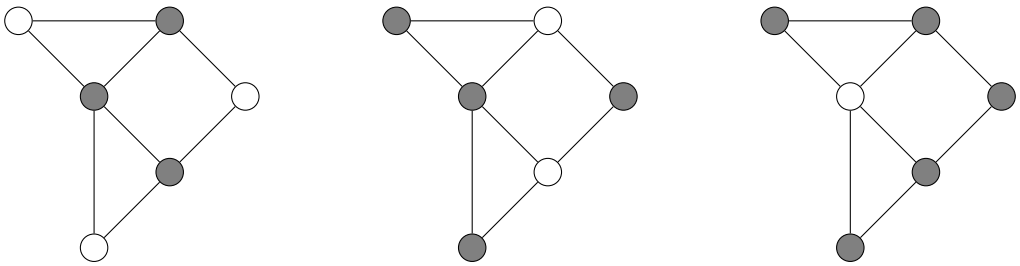
7.5 Das Knotenüberdeckungsproblem

Die bisher betrachteten gierigen Algorithmen haben jeweils ein Berechnungs- bzw. Optimierungsproblem exakt gelöst, sie bestimmten einen *kürzesten* Pfad, einen *minimalen* Spannbaum, etc. Oft treten in der Praxis aber Probleme auf, für deren exakte optimale Lösung keine effizienten Algorithmen bekannt sind. In einer solchen Situation ist es dann oft nützlich einen *Approximationsalgorithmus* zu verwenden. Ein solcher findet zwar im Allgemeinen keine optimale Lösung, aber er findet eine die, in einem noch zu präzisierenden Sinn, nicht allzu weit davon entfernt liegt. Wesentlich ist dann natürlich dass der Approximationsalgorithmus effizient genug für praktische Anwendungen ist. Für derartige Approximationsverfahren spielen gierige Algorithmen eine wichtige Rolle. Wir werden dazu in diesem Abschnitt ein Beispiel sehen.

Definition 7.6. Sei $G = (V, E)$ ein Graph. Eine *Knotenüberdeckung* von V ist eine Menge $V' \subseteq V$ so dass $\{u, v\} \in E \Rightarrow u \in V'$ oder $v \in V'$.

Beispiel 7.8. Ein Graph und drei seiner Knotenüberdeckungen (in grau):



Wie man sich leicht überlegen kann hat dieser Graph keine Knotenüberdeckung der Größe 2.

Jeder Graph hat die triviale Knotenüberdeckung $V = V'$. Wenn wir nach einer optimalen Knotenüberdeckung fragen, d.h. eine minimaler Kardinalität, dann erhalten wir das folgende Berechnungsproblem:

Optimale Knotenüberdeckung
Eingabe: ungerichteter Graph $G = (V, E)$
Ausgabe: Knotenüberdeckung V' von G so dass $\|V'\|$ minimal ist

Natürlich gibt es oft triviale Approximationsalgorithmen, z.B. im Fall der Knotenüberdeckung einfach V zu verwenden. Deshalb interessiert man sich im Kontext von Approximationsalgorithmen oft für Approximationsgarantien.

Definition 7.7. Wir sagen dass ein Algorithmus für ein Minimierungsproblem *Approximationsrate* $\rho(n)$ hat falls für jede Eingabe x der Größe n, für die vom Algorithmus gelieferte Ausgabe mit Kosten $c(x)$ und die optimalen Kosten $c^*(x)$ bei Eingabe x gilt dass $\frac{c(x)}{c^*(x)} \leq \rho(n)$.

Mit Kosten in der obigen Definition ist die Qualität der Lösung gemeint, also z.B. im Fall der Knotenüberdeckung die Anzahl der Knoten in der Überdeckung. Die Frage der Laufzeit eines Algorithmus ist unabhängig von der Frage nach seiner Approximationsrate. Ein Algorithmus mit Approximationsrate $\rho(n)$ wird oft auch als $\rho(n)$-*Approximationsalgorithmus* bezeichnet. Oft ist es möglich, für Optimierungsprobleme, auch wenn kein polynomialer Algorithmus zu ihrer exakten Lösung bekannt ist, einen polynomialen Approximationsalgorithmus zu finden, gelegentlich sogar mit konstanter, d.h. nicht von n abhängiger, Approximationsrate. Gierige Algorithmen

Algorithmus 31 Gierige Knotenüberdeckung

Prozedur GierigeKnotenüberdeckung(V, E)
  $U := \emptyset$
  $A := E$
  Solange A nicht leer
    Sei $\{u, v\} \in A$
    $U := U \cup \{u, v\}$
    Entferne alle e aus A die u oder v enthalten
  Ende Solange
  Antworte U
Ende Prozedur

eignen sich dafür besonders gut, da sie immer eine geringe Laufzeit haben. Betrachten wir zum Beispiel Algorithmus 31 zur Berechnung einer Knotenüberdeckung. Dieser berechnet keine optimale Knotenüberdeckung. Das sieht man allein schon daran, dass er nur Knotenüberdeckungen gerader Kardinalität berechnet, es gibt aber Graphen deren optimale Knotenüberdeckung ungerade Kardinalität hat, siehe Beispiel 7.8. Dieser Algorithmus hat Laufzeit $O(|E|)$.

Satz 7.7. *Die Prozedur* GierigeKnotenüberdeckung *ist ein* 2*-Approximationsalgorithmus.*

Beweis. Sei $G = (V, E)$ der Eingabegraph, sei $V^* \subseteq V$ eine optimale Knotenüberdeckung, sei V' die Ausgabe des Algorithmus und sei E' die Menge der aus A gewählten Kanten, dann ist $|V'| = 2|E'|$. Nun gibt es nach der Definition des Algorithmus keine zwei Kanten in E' die einen Knoten teilen. Also muss eine beliebige Knotenüberdeckung für jede Kante $e \in E'$ mindestens einen Knoten enthalten, um e zu überdecken. Insbesondere gilt das auch für V^*. Damit ist $|V^*| \geq |E'| = \frac{1}{2}|V'|$ und damit $\frac{|V'|}{|V^*|} \leq 2$. □

Es ist nicht bekannt ob das Problem der optimalen Knotenüberdeckung durch einen Algorithmus mit polynomialer Laufzeit (exakt) gelöst werden kann. An sich kann das schon Grund genug sein, sich einen Approximationsalgorithmus zu überlegen. In diesem (und vielen vergleichbaren Fällen) weiß man allerdings sogar dass die Existenz eines solchen Algorithmus implizieren würde dass $\mathbf{P} = \mathbf{NP}$. Die Frage ob $\mathbf{P} = \mathbf{NP}$ ist eines der schwierigsten offenen Probleme der Mathematik.

Um das genauer zu erklären, muss kurz ausgeholt werden. Ein Entscheidungsproblem ist ein Berechnungsproblem dessen Ausgabe entweder “ja” oder “nein” ist. Berechnungsprobleme und insbesondere auch Optimierungsprobleme können üblicherweise recht direkt in Entscheidungsprobleme übertragen werden, z.B.

Knotenüberdeckung

Eingabe: endlicher Graph $G = (V, E)$, $k \in \mathbb{N}$

Ausgabe: “ja” wenn G eine Knotenüberdeckung V' hat mit $|V'| \leq k$ und “nein” sonst

Ein Entscheidungsproblem wird als Menge betrachtet indem es mit der Menge seiner “ja”-Instanzen identifiziert wird. So wird zum Beispiel das Entscheidungsproblem der Knotenüberdeckung identifiziert mit der Menge

$$K = \{(G, k) \mid G \text{ ist endlicher Graph der Knotenüberdeckung } V' \text{ mit } |V'| \leq k \text{ hat}\}.$$

Wir schränken unsere Betrachtung nun ein auf Entscheidungsprobleme die Teilmengen von $\{0,1\}^*$ sind. Graphen, natürliche Zahlen, Tupel von Graphen und natürlichen Zahlen, etc. können als Elemente eines Entscheidungsproblems betrachtet werden indem eine Kodierung in Wörter über $\{0,1\}$ fixiert wird und der Graph, die Zahl, etc. mit seiner/ihrer Codierung identifiziert wird.

P bezeichnet nun die Menge aller Entscheidungsprobleme für die ein Algorithmus existiert dessen Laufzeit polynomial in der Größe der Eingabe ist[2]. Ein Entscheidungsproblem S ist[3] in **NP** genau dann wenn es ein Entscheidungsproblem $R \in \mathbf{P}$ sowie ein $c \in \mathbb{N}$ gibt so dass

$$x \in S \Leftrightarrow \exists y \in \{0,1\}^* \text{ mit } |y| \leq |x|^c \text{ und } (x,y) \in R.$$

An der Definition des Entscheidungsproblems K der Knotenüberdeckung ist jetzt unmittelbar sichtbar, dass es in **NP** ist indem wir für R die folgende Relation wählen:

$$((G,k),V') \in R \Leftrightarrow V' \text{ ist Knotenüberdeckung von } G \text{ mit } |V'| \leq k$$

und beobachten dass $R \in \mathbf{P}$. Dann gibt es ein c so dass

$$\begin{aligned}((V,E),k) \in K \Leftrightarrow \exists y \in \{0,1\}^* \text{ mit } |y| \leq |x|^c,\ x \text{ codiert } ((V,E),k),\\ y \text{ codiert } V' \subseteq V \text{ und } ((V,E),V') \in R\end{aligned}$$

Überdies kann vom Entscheidungsproblem der Knotenüberdeckung (mit etwas mehr technischem Aufwand) gezeigt werden, dass es **NP**-vollständig ist. **NP**-vollständige Probleme sind in einem gewissen technischen Sinn die schwierigsten Probleme in **NP**. Für ein beliebiges **NP**-vollständiges Problem S gilt dass $S \in \mathbf{P} \Leftrightarrow \mathbf{P} = \mathbf{NP}$.

Wenn nun das Optimierungsproblem der optimalen Knotenüberdeckung einen polynomialen Algorithmus besitzt, dann hat auch das Entscheidungsproblem einen polynomialen Algorithmus und, da dieses **NP**-vollständig ist, wäre dann $\mathbf{P} = \mathbf{NP}$.

Es gibt tausende **NP**-vollständige Probleme von denen viele praxisrelevant sind. Die Bedeutung der Frage ob $\mathbf{P} = \mathbf{NP}$ rührt daher dass die (meisten) Probleme in **P** in der Praxis effizient lösbar sind und viele praxisrelevante Probleme **NP**-vollständig sind.

[2]Diese Definition bleibt insofern informell als wir im Rahmen dieser Vorlesung nicht formell definieren was ein Algorithmus ist. Für die Definition von **P** hat das aber kaum Bedeutung da sich unterschiedliche derartige formale Definitionen modulo polynomialer Berechenbarkeit nicht unterscheiden.

[3]**NP** steht für "lösbar in **n**icht-deterministisch **p**olynomialer Zeit" nach einer anderen Definition von **NP** über nicht-deterministische Berechnung.

Kapitel 8

Dynamische Programmierung

In Kapitel 3 haben wir Teile-und-herrsche Algorithmen kennengelernt. Diese gehen so vor, dass sie eine Instanz eines Problems in kleinere, disjunkte, Teilinstanzen zerlegen, diese unabhängig voneinander lösen und deren Lösungen dann zu einer der ursprünglichen Instanz kombinieren. Solche Algorithmen sind nicht effizient, wenn im Laufe der Berechnung die selben Teilinstanzen oft auftreten. Dann werden diese nämlich unabhängig voneinander mehrfach gelöst. *Dynamische Programmierung* ist ein Designprinzip für Algorithmen, das solche mehrfach auftretenden Teilinstanzen auf eine Weise organisiert die doppelte Berechnungen vermeidet. Damit das effizient möglich ist, ist es notwendig dass die Anzahl der zur Bestimmung einer Lösung notwendigen Teilinstanzen nicht allzu groß ist (also z.B. nur polynomial).

Dieses Prinzip kann gut am folgenden Berechnungsproblem illustriert werden:

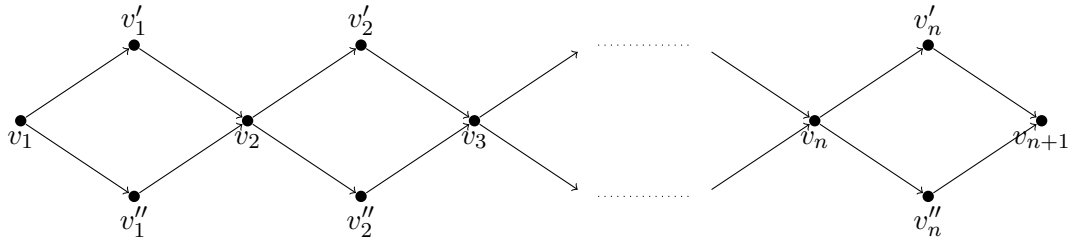
Anzahl der Pfade
Eingabe: endlicher gerichteter azyklischer Graph (V, E), $s, t \in V$
Ausgabe: Anzahl der Pfade von s nach t

Zunächst beobachten wir dass die Anzahl $n_{v,t}$ der Pfade von v nach t gleich $\sum_{(v,w)\in E} n_{w,t}$ ist. Eine Prozedur die diese Beobachtung direkt der teile-und-herrsche Methode folgend realisiert ist in Algorithmus 32 angegeben. Dieser Algorithmus ist zwar korrekt aber nicht sehr effizient.

Algorithmus 32 Anzahl der Pfade (exponentiell)

```
Prozedur AnzahlPfade(E, v, t)
    Falls v = t dann
        n := 1
    sonst
        n := 0
        Für (v, w) ∈ E
            n := n + AnzahlPfade(E, w, t)
        Ende Für
    Ende Falls
    Antworte n
Ende Prozedur
```

Zum Beispiel wird im Graphen



bei der Berechnung von AnzahlPfade(E, v_1, v_{n+1}) die Prozedur zwei Mal auf v_2 aufgerufen, vier Mal auf v_3, ..., 2^{n-1} mal auf v_n, insgesamt benötigt sie also Laufzeit $\Omega(2^n)$ auf einem Graphen mit $|V| = O(n)$ und $|E| = O(n)$. Man sieht also dass dieser Algorithmus nicht nur die Anzahl der Pfade berechnet, sondern dass er sie auch alle durchläuft und dadurch im schlechtesten Fall exponentielle Laufzeit benötigt.

Wir interessieren uns aber nicht für die Menge der Pfade, sondern nur für deren Kardinalität. So können wir die zur Entwicklung eines effizienteren Algorithmus entscheidene Beobachtung machen, dass es ausreicht pro $v \in V$ nur einmal zu berechnen wie viele Pfade von v nach t führen, selbst wenn v von s aus auf verschiedenen Pfaden erreichbar ist. Eine, effizientere, Vorgehensweise kann nun auf zwei Arten realisiert werden: 1. von oben nach unten wobei bereits berechnete Werte zwischengespeichert werden (*top-down mit Memoisierung*) oder 2. von unten nach oben (*bottom up*), dann entfällt die Notwendigkeit der expliziten Zwischenspeicherung. Die erste ist in Algorithmus 33 angegeben, ein Beispiel für die zweite sehen wir im nächsten Abschnitt. Algorithmus 33 erweitert die Datenstruktur für einen Knoten v um ein Attribut n

Algorithmus 33 Anzahl der Pfade (top-down mit Memoisierung)

```
Prozedur AnzahlPfade(E, v, t)
    Setze v.n := +∞ für alle v ∈ V
    Antworte AnzahlPfadeRek(E, s, t)
Ende Prozedur

Prozedur AnzahlPfadeRek(E, v, t)
    Falls v.n = +∞ dann
        Falls v = t dann
            v.n := 1
        sonst
            v.n := 0
            Für (v, w) ∈ E
                v.n := v.n + AnzahlPfadeRek(E, w, t)
            Ende Für
        Ende Falls
    Ende Falls
    Antworte v.n
Ende Prozedur
```

das die Anzahl der Pfade von v nach t angibt. Der Wert $+\infty$ bedeutet dabei dass diese Anzahl noch nicht bekannt ist. Dieser Algorithmus hat die Laufzeit $O(|V| + |E|)$. Man beachte auch die Ähnlichkeit dieses Verfahrens mit der Tiefensuche.

8.1 Das Stabzerlegungsproblem

Wir betrachten nun das folgende Optimierungsproblem: gegeben ist ein Eisenstab der Länge $n \in \mathbb{N}$ sowie eine Liste von Preisen $p_1, \ldots, p_n$ wobei p_i der erzielbare Preis für einen Eisenstab der Länge i ist. Der gegebene Stab der Länge n soll so zerschnitten werden, dass der durch Verkauf der Einzelteile erzielbare Gesamtpreis (d.h. die Summe der Preise der Einzelteile) maximal ist.

Stabzerlegungsproblem

Eingabe: $n \in \mathbb{N}$, $p_1, \ldots, p_n \in \mathbb{N}$

Ausgabe: $i_1, \ldots, i_k \in \mathbb{N}$ so dass $n = i_1 + \cdots + i_k$ und $p_{i_1} + \cdots + p_{i_k}$ maximal

Für einen Stab der Länge n gibt es 2^{n-1} verschiedene Möglichkeiten eine Zerteilung zu wählen, da an jedem ganzzahligen Punkt entweder geschnitten oder nicht geschnitten werden kann. Alle diese Zerteilungen durchzuprobieren und davon jene mit maximalem Gesamtpreis zu wählen würde also Zeit $\Omega(2^n)$ benötigen. Mit dynamischer Programmierung ist das auf effizientere Weise lösbar.

Beispiel 8.1. Gegeben $n = 8$ und $p_1, \ldots, p_n$ wie folgt:

i	1	2	3	4	5	6	7	8
p_i	1	3	6	9	13	15	16	18

Dann ist die optimale Zerlegung des Stabs der Länge 8 in einen Stab der Länge 3 und einen der Länge 5. Damit wird ein Gesamtpreis von $p_3 + p_5 = 19$ erzielt.

Sei, für $1 \leq i \leq n$, g_i der für einen Stab der Länge i mit Preisen $p_1, \ldots, p_n$ maximal erzielbare Gewinn. Dann ist

$$g_i = \max\{p_1 + g_{i-1}, p_2 + g_{i-2} \ldots, p_{i-1} + g_1, p_i\}$$

da wir unterscheiden können ob nicht geschnitten wird (p_i) oder mindestens einmal geschnitten wird und dann nach Länge des ersten Stücks. Wir sehen also dass die optimale Lösung (die g_i bestimmt) zusammengesetzt ist durch optimale Lösungen für kleinere Instanzen des selben Problems (jene die die g_j bestimmen für $1 \leq j < i$). Das ist entscheidend für die Anwendbarkeit des dynamischen Programmierens. Außerdem ist die Gesamtanzahl der kleineren Instanzen, d.h. der g_j, die rekursiv zur Bestimmung von g_i benötigt werden und damit die Anzahl der Knoten im Teilproblemgraph klein (hier gleich i). Wenn wir $g_0 = 0$ setzen können wir g_i auch schreiben als

$$g_i = \max\{p_j + g_{i-j} \mid 1 \leq j \leq i\}.$$

Wir können jetzt eine bottom-up Berechnung der g_i in einem Datenfeld G durchführen, d.h. beginnend bei kleinen i in Richtung der größeren i. Um nicht nur den maximal erzielbaren Gewinn g_n sondern auch die Teilung $i_1 + \cdots + i_k = n$ des Stabs zu berechnen, verwenden wir das zusätzliche Datenfeld S das, für $i = 1, \ldots, n$, in $S[i]$ speichert wie lange das erste Stück der optimalen Zerteilung eines Stabs der Länge i ist. Die globale Zerteilung kann dann leicht aus S in Zeit $O(n)$ berechnet werden. Diese Vorgehensweise ist in Algorithmus 34 als Pseudocode formuliert. Die Laufzeit dieser Prozedur ist aufgrund der beiden verschachtelten Schleifen $\Theta(n^2)$.

Algorithmus 34 Stabzerlegung (bottom-up)

```
Prozedur StabZerlegung(P, n)
    Sei G[0, ..., n] neues Datenfeld
    Sei S[1, ..., n] neues Datenfeld
    G[0] := 0
    Für i = 1, ..., n
        m := 0
        Für j = 1, ..., i
            Falls P[j] + G[i − j] ≥ m dann
                m := P[j] + G[i − j]
                S[i] := j
            Ende Falls
        Ende Für
        G[i] := m
    Ende Für
    Antworte (G[n], S)
Ende Prozedur
```