus, liegt $a(i_{r+1})$ noch in R, da nach Ordnung gilt $e(i_r) \geq a(i_r) \overset{I.V.}{\geq} a(j_r) \geq e(j_{r+1}) \geq a(j_{r+1})$.

Der neue Anfang von $a(i_{r+1})$ muss also im Intervall $(a(j_{r+1}), a(j_r))$ (Wir haben hier an dieser Stelle das ganze umgedreht, weil Intervalle so rum aufgeschrieben werden) liegen. Damit muss $a(i_{r+1}) \geq a(j_{r+1})$ sein. □

Beweis der Optimalität:

Angenommen $k < m$ dann können wir aus unserer oberen Behauptung ziehen, dass noch min ein Element in R sein muss, nämlich I_{k+1}. Der Algortihmus kann also nicht terminiert sein.

Angenommen $k > m$, dann wäre S nicht optimal gewesen.

$\Rightarrow A$ ist eine optimale Lösung. □

(c) Sortiere absteigend nach den Endzeiten.

Dieser Fall verhält sich analog zu *a)*. Nehmen wir die selben Intervalle $R = \{I_1, I_2, I_3\}$, wobei $I_1 = (1, 100),\ I_2 = (2, 10),\ I_3 = (12, 20)$.

Da I_1 auch das letzte Ende hat, wird der Algorithmus dieses Intervall zuerst nehme und alle anderen verwerfen, da diese überlappen. Optimal hier wäre wie gehabt I_2, I_3.

Wir haben wieder ein Gegenbeispiel gefunden, damit kann der Algorithmus nicht korrekt sein.

(d) Sortiere aufsteigend nach der Länge

Diese Ordnung bildet keine valide Lösung.
Nehmen wir als Beispiel $R = \{I_1, I_2, I_3\}$ mit $I_1 = (1, 10)$, $I_2 = (9, 12)$, $I_3 = (11, 20)$.

Der Algorithmus würde uns als kürzestes Intervall I_2 liefern. Damit fallen I_1, I_3 aus der Menge, weil sie nicht veträglich sind. Die maximale Anzahl von Intervallen ist aber I_1, I_2, da diese beiden *verträglich* sind.

(e) Sortiere absteigend nach der Länge

Dieser Fall verhält sich analog zu *a), b)*. Nehmen wieder die selben Intervalle $R = \{I_1, I_2, I_3\}$, wobei $I_1 = (1, 100),\ I_2 = (2, 10),\ I_3 = (12, 20)$.

Das längeste Intervall ist I_1. Da dieses I_2, I_3 überdeckt, werden die beiden aus der Menge geschmissen. Damit liefert der Algorithmus I_1. Das maximale Menge von *verträglichen* Intervallen wäre allerdings I_2, I_3.

Wir haben ein Gegenbeispiel gefunden und damit ist der Algorithmus nicht korrekt.

Aufgabe 3 Autobahnfahrt

Gegeben ist eine Menge von Tankstellen entlang eines Weges und ein Auto, dass n Kilometer fahren kann.
Wir suchen nun eine minimale Anzahl von Tankstellen, mit der wir unser Ziel erreichen können (ohne mit einem leeren Tank liegen zu bleiben).

Lösung:

Notation und Idee:
Wir setzen hier einmal vorraus, dass die Tankstellen uns in einer Liste gegeben werden, die sie nach Abstand zum Startpunkt sortiert. Zwischen 2 Tankstellen möchten wir den Abstand dieser beiden Tankstellen haben.

Als Darstellung wählen wir eine Liste des Treibstoffverbrauches $S = [\delta_1, \delta_2, ...\delta_{m+1}]$, wobei δ_1 der verbrauchte Treibstoff vom Startpunkt zur ersten Tankstelle ist, δ_2 der verbrauchte Treibstoff zwischen der ersten und der zweiten Tankstelle usw.
δ_{n+1} ist nun der verbrauchte Treibstoff von der letzten Tankstelle zum Zielpunkt.

Für den Beweis konstruieren wir uns noch eine Funktion $w : Tankstelle \longrightarrow \mathbb{N}$, die den Abstand einer Tankstelle zum Start angibt.

Um das Ziel zu finden gehen wir nun die Tankstellen durch, bis wir nicht mehr genügend Sprit bis zur nächsten Tankstelle haben.

Algorithmus:

```
stops := 0;
tank := n;
for i := 0 to m do
  if S[i] > n then
    error // Die Strecke kann nicht befahren werden
  if S[i] > tank then
    stops++;
    tank := n;
  tank := tank - S[i];
```

Korrektheit:

Sei k eine optimale Anzahl von Stopps und l die Anzahl von Stopps, die uns der Algorithmus zurückgibt. Nun seien $i_1, i_2, ..., i_k$ und $j_1, j_2, ..., j_l$ die Indizes der Tankstellen bei der k bzw. l getankt haben.

Beh.: $\forall r \leq k \ : \ w(i_r) \geq w(j_r)$

Bew.: Induktion über k

I.A. $w(i_1) \geq w(j_1)$ gilt, da unser Algorithmus ersteinmal soweit fährt, bis die nächste Etappe nicht mehr zu schaffen ist. Da die nächste Tankstelle vom Start also nicht zu erreichen ist, muss der optimale stopp von k vorher tanken.

I.S. $r \longrightarrow r+1$

Nach I.V. gilt $w(i_r) \geq w(j_r)$. Der Algorithmus fährt mit dem Auto nun so weit, dass für den Weg bis zur nächsten Tankstelle der Sprit nicht mehr genügt.
Da die optimale Tour höchstens genau so weit ist, wie die Greedy - Tour, kann der Sprit maximal genau so weit reichen, wie der Sprit der Greedy-Tour.

Formal ist: $w(j_r) \leq w(i_r) \leq w(i_{r+1}) \leq w(t)$, wobei t die Tankstelle direkt nach i_{r+1} ist. Da der Sprit bis nach t voni_r nicht reicht, werden wir von j_r nicht bis t kommen. Deshalb muss der nächste Stop der optimalen Tour zwischen $w(j_r)$ und $w(i_r)$ liegen. Damit gilt: $w(j_{r+1} \leq w(i_{r+1})$

Beweis der Optimalität

Ist $l > k$, wissen wir, dass die Greedy-Tour nach k Stops mindestens genau so viel Strecke zurückgelegt hat, wie die optimale Tour. Der Weg zum Ziel sollte von dieser Tankstelle also aus möglich sein, deswegen würde der Algorithmus keinen weiteren Stop mehr machen. Dieser Fall ist also nicht möglich.

Ist $k > l$ wäre k keine optimale Tour.

Damit kann nur $k = l$ gelten. Unser Algorithmus liefert also eine minimale Anzahl von Stops.

Laufzeit:

Sollte die Darstellung nicht sortiert sein, so müssen wir alle Tankstellen nach Abstand zum Startpunkt sortieren. Dies ist mit $\Omega(m \cdot \log m)$ nach unten beschränkt. Kann aber je nach Darstellung der Tankstellen beliebig schlecht werden.

Nehmen wir in, dass die Tankstellen, wie angenommen, schon sortiert nach Abstand zum Start sortiert sind. Nun können wir in $O(m)$ die Abstände zwischen den einzelnen Tankstellen bestimmen, damit den Treibstoffverbraucht und diesen in die Liste S eintragen.

Der angegebene Algorithmus läuft nun in $\Theta(m)$ durch, da wir für jede Tankstelle entscheiden, ob wir den nächsten Weg noch schaffen. Da wir 2 Vergleiche haben und maximal 2 Operationen in der Schleife.