Algorithmen und Datenstrukturen
Kapitel 6
Komplexitätstheorie
Einführung in *P* und *NP*

Frank Heitmann
heitmann@informatik.uni-hamburg.de

11. November 2015

Organisatorisches

1. Raumwechsel: Gr. 19 (Fr 12-14, F-334) diese Woche in D-129.
2. Studie zum Arbeitsverhalten von Studierenden unter Leitung von Prof. Maria Knobelsdorf startet!

Motivation

Alle bisher behandelten Probleme haben eins gemeinsam:

$\Rightarrow$ Algorithmen in $O(n^k)$ (Polynomialzeit) waren möglich!

Dies ist nun (scheinbar) nicht immer der Fall, aber unsere Fähigkeiten untere Schranken zu beweisen sind bisher sehr beschränkt!

Motivation

Alle bisher behandelten Probleme haben eins gemeinsam:

$\Rightarrow$ Algorithmen in $O(n^k)$ (Polynomialzeit) waren möglich!

Dies ist nun (scheinbar) nicht immer der Fall, aber unsere Fähigkeiten untere Schranken zu beweisen sind bisher sehr beschränkt!

Motivation

Alle bisher behandelten Probleme haben eins gemeinsam:

$\Rightarrow$ Algorithmen in $O(n^k)$ (Polynomialzeit) waren möglich!

Dies ist nun (scheinbar) nicht immer der Fall, aber unsere Fähigkeiten untere Schranken zu beweisen sind bisher sehr beschränkt!

Motivation

- Bestimmung eines *kürzesten Pfades* in einem Graphen können wir in $O(V^2)$ lösen (Dijkstra).
- Die Bestimmung eines *längsten Pfades* scheint viel schwieriger zu sein. Bisher ist kein Algorithmus bekannt, der dieses Problem in $O(n^k)$ für *irgendein* k löst!

Anmerkung (zur Nachbereitung)

Es gibt viele ähnlich klingende Probleme bei denen wir das eine lösen können, das andere nicht (z.B. Euler-Zug und Hamilton-Kreis, 2-Färbbarkeit und 3-Färbbarkeit, ...). Wichtiger ist aber, dass es unzählige Probleme gibt, die wir bisher nicht in Polynomialzeit lösen können - und wir wissen weder, ob dies tatsächlich der Fall ist (oder wir nur nicht die richtige Idee haben), noch, woran dies eigentlich liegt.

Motivation

- Bestimmung eines *kürzesten Pfades* in einem Graphen können wir in $O(V^2)$ lösen (Dijkstra).
- Die Bestimmung eines *längsten Pfades* scheint viel schwieriger zu sein. Bisher ist kein Algorithmus bekannt, der dieses Problem in $O(n^k)$ für *irgendein* k löst!

Anmerkung (zur Nachbereitung)

Es gibt viele ähnlich klingende Probleme bei denen wir das eine lösen können, das andere nicht (z.B. Euler-Zug und Hamilton-Kreis, 2-Färbbarkeit und 3-Färbbarkeit, ...). Wichtiger ist aber, dass es unzählige Probleme gibt, die wir bisher nicht in Polynomialzeit lösen können - und wir wissen weder, ob dies tatsächlich der Fall ist (oder wir nur nicht die richtige Idee haben), noch, woran dies eigentlich liegt.

Motivation

- Bestimmung eines *kürzesten Pfades* in einem Graphen können wir in $O(V^2)$ lösen (Dijkstra).
- Die Bestimmung eines *längsten Pfades* scheint viel schwieriger zu sein. Bisher ist kein Algorithmus bekannt, der dieses Problem in $O(n^k)$ für *irgendein* k löst!

Anmerkung (zur Nachbereitung)

Es gibt viele ähnlich klingende Probleme bei denen wir das eine lösen können, das andere nicht (z.B. Euler-Zug und Hamilton-Kreis, 2-Färbbarkeit und 3-Färbbarkeit, ...). Wichtiger ist aber, dass es unzählige Probleme gibt, die wir bisher nicht in Polynomialzeit lösen können - und wir wissen weder, ob dies tatsächlich der Fall ist (oder wir nur nicht die richtige Idee haben), noch, woran dies eigentlich liegt.

Motivation

- Bestimmung eines *kürzesten Pfades* in einem Graphen können wir in $O(V^2)$ lösen (Dijkstra).
- Die Bestimmung eines *längsten Pfades* scheint viel schwieriger zu sein. Bisher ist kein Algorithmus bekannt, der dieses Problem in $O(n^k)$ für *irgendein* k löst!

Anmerkung (zur Nachbereitung)

Es gibt viele ähnlich klingende Probleme bei denen wir das eine lösen können, das andere nicht (z.B. Euler-Zug und Hamilton-Kreis, 2-Färbbarkeit und 3-Färbbarkeit, ...). Wichtiger ist aber, dass es unzählige Probleme gibt, die wir bisher nicht in Polynomialzeit lösen können - und wir wissen weder, ob dies tatsächlich der Fall ist (oder wir nur nicht die richtige Idee haben), noch, woran dies eigentlich liegt.

Motivation

Wir würden nun gerne eine untere Schranke für obiges Problem zeigen, also z.B.

$$\texttt{LONGEST-PATH} \in \Omega(n^k)$$

für alle k (bzw. $\texttt{LONGEST-PATH} \notin O(n^k)$ für alle k). Das schaffen wir aber bisher nicht!

Anmerkung

Von praktischer Relevanz ist dies, weil man bei neu aufkommenden Problemen ja wissen möchte, ob es sich lohnt Mühe, in die Entwicklung eines schnellen Algorithmus zu stecken!

Motivation

Wir würden nun gerne eine untere Schranke für obiges Problem zeigen, also z.B.

$$\texttt{LONGEST-PATH} \in \Omega(n^k)$$

für alle k (bzw. $\texttt{LONGEST-PATH} \notin O(n^k)$ für alle k). Das schaffen wir aber bisher nicht!

Anmerkung

Von praktischer Relevanz ist dies, weil man bei neu aufkommenden Problemen ja wissen möchte, ob es sich lohnt Mühe, in die Entwicklung eines schnellen Algorithmus zu stecken!

Motivation

Wir würden nun gerne eine untere Schranke für obiges Problem zeigen, also z.B.

$$\texttt{LONGEST-PATH} \in \Omega(n^k)$$

für alle k (bzw. $\texttt{LONGEST-PATH} \notin O(n^k)$ für alle k). Das schaffen wir aber bisher nicht!

Anmerkung

Von praktischer Relevanz ist dies, weil man bei neu aufkommenden Problemen ja wissen möchte, ob es sich lohnt Mühe, in die Entwicklung eines schnellen Algorithmus zu stecken!

Motivation - Zusammenfassung

Zusammengefasst:

- Beobachtung 1: Unsere Fähigkeit untere Schranken für Probleme zu beweisen sind im Moment sehr beschränkt.
- Beobachtung 2: Wir kennen aber viele Probleme, für die uns keine effizienten Algorithmen einfallen.
- Das Ziel: Wie kann man die Annahme, dass es für ein (möglicherweise neues) Problem keinen effizienten Algorithmus gibt, möglichst gut untermauern?
- Auftritt *NP*-Vollständigkeit! Hiermit werden wir die 'Kompliziertheit' eines Problems formalisieren.

Motivation - Zusammenfassung

Zusammengefasst:

- Beobachtung 1: Unsere Fähigkeit untere Schranken für Probleme zu beweisen sind im Moment sehr beschränkt.
- Beobachtung 2: Wir kennen aber viele Probleme, für die uns keine effizienten Algorithmen einfallen.
- Das Ziel: Wie kann man die Annahme, dass es für ein (möglicherweise neues) Problem keinen effizienten Algorithmus gibt, möglichst gut untermauern?
- Auftritt NP-Vollständigkeit! Hiermit werden wir die 'Kompliziertheit' eines Problems formalisieren.

Motivation - Zusammenfassung

Zusammengefasst:

- Beobachtung 1: Unsere Fähigkeit untere Schranken für Probleme zu beweisen sind im Moment sehr beschränkt.
- Beobachtung 2: Wir kennen aber viele Probleme, für die uns keine effizienten Algorithmen einfallen.
- Das Ziel: Wie kann man die Annahme, dass es für ein (möglicherweise neues) Problem keinen effizienten Algorithmus gibt, möglichst gut untermauern?
- Auftritt NP-Vollständigkeit! Hiermit werden wir die 'Kompliziertheit' eines Problems formalisieren.

Motivation - Zusammenfassung

Zusammengefasst:

- Beobachtung 1: Unsere Fähigkeit untere Schranken für Probleme zu beweisen sind im Moment sehr beschränkt.
- Beobachtung 2: Wir kennen aber viele Probleme, für die uns keine effizienten Algorithmen einfallen.
- Das Ziel: Wie kann man die Annahme, dass es für ein (möglicherweise neues) Problem keinen effizienten Algorithmus gibt, möglichst gut untermauern?
- Auftritt *NP*-Vollständigkeit! Hiermit werden wir die 'Kompliziertheit' eines Problems formalisieren.

Motivation - Zusammenfassung

Zusammengefasst:

- Beobachtung 1: Unsere Fähigkeit untere Schranken für Probleme zu beweisen sind im Moment sehr beschränkt.
- Beobachtung 2: Wir kennen aber viele Probleme, für die uns keine effizienten Algorithmen einfallen.
- Das Ziel: Wie kann man die Annahme, dass es für ein (möglicherweise neues) Problem keinen effizienten Algorithmus gibt, möglichst gut untermauern?
- Auftritt *NP*-Vollständigkeit! Hiermit werden wir die 'Kompliziertheit' eines Problems formalisieren.

Zur Nachbereitung

Anmerkung (zur Nachbereitung)

Mit letzterem wird es dann möglich sein für ein neues Problem X zu zeigen, dass es NP-vollständig ist und dann Aussagen zu treffen wie: Wenn man X schnell lösen könnte, dann auch all diese anderen komplizierten Probleme, für die uns (und vielen, vielen anderen Menschen) schon seit Jahrzehnten (und länger) nichts gutes eingefallen ist! - Dann kann man seine Kraft ggf. lieber darauf verwenden für diese Problem z.B. Näherungslösungen zu finden oder 'nur' Spezialfälle zu lösen.

Ablauf

Der Ablauf für heute und nächstes Mal:

- Formal fassen was ein Problem und eine Kodierung ist.
- P und NP einführen
  - Sinnhaftigkeit der Definitionen erläutern.
- Reduktion einführen.
- NP-Vollständigkeit einführen.
- Probleme als NP-vollständig nachweisen und begründen, warum dies unser Ziel von oben erfüllt.

Ablauf

Der Ablauf für heute und nächstes Mal:

- Formal fassen was ein Problem und eine Kodierung ist.
- P und NP einführen
  - Sinnhaftigkeit der Definitionen erläutern.
- Reduktion einführen.
- NP-Vollständigkeit einführen.
- Probleme als NP-vollständig nachweisen und begründen, warum dies unser Ziel von oben erfüllt.

Ablauf

Der Ablauf für heute und nächstes Mal:

- Formal fassen was ein Problem und eine Kodierung ist.
- P und NP einführen
  - Sinnhaftigkeit der Definitionen erläutern.
- Reduktion einführen.
- NP-Vollständigkeit einführen.
- Probleme als NP-vollständig nachweisen und begründen, warum dies unser Ziel von oben erfüllt.

Ablauf

Der Ablauf für heute und nächstes Mal:

- Formal fassen was ein Problem und eine Kodierung ist.
- P und NP einführen
  - Sinnhaftigkeit der Definitionen erläutern.
- Reduktion einführen.
- NP-Vollständigkeit einführen.
- Probleme als NP-vollständig nachweisen und begründen, warum dies unser Ziel von oben erfüllt.

Ablauf

Der Ablauf für heute und nächstes Mal:

- Formal fassen was ein Problem und eine Kodierung ist.
- P und NP einführen
  - Sinnhaftigkeit der Definitionen erläutern.
- Reduktion einführen.
- NP-Vollständigkeit einführen.
- Probleme als NP-vollständig nachweisen und begründen, warum dies unser Ziel von oben erfüllt.

Probleminstanzen und Lösungen

Definition (Problem)

Sei I eine Menge von *Probleminstanzen*, S eine Menge von *Problemlösungen*. Ein *abstraktes Problem* Q definieren wir dann als binäre Relation $Q \subseteq I \times S$.

Beispiel

Um z.B. das `SHORTEST-PATH` Problem zu beschreiben, könnte die Menge I alle Tupel (G, s, t) bestehend aus einem gerichteten Graphen G und zwei Knoten s und t enthalten. Die Menge S dann Folgen von Knoten und Q enthält Tupel $((G, s, t), sv_0v_1 \dots t)$ derart, dass $sv_0v_1 \dots t$ ein kürzester Pfad in G von s nach t ist.

Probleminstanzen und Lösungen

Definition (Problem)

Sei I eine Menge von *Probleminstanzen*, S eine Menge von *Problemlösungen*. Ein *abstraktes Problem* Q definieren wir dann als binäre Relation $Q \subseteq I \times S$.

Beispiel

Um z.B. das `SHORTEST-PATH` Problem zu beschreiben, könnte die Menge I alle Tupel (G, s, t) bestehend aus einem gerichteten Graphen G und zwei Knoten s und t enthalten. Die Menge S dann Folgen von Knoten und Q enthält Tupel $((G, s, t), sv_0v_1 \ldots t)$ derart, dass $sv_0v_1 \ldots t$ ein kürzester Pfad in G von s nach t ist.

Codierungen

Da wir mit Computern arbeiten und insb. die Laufzeit von Algorithmen bewerten wollen und diese Bewertung abhängig ist von der Länge der Eingabe, müssen wir uns mit Codierungen beschäftigen.

- Zahlen $1, 2, 3, 4 \in \mathbb{N}$ werden binär codiert.
- Buchstaben etc. kann man z.B. im ASCII-Code binär codieren.

Wie die Codierung genau geschieht ist für uns nicht wichtig. Man sollte sich nur bewusst machen, dass obige Probleminstanzen und Lösungen irgendwie codiert werden müssen. Wir machen dies meistens mit einer sinnvollen Binärcodierung.

Nebenbemerkung

Für die Betrachtungen bzgl. P und NP macht die Codierung meist kaum einen Unterschied. Nur unär/monadisch sollte sie nicht sein!

Codierungen

Da wir mit Computern arbeiten und insb. die Laufzeit von Algorithmen bewerten wollen und diese Bewertung abhängig ist von der Länge der Eingabe, müssen wir uns mit Codierungen beschäftigen.

- Zahlen $1, 2, 3, 4 \in \mathbb{N}$ werden binär codiert.
- Buchstaben etc. kann man z.B. im ASCII-Code binär codieren.

Wie die Codierung genau geschieht ist für uns nicht wichtig. Man sollte sich nur bewusst machen, dass obige Probleminstanzen und Lösungen irgendwie codiert werden müssen. Wir machen dies meistens mit einer sinnvollen Binärcodierung.

Nebenbemerkung

Für die Betrachtungen bzgl. P und NP macht die Codierung meist kaum einen Unterschied. Nur unär/monadisch sollte sie nicht sein!

Codierungen

Da wir mit Computern arbeiten und insb. die Laufzeit von Algorithmen bewerten wollen und diese Bewertung abhängig ist von der Länge der Eingabe, müssen wir uns mit Codierungen beschäftigen.

- Zahlen $1, 2, 3, 4 \in \mathbb{N}$ werden binär codiert.
- Buchstaben etc. kann man z.B. im ASCII-Code binär codieren.

Wie die Codierung genau geschieht ist für uns nicht wichtig. Man sollte sich nur bewusst machen, dass obige Probleminstanzen und Lösungen irgendwie codiert werden müssen. Wir machen dies meistens mit einer sinnvollen Binärcodierung.

Nebenbemerkung

Für die Betrachtungen bzgl. P und NP macht die Codierung meist kaum einen Unterschied. Nur unär/monadisch sollte sie nicht sein!

Exkurs: Mehr zu Codierungen

Exkurs: Codierungen

Die meisten Codierungen sind *polynomial verwandt*, d.h. sie lassen sich mit polynomialen Aufwand ineinander überführen (z.B. Binärcodierung in eine Codierung zur Basis 3).
Eine Ausnahme bildet hier eine monadische (oder unäre) Codierung, in der eine Zahl k durch k Nullen (z.B.) codiert wird. Dies führt zu exponentiellem Mehraufwand im Vergleich zu einer Binärcodierung.

Konkrete Probleme

Definition (Konkretes Problem)

Ist die Instanzmenge die Menge der binären Strings (von denen einige dann syntaktischen Unsinn codieren), so sprechen wir von einem *konkreten Problem*

Anmerkung

Wenn man deutlich machen will, dass ein mathematisches Objekt codiert wird, nutzt man meist die Klammern $\langle, \rangle$. Für die Menge I aller Tupel (G, s, t) also z.B. $\{\langle (G, s, t) \rangle\}$.

Konkrete Probleme

Definition (Konkretes Problem)

Ist die Instanzmenge die Menge der binären Strings (von denen einige dann syntaktischen Unsinn codieren), so sprechen wir von einem *konkreten Problem*

Anmerkung

Wenn man deutlich machen will, dass ein mathematisches Objekt codiert wird, nutzt man meist die Klammern $\langle, \rangle$. Für die Menge I aller Tupel (G, s, t) also z.B. $\{\langle (G, s, t) \rangle\}$.

Entscheidungsprobleme und Optimierungsprobleme

- Viele Probleme in der Informatik sind *Optimierungsprobleme*, z.B.
  - Wert eines kürzesten Pfad.
  - Wert eines maximalen Flusses.
- Die Theorie wird aber einfacher, wenn man sich auf *Entscheidungsprobleme* beschränkt:
  - Eine Eingabe wird akzeptiert oder nicht, bzw.
  - Ausgabe ist 1 oder 0.
- I.A. ist es jedoch leicht ein Optimierungsproblem in ein Entscheidungsproblem umzuwandeln, das nicht schwieriger ist - und für die Ziele, die wir hier verfolgen reicht dies!

Entscheidungsprobleme und Optimierungsprobleme

- Viele Probleme in der Informatik sind *Optimierungsprobleme*, z.B.
  - Wert eines kürzesten Pfad.
  - Wert eines maximalen Flusses.
- Die Theorie wird aber einfacher, wenn man sich auf *Entscheidungsprobleme* beschränkt:
  - Eine Eingabe wird akzeptiert oder nicht, bzw.
  - Ausgabe ist 1 oder 0.
- I.A. ist es jedoch leicht ein Optimierungsproblem in ein Entscheidungsproblem umzuwandeln, das nicht schwieriger ist - und für die Ziele, die wir hier verfolgen reicht dies!

Entscheidungsprobleme und Optimierungsprobleme II

Statt

- Gibt es einen kürzesten Pfad von s nach t (in G) - und was ist sein Wert?

fragen wir jetzt

- Gibt es einen Pfad von s nach t (in G) mit Kosten $\leq k$?

Pause to Ponder

Kann man das Entscheidungsproblem lösen, kann man (i.A.) auch das Optimierungsproblem lösen, wobei sich der zusätzliche Aufwand in Grenzen hält. Wie geht dies hier? (Und die Umkehrung?)

Entscheidungsprobleme und Optimierungsprobleme II

Statt

- Gibt es einen kürzesten Pfad von s nach t (in G) - und was ist sein Wert?

fragen wir jetzt

- Gibt es einen Pfad von s nach t (in G) mit Kosten $\leq k$?

Pause to Ponder

Kann man das Entscheidungsproblem lösen, kann man (i.A.) auch das Optimierungsproblem lösen, wobei sich der zusätzliche Aufwand in Grenzen hält. Wie geht dies hier? (Und die Umkehrung?)

Entscheidungsprobleme und Optimierungsprobleme II

Statt

- Gibt es einen kürzesten Pfad von s nach t (in G) - und was ist sein Wert?

fragen wir jetzt

- Gibt es einen Pfad von s nach t (in G) mit Kosten $\leq k$?

Pause to Ponder

Kann man das Entscheidungsproblem lösen, kann man (i.A.) auch das Optimierungsproblem lösen, wobei sich der zusätzliche Aufwand in Grenzen hält. Wie geht dies hier? (Und die Umkehrung?)

Die Komplexitätsklasse P

Definition (Polynomialzeit)

- Sei Q ein konkretes Problem und i eine (binär codierte) Probleminstanz der Länge $n = |i|$. Erzeugt ein Algorithmus in Zeit $O(T(n))$ eine Lösung (d.h. formal ein $s \in S$, so dass $(i, s) \in Q$ gilt), so sagen wir, dass er das Problem *löst*.
- Existiert zu einem Problem ein Algorithmus, der die Laufzeit $O(n^k)$ für eine Konstante k hat, so sagen wir dass das Problem in polynomialer Zeit (auch: Polynomialzeit) lösbar ist.
- Die Menge aller konkreten Entscheidungsprobleme, die in polymomialer Zeit lösbar sind, bilden die *Komplexitätsklasse* P.

Die Komplexitätsklasse P

Definition (Polynomialzeit)

- Sei Q ein konkretes Problem und i eine (binär codierte) Probleminstanz der Länge $n = |i|$. Erzeugt ein Algorithmus in Zeit $O(T(n))$ eine Lösung (d.h. formal ein $s \in S$, so dass $(i, s) \in Q$ gilt), so sagen wir, dass er das Problem *löst*.
- Existiert zu einem Problem ein Algorithmus, der die Laufzeit $O(n^k)$ für eine Konstante k hat, so sagen wir dass das Problem in polynomialer Zeit (auch: Polynomialzeit) lösbar ist.
- Die Menge aller konkreten Entscheidungsprobleme, die in polymomialer Zeit lösbar sind, bilden die *Komplexitätsklasse* P.

Die Komplexitätsklasse P

Definition (Polynomialzeit)

- Sei Q ein konkretes Problem und i eine (binär codierte) Probleminstanz der Länge $n = |i|$. Erzeugt ein Algorithmus in Zeit $O(T(n))$ eine Lösung (d.h. formal ein $s \in S$, so dass $(i, s) \in Q$ gilt), so sagen wir, dass er das Problem *löst*.
- Existiert zu einem Problem ein Algorithmus, der die Laufzeit $O(n^k)$ für eine Konstante k hat, so sagen wir dass das Problem in polynomialer Zeit (auch: Polynomialzeit) lösbar ist.
- Die Menge aller konkreten Entscheidungsprobleme, die in polymomialer Zeit lösbar sind, bilden die *Komplexitätsklasse* P.

Zur Bedeutung von P

Die Probleme in P werden als *'effizient lösbar'* betrachtet. Warum?

- Ein Problem in $\Theta(n^{50})$ ist zwar eher unhandlich (als effizient lösbar), aber solch große Exponenten treten kaum auf! Hat man erstmal einen Algorithmus in P gefunden, so ist es meist möglich diesen zu verbessern, so dass man letztendlich einen Algorithmus in $O(n^k)$ mit kleinem k hat.
- Gängige Rechnermodelle simulieren sich gegenseitig mit polynomiellen Mehraufwand.
- Mit Polynomen lässt sich gut rechnen.
- Insgesamt hat diese Herangehensweise zu einer handhabbaren und eleganten Theorie geführt, mit der für die Praxis relevante Aussagen möglich sind.

Zur Bedeutung von P

Die Probleme in P werden als *'effizient lösbar'* betrachtet. Warum?

- Ein Problem in $\Theta(n^{50})$ ist zwar eher unhandlich (als effizient lösbar), aber solch große Exponenten treten kaum auf! Hat man erstmal einen Algorithmus in P gefunden, so ist es meist möglich diesen zu verbessern, so dass man letztendlich einen Algorithmus in $O(n^k)$ mit kleinem k hat.
- Gängige Rechnermodelle simulieren sich gegenseitig mit polynomiellen Mehraufwand.
- Mit Polynomen lässt sich gut rechnen.
- Insgesamt hat diese Herangehensweise zu einer handhabbaren und eleganten Theorie geführt, mit der für die Praxis relevante Aussagen möglich sind.

Zur Bedeutung von P

Die Probleme in P werden als *'effizient lösbar'* betrachtet. Warum?

- Ein Problem in $\Theta(n^{50})$ ist zwar eher unhandlich (als effizient lösbar), aber solch große Exponenten treten kaum auf! Hat man erstmal einen Algorithmus in P gefunden, so ist es meist möglich diesen zu verbessern, so dass man letztendlich einen Algorithmus in $O(n^k)$ mit kleinem k hat.
- Gängige Rechnermodelle simulieren sich gegenseitig mit polynomiellen Mehraufwand.
- Mit Polynomen lässt sich gut rechnen.
- Insgesamt hat diese Herangehensweise zu einer handhabbaren und eleganten Theorie geführt, mit der für die Praxis relevante Aussagen möglich sind.

Zur Bedeutung von P

Die Probleme in P werden als *'effizient lösbar'* betrachtet. Warum?

- Ein Problem in $\Theta(n^{50})$ ist zwar eher unhandlich (als effizient lösbar), aber solch große Exponenten treten kaum auf! Hat man erstmal einen Algorithmus in P gefunden, so ist es meist möglich diesen zu verbessern, so dass man letztendlich einen Algorithmus in $O(n^k)$ mit kleinem k hat.
- Gängige Rechnermodelle simulieren sich gegenseitig mit polynomiellen Mehraufwand.
- Mit Polynomen lässt sich gut rechnen.
- Insgesamt hat diese Herangehensweise zu einer handhabbaren und eleganten Theorie geführt, mit der für die Praxis relevante Aussagen möglich sind.

Zur Nachbereitung

Anmerkung (zur Nachbereitung)

Man kann Beispiele konstruieren, mit denen man die Sinnhaftigkeit der Definition von P hinterfragen kann, aber meist sind diese tatsächlich arg konstruiert. Aus praktischer und theoretischer Sicht ist die aktuelle Definition von P sinnvoll. (In der Zukunft ändert sich das vielleicht wieder, aber das führt hier zu weit... - ist aber spannend ;-))

Refresher: Formale Sprachen

Vorteil bei Entscheidungsproblemen: Wir können die Theorie der formalen Sprachen nutzen!

Definition (Grundlagen formaler Sprachen)

- Ein Alphabet Σ ist eine endliche Menge von Symbolen. Hier meist $\Sigma = \{0, 1\}$.
- $\Sigma^* = \{\epsilon, 0, 1, 00, 01, 10, 11, 000, \ldots\}$ ist die Menge aller Worte, die aus Elementen aus Σ gebildet werden können.
- Jede Teilmenge $L \subseteq \Sigma^*$ ist eine Sprache (über Σ).
- ϵ ist das leere Wort, $\emptyset$ die leere Sprache.

Refresher: Formale Sprachen

Vorteil bei Entscheidungsproblemen: Wir können die Theorie der formalen Sprachen nutzen!

Definition (Grundlagen formaler Sprachen)

- Ein Alphabet Σ ist eine endliche Menge von Symbolen. Hier meist $\Sigma = \{0, 1\}$.
- $\Sigma^* = \{\epsilon, 0, 1, 00, 01, 10, 11, 000, \ldots\}$ ist die Menge aller Worte, die aus Elementen aus Σ gebildet werden können.
- Jede Teilmenge $L \subseteq \Sigma^*$ ist eine Sprache (über Σ).
- ϵ ist das leere Wort, $\emptyset$ die leere Sprache.

Refresher: Formale Sprachen

Vorteil bei Entscheidungsproblemen: Wir können die Theorie der formalen Sprachen nutzen!

Definition (Grundlagen formaler Sprachen)

- Ein Alphabet Σ ist eine endliche Menge von Symbolen. Hier meist $\Sigma = \{0, 1\}$.
- $\Sigma^* = \{\epsilon, 0, 1, 00, 01, 10, 11, 000, \ldots\}$ ist die Menge aller Worte, die aus Elementen aus Σ gebildet werden können.
- Jede Teilmenge $L \subseteq \Sigma^*$ ist eine Sprache (über Σ).
- ϵ ist das leere Wort, $\emptyset$ die leere Sprache.

Refresher: Formale Sprachen

Vorteil bei Entscheidungsproblemen: Wir können die Theorie der formalen Sprachen nutzen!

Definition (Grundlagen formaler Sprachen)

- Ein Alphabet Σ ist eine endliche Menge von Symbolen. Hier meist $\Sigma = \{0, 1\}$.
- $\Sigma^* = \{\epsilon, 0, 1, 00, 01, 10, 11, 000, \ldots\}$ ist die Menge aller Worte, die aus Elementen aus Σ gebildet werden können.
- Jede Teilmenge $L \subseteq \Sigma^*$ ist eine Sprache (über Σ).
- ϵ ist das leere Wort, $\emptyset$ die leere Sprache.

Refresher: Formale Sprachen

Vorteil bei Entscheidungsproblemen: Wir können die Theorie der formalen Sprachen nutzen!

Definition (Grundlagen formaler Sprachen)

- Ein Alphabet Σ ist eine endliche Menge von Symbolen. Hier meist $\Sigma = \{0, 1\}$.
- $\Sigma^* = \{\epsilon, 0, 1, 00, 01, 10, 11, 000, \ldots\}$ ist die Menge aller Worte, die aus Elementen aus Σ gebildet werden können.
- Jede Teilmenge $L \subseteq \Sigma^*$ ist eine Sprache (über Σ).
- ϵ ist das leere Wort, $\emptyset$ die leere Sprache.

Refresher: Formale Sprachen II

Definition (Formale Sprachen - Operationen)

- Vereinigung und Schnitt zweier Sprachen ist wie bei Mengen definiert.
- Das Komplement von L ist $\overline{L} = \Sigma^* \setminus L$.
- Die Konkatenation zweier Sprachen L_1, L_2 ist

$$L = L_1 \cdot L_2 = \{w_1 w_2 \mid w_1 \in L_1, w_2 \in L_2\}.$$

- Der Kleene-Stern (oder Abschluss) von L ist

$$L^* = \{\epsilon\} \cup L \cup L^2 \cup L^3 \cup \ldots$$

wobei L^k die Sprache L k-mal mit sich selbst konkateniert ist.

Refresher: Formale Sprachen II

Definition (Formale Sprachen - Operationen)

- Vereinigung und Schnitt zweier Sprachen ist wie bei Mengen definiert.
- Das Komplement von L ist $\overline{L} = \Sigma^* \setminus L$.
- Die Konkatenation zweier Sprachen L_1, L_2 ist

$$L = L_1 \cdot L_2 = \{w_1 w_2 \mid w_1 \in L_1, w_2 \in L_2\}.$$

- Der Kleene-Stern (oder Abschluss) von L ist

$$L^* = \{\epsilon\} \cup L \cup L^2 \cup L^3 \cup \ldots$$

wobei L^k die Sprache L k-mal mit sich selbst konkateniert ist.

Refresher: Formale Sprachen II

Definition (Formale Sprachen - Operationen)

- Vereinigung und Schnitt zweier Sprachen ist wie bei Mengen definiert.
- Das Komplement von L ist $\overline{L} = \Sigma^* \setminus L$.
- Die Konkatenation zweier Sprachen L_1, L_2 ist

$$L = L_1 \cdot L_2 = \{w_1 w_2 \mid w_1 \in L_1, w_2 \in L_2\}.$$

- Der Kleene-Stern (oder Abschluss) von L ist

$$L^* = \{\epsilon\} \cup L \cup L^2 \cup L^3 \cup \ldots$$

wobei L^k die Sprache L k-mal mit sich selbst konkateniert ist.

Refresher: Formale Sprachen II

Definition (Formale Sprachen - Operationen)

- Vereinigung und Schnitt zweier Sprachen ist wie bei Mengen definiert.
- Das Komplement von L ist $\overline{L} = \Sigma^* \setminus L$.
- Die Konkatenation zweier Sprachen L_1, L_2 ist

$$L = L_1 \cdot L_2 = \{w_1 w_2 \mid w_1 \in L_1, w_2 \in L_2\}.$$

- Der Kleene-Stern (oder Abschluss) von L ist

$$L^* = \{\epsilon\} \cup L \cup L^2 \cup L^3 \cup \ldots$$

wobei L^k die Sprache L k-mal mit sich selbst konkateniert ist.

Formale Sprachen - Beispiel

Beispiel

Sei $\Sigma = \{a, b\}$ und $A = \{a, aa, ab\}, B = \{a, ba\}, C = \{a\}^*$ dann ist

- $A \cup B =$
- $A \cap B =$
- $\overline{C} =$
- $A \cdot B =$
- $B^* =$

Formale Sprachen - Beispiel

Beispiel

Sei $\Sigma = \{a, b\}$ und $A = \{a, aa, ab\}, B = \{a, ba\}, C = \{a\}^*$ dann ist

- $A \cup B = \{a, aa, ab, ba\}$
- $A \cap B =$
- $\overline{C} =$
- $A \cdot B =$
- $B^* =$

Formale Sprachen - Beispiel

Beispiel

Sei $\Sigma = \{a, b\}$ und $A = \{a, aa, ab\}, B = \{a, ba\}, C = \{a\}^*$ dann ist

- $A \cup B = \{a, aa, ab, ba\}$
- $A \cap B =$
- $\overline{C} =$
- $A \cdot B =$
- $B^* =$

Formale Sprachen - Beispiel

Beispiel

Sei $\Sigma = \{a, b\}$ und $A = \{a, aa, ab\}, B = \{a, ba\}, C = \{a\}^*$ dann ist

- $A \cup B = \{a, aa, ab, ba\}$
- $A \cap B = \{a\}$
- $\overline{C} =$
- $A \cdot B =$
- $B^* =$

Formale Sprachen - Beispiel

Beispiel

Sei $\Sigma = \{a, b\}$ und $A = \{a, aa, ab\}, B = \{a, ba\}, C = \{a\}^*$ dann ist

- $A \cup B = \{a, aa, ab, ba\}$
- $A \cap B = \{a\}$
- $\overline{C} =$
- $A \cdot B =$
- $B^* =$

Formale Sprachen - Beispiel

Beispiel

Sei $\Sigma = \{a, b\}$ und $A = \{a, aa, ab\}, B = \{a, ba\}, C = \{a\}^*$ dann ist

- $A \cup B = \{a, aa, ab, ba\}$
- $A \cap B = \{a\}$
- $\overline{C} = \{b, ba, ab, bb, aab, aba, baa, abb, bab, bba, bbb, \ldots\}$
- $A \cdot B =$
- $B^* =$

Formale Sprachen - Beispiel

Beispiel

Sei $\Sigma = \{a, b\}$ und $A = \{a, aa, ab\}, B = \{a, ba\}, C = \{a\}^*$ dann ist

- $A \cup B = \{a, aa, ab, ba\}$
- $A \cap B = \{a\}$
- $\overline{C} = \{b, ba, ab, bb, aab, aba, baa, abb, bab, bba, bbb, \ldots\}$
- $A \cdot B =$
- $B^* =$

Formale Sprachen - Beispiel

Beispiel

Sei $\Sigma = \{a, b\}$ und $A = \{a, aa, ab\}, B = \{a, ba\}, C = \{a\}^*$ dann ist

- $A \cup B = \{a, aa, ab, ba\}$
- $A \cap B = \{a\}$
- $\overline{C} = \{b, ba, ab, bb, aab, aba, baa, abb, bab, bba, bbb, \ldots\}$
- $A \cdot B = \{aa, aba, aaa, aaba, abba\}$
- $B^* =$

Formale Sprachen - Beispiel

Beispiel

Sei $\Sigma = \{a, b\}$ und $A = \{a, aa, ab\}, B = \{a, ba\}, C = \{a\}^*$ dann ist

- $A \cup B = \{a, aa, ab, ba\}$
- $A \cap B = \{a\}$
- $\overline{C} = \{b, ba, ab, bb, aab, aba, baa, abb, bab, bba, bbb, \ldots\}$
- $A \cdot B = \{aa, aba, aaa, aaba, abba\}$
- $B^* =$

Formale Sprachen - Beispiel

Beispiel

Sei $\Sigma = \{a, b\}$ und $A = \{a, aa, ab\}, B = \{a, ba\}, C = \{a\}^*$ dann ist

- $A \cup B = \{a, aa, ab, ba\}$
- $A \cap B = \{a\}$
- $\overline{C} = \{b, ba, ab, bb, aab, aba, baa, abb, bab, bba, bbb, \ldots\}$
- $A \cdot B = \{aa, aba, aaa, aaba, abba\}$
- $B^* = \{\epsilon, a, ba, aa, aba, baa, baba, aaa, aaba, \ldots\}$

Entscheidungsprobleme und Formale Sprachen

Sei Q ein Entscheidungsproblem, dann ist die Instanzmenge durch Σ^* gegeben ($\Sigma = \{0, 1\}$). Q ist nun durch jene Instanzen, die 'ja' als Antwort haben vollständig charakterisiert, d.h. Q kann mit folgender Sprache L über Σ identifiziert werden:

$$L = \{x \in \Sigma^* \mid Q(x) = 1\}.$$

Entscheidungsprobleme und Formale Sprachen

Sei Q ein Entscheidungsproblem, dann ist die Instanzmenge durch Σ^* gegeben ($\Sigma = \{0, 1\}$). Q ist nun durch jene Instanzen, die 'ja' als Antwort haben vollständig charakterisiert, d.h. Q kann mit folgender Sprache L über Σ identifiziert werden:

$$L = \{x \in \Sigma^* \mid Q(x) = 1\}.$$

Entscheidungsprobleme und Formale Sprachen II

Praktischerweise schreibt man das dann so:

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

Anmerkung

Die $\langle, \rangle$ Klammern sollen die Codierung andeuten.

Komplexitätsklassen und formale Sprachen

Wir drücken nun die Beziehung zwischen Entscheidungsproblemen und Algorithmen formaler aus als bisher. Sei A dazu ein Algorithmus.

- Eine Eingabe $x \in \{0,1\}^*$ wird von A *akzeptiert*, wenn $A(x) = 1$ gilt. Wenn $A(x) = 0$ gilt, wird x von A *zurückgewiesen*.
- Die von A akzeptierte Sprache ist

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

Anmerkung

Wird eine Sprache L akzeptiert, muss nicht jedes $x \notin L$ zurückgewiesen werden. Dies würde bedeuten, dass A stets endet (oder: anhält). Dies ist aber gar nicht gefordert! Eine Endlosschleife ist bspw. auch möglich.

Komplexitätsklassen und formale Sprachen

Wir drücken nun die Beziehung zwischen Entscheidungsproblemen und Algorithmen formaler aus als bisher. Sei A dazu ein Algorithmus.

- Eine Eingabe $x \in \{0,1\}^*$ wird von A *akzeptiert*, wenn $A(x) = 1$ gilt. Wenn $A(x) = 0$ gilt, wird x von A *zurückgewiesen*.
- Die von A akzeptierte Sprache ist

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

Anmerkung

Wird eine Sprache L akzeptiert, muss nicht jedes $x \notin L$ zurückgewiesen werden. Dies würde bedeuten, dass A stets endet (oder: anhält). Dies ist aber gar nicht gefordert! Eine Endlosschleife ist bspw. auch möglich.

Komplexitätsklassen und formale Sprachen

Wir drücken nun die Beziehung zwischen Entscheidungsproblemen und Algorithmen formaler aus als bisher. Sei A dazu ein Algorithmus.

- Eine Eingabe $x \in \{0,1\}^*$ wird von A *akzeptiert*, wenn $A(x) = 1$ gilt. Wenn $A(x) = 0$ gilt, wird x von A *zurückgewiesen*.
- Die von A akzeptierte Sprache ist

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

Anmerkung

Wird eine Sprache L akzeptiert, muss nicht jedes $x \notin L$ zurückgewiesen werden. Dies würde bedeuten, dass A stets endet (oder: anhält). Dies ist aber gar nicht gefordert! Eine Endlosschleife ist bspw. auch möglich.

Komplexitätsklassen und formale Sprachen

Wir drücken nun die Beziehung zwischen Entscheidungsproblemen und Algorithmen formaler aus als bisher. Sei A dazu ein Algorithmus.

- Eine Eingabe $x \in \{0,1\}^*$ wird von A *akzeptiert*, wenn $A(x) = 1$ gilt. Wenn $A(x) = 0$ gilt, wird x von A *zurückgewiesen*.
- Die von A akzeptierte Sprache ist

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

Anmerkung

Wird eine Sprache L akzeptiert, muss nicht jedes $x \notin L$ zurückgewiesen werden. Dies würde bedeuten, dass A stets endet (oder: anhält). Dies ist aber gar nicht gefordert! Eine Endlosschleife ist bspw. auch möglich.

Komplexitätsklassen und formale Sprachen II

Definition (Entscheiden von Sprachen)

Eine Sprache L wird von einem Algorithmus A *entschieden*, wenn

- jedes $x \in L$ von A akzeptiert wird ($A(x) = 1$) und
- jedes $x \notin L$ von A zurückgewiesen wird ($A(x) = 0$).

A hält also insb. auf jeder Eingabe an und liefert stets das richtige Ergebnis.

Komplexitätsklassen und formale Sprachen II

Definition (Entscheiden von Sprachen)

Eine Sprache L wird von einem Algorithmus A *entschieden*, wenn

- jedes $x \in L$ von A akzeptiert wird ($A(x) = 1$) und
- jedes $x \notin L$ von A zurückgewiesen wird ($A(x) = 0$).

A hält also insb. auf jeder Eingabe an und liefert stets das richtige Ergebnis.

Komplexitätsklassen und formale Sprachen II

Definition (Entscheiden von Sprachen)

Eine Sprache L wird von einem Algorithmus A *entschieden*, wenn

- jedes $x \in L$ von A akzeptiert wird ($A(x) = 1$) und
- jedes $x \notin L$ von A zurückgewiesen wird ($A(x) = 0$).

A hält also insb. auf jeder Eingabe an und liefert stets das richtige Ergebnis.

Komplexitätsklassen und formale Sprachen II

Definition (Entscheiden von Sprachen)

Eine Sprache L wird von einem Algorithmus A *entschieden*, wenn

- jedes $x \in L$ von A akzeptiert wird ($A(x) = 1$) und
- jedes $x \notin L$ von A zurückgewiesen wird ($A(x) = 0$).

A hält also insb. auf jeder Eingabe an und liefert stets das richtige Ergebnis.

Komplexitätsklassen und formale Sprachen III

Definition (Polynomielle Zeitschranken)

- Ein Algorithmus A akzeptiert eine Sprache L in *polynomialer Zeit*, wenn sie von A akzeptiert wird und zusätzlich ein $k \in \mathbb{N}$ existiert, so dass jedes $x \in L$ mit $|x| = n$ in Zeit $O(n^k)$ akzeptiert wird.
- Soll A die Sprache L entscheiden, wird für jeden String x der Länge n in Zeit $O(n^k)$ entschieden, ob $x \in L$ gilt (oder nicht).

Komplexitätsklassen und formale Sprachen IV

Definition (Neue Defintion von P)

$$P = \{L \subseteq \{0,1\}^* \mid \text{es existiert ein Algorithmus } A, \text{ der } L \text{ in Polynomialzeit entscheidet.}\}$$

Akzeptieren und Entscheiden

Satz (Theorem 34.2 im Cormen)

$P = \{L \mid L \textit{ wird von einem Algo. in Polynomialzeit akzeptiert}\}$

Akzeptieren und Entscheiden

Satz (Theorem 34.2 im Cormen)

$P = \{L \mid L \textit{ wird von einem Algo. in Polynomialzeit akzeptiert}\}$

Beweis.

Die Richtung von links nach rechts ist klar (Warum?), es ist also zu zeigen,

□

Akzeptieren und Entscheiden

Satz (Theorem 34.2 im Cormen)

$P = \{L \mid L \textit{ wird von einem Algo. in Polynomialzeit akzeptiert}\}$

Beweis.

Die Richtung von links nach rechts ist klar (Warum?), es ist also zu zeigen, dass jede in Polynomialzeit akzeptierbare Sprache auch in Polynomialzeit entscheidbar ist.

□

Akzeptieren und Entscheiden

Satz (Theorem 34.2 im Cormen)

$P = \{L \mid L \textit{ wird von einem Algo. in Polynomialzeit akzeptiert}\}$

Beweis.

Die Richtung von links nach rechts ist klar (Warum?), es ist 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.

□

Akzeptieren und Entscheiden

Satz (Theorem 34.2 im Cormen)

$P = \{L \mid L \textit{ wird von einem Algo. in Polynomialzeit akzeptiert}\}$

Beweis.

Die Richtung von links nach rechts ist klar (Warum?), es ist 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 $s = c \cdot |x|^k$

□

Akzeptieren und Entscheiden

Satz (Theorem 34.2 im Cormen)

$P = \{L \mid L \textit{ wird von einem Algo. in Polynomialzeit akzeptiert}\}$

Beweis.

Die Richtung von links nach rechts ist klar (Warum?), es ist 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.

□

Akzeptieren und Entscheiden

Satz (Theorem 34.2 im Cormen)

$P = \{L \mid L \textit{ wird von einem Algo. in Polynomialzeit akzeptiert}\}$

Beweis.

Die Richtung von links nach rechts ist klar (Warum?), es ist 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 $O(n^k)$. □

Akzeptieren und Entscheiden

Satz (Theorem 34.2 im Cormen)

$P = \{L \mid L \textit{ wird von einem Algo. in Polynomialzeit akzeptiert}\}$

Anmerkung

Man beachte, dass der obige Beweis nicht konstruktiv war! Wenn man weiss, dass ein Algorithmus mit polynomialer Laufzeit existiert, weiss man nicht zwingend wie dieser aussieht geschweige denn, wie die Schranke aussieht.

Abschlusseigenschaften

Satz

Sind $L_1, L_2 \in P$*, so gilt auch* $L_1 \cup L_2, L_1 \cap L_2, L_1 \cdot L_2, \overline{L_1}, L_1^* \in P$*.*

Beweis.

Wir zeigen hier nur $L_1 \cup L_2 \in P$. Seien $L_1, L_2 \in P$, dann

□

Abschlusseigenschaften

Satz

Sind $L_1, L_2 \in P$*, so gilt auch* $L_1 \cup L_2, L_1 \cap L_2, L_1 \cdot L_2, \overline{L_1}, L_1^* \in P$*.*

Beweis.

Wir zeigen hier nur $L_1 \cup L_2 \in P$. Seien $L_1, L_2 \in P$, dann gibt es Algorithmen A_1, A_2, die L_1 und L_2 entscheiden. Wir konstruieren einen Polynomialzeitalgorithmus A für $L_1 \cup L_2$ wie folgt:

□

Abschlusseigenschaften

Satz

Sind $L_1, L_2 \in P$, so gilt auch $L_1 \cup L_2, L_1 \cap L_2, L_1 \cdot L_2, \overline{L_1}, L_1^* \in P$.

Beweis.

Wir zeigen hier nur $L_1 \cup L_2 \in P$. Seien $L_1, L_2 \in P$, dann gibt es Algorithmen A_1, A_2, die L_1 und L_2 entscheiden. Wir konstruieren einen Polynomialzeitalgorithmus A für $L_1 \cup L_2$ wie folgt: Bei Eingabe x wird zunächst $A_1(x)$ berechnet (man beachte, dass A_1 die Sprache L_1 entscheidet, also auf jeden Fall anhält). Ist $A_1(x) = 1$, so akzeptiert A sofort, ansonsten wird $A_2(x)$ berechnet. Ist $A_2(x) = 1$ akzeptiert A wieder. Ist hingegen auch $A_2(x) = 0$, so liefert A den Wert 0.

□

Abschlusseigenschaften

Satz

Sind $L_1, L_2 \in P$, so gilt auch $L_1 \cup L_2, L_1 \cap L_2, L_1 \cdot L_2, \overline{L_1}, L_1^* \in P$.

Beweis.

Wir zeigen hier nur $L_1 \cup L_2 \in P$. Seien $L_1, L_2 \in P$, dann gibt es Algorithmen A_1, A_2, die L_1 und L_2 entscheiden. Wir konstruieren einen Polynomialzeitalgorithmus A für $L_1 \cup L_2$ wie folgt: Bei Eingabe x wird zunächst $A_1(x)$ berechnet (man beachte, dass A_1 die Sprache L_1 entscheidet, also auf jeden Fall anhält). Ist $A_1(x) = 1$, so akzeptiert A sofort, ansonsten wird $A_2(x)$ berechnet. Ist $A_2(x) = 1$ akzeptiert A wieder. Ist hingegen auch $A_2(x) = 0$, so liefert A den Wert 0. Da die Laufzeit von A_1 und A_2 durch ein Polynom beschränkt ist, gilt dies auch für die Laufzeit von A. (Warum genau?) □

Zwischenzusammenfassung

Zusammenfassung

- Die Eingabe für einen Algorithmus muss codiert sein. Wir gehen hier von einer Codierung im Binärcode aus. Alle Eingaben sind dann Zeichenketten $x \in \{0,1\}^*$. Die Menge der Instanzen ist dann $\{0,1\}^*$.
- Ein Entscheidungs*problem* kann dann als Sprache $L \subseteq \{0,1\}^*$ verstanden werden. Es gibt dann Eingaben x,
  - die der Algorithmus akzeptieren soll ($x \in L$) und solche
  - die der Algorithmus ablehnen soll ($x \notin L$).
- Sprachen können akzeptiert oder entschieden werden. (Beim Entscheiden, hält der Algorithmus auch stets auf Eingaben, die er nicht akzeptieren soll.)
- Gibt es zu einem Problem $L \subseteq \{0,1\}^*$ einen Algorithmus, der das Problem in Polynomialzeit entscheidet, so gilt $L \in P$.

Zwischenzusammenfassung

Zusammenfassung

- Die Eingabe für einen Algorithmus muss codiert sein. Wir gehen hier von einer Codierung im Binärcode aus. Alle Eingaben sind dann Zeichenketten $x \in \{0,1\}^*$. Die Menge der Instanzen ist dann $\{0,1\}^*$.
- Ein Entscheidungs*problem* kann dann als Sprache $L \subseteq \{0,1\}^*$ verstanden werden. Es gibt dann Eingaben x,
  - die der Algorithmus akzeptieren soll ($x \in L$) und solche
  - die der Algorithmus ablehnen soll ($x \notin L$).
- Sprachen können akzeptiert oder entschieden werden. (Beim Entscheiden, hält der Algorithmus auch stets auf Eingaben, die er nicht akzeptieren soll.)
- Gibt es zu einem Problem $L \subseteq \{0,1\}^*$ einen Algorithmus, der das Problem in Polynomialzeit entscheidet, so gilt $L \in P$.

Zwischenzusammenfassung

Zusammenfassung

- Die Eingabe für einen Algorithmus muss codiert sein. Wir gehen hier von einer Codierung im Binärcode aus. Alle Eingaben sind dann Zeichenketten $x \in \{0,1\}^*$. Die Menge der Instanzen ist dann $\{0,1\}^*$.
- Ein Entscheidungs*problem* kann dann als Sprache $L \subseteq \{0,1\}^*$ verstanden werden. Es gibt dann Eingaben x,
  - die der Algorithmus akzeptieren soll ($x \in L$) und solche
  - die der Algorithmus ablehnen soll ($x \notin L$).
- Sprachen können akzeptiert oder entschieden werden. (Beim Entscheiden, hält der Algorithmus auch stets auf Eingaben, die er nicht akzeptieren soll.)
- Gibt es zu einem Problem $L \subseteq \{0,1\}^*$ einen Algorithmus, der das Problem in Polynomialzeit entscheidet, so gilt $L \in P$.

Zwischenzusammenfassung

Zusammenfassung

- Die Eingabe für einen Algorithmus muss codiert sein. Wir gehen hier von einer Codierung im Binärcode aus. Alle Eingaben sind dann Zeichenketten $x \in \{0,1\}^*$. Die Menge der Instanzen ist dann $\{0,1\}^*$.
- Ein Entscheidungs*problem* kann dann als Sprache $L \subseteq \{0,1\}^*$ verstanden werden. Es gibt dann Eingaben x,
  - die der Algorithmus akzeptieren soll ($x \in L$) und solche
  - die der Algorithmus ablehnen soll ($x \notin L$).
- Sprachen können akzeptiert oder entschieden werden. (Beim Entscheiden, hält der Algorithmus auch stets auf Eingaben, die er nicht akzeptieren soll.)
- Gibt es zu einem Problem $L \subseteq \{0,1\}^*$ einen Algorithmus, der das Problem in Polynomialzeit entscheidet, so gilt $L \in P$.

Verifikation in polynomialer Zeit

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

Anmerkung

Schwierig zu lösen (zumindest effizient), aber gegeben ein Pfad, kann schnell *überprüft* werden, ob er die Kriterien erfüllt.

Verifikation in polynomialer Zeit

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

Anmerkung

Schwierig zu lösen (zumindest effizient), aber gegeben ein Pfad, kann schnell *überprüft* werden, ob er die Kriterien erfüllt.

Verifikation in polynomialer Zeit

Definition (Verifikationsalgorithmus)

Ein *Verifikationsalgorithmus* A ist ein 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

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

Anmerkung

Es geht also insb. um die Eingabe x. Diese bilden die Sprache. Das Zertifikat y kann vom Algorithmus genutzt werden, um zu entscheiden, ob $x \in L$ gilt, oder nicht.

Verifikation in polynomialer Zeit

Definition (Verifikationsalgorithmus)

Ein *Verifikationsalgorithmus* A ist ein 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

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

Anmerkung

Es geht also insb. um die Eingabe x. Diese bilden die Sprache. Das Zertifikat y kann vom Algorithmus genutzt werden, um zu entscheiden, ob $x \in L$ gilt, oder nicht.

Die Klasse NP

In NP sind nun jene Sprachen, die durch einen Algoritmus in polynomialer Zeit verifiziert werden können. Für das Zertifikat y verlangen wir zusätzlich, dass $|y| \in O(|x|^c)$ (für eine Konstante c) gilt. (Ist ein Algorithmus dann polynomiell in x (genauer: in $|x|$), so auch in x und y.)

Definition (NP)

$L \in NP$ gdw. ein Algorithmus 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.

Die Klasse NP

In NP sind nun jene Sprachen, die durch einen Algoritmus in polynomialer Zeit verifiziert werden können. Für das Zertifikat y verlangen wir zusätzlich, dass $|y| \in O(|x|^c)$ (für eine Konstante c) gilt. (Ist ein Algorithmus dann polynomiell in x (genauer: in $|x|$), so auch in x und y.)

Definition (NP)

$L \in NP$ gdw. ein Algorithmus 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.

Die Klasse NP

In NP sind nun jene Sprachen, die durch einen Algoritmus in polynomialer Zeit verifiziert werden können. Für das Zertifikat y verlangen wir zusätzlich, dass $|y| \in O(|x|^c)$ (für eine Konstante c) gilt. (Ist ein Algorithmus dann polynomiell in x (genauer: in $|x|$), so auch in x und y.)

Definition (NP)

$L \in NP$ gdw. ein Algorithmus 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.

Nichtdeterminismus

Nichtdeterminismus

Ein nichtdeterministischer Algorithmus A kann 'raten'. In einem Zustand z kann A z.B. eine Variable auf 0 und auf 1 setzen und damit in zwei Nachfolgezuständen z_0 und z_1 sein. Man interpretiert dies so, dass A in beiden Zuständen zugleich ist. Eine Eingabe x wird dann akzeptiert, wenn es *eine* Rechnung gibt, in der A x akzeptiert.

Das 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.

Satz (Nichtdeterminismus = Verifikation)

Die Definitionen von NP mittels Nichtdeterminismus und Verifikationsalgorithmen sind äquivalent.

Nichtdeterminismus

Nichtdeterminismus

Ein nichtdeterministischer Algorithmus A kann 'raten'. In einem Zustand z kann A z.B. eine Variable auf 0 und auf 1 setzen und damit in zwei Nachfolgezuständen z_0 und z_1 sein. Man interpretiert dies so, dass A in beiden Zuständen zugleich ist. Eine Eingabe x wird dann akzeptiert, wenn es *eine* Rechnung gibt, in der A x akzeptiert.

Das 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.

Satz (Nichtdeterminismus = Verifikation)

Die Definitionen von NP mittels Nichtdeterminismus und Verifikationsalgorithmen sind äquivalent.

Nichtdeterminismus

Nichtdeterminismus

Ein nichtdeterministischer Algorithmus A kann 'raten'. In einem Zustand z kann A z.B. eine Variable auf 0 und auf 1 setzen und damit in zwei Nachfolgezuständen z_0 und z_1 sein. Man interpretiert dies so, dass A in beiden Zuständen zugleich ist. Eine Eingabe x wird dann akzeptiert, wenn es *eine* Rechnung gibt, in der A x akzeptiert.

Das 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.

Satz (Nichtdeterminismus = Verifikation)

Die Definitionen von NP mittels Nichtdeterminismus und Verifikationsalgorithmen sind äquivalent.

P vs. NP

Zusammenhänge zwischen P und NP:

- $P \subseteq NP$ ist klar.
- $P \supseteq NP$ und damit $P = NP$ ist ungelöst.
- Die Theorie der NP-vollständigen Sprachen liefert einen starken Hinweis darauf, dass $P \neq NP$ gilt, denn es wird vermutet, dass keine NP-vollständige Sprache in P liegt.
- Es gibt weitere Hinweise hierauf ... aber genaues weiß man nicht...

P vs. NP

Zusammenhänge zwischen P und NP:

- $P \subseteq NP$ ist klar.
- $P \supseteq NP$ und damit $P = NP$ ist ungelöst.
- Die Theorie der NP-vollständigen Sprachen liefert einen starken Hinweis darauf, dass $P \neq NP$ gilt, denn es wird vermutet, dass keine NP-vollständige Sprache in P liegt.
- Es gibt weitere Hinweise hierauf ... aber genaues weiß man nicht...

P vs. NP

Zusammenhänge zwischen P und NP:

- $P \subseteq NP$ ist klar.
- $P \supseteq NP$ und damit $P = NP$ ist ungelöst.
- Die Theorie der NP-vollständigen Sprachen liefert einen starken Hinweis darauf, dass $P \neq NP$ gilt, denn es wird vermutet, dass keine NP-vollständige Sprache in P liegt.
- Es gibt weitere Hinweise hierauf ... aber genaues weiß man nicht...

NP-Probleme lösen

Satz

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.

□

NP-Probleme lösen

Satz

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).

□

NP-Probleme lösen

Satz

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 $O(|x|^c)$.

□

NP-Probleme lösen

Satz

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 $O(|x|^c)$. Man geht alle $2^{O(|x|^c)}$ Zertifikate durch und führt für jeden den Verifikationsalgorithmus aus. □

NP-Probleme lösen

Satz

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 $O(|x|^c)$. Man geht alle $2^{O(|x|^c)}$ Zertifikate durch und führt für jeden den Verifikationsalgorithmus aus. Dieses Verfahren ist in $2^{O(n^k)}$. □

Mengenpartitionsproblem - Deterministisch

Das 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.

Algorithmus 1 Suchraum durchsuchen (deterministisch!)

```
1: for all A ⊆ S do
2:     if Σ_{x∈A} x = Σ_{x∈Ā} x then
3:         return true
4:     end if
5: end for
6: return false
```

Laufzeit ist *deterministisch* in $O(2^{|S|})$

Mengenpartitionsproblem - Nichtdeterministisch

Das 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.

Algorithmus 2 Suchraum durchsuchen (nichtdeterministisch!)

1: Rate ein $A \subseteq S$
2: **if** $\sum_{x \in A} x = \sum_{x \in \overline{A}} x$ **then**
3: **return true**
4: **end if**
5: **return false**

Laufzeit ist *nichtdeterministisch* in $O(|S|)$ (also in NP).

NP-Probleme

Das Teilsummenproblem

Gegeben ist eine Menge $S \subset \mathbb{N}$ und ein $t \in \mathbb{N}$. Gibt es eine Menge $S' \subseteq S$ mit $\sum_{s \in S'} s = t$?

Das Cliquenproblem

Gegeben ist ein ungerichteter Graph $G = (V, E)$ und ein $k \in \mathbb{N}$. Enthält G eine Clique, d.h. ein vollständigen Graphen, der Größe k als Teilgraph?

Das Färbungsproblem

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

NP-Probleme

Das Teilsummenproblem

Gegeben ist eine Menge $S \subset \mathbb{N}$ und ein $t \in \mathbb{N}$. Gibt es eine Menge $S' \subseteq S$ mit $\sum_{s \in S'} s = t$?

Das Cliquenproblem

Gegeben ist ein ungerichteter Graph $G = (V, E)$ und ein $k \in \mathbb{N}$. Enthält G eine Clique, d.h. ein vollständigen Graphen, der Größe k als Teilgraph?

Das Färbungsproblem

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

NP-Probleme

Das Teilsummenproblem

Gegeben ist eine Menge $S \subset \mathbb{N}$ und ein $t \in \mathbb{N}$. Gibt es eine Menge $S' \subseteq S$ mit $\sum_{s \in S'} s = t$?

Das Cliquenproblem

Gegeben ist ein ungerichteter Graph $G = (V, E)$ und ein $k \in \mathbb{N}$. Enthält G eine Clique, d.h. ein vollständigen Graphen, der Größe k als Teilgraph?

Das Färbungsproblem

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

$coNP$

P ist komplementabgeschlossen. Bei NP weiss man dies nicht.

Definition ($coNP$)

Wir definieren die Komplexitätsklasse $coNP$ durch

$$L \in coNP \text{ gdw. } \overline{L} \in NP.$$

Anmerkung

Die Frage, ob NP gegenüber Komplementbildung abgeschlossen ist, kann so als die Frage, ob $NP = coNP$ gilt, formuliert werden.

$coNP$

P ist komplementabgeschlossen. Bei NP weiss man dies nicht.

Definition ($coNP$)

Wir definieren die Komplexitätsklasse $coNP$ durch

$$L \in coNP \text{ gdw. } \overline{L} \in NP.$$

Anmerkung

Die Frage, ob NP gegenüber Komplementbildung abgeschlossen ist, kann so als die Frage, ob $NP = coNP$ gilt, formuliert werden.

$coNP$

P ist komplementabgeschlossen. Bei NP weiss man dies nicht.

Definition ($coNP$)

Wir definieren die Komplexitätsklasse $coNP$ durch

$$L \in coNP \text{ gdw. } \overline{L} \in NP.$$

Anmerkung

Die Frage, ob NP gegenüber Komplementbildung abgeschlossen ist, kann so als die Frage, ob $NP = coNP$ gilt, formuliert werden.

Vier Möglichkeiten

Eine der folgenden vier Möglichkeiten gilt:

- $P = NP = coNP$
- $P \subset NP = coNP$
- $P = NP \cap coNP \subset coNP$ und $P = NP \cap coNP \subset NP$ und $NP \neq coNP$
- $P \subset NP \cap coNP \subset coNP$ und $P \subset NP \cap coNP \subset NP$ und $NP \neq coNP$

Wir haben leider keine Ahnung welche. Die vierte ist am wahrscheinlichsten...

Vier Möglichkeiten

Eine der folgenden vier Möglichkeiten gilt:

- $P = NP = coNP$
- $P \subset NP = coNP$
- $P = NP \cap coNP \subset coNP$ und $P = NP \cap coNP \subset NP$ und $NP \neq coNP$
- $P \subset NP \cap coNP \subset coNP$ und $P \subset NP \cap coNP \subset NP$ und $NP \neq coNP$

Wir haben leider keine Ahnung welche. Die vierte ist am wahrscheinlichsten...

Vier Möglichkeiten

Eine der folgenden vier Möglichkeiten gilt:

- $P = NP = coNP$
- $P \subset NP = coNP$
- $P = NP \cap coNP \subset coNP$ und $P = NP \cap coNP \subset NP$ und $NP \neq coNP$
- $P \subset NP \cap coNP \subset coNP$ und $P \subset NP \cap coNP \subset NP$ und $NP \neq coNP$

Wir haben leider keine Ahnung welche. Die vierte ist am wahrscheinlichsten...

Vier Möglichkeiten

Eine der folgenden vier Möglichkeiten gilt:

- $P = NP = coNP$
- $P \subset NP = coNP$
- $P = NP \cap coNP \subset coNP$ und $P = NP \cap coNP \subset NP$ und $NP \neq coNP$
- $P \subset NP \cap coNP \subset coNP$ und $P \subset NP \cap coNP \subset NP$ und $NP \neq coNP$

Wir haben leider keine Ahnung welche. Die vierte ist am wahrscheinlichsten...

Vier Möglichkeiten

Eine der folgenden vier Möglichkeiten gilt:

- $P = NP = coNP$
- $P \subset NP = coNP$
- $P = NP \cap coNP \subset coNP$ und $P = NP \cap coNP \subset NP$ und $NP \neq coNP$
- $P \subset NP \cap coNP \subset coNP$ und $P \subset NP \cap coNP \subset NP$ und $NP \neq coNP$

Wir haben leider keine Ahnung welche. Die vierte ist am wahrscheinlichsten...

Abschlusseigenschaften

Satz

Sind $L_1, L_2 \in NP$, so gilt auch $L_1 \cup L_2, L_1 \cap L_2, L_1 \cdot L_2, L_1^* \in NP$.

Beweis.

Beweis ähnlich zu den Beweisen bei P. □

Pause to Ponder

Was ist das Problem beim Komplementabschluss? Warum geht der Beweis nicht?

Abschlusseigenschaften

Satz

Sind $L_1, L_2 \in NP$, so gilt auch $L_1 \cup L_2, L_1 \cap L_2, L_1 \cdot L_2, L_1^* \in NP$.

Beweis.

Beweis ähnlich zu den Beweisen bei P. □

Pause to Ponder

Was ist das Problem beim Komplementabschluss? Warum geht der Beweis nicht?

Zwischenzusammenfassung

Definition (NP)

$L \in NP$ gdw. ein Algorithmus A mit zwei Eingaben und mit polynomialer Laufzeit existiert, so dass für ein c gilt

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

Satz (Nichtdeterminismus = Verifikation)

Die Definitionen von NP mittels Nichtdeterminismus und Verifikationsalgorithmen sind äquivalent.

Fazit

Man weiß über NP nur sehr wenig gesichert, aber viele Vermutungen kann man gut untermauern. Insb. hilft die Klasse 'unhandbare' Probleme zu klassifizieren.

Zwischenzusammenfassung

Definition (NP)

$L \in NP$ gdw. ein Algorithmus A mit zwei Eingaben und mit polynomialer Laufzeit existiert, so dass für ein c gilt

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

Satz (Nichtdeterminismus = Verifikation)

Die Definitionen von NP mittels Nichtdeterminismus und Verifikationsalgorithmen sind äquivalent.

Fazit

Man weiß über NP nur sehr wenig gesichert, aber viele Vermutungen kann man gut untermauern. Insb. hilft die Klasse 'unhandbare' Probleme zu klassifizieren.

Zwischenzusammenfassung

Definition (NP)

$L \in NP$ gdw. ein Algorithmus A mit zwei Eingaben und mit polynomialer Laufzeit existiert, so dass für ein c gilt

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

Satz (Nichtdeterminismus = Verifikation)

Die Definitionen von NP mittels Nichtdeterminismus und Verifikationsalgorithmen sind äquivalent.

Fazit

Man weiß über NP nur sehr wenig gesichert, aber viele Vermutungen kann man gut untermauern. Insb. hilft die Klasse 'unhandbare' Probleme zu klassifizieren.

Reduktionen / Ausblick

Nächstes Mal:

- Definition der Reduktion
  - Problem A wird in Problem B umgewandelt.
  - Wird Problem B gelöst, löst dies auch A.
  - A ist also höchstens so kompliziert wie B.
- Definition der *NP*-vollständigkeit
  - Damit: Die kompliziertesten Probleme in *NP* charakterisieren.
  - Löst man eins, löst man alle.
  - Nachweis der *NP*-vollständigkeit eines Problems heißt dann, dass es (sehr wahrscheinlich) nicht effizient lösbar ist (denn sonst wären alle effizient lösbar).
- Beweis der *NP*-Vollständigkeit einiger Probleme.

Reduktionen / Ausblick

Nächstes Mal:

- Definition der Reduktion
  - Problem A wird in Problem B umgewandelt.
  - Wird Problem B gelöst, löst dies auch A.
  - A ist also höchstens so kompliziert wie B.
- Definition der *NP*-vollständigkeit
  - Damit: Die kompliziertesten Probleme in *NP* charakterisieren.
  - Löst man eins, löst man alle.
  - Nachweis der *NP*-vollständigkeit eines Problems heißt dann, dass es (sehr wahrscheinlich) nicht effizient lösbar ist (denn sonst wären alle effizient lösbar).
- Beweis der *NP*-Vollständigkeit einiger Probleme.

Reduktionen / Ausblick

Nächstes Mal:

- Definition der Reduktion
  - Problem A wird in Problem B umgewandelt.
  - Wird Problem B gelöst, löst dies auch A.
  - A ist also höchstens so kompliziert wie B.
- Definition der *NP*-vollständigkeit
  - Damit: Die kompliziertesten Probleme in *NP* charakterisieren.
  - Löst man eins, löst man alle.
  - Nachweis der *NP*-vollständigkeit eines Problems heißt dann, dass es (sehr wahrscheinlich) nicht effizient lösbar ist (denn sonst wären alle effizient lösbar).
- Beweis der *NP*-Vollständigkeit einiger Probleme.

Reduktionen / Ausblick

Nächstes Mal:

- Definition der Reduktion
  - Problem A wird in Problem B umgewandelt.
  - Wird Problem B gelöst, löst dies auch A.
  - A ist also höchstens so kompliziert wie B.
- Definition der *NP*-vollständigkeit
  - Damit: Die kompliziertesten Probleme in *NP* charakterisieren.
  - Löst man eins, löst man alle.
  - Nachweis der *NP*-vollständigkeit eines Problems heißt dann, dass es (sehr wahrscheinlich) nicht effizient lösbar ist (denn sonst wären alle effizient lösbar).
- Beweis der *NP*-Vollständigkeit einiger Probleme.

Reduktionen / Ausblick

Nächstes Mal:

- Definition der Reduktion
  - Problem A wird in Problem B umgewandelt.
  - Wird Problem B gelöst, löst dies auch A.
  - A ist also höchstens so kompliziert wie B.
- Definition der *NP*-vollständigkeit
  - Damit: Die kompliziertesten Probleme in *NP* charakterisieren.
  - Löst man eins, löst man alle.
  - Nachweis der *NP*-vollständigkeit eines Problems heißt dann, dass es (sehr wahrscheinlich) nicht effizient lösbar ist (denn sonst wären alle effizient lösbar).
- Beweis der *NP*-Vollständigkeit einiger Probleme.

Reduktionen / Ausblick

Nächstes Mal:

- Definition der Reduktion
  - Problem A wird in Problem B umgewandelt.
  - Wird Problem B gelöst, löst dies auch A.
  - A ist also höchstens so kompliziert wie B.
- Definition der *NP*-vollständigkeit
  - Damit: Die kompliziertesten Probleme in *NP* charakterisieren.
  - Löst man eins, löst man alle.
  - Nachweis der *NP*-vollständigkeit eines Problems heißt dann, dass es (sehr wahrscheinlich) nicht effizient lösbar ist (denn sonst wären alle effizient lösbar).
- Beweis der *NP*-Vollständigkeit einiger Probleme.

Wiederholung: Codierung, P

Zusammenfassung

- Die Eingabe für einen Algorithmus muss codiert sein. Wir gehen hier von einer Codierung im Binärcode aus. Alle Eingaben sind dann Zeichenketten $x \in \{0,1\}^*$. Die Menge der Instanzen ist dann $\{0,1\}^*$.
- Ein Entscheidungs*problem* kann dann als Sprache $L \subseteq \{0,1\}^*$ verstanden werden. Es gibt dann Eingaben x,
  - die der Algorithmus akzeptieren soll ($x \in L$) und solche
  - die der Algorithmus ablehnen soll ($x \notin L$).
- Sprachen können akzeptiert oder entschieden werden. (Beim Entscheiden, hält der Algorithmus auch stets auf Eingaben, die er nicht akzeptieren soll.)
- Gibt es zu einem Problem $L \subseteq \{0,1\}^*$ einen Algorithmus, der das Problem in Polynomialzeit entscheidet, so gilt $L \in P$.

Wiederholung: NP

Definition (NP)

$L \in NP$ gdw. ein Algorithmus A mit zwei Eingaben und mit polynomialer Laufzeit existiert, so dass für ein c gilt

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

Satz (Nichtdeterminismus = Verifikation)

Die Definitionen von NP mittels Nichtdeterminismus und Verifikationsalgorithmen sind äquivalent.

Fazit

Man weiß über NP nur sehr wenig gesichert, aber viele Vermutungen kann man gut untermauern. Insb. hilft die Klasse 'unhandbare' Probleme zu klassifizieren.