P, NP und NP-Vollständigkeit

Mit der Turing-Maschine haben wir einen Formalismus kennengelernt, um über das Berechenbare nachdenken und argumentieren zu können. Wie unsere bisherigen Automatenmodelle kann die Turing-Maschine Sprachen akzeptieren. Aus algorithmischer Sicht interessanter ist aber, dass die Turing-Maschine Sprachen *entscheiden* kann. Bei solchen Sprachen hält sie dann auf jeder Eingabe nach endlich vielen Schritten an und gibt ein Ergebnis aus. Da man Probleme in Sprachen "kodieren" kann, können wir so über algorithmische Probleme und ihre Lösbarkeit nachdenken.

Ein Problem kann dann entscheidbar oder unentscheidbar sein. Wenn wir jedoch wissen, dass ein Problem entscheidbar ist, genügt dies im Allgemeinen noch nicht, um das Problem in der Praxis tatsächlich lösen zu können. Selbst wenn wir einen Algorithmus haben, könnte es nämlich sein, dass dieser viel zu ineffizient ist, als dass man ihn einsetzen könnte. Nehmen wir beispielsweise an, wir haben einen Algorithmus, der n Zahlen mit 2^n vielen Schritten sortieren kann. Solch ein Algorithmus würde schon bei nur 50 zu sortierenden Zahlen $2^{50} \approx 1$ Billarde Schritte benötigen. Bei 100 oder gar 1000 zu sortierenden Zahlen wächst die Anzahl der Schritte in kaum vorstellbare Größen.

Neben der recht groben Unterteilung in entscheidbare und unentscheidbare Probleme, wäre es also wünschenswert die entscheidbaren Probleme danach weiter zu unterteilen, wie schnell sie gelöst werden können. Für die Turing-Maschine werden hierfür Komplexitätsmaße derart eingeführt, dass jeder Schritt einer DTM eine Zeiteinheit zählt und jede besuchte Zelle auf dem Band eine Platzeinheit. Um wie im Beispiel oben eine Funktion wie z.B. $t(n) = 2^n$ als Schranke für die Anzahl der benutzten Schritte benutzen zu können, betrachtet man nun alle Wörter der Länge n. Für jedes dieser Wörter ermittelt man, wie lange die DTM arbeitet. Die Länge der längsten Rechnung muss dann stets kleiner-gleich $t(n)$ sein. Ein Beispiel: Hat man als Zeitschranke für die Laufzeit einer DTM die Funktion $t(n) = n^2 + 4 \cdot n$, so darf die DTM auf einem Wort der Länge 5 maximal $t(5) = 5^2 + 4 \cdot 5 = 45$ Schritte arbeiten. Dass die DTM bei einigen Eingabe der Länge 5 weniger Schritte braucht, ist nicht wichtig. Wichtig hier ist, dass sie bei keinem Wort der Länge 5 mehr als $t(5)$ Schritte benötigt. (Es kann sogar sein, dass sie bei keinem Wort der Länge 5 die Grenze von 45 erreicht. Evtl. hat man die Funktion nämlich deswegen so gewählt, weil dies gut zu anderen n passt.)

Bei einer NTM wählt man für die Zeitschranke die kürzeste akzeptierende Rechnung, verfährt sonst aber wie eben. Hat eine NTM also die Zeitschranke

$t(n) = n^2 + 4 \cdot n$, so bedeutet dies, dass sie ein Wort der Länge 5 nach spätestens 45 Schritten akzeptiert, d.h. dass es mindestens eine Erfolgsrechnung mit maximal dieser Länge gibt, wenn sie das Wort akzeptiert. Man kann eine NTM dann aber auch so bauen, dass sie $t(n)$ zunächst berechnet und dann jede Rechnung nach $t(n)$ vielen Schritten beendet. Die neue NTM hat dann eine etwas andere Zeitschranke (weil sie $t(n)$ noch berechnen muss und bei jeder Rechnung einen Zähler erhöht und darauf achtet, wann $t(n)$ erreicht wird), aber diese macht kaum einen Unterschied insb. wenn es um die Komplexitätsklassen geht, die wir jetzt einführen wollen.

Einmal bis hierhin zusammengefasst:

- Die Einteilung in entscheidbar / unentscheidbar genügt uns nicht. Wir wollen auch Aussagen darüber machen können, ob ein entscheidbares Problem *effizient* gelöst werden kann, d.h. wir wollen über den Zeit- und ggf. auch den Speicherbedarf eines Algorithmus zur Lösung eines Problems Aussagen treffen können.
- Hierzu führen wir Zeit- und Platzkomplexitätsmaße für DTMs und NTMs ein.
- Bei der DTM zählt bei einer Rechnung jede Konfiguration als ein Schritt und jede in der Rechnung benutzte Bandzelle als eine Platzeinheit.
- Bei der NTM nehmen wir die kürzestes Erfolgsrechnung auf einem Wort und zählen dann so wie bei der DTM.
- Eine Zeitschranke ist dann eine Funktion t, die als Argument die Länge eines Wortes kriegt und dann die maximale Anzahl der Schritte, die die DTM machen darf zurückliefert (bzw. die maximale Anzahl der Schritte in der kürzesten Erfolgsrechnung bei einer NTM).
- Bei der Platzschranke verfährt man im Grunde genauso, nur dass man die Anzahl der benutzten Bandzellen zählt.

Da man bei der Funktion t alle Wörter einer Länge betrachtet, spricht man von einer *worst-case*-Abschätzung.

Wir interessieren uns nun für Probleme, die von DTMs mit bestimmten Zeit- oder auch Platzschranken akzeptiert werden können. Von herausragender Bedeutung sind hierbei Polynome. Das hängt damit zusammen, dass man einerseits mit Polynomen gut rechnen kann, andererseits Polynome aber ein im Vergleich mit dem Argument akzeptables Wachstum aufweisen. Im Allgemeinen geht man davon aus, dass wenn die Laufzeit eines Algorithmus oder einer Turing-Maschine durch ein Polynom beschränkt ist, der Algorithmus effizient ist.

Die Klasse aller Probleme, die von einer deterministischen Turing-Maschine in Polynomialzeit entschieden werden können, wird mit P bezeichnet. Zu jeder solchen Turing-Maschine gibt es also ein Polynom p derart, dass die DTM bei Eingaben der Länge n maximal $p(n)$ viele Schritte macht.

Ganz ähnlich definiert man sich die Klasse NP als die Klasse aller Probleme, die von einer *nicht*deterministischen Turing-Maschine in Polynomialzeit entschieden werden können.

Mit Polynomen zu arbeiten hat noch einen weiteren wichtigen Grund: Die erweiterte Church-Turing-These besagt, dass alle Berechnungsmodelle, die wie die Turing-Maschine die Klasse der intuitiv berechenbaren Funktionen erfassen, mit polynomiellen Aufwand ineinander transformiert werden können. D.h. hat man eine Rechnung der Länge k in einem Berechnungsmodell B, so gibt es ein Polynom p derart, dass man die Rechnung in B in eine Turing-Maschine und in eine Rechnung dieser Turing-Maschine transformieren kann. Die neue Rechnung hat dabei höchstens die Länge $p(k)$ und ist damit höchstens polynomiell länger als die ursprüngliche Rechnung. Während dies eine These ist, die man für jedes neu aufkommende Berechnungsmodell erneut bestätigen muss, kann man tatsächlich zeigen, dass dies für unsere heutigen Computer und die Turing-Maschine gilt. Wir können also genauso gut in einer imperativen Programmiersprache einen Algorithmus schreiben und argumentieren, dass dieser das Problem in Polynomialzeit löst und wissen dann auch, dass dieses Problem auch auf der TM in Polynomialzeit lösbar ist, also in der Klasse P liegt. Daher trifft man in der Literatur auch oft nur auf Beschreibungen, was eine TM tun müsste, oder auch gleich auf Pseudocode in einer meist imperativen Programmiersprache.

Wenn wir nun also für ein Problem einen Algorithmus im Pseudocode angeben können und argumentieren können, dass dieser Algorithmus im worst-case in Polynomialzeit läuft, dann wissen wir, dass das Problem in P ist - was bedeutet, dass das Problem effizient gelöst werden kann. Für das ganz zu Anfang angesprochene Sortierproblem gibt es bspw. sehr effiziente Algorithmen, die das Problem ungefähr in $n \cdot \log n$ lösen. Um 1000 Zahlen zu sortieren braucht man dann nur ungefähr 10000 Schritte. (Man beachte wieder, dass dies eine worst-case-Abschätzung ist. Liegen die Zahlen schon sortiert vor, so braucht man vermutlich nur einmal über die Zahlen zu lesen, um dies zu erkennen, und braucht dann nur 1000 Schritte.)

Es gibt nun aber viele Probleme, für die es uns nicht so recht gelingen will, einen Algorithmus in P zu entwickeln. Viele dieser Probleme liegen aber in NP. Dies hängt damit zusammen, dass ein NP-Algorithmus zunächst eine mögliche Lösung nichtdeterministisch *raten* kann und diese dann *überprüfen* (oder *verifizieren*) kann. Betrachten wir bspw. das Problem in einem ungerichteten Graphen G mit n Knoten eine Clique der Größe k zu finden, d.h. k Knoten, die alle paarweise miteinander verbunden sind. Hierfür ist kein Algorithmus in P bekannt. Ein NP-Algorithmus ist aber schnell gefunden: Zunächst rät man sich k Knoten und überprüft dann lediglich noch, ob diese tatsächlich alle miteinander verbunden sind. Wenn der Graph eine k Clique hat, dann gibt es auch eine Erfolgsrechnung in der genau diese k Knoten geraten werden. Da dieses Raten schnell geht (man geht einmal die Knotenmenge durch und bestimmt für jeden Knoten, ob er in der angenommenen Clique sein soll oder nicht) und auch das Überprüfen, ist das ganze Problem in NP lösbar. (Wir machen dies noch in der Vorlesung genauer.)

Ein NP-Algorithmus ist nun aber nichtdeterministisch und dies lässt sich

so nicht implementieren. Kann man den Algorithmus in einen deterministischen umwandeln und so das Problem lösen? (Wir hatten ja zumindest schon gesehen, dass man eine NTM in eine DTM umwandeln kann!) Tatsächlich geht dies, aber die Zeitschranke des deterministischen Algorithmus ist dann *exponentiell*! Damit ist der Algorithmus dann nicht mehr praktikabel.

Geht dies auch schneller? Kann man eine Clique also vielleicht auch deterministisch in Polynomialzeit berechnen? Genau hier scheitern wir kläglich. Wir können weder zeigen, dass dies geht, noch, dass dies nicht geht! Wir vermuten zwar sehr stark, dass das Problem nicht in P ist, aber es gelingt uns nicht, dies zu zeigen. Man sagt, dass unsere Möglichkeiten *untere Schranke* von Algorithmen zu zeigen noch sehr eingeschränkt sind. (Wobei hier mit unteren Schranken untere Zeitschranken gemeint sind. Für Platzschranken haben wir diese Probleme aber auch.)

Wir können aber immerhin zeigen, dass es sehr, sehr unwahrscheinlich ist, dass bestimmte Probleme (wie z.B. auch Clique) in P liegen. Dies ist die Theorie der NP-Vollständigkeit. Hier versucht man zu formalisieren, dass ein Problem zu den "schwierigsten" in NP gehört (und daher dann wahrscheinlich nicht in P liegt). Man macht dies so: Ein Problem L ist genau dann NP-vollständig, wenn es 1. in NP liegt und 2. jedes Problem aus NP in Polynomialzeit auf L reduziert werden kann. (Gilt nur 2. ist das Problem NP-schwierig.) Eine Reduktion von L' nach L ist dabei eine berechenbare Funktion f mit der Eigenschaft $x \in L'$ gilt genau dann, wenn $f(x) \in L$ gilt. Die Idee ist, dass eine Reduktion ein Problem in ein anderes "transformiert". Nehmen wir an, wir hätten eine Reduktion von L' nach L und könnten L schnell lösen. Dann könnten wir jetzt eine Eingabe x, von der wir testen wollen, ob sie in L' liegt, mit der Reduktion in eine Instanz $f(x)$ von L transformieren und schnell überprüfen, ob $f(x) \in L$ gilt. Gilt dies, gilt auch $x \in L'$. Gilt $f(x) \in L$ nicht, so gilt auch $x \in L$ nicht. Wir können dann also auch das Problem L' schnell lösen. Hierfür ist aber noch wichtig, dass auch die Reduktion schnell geht. Daher ist oben auch gefordert, dass jedes Problem aus NP sich *in Polynomialzeit* auf unser neues Problem reduzieren lässt.

Mittlerweile sind viele NP-vollständige Probleme bekannt. Könnte man eines davon, sagen wir L, effizient (soll heißen: in P) lösen, so würde $P = NP$ folgen. Denn man könnte wie eben beschrieben ein beliebiges Problem L' aus NP nehmen, es auf L in Polynomialzeit reduzieren und dann noch den P-Algorithmus für L nutzen. Damit wäre dann auch L' in Polynomialzeit gelöst und liegt also in P.

Da man allgemein vermutet, dass $P = NP$ nicht gilt, ist ein Nachweis, dass ein Problem NP-vollständig ist, gleichbedeutend damit, dass es (aller Wahrscheinlichkeit nach) nicht in P liegt.

Um zu zeigen, dass ein Problem L NP-vollständig ist, reduziert man aber üblicherweise nicht alle Probleme aus NP auf L, sondern man nimmt sich ein bereits als NP-vollständig nachgewiesenes Problem L' und reduziert dieses auf L. Man kann dann jedes Problem aus NP auf L reduzieren, indem man es erst auf L' und dieses dann auf L reduziert. Man kann zeigen, dass zwei so hintereinander ausgeführte Polynomialzeitreduktionen wieder eine Polynomialzeitreduktion ergeben, daher klappt dies.

In der Vorlesung werden wir die Komplexitätsklassen P und NP sowie die der NP-schwierigen und der NP-vollständigen Probleme einführen und mit einem Reduktionsbeweis das Clique-Problem als NP-vollständig nachweisen.

Selbsttest (Lösung auf Seite 6)

1. Wann ist ein Problem in P?
2. Wann ist ein Problem in NP?
3. Wann ist ein Problem NP-schwierig?
4. Wann ist ein Problem NP-vollständig?
5. Was müssen Sie tun, wenn Sie ein Problem als NP-vollständig nachweisen wollen?
6. Was wissen Sie, wenn ein Problem NP-vollständig ist?

Lösungen zum Selbsttest auf Seite 5

1. **F:** Wann ist ein Problem in P?

   A: Wenn es eine deterministische Turing-Maschine gibt, die das Problem in Polynomialzeit entscheidet. D.h. es gibt ein Polynom p, so dass bei Eingabe w mit $|w| = n$ die TM maximal $p(n)$ Schritte macht und dann entscheidet, ob w in der Sprache, die zu dem Problem gehört, ist oder nicht.

2. **F:** Wann ist ein Problem in NP?

   A: Wenn es eine *nicht*deterministische Turing-Maschine gibt, die das Problem in Polynomialzeit entscheidet. Man beachte, dass dafür nur verlangt wird, dass die kürzeste Erfolgsrechnung innerhalb der Zeitschranke ist. Man kann dies aber so umbauen, dass alle Rechnungen innerhalb der Zeitschranke sind, indem man die Turingmaschine vorher $p(n)$ berechnen lässt (wobei p die Zeitschranke ist) und sie dann ihre Schritte zählen lässt.

3. **F:** Wann ist ein Problem NP-schwierig?

   A: Lässt sich jedes $L \in NP$ in Polynomialzeit auf ein L' reduzieren, so ist L' NP-schwierig. Die Reduktion f erfüllt dabei die Eigenschaft $x \in L \Leftrightarrow f(x) \in L'$, d.h. wenn x in L ist (eine Ja-Instanz), dann ist $f(x)$ in L'. Ist hingegen x nicht in L (eine Nein-Instanz), so ist auch $f(x)$ nicht in L'. Ja-Instanzen werden also auf Ja-Instanzen abgebilet und Nein-Instanzen auf Nein-Instanzen. Die Reduktion *transformiert* also ein Problem in ein anderes, erhält dabei aber die Eigenschaft, ob eine Ja- oder Nein-Instanz vorliegt (und das i.A. ohne zu berechnen, welche Art von Instanz vorliegt). Wichtig ist noch, dass die Reduktion in Polynomialzeit berechnet werden kann!

4. **F:** Wann ist ein Problem NP-vollständig?

   A: Wenn es NP-schwierig ist und zusätzlich noch selbst in NP liegt.

5. **F:** Was müssen Sie tun, wenn Sie ein Problem als NP-vollständig nachweisen wollen?

   A: Zeigen, dass es in NP ist und zeigen, dass jedes Problem aus NP sich auf dieses Problem in Polynomialzeit reduzieren lässt. Letzteres macht man üblicherweise, indem man ein NP-vollständiges Problem nimmt und (nur) dieses dann auf das neue in Polynomialzeit reduziert. Aufgrund der Transitivität der Reduktionen lassen sich dann alle Probleme aus NP auf das neue reduzieren.

6. **F:** Was wissen Sie, wenn ein Problem NP-vollständig ist?

   A: Wir wissen dann, dass es aller Wahrscheinlichkeit nach nicht in P liegt. Statt einen P-Algorithmus zu suchen, was vermutlich hoffnungslos ist, kann man sich dann mit anderen Dingen beschäftigen, wie z.B. einen Approximationsalgorithmus zu entwickeln oder zu überprüfen, ob man das Problem vielleicht gar nicht in seiner Allgemeinheit zu lösen braucht.