man dann ja den besten Wert kennt. Andersherum funktioniert es meist, indem man z.B. den Wert k aus dem obigen Beispiel wie bei einer binären Suche immer weiter halbiert oder indem die Probleminstanz sukzessive geändert wird, hier z.B. indem einzelne Knoten entfernt werden und geprüft wird, ob ein Entscheidungsverfahren immer noch akzeptiert. Wir merken uns an dieser Stelle einfach, dass die Entscheidungs- und die Optimierungsvariante eines Problems so ähnlich sind, dass man, wenn man das eine schnell (also mit einem Polynomialzeitalgorithmus) lösen kann, auch das andere schnell (mit einem Polynomialzeitalgorithmus) lösen kann. Es genügt daher, sich auf die, für die Theorie angenehmeren, Entscheidungsvarianten zu konzentrieren und mit diesen eine für die Praxis relevante Theorie aufzubauen.

Fragen

1. Wenn sie für eine DTM eine Zeitschranke von $t(n) = n^2 + 2n$ haben, was gilt dann bei einer Eingabe w der Länge 5?
   a) Die Rechnung dauert mindestens 35 Schritte
   b) Die Rechnung dauert maximal 35 Schritte
   c) Die Rechnung dauert genau 35 Schritte
   d) w wird nach 35 Schritten akzeptiert
2. Das P in NP steht für Polynomialzeit. Wofür steht das N?
   a) nicht(-polynomiell)
   b) nicht-deterministisch
   c) nicht-(praktikabel)
3. Was gilt?
   a) $P \subseteq NP$
   b) $NP \subseteq P$
   c) $P \cap NP = \emptyset$
   d) unvergleichbar!

5.2 P und NP

Wir haben in dem vorherigen Abschnitt definiert, wie die Zeit- und die Platzkomplexität einer Turingmaschine bestimmt werden kann. Betrachtet man nun Polynome als Schranken, d.h. nimmt man z.B. $p(n) = n^2$ und betrachtet nun die Sprachen, die von Turingmaschinen akzeptiert werden, die bei einer Eingabe der Länge n maximal n^2 viele Schritte tätigen dürfen, so gelangt man

zur Komplexitätsklasse P, die wir ebenfalls schon im letzten Abschnitt gesehen haben. Die Komplexitätsklasse P umfasst gerade jene Sprache L, die von einer *deterministischen* Turingmaschine M akzeptiert werden können, deren Laufzeit durch ein Polynom beschränkt ist. Entsprechend wird die Komplexitätsklasse NP mittels *nichtdeterministischer* Turingmaschinen eingeführt. Wir wiederholen die Definition von P und NP hier einmal.

Definition 5.2.1 (Die Klassen P und NP)**.** *Wir definieren die Komplexitätsklassen P und NP. Sie enthalten Probleme, die von einer deterministischen bzw. einer nichtdeterministischen Turingmaschine mit einer polynomiellen Zeitschranke akzeptiert werden können.*

$$\begin{aligned} P &:= \{L \mid \textit{es gibt ein Polynom } p \textit{ und eine} \\ &\qquad p(n)\textit{-zeitbeschränkte DTM } A \textit{ mit } L(A) = L\} \\ &= \cup_{i\geq 1} DTIME(n^i) \\ NP &:= \{L \mid \textit{es gibt ein Polynom } p \textit{ und eine} \\ &\qquad p(n)\textit{-zeitbeschränkte NTM } A \textit{ mit } L(A) = L\} \\ &= \cup_{i\geq 1} NTIME(n^i) \end{aligned}$$

Die Komplexitätsklassen P und NP sind im Grunde Sprachfamilien (d.h. Mengen von Sprachen). Die Sprachen, die in P bzw. NP enthalten sind, sind aber gerade über Komplexitätsmaße definiert. Daher die Bezeichnung Komplexitätsklassen.

In der Definition von P wird nur gefordert, dass L von einer DTM A akzeptiert wird. Die Sprache ist also aufzählbar. Tatsächlich können die Sprachen aber auch entschieden werden. Die Kernidee ist, die Turingmaschine zunächst $p(n)$ ausrechnen zu lassen und dann einen Zähler bei jedem Schritt zu erhöhen. Erreicht man die $p(n)$ und hat noch nicht akzeptiert, so kann abgelehnt werden. Wir führen dies noch einmal aus.

Satz 5.2.2.

$$\begin{aligned} P &= \{L \mid L \textit{ wird von einem Algorithmus} \\ &\qquad \textit{in Polynomialzeit entschieden}\} \end{aligned}$$

Beweis. Die Richtung von rechts nach links ist klar, denn wenn L in Polynomialzeit entschieden wird, dann wird es auch in Polynomialzeit aufgezählt und damit ist es in P. Es ist nun also zu zeigen, dass jede in Polynomialzeit akzeptierbare Sprache auch in Polynomialzeit entscheidbar ist. Sei A ein Algorithmus der L in Zeit $O(n^k)$ akzeptiert. Es gibt dann eine Konstante c, so dass A die Sprache in höchstens $c \cdot n^k$ Schritten akzeptiert. Ein Algorithmus A', der L entscheidet, berechnet bei Eingabe x zunächst $s = c \cdot |x|^k$ und simuliert A dann s Schritte lang. Hat A akzeptiert, so akzeptiert auch A', hat A bisher

nicht akzeptiert, so lehnt A' die Eingabe ab. Damit entscheidet A' die Sprache L in $O(n^k)$. □

Die Klasse P ist deswegen von besonderer Bedeutung, weil man davon ausgeht, dass Probleme, die in P liegen effizient gelöst werden können. Dies ist natürlich nur eine auf Intuition basierende Aussage und bestimmt ist ein Algorithmus mit einer Laufzeit von n^{100} kein effizienter Algorithmus und ein Algorithmus mit einer Laufzeit von $2^{(n/10^{50})}$ dürfte für die allermeisten Anwendungen sehr effizient sein. Aus Erfahrung weiß man aber, dass solche Laufzeiten faktisch nicht oder nur bei konstruierten Beispielen auftreten. In der Praxis hat sich die Merkregel "Problem ist in P bedeutet, das Problem ist effizient lösbar" bewährt.

Ein typisches Problem in P ist z.B. die Frage nach einem kürzesten Pfad in einem Graphen

$$\texttt{PATH} = \{\langle G, s, t, k\rangle \mid \begin{array}{l} G = (V, E) \text{ ist ein ungerichteter Graph,} \\ s, t \in V, \\ k \geq 0 \text{ ist eine ganze Zahl und} \\ \text{es existiert ein } s\text{-}t\text{-Pfad in } G, \\ \text{der aus höchstens } k \text{ Kanten besteht.} \end{array} \}$$

Obige Formulierung sucht nicht nach einem kürzesten Pfad, sondern ist das zugehörige Entscheidungsproblem. Wir haben aber in dem vorherigen Abschnitt besprochen, wie man zu einem Entscheidungsproblem das zugehörige Optimierungsproblem löst und andersherum. Hier würde man für fallende k immer wieder das Entscheidungsproblem starten und so das Optimierungsproblem lösen.

Wandelt man dieses Problem nun geringfügig ab und sucht nicht nach einem kürzesten Pfad, sondern nach einem *längsten* Pfad, so wird das Problem deutlich komplizierter und scheint tatsächlich nicht mehr in P zu liegen:

$$\texttt{L-PATH} = \{\langle G, s, t, k\rangle \mid \begin{array}{l} G = (V, E) \text{ ist ein ungerichteter Graph,} \\ s, t \in V, \\ k \geq 0 \text{ ist eine ganze Zahl und} \\ \text{es existiert ein } s\text{-}t\text{-Pfad in } G, \\ \text{der aus } \textit{mindestens } k \text{ Kanten besteht.} \end{array} \}$$

In NP liegt dieses Problem aber ganz bestimmt, denn wir können mit einer nichtdeterministischen Turingmaschine alle Pfade zwischen s und t raten und dann prüfen, ob ein Pfad der Länge k enthalten ist.

Etwas anderes ist hier aber auch auffällig. Während es zwar schwierig zu sein scheint eine Lösung für `L-PATH` zu berechnen, so kann, *gegeben* ein Pfad, schnell *überprüft* werden, ob dies eine Lösung ist. Diese Überlegung führt zu einer alternativen Beschreibung von NP. Wir definieren dazu zunächst einen *Verifikationsalgorithmus*. Dieser erhält zwei Eingaben. Die eigentliche Probleminstanz

x und eine weitere Zeichenkette y, *Zertifikat* genannt. Der Algorithmus prüft dann, ob y eine Lösung der Probleminstanz x ist.

Definition 5.2.3 (Verifikationsalgorithmus)**.** *Ein* Verifikationsalgorithmus A *ist ein deterministischer Algorithmus mit zwei Argumenten $x, y \in \Sigma^*$, wobei x die gewöhnliche Eingabe und y ein* Zertifikat *ist. A* verifiziert *x, wenn es ein Zertifikat y gibt mit $A(x, y) = 1$. Die von A verifizierte Sprache ist*

$$L = \{x \in \{0,1\}^* \mid \exists y \in \{0,1\}^* : A(x, y) = 1\}.$$

Man beachte, dass auch in diesem Fall die x die Sprache ausmachen. Das Zertifikat y kann vom Algorithmus x genutzt werden, um zu entscheiden, ob $x \in L$ gilt oder nicht.

Die Klasse NP kann nun auch beschrieben werden, in dem man verlangt, dass ein Verifikationsalgorithmus mit polynomieller Laufzeit existiert. Zusätzlich verlangt man, dass y nur eine polynomielle Länge in x hat.

Definition 5.2.4 (NP (alternative Definition))**.** *$L \in NP$ gdw. ein Verifikationsalgorithmus A mit zwei Eingaben und mit polynomialer Laufzeit existiert, so dass für ein c*

$$L = \{x \in \{0,1\}^* \mid \begin{array}{l} \text{es existiert ein Zertifikat } y \text{ mit } |y| \in O(|x|^c), \\ \text{so dass } A(x, y) = 1 \text{ gilt} \end{array} \}$$

gilt.

Betrachten wir die Definition noch einmal genauer. Ein Problem ist in NP, wenn ein Verifikationsalgorithmus existiert, der L akzeptiert. Soweit so gut. Ein Verifikationsalgorithmus unterscheidet sich von einem "normalen" Algorithmus dadurch, dass er neben der Instanz x des Problems, das er lösen soll, noch eine zusätzliche Eingabe y erhält. Dieses y, das Zertifikat, soll ihm helfen, zu entscheiden, ob er x akzeptieren soll oder nicht. Betrachten wir z.B. das Erfüllbarkeitsproblem der Aussagenlogik. Die Eingabeinstanz ist eine Formel F, das Zertifikat könnte dann eine (erfüllende) Belegung sein. Der Verifikationsalgorithmus überprüft dann lediglich, ob die Belegung tatsächlich eine erfüllende ist. Um NP zu erfassen darf man nun nicht einen uneingeschränkten Verifikationsalgorithmus nehmen, da man damit dann zu viel könnte. Der Verifikationsalgorithmus darf nur eine polynomielle Laufzeit haben. Ferner darf y nur polynomiell in der Länge von x sein. Z.B. könnte es sein, dass y nur maximal von der Länge $|x|^2$ sein darf. Warum diese Einschränkung? Sonst wird das Zertifikat wieder zu mächtig und außerdem könnte man sonst eine deutlich längere Laufzeit im Vergleich mit x haben. Ist y nämlich deutlich länger als x, also z.B. exponentiell in $|x|$, dann arbeitet der Verifikationsalgorithmus zwar vielleicht polynomiell in der Länge von x *und* y, was dann aber exponentiell in der Länge von x sein kann (da y ja so lang ist) und dies könnte der nichtdeterministische Algorithmus dann nicht mehr schaffen (siehe die nachfolgende

Konstruktionsidee). Ist außerdem das Zertifikat länger als polynomiell in x, so kann es von einer nichtdeterministischen, polynomialzeitbeschränkten Turingmaschine nicht geraten werden, da diese ja nur polynomiell viele Schritte in der Länge von x machen darf (siehe ebenfalls die nachfolgende Konstruktionsidee).

Wir haben oben gesagt, dass wir NP alternativ mittels der Verifkationsalgorithmen definieren. Streng genommen würde man sagen, dass eines die Definition ist und dann von der zweiten zeigen, dass sie zur ersten äquivalent ist. Wir wollen dies hier nicht im Detail machen, sondern einfach mit beiden Formulierungen als Definition arbeiten.

Es lohnt sich aber, sich kurz zu überlegen, warum die beiden Definitionen äquivalent sind. Die Kernidee ist die folgende: Ein nichtdeterministischer Algorithmus kann Werte raten. Eine bestimmte Folge dieser geratenen Werte führt dann zum Erfolg. Ein Zertifikat kann dann gerade aus den richtigen dieser geratenen Werte bestehen. Betrachten wir z.B. das folgende Mengenpartitionsproblem:

> Gegeben sei eine Menge $S \subseteq \mathbb{N}$. Gesucht ist eine Menge $A \subseteq S$, so dass $\sum_{x \in A} x = \sum_{x \in \overline{A}} x$ gilt.

Ein nichtdeterministischer Algorithmus rät für jedes Element aus S, ob es in A sein soll oder nicht und überprüft dann, ob die beiden Summen identisch sind. Ein Verifikationsalgorithmus erhält als Zertifikat gerade die Elemente der Menge A und prüft dann, ob die Summe stimmt. Wenn es eine Menge A gibt, mit der dies klappt, so wird der nichtdeterministische diese raten und für den Verifikationsalgorithmus gibt es diese als Zertifikat.

Wir kennen nun zwei Darstellungen für NP. Will man nun also zeigen, dass ein Problem in NP ist, so kann man entweder zeigen, dass es eine nichtdeterministische Turingmaschine gibt, die dieses Problem in Polynomialzeit löst oder man kann zeigen, dass es einen Verifikationsalgorithmus gibt, der in Polynomialzeit arbeitet und das Problem löst. Beides genügt zwar, um zu zeigen, dass das Problem in NP ist. Wenn man ein solche Problem aber in der Praxis lösen will, dann hilft das so nicht viel, denn man kann ja weder eine nichtdeterministische Maschine bauen, noch einen Verifikationsalgorithmus entwerfen (denn wo sollte das Zertifikat herkommen?). Wie also löst man ein solches Problem *deterministisch*?

Pause to Ponder: Wie löst man ein NP-Problem mit einer deterministischen Turingmaschine?

Tatsächlich haben wir dieses Problem schon gelöst. Wir haben ja bereits im vorherigen Kapitel gezeigt, dass jede nichtdeterministische Turingmaschine von einer deterministischen Turingmaschine simuliert werden kann. Dies könnten wir hier also einfach nutzen, um die NTM, die ein Problem aus NP akzeptiert, durch eine DTM zu simulieren und so das Problem deterministisch zu entscheiden. Äquivalent ist die Überlegung jedes mögliche Zertifikat durchzuprobieren. Man beachte hierfür, dass die Zertifikate in der Länge beschränkt sind. Es kann

also nur endlich viele geben. Dies ist allerdings sehr aufwändig, da es sehr viele Zertifikate einer Länge k geben kann (hat man m Symbole, so gibt es dann m^k Zeichenketten, die im schlimmsten Fall alle probiert werden müssen). Die Simulation einer NTM ist allerdings auch sehr teuer. Im Grunde genommen sind beides *Brute-Force-Methoden*, bei denen alle Möglichkeiten durchprobiert werden – alle Rechnungen der NTM oder eben alle Zertifikate. Dies ergibt eine obere Schranke für einen deterministischen Algorithmus, der ein NP-Problem löst.

Satz 5.2.5. *Sei $L \in NP$, dann gibt es ein $k \in \mathbb{N}$ und einen deterministischen Algorithmus, der L in $2^{O(n^k)}$ entscheidet.*

Beweis. Beweisskizze/Idee: Ist $L \in NP$, so gibt es einen Verifikationsalgorithmus in $O(n^k)$ (n ist die Eingabelänge). Das Zertifikat y hat eine Länge in $O(n^c)$. Man geht alle $2^{O(n^c)}$ Zertifikate durch und führt für jeden den Verifikationsalgortihmus aus. Dieses Verfahren ist (unter der sinnvollen Annahme $k > c$) in $2^{O(n^k)}$.

Will man hier genauer sein, so müsste man zunächst sagen, dass das Verfahren eigentlich in $2^{O(n^c)} \cdot O(n^k)$ ist. Wegen $n^k \leq 2^{n^k}$ kann man dies nach oben mit $2^{O(n^c)} \cdot 2^{O(n^k)}$ abschätzen. Nun darf man $k \geq c$ annehmen, da sonst der Verifikationsalgorithmus eine Laufzeit hätte bei der er sich gar nicht das ganze Zertifikat ansehen kann. Damit kann man nach oben durch $2^{O(n^k)} \cdot 2^{O(n^k)}$ abschätzen. Dies ist gleich $2^{2 \cdot O(n^k)} = 2^{O(n^k)}$. □

Definition 5.2.6 (Typische NP-Probleme)**.** *Die folgenden Probleme sind typische Beispiele für Problem in NP.*

1. ***Name**: Mengenpartitonsproblem*

   ***Gegeben**: Eine endliche Menge $S \subseteq \mathbb{N}$*

   ***Frage**: Gibt es eine Menge $A \subseteq S$, so dass $\sum_{x \in A} x = \sum_{x \in \overline{A}} x$ gilt?*

2. ***Name**: Teilsummenproblem*

   ***Gegeben**: Eine endliche Menge $S \subset \mathbb{N}$ und ein $t \in \mathbb{N}$.*

   ***Frage**: Gibt es eine Menge $S' \subseteq S$ mit $\sum_{s \in S'} s = t$?*

3. ***Name**: Cliquenproblem*

   ***Gegeben**: Ein ungerichteter Graph $G = (V, E)$ und ein $k \in \mathbb{N}$.*

   ***Frage**: Enthält G eine Clique, d.h. ein vollständigen Graphen, der Größe k als Teilgraph?*

4. ***Name**: Färbungsproblem*

   ***Gegeben**: Ein ungerichteter Graph $G = (V, E)$ und ein $k \in \mathbb{N}$.*

> ***Frage**: Kann G mit k Farben gefärbt werden? D.h. gibt es eine Funktion $c : V \to \{1, \ldots, k\}$ derart, dass $c(u) \neq c(v)$ für jede Kante $\{u, v\} \in E$ gilt?*

Alle oben genannten Probleme sind in NP und damit schnell nichtdeterministisch lösbar. Die besten bekannten deterministischen Algorithmen arbeiten allerdings im Grunde so wie oben im allgemeinen Fall illustriert mit einer Brute-Force-Methode und benötigen daher expoentielle Laufzeit.

Die Aussage stimmt nicht ganz, da mittlerweile für etliche NP-Probleme auch etwas bessere Algorithmen bekannt sind. Diese Algorithmen sind allerdings weiterhin exponentiell, so dass die generelle Aussage gültig bleibt. Aus Praxissicht sind diese Algorithmen dann aber immerhin etwas besser als die Brute-Force-Methode und tatsächlich ist es ein aktives Foschungsfeld hier Algorithmen zu entwickeln, die mit immer größeren Eingabeinstanzen fertig werden.

Die Frage ist nun, geht es wirklich nicht schneller? Wenn wir dies beantworten könnten, so könnten wir entweder versuchen schnellere Algorithmen zu entwerfen oder können dies ohne schlechtes Gewissen unterlassen. Um die Suche nach unteren Schranken soll es im nächsten Abschnitt zur NP-Vollständigkeit gehen.