Algorithmen und Komplexität
Vorlesung 18

Wolfgang Globke

universität wien

DHBW
Duale Hochschule
Baden-Württemberg

6. Juni 2019

Wiederholung: TSP und HC

Beim Problem des Handelsreisenden, kurz TSP, suchen wir in einem gewichteten Graphen $G = (V, E)$ nach einem Zyklus Z, der jeden Knoten des Graphen enthält und minimales Gewicht hat unter allen solchen Zyklen
(als Entscheidungsproblem: es soll $\mathrm{w}(Z) \leq c$ für eine gegeben Konstante c sein).

Wiederholung: TSP und HC

Beim Problem des Handelsreisenden, kurz TSP, suchen wir in einem gewichteten Graphen $G = (V, E)$ nach einem Zyklus Z, der jeden Knoten des Graphen enthält und minimales Gewicht hat unter allen solchen Zyklen
(als Entscheidungsproblem: es soll $\mathrm{w}(Z) \leq c$ für eine gegeben Konstante c sein).

- Sehr aufwändiges Problem:
  Bekannte Algorithmen für exakte Lösungen haben exponentiellen Aufwand.

Wiederholung: TSP und HC

Beim Problem des Handelsreisenden, kurz TSP, suchen wir in einem gewichteten Graphen $G = (V, E)$ nach einem Zyklus Z, der jeden Knoten des Graphen enthält und minimales Gewicht hat unter allen solchen Zyklen
(als Entscheidungsproblem: es soll $\mathrm{w}(Z) \leq c$ für eine gegeben Konstante c sein).

- Sehr aufwändiges Problem:
  Bekannte Algorithmen für exakte Lösungen haben exponentiellen Aufwand.
- Auch Approximationsalgorithmen könnten für das allgemeine TSP nur dann polynomialen Aufwand haben, wenn $\mathbf{P} = \mathbf{NP}$ gilt.

Wiederholung: TSP und HC

Beim Problem des Handelsreisenden, kurz TSP, suchen wir in einem gewichteten Graphen $G = (V, E)$ nach einem Zyklus Z, der jeden Knoten des Graphen enthält und minimales Gewicht hat unter allen solchen Zyklen
(als Entscheidungsproblem: es soll $\mathrm{w}(Z) \leq c$ für eine gegeben Konstante c sein).

- Sehr aufwändiges Problem:
  Bekannte Algorithmen für exakte Lösungen haben exponentiellen Aufwand.
- Auch Approximationsalgorithmen könnten für das allgemeine TSP nur dann polynomialen Aufwand haben, wenn $\mathbf{P} = \mathbf{NP}$ gilt.
- Im Spezialfall, dass die Gewichte auf G durch euklidische Abstände gegeben sind, kann man aber mit polynomialem Aufwand Näherungslösungen $\tilde{Z}$ bestimmen, die $\mathrm{w}(\tilde{Z}) \leq 2\mathrm{w}(Z)$ erfüllen (wenn Z eine exakte Lösung ist).

Wiederholung: TSP und HC

Beim Problem des Handelsreisenden, kurz TSP, suchen wir in einem gewichteten Graphen $G = (V, E)$ nach einem Zyklus Z, der jeden Knoten des Graphen enthält und minimales Gewicht hat unter allen solchen Zyklen
(als Entscheidungsproblem: es soll $\mathrm{w}(Z) \leq c$ für eine gegeben Konstante c sein).

- Sehr aufwändiges Problem:
  Bekannte Algorithmen für exakte Lösungen haben exponentiellen Aufwand.
- Auch Approximationsalgorithmen könnten für das allgemeine TSP nur dann polynomialen Aufwand haben, wenn $\mathbf{P} = \mathbf{NP}$ gilt.
- Im Spezialfall, dass die Gewichte auf G durch euklidische Abstände gegeben sind, kann man aber mit polynomialem Aufwand Näherungslösungen $\tilde{Z}$ bestimmen, die $\mathrm{w}(\tilde{Z}) \leq 2\mathrm{w}(Z)$ erfüllen (wenn Z eine exakte Lösung ist).

Beim Problem der Hamilton-Zyklen, kurz HC, suchen wir einen Zyklus, der jeden Knoten in G genau einmal enthält.

Wiederholung: TSP und HC

Beim Problem des Handelsreisenden, kurz TSP, suchen wir in einem gewichteten Graphen $G = (V, E)$ nach einem Zyklus Z, der jeden Knoten des Graphen enthält und minimales Gewicht hat unter allen solchen Zyklen
(als Entscheidungsproblem: es soll $\mathrm{w}(Z) \leq c$ für eine gegeben Konstante c sein).

- Sehr aufwändiges Problem:
  Bekannte Algorithmen für exakte Lösungen haben exponentiellen Aufwand.
- Auch Approximationsalgorithmen könnten für das allgemeine TSP nur dann polynomialen Aufwand haben, wenn $\mathbf{P} = \mathbf{NP}$ gilt.
- Im Spezialfall, dass die Gewichte auf G durch euklidische Abstände gegeben sind, kann man aber mit polynomialem Aufwand Näherungslösungen $\tilde{Z}$ bestimmen, die $\mathrm{w}(\tilde{Z}) \leq 2\mathrm{w}(Z)$ erfüllen (wenn Z eine exakte Lösung ist).

Beim Problem der Hamilton-Zyklen, kurz HC, suchen wir einen Zyklus, der jeden Knoten in G genau einmal enthält.

- HC kann als „ungewichtete” Version von TSP aufgefasst werden.

Wiederholung: TSP und HC

Beim Problem des Handelsreisenden, kurz TSP, suchen wir in einem gewichteten Graphen $G = (V, E)$ nach einem Zyklus Z, der jeden Knoten des Graphen enthält und minimales Gewicht hat unter allen solchen Zyklen
(als Entscheidungsproblem: es soll $\mathrm{w}(Z) \leq c$ für eine gegeben Konstante c sein).

- Sehr aufwändiges Problem:
  Bekannte Algorithmen für exakte Lösungen haben exponentiellen Aufwand.
- Auch Approximationsalgorithmen könnten für das allgemeine TSP nur dann polynomialen Aufwand haben, wenn $\mathbf{P} = \mathbf{NP}$ gilt.
- Im Spezialfall, dass die Gewichte auf G durch euklidische Abstände gegeben sind, kann man aber mit polynomialem Aufwand Näherungslösungen $\tilde{Z}$ bestimmen, die $\mathrm{w}(\tilde{Z}) \leq 2\mathrm{w}(Z)$ erfüllen (wenn Z eine exakte Lösung ist).

Beim Problem der Hamilton-Zyklen, kurz HC, suchen wir einen Zyklus, der jeden Knoten in G genau einmal enthält.

- HC kann als „ungewichtete" Version von TSP aufgefasst werden.
- HC kann auf TSP effizient reduziert werden ($\mathrm{HC} \preceq_{\mathrm{p}} \mathrm{TSP}$), d.h. ein Algorithmus der TSP löst kann auch HC lösen, ohne dass das Übertragen der Problemstellung wesentlich zum Aufwand beiträgt.

Wiederholung: TSP und HC

Beim Problem des Handelsreisenden, kurz TSP, suchen wir in einem gewichteten Graphen $G = (V, E)$ nach einem Zyklus Z, der jeden Knoten des Graphen enthält und minimales Gewicht hat unter allen solchen Zyklen
(als Entscheidungsproblem: es soll $\mathrm{w}(Z) \leq c$ für eine gegeben Konstante c sein).

- Sehr aufwändiges Problem:
  Bekannte Algorithmen für exakte Lösungen haben exponentiellen Aufwand.
- Auch Approximationsalgorithmen könnten für das allgemeine TSP nur dann polynomialen Aufwand haben, wenn $\mathbf{P} = \mathbf{NP}$ gilt.
- Im Spezialfall, dass die Gewichte auf G durch euklidische Abstände gegeben sind, kann man aber mit polynomialem Aufwand Näherungslösungen $\tilde{Z}$ bestimmen, die $\mathrm{w}(\tilde{Z}) \leq 2\mathrm{w}(Z)$ erfüllen (wenn Z eine exakte Lösung ist).

Beim Problem der Hamilton-Zyklen, kurz HC, suchen wir einen Zyklus, der jeden Knoten in G genau einmal enthält.

- HC kann als „ungewichtete" Version von TSP aufgefasst werden.
- HC kann auf TSP effizient reduziert werden ($\mathrm{HC} \preceq_{\mathrm{p}} \mathrm{TSP}$), d.h. ein Algorithmus der TSP löst kann auch HC lösen, ohne dass das Übertragen der Problemstellung wesentlich zum Aufwand beiträgt.
- Die Komplexität von TSP ist also mindestens so hoch wie die von HC. Leider wissen wir nicht, ob wir HC besser als in exponentieller Zeit lösen können.

Wiederholung: Problemreduktion

Ein Problem $\mathscr{P}$ ist auf das Problem $\mathscr{Q}$ reduzierbar, wenn ein Algorithmus A, der $\mathscr{Q}$ löst, auch $\mathscr{P}$ lösen kann. Schreibe $\mathscr{P} \preceq \mathscr{Q}$.

Wiederholung: Problemreduktion

Ein Problem $\mathcal{P}$ ist auf das Problem $\mathcal{Q}$ reduzierbar, wenn ein Algorithmus A, der $\mathcal{Q}$ löst, auch $\mathcal{P}$ lösen kann. Schreibe $\mathcal{P} \preceq \mathcal{Q}$.

- Wenn $\mathcal{P}$ mit unwesentlichem Aufwand (polynomial) in eine Instanz des Problems $\mathcal{Q}$ umgewandelt werden kann, so bedeutete dies, dass $\mathcal{Q}$ mindestens so schwer ist wie $\mathcal{P}$. Schreibe $\mathcal{P} \preceq_{\mathrm{p}} \mathcal{Q}$.

Ein Problem $\mathcal{P}$ ist auf das Problem $\mathcal{Q}$ reduzierbar, wenn ein Algorithmus A, der $\mathcal{Q}$ löst, auch $\mathcal{P}$ lösen kann. Schreibe $\mathcal{P} \preceq \mathcal{Q}$.

- Wenn $\mathcal{P}$ mit unwesentlichem Aufwand (polynomial) in eine Instanz des Problems $\mathcal{Q}$ umgewandelt werden kann, so bedeutete dies, dass $\mathcal{Q}$ mindestens so schwer ist wie $\mathcal{P}$. Schreibe $\mathcal{P} \preceq_{\mathrm{p}} \mathcal{Q}$.
- Ist $\mathcal{P} \preceq_{\mathrm{p}} \mathcal{Q}$ und ist eine untere Schranke für den Aufwand von $\mathcal{P}$ bekannt, so gilt diese untere Schranke auch für $\mathcal{Q}$.

Wiederholung: Turing-Maschinen

Turing-Maschinen bieten ein schlichtes und sehr mächtiges Rechenmodell, dass ich für die Komplexitätstheorie gut eignet.

Turing-Maschinen bieten ein schlichtes und sehr mächtiges Rechenmodell, dass ich für die Komplexitätstheorie gut eignet.

Beispiel

Eine Turing-Maschine, die zwei Zahlen addiert, die durch Strichfolgen auf dem Band gegeben und durch # getrennt sind.

Q	z	Q'	z'	α
Q_0	$\vert$	Q_0	$\vert$	$\rightarrow$
Q_0	#	Q_1	$\vert$	$\rightarrow$
Q_1	$\vert$	Q_1	$\vert$	$\rightarrow$
Q_1	␣	Q_2	␣	$\leftarrow$
Q_2	?	Q_{end}	␣	$\downarrow$

Hier steht „?” für ein beliebiges Zeichen.

Wiederholung: Turing-Maschinen

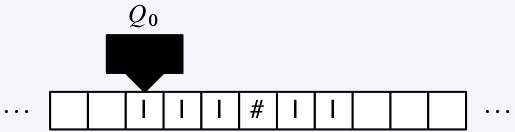
Turing-Maschinen bieten ein schlichtes und sehr mächtiges Rechenmodell, dass ich für die Komplexitätstheorie gut eignet.

Beispiel

Eine Turing-Maschine, die zwei Zahlen addiert, die durch Strichfolgen auf dem Band gegeben und durch # getrennt sind.

Q	z	Q'	z'	α
Q_0	$\vert$	Q_0	$\vert$	$\rightarrow$
Q_0	#	Q_1	$\vert$	$\rightarrow$
Q_1	$\vert$	Q_1	$\vert$	$\rightarrow$
Q_1	␣	Q_2	␣	$\leftarrow$
Q_2	?	Q_{end}	␣	$\downarrow$

Hier steht „?” für ein beliebiges Zeichen.



Wiederholung: Turing-Maschinen

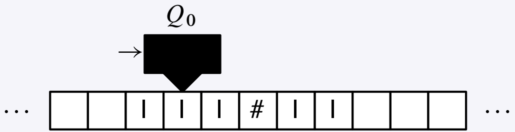
Turing-Maschinen bieten ein schlichtes und sehr mächtiges Rechenmodell, dass ich für die Komplexitätstheorie gut eignet.

Beispiel

Eine Turing-Maschine, die zwei Zahlen addiert, die durch Strichfolgen auf dem Band gegeben und durch # getrennt sind.

Q	z	Q'	z'	α
Q_0	$\vert$	Q_0	$\vert$	$\rightarrow$
Q_0	#	Q_1	$\vert$	$\rightarrow$
Q_1	$\vert$	Q_1	$\vert$	$\rightarrow$
Q_1	␣	Q_2	␣	$\leftarrow$
Q_2	?	Q_{end}	␣	$\downarrow$

Hier steht „?" für ein beliebiges Zeichen.



$$(Q_0, |) \mapsto (Q_0, |, \rightarrow)$$

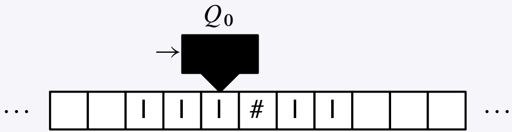
Turing-Maschinen bieten ein schlichtes und sehr mächtiges Rechenmodell, dass ich für die Komplexitätstheorie gut eignet.

Beispiel

Eine Turing-Maschine, die zwei Zahlen addiert, die durch Strichfolgen auf dem Band gegeben und durch # getrennt sind.

Q	z	Q'	z'	α
Q_0	$\vert$	Q_0	$\vert$	$\rightarrow$
Q_0	#	Q_1	$\vert$	$\rightarrow$
Q_1	$\vert$	Q_1	$\vert$	$\rightarrow$
Q_1	␣	Q_2	␣	$\leftarrow$
Q_2	?	Q_{end}	␣	$\downarrow$

Hier steht „?" für ein beliebiges Zeichen.



$(Q_0, |) \mapsto (Q_0, |, \rightarrow)$

Wiederholung: Turing-Maschinen

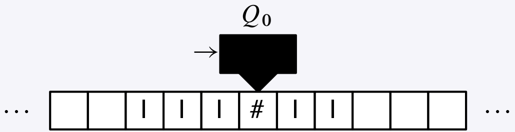
Turing-Maschinen bieten ein schlichtes und sehr mächtiges Rechenmodell, dass ich für die Komplexitätstheorie gut eignet.

Beispiel

Eine Turing-Maschine, die zwei Zahlen addiert, die durch Strichfolgen auf dem Band gegeben und durch # getrennt sind.

Q	z	Q'	z'	α
Q_0	$\vert$	Q_0	$\vert$	$\rightarrow$
Q_0	#	Q_1	$\vert$	$\rightarrow$
Q_1	$\vert$	Q_1	$\vert$	$\rightarrow$
Q_1	$\sqcup$	Q_2	$\sqcup$	$\leftarrow$
Q_2	?	Q_{end}	$\sqcup$	$\downarrow$

Hier steht „?” für ein beliebiges Zeichen.



$$(Q_0, |) \mapsto (Q_0, |, \rightarrow)$$

Wiederholung: Turing-Maschinen

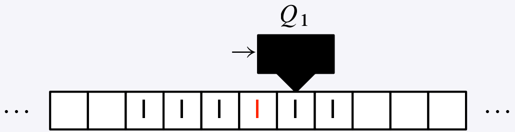
Turing-Maschinen bieten ein schlichtes und sehr mächtiges Rechenmodell, dass ich für die Komplexitätstheorie gut eignet.

Beispiel

Eine Turing-Maschine, die zwei Zahlen addiert, die durch Strichfolgen auf dem Band gegeben und durch # getrennt sind.

Q	z	Q'	z'	α
Q_0	$\vert$	Q_0	$\vert$	$\rightarrow$
Q_0	#	Q_1	$\vert$	$\rightarrow$
Q_1	$\vert$	Q_1	$\vert$	$\rightarrow$
Q_1	$\sqcup$	Q_2	$\sqcup$	$\leftarrow$
Q_2	?	Q_{end}	$\sqcup$	$\downarrow$

Hier steht „?” für ein beliebiges Zeichen.



$$(Q_0, \#) \mapsto (Q_1, |, \rightarrow)$$

Wiederholung: Turing-Maschinen

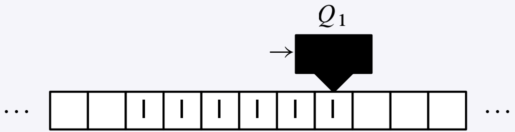
Turing-Maschinen bieten ein schlichtes und sehr mächtiges Rechenmodell, dass ich für die Komplexitätstheorie gut eignet.

Beispiel

Eine Turing-Maschine, die zwei Zahlen addiert, die durch Strichfolgen auf dem Band gegeben und durch # getrennt sind.

Q	z	Q'	z'	α
Q_0	$\vert$	Q_0	$\vert$	$\rightarrow$
Q_0	#	Q_1	$\vert$	$\rightarrow$
Q_1	$\vert$	Q_1	$\vert$	$\rightarrow$
Q_1	$\sqcup$	Q_2	$\sqcup$	$\leftarrow$
Q_2	?	Q_{end}	$\sqcup$	$\downarrow$

Hier steht „?” für ein beliebiges Zeichen.



$$(Q_1, |) \mapsto (Q_1, |, \rightarrow)$$

Wiederholung: Turing-Maschinen

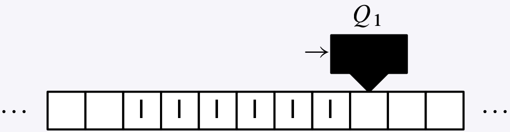
Turing-Maschinen bieten ein schlichtes und sehr mächtiges Rechenmodell, dass ich für die Komplexitätstheorie gut eignet.

Beispiel

Eine Turing-Maschine, die zwei Zahlen addiert, die durch Strichfolgen auf dem Band gegeben und durch # getrennt sind.

Q	z	Q'	z'	α
Q_0	$\vert$	Q_0	$\vert$	$\rightarrow$
Q_0	#	Q_1	$\vert$	$\rightarrow$
Q_1	$\vert$	Q_1	$\vert$	$\rightarrow$
Q_1	$\sqcup$	Q_2	$\sqcup$	$\leftarrow$
Q_2	?	Q_{end}	$\sqcup$	$\downarrow$

Hier steht „?” für ein beliebiges Zeichen.



$$(Q_1, |) \mapsto (Q_1, |, \rightarrow)$$

Wiederholung: Turing-Maschinen

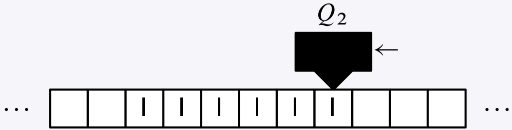
Turing-Maschinen bieten ein schlichtes und sehr mächtiges Rechenmodell, dass ich für die Komplexitätstheorie gut eignet.

Beispiel

Eine Turing-Maschine, die zwei Zahlen addiert, die durch Strichfolgen auf dem Band gegeben und durch # getrennt sind.

Q	z	Q'	z'	α
Q_0	$\mid$	Q_0	$\mid$	$\rightarrow$
Q_0	#	Q_1	$\mid$	$\rightarrow$
Q_1	$\mid$	Q_1	$\mid$	$\rightarrow$
Q_1	␣	Q_2	␣	$\leftarrow$
Q_2	?	Q_{end}	␣	$\downarrow$

Hier steht „?” für ein beliebiges Zeichen.



$$(Q_1, \text{␣}) \mapsto (Q_2, \text{␣}, \leftarrow)$$

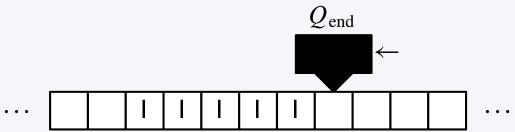
Turing-Maschinen bieten ein schlichtes und sehr mächtiges Rechenmodell, dass ich für die Komplexitätstheorie gut eignet.

Beispiel

Eine Turing-Maschine, die zwei Zahlen addiert, die durch Strichfolgen auf dem Band gegeben und durch # getrennt sind.

Q	z	Q'	z'	α
Q_0	$\vert$	Q_0	$\vert$	$\rightarrow$
Q_0	#	Q_1	$\vert$	$\rightarrow$
Q_1	$\vert$	Q_1	$\vert$	$\rightarrow$
Q_1	␣	Q_2	␣	$\leftarrow$
Q_2	?	Q_{end}	␣	$\downarrow$

Hier steht „?” für ein beliebiges Zeichen.



$$(Q_2, ?) \mapsto (Q_{\text{end}}, ␣, \downarrow)$$

Wiederholung: Turing-Maschinen

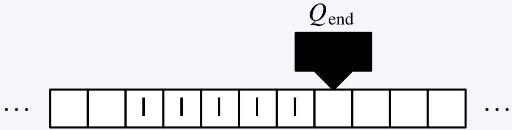
Turing-Maschinen bieten ein schlichtes und sehr mächtiges Rechenmodell, dass ich für die Komplexitätstheorie gut eignet.

Beispiel

Eine Turing-Maschine, die zwei Zahlen addiert, die durch Strichfolgen auf dem Band gegeben und durch # getrennt sind.

Q	z	Q'	z'	α
Q_0	$\mid$	Q_0	$\mid$	$\rightarrow$
Q_0	#	Q_1	$\mid$	$\rightarrow$
Q_1	$\mid$	Q_1	$\mid$	$\rightarrow$
Q_1	␣	Q_2	␣	$\leftarrow$
Q_2	?	Q_{end}	␣	$\downarrow$

Hier steht „?” für ein beliebiges Zeichen.



Endzustand erreicht
Berechnung zuende

Die Klasse **P** der deterministisch polynomialen Entscheidungsprobleme besteht aus allen (entscheidbaren) Prädikaten, die von einer Turing-Maschine in polynomialer Zeit entschieden werden können.

Die Klasse **P** der deterministisch polynomialen Entscheidungsprobleme besteht aus allen (entscheidbaren) Prädikaten, die von einer Turing-Maschine in polynomialer Zeit entschieden werden können.

- T entscheidet ein Prädikat p, wenn T für Eingaben aus der Menge $M_p = \{x \mid p(x) = \text{wahr}\}$ in einem Endzustand Q_{wahr} endet, und für $x \notin M_p$ in einem Endzustand Q_{falsch} endet.

Die Klasse **P** der deterministisch polynomialen Entscheidungsprobleme besteht aus allen (entscheidbaren) Prädikaten, die von einer Turing-Maschine in polynomialer Zeit entschieden werden können.

- T entscheidet ein Prädikat p, wenn T für Eingaben aus der Menge $M_p = \{x \mid p(x) = \text{wahr}\}$ in einem Endzustand Q_{wahr} endet, und für $x \notin M_p$ in einem Endzustand Q_{falsch} endet.
- „Zeit" wird für Turing-Maschinen in der Anzahl der Zustandsübergänge gemessen.

Die Klasse **P** der deterministisch polynomialen Entscheidungsprobleme besteht aus allen (entscheidbaren) Prädikaten, die von einer Turing-Maschine in polynomialer Zeit entschieden werden können.

- T entscheidet ein Prädikat p, wenn T für Eingaben aus der Menge $M_p = \{x \mid p(x) = \text{wahr}\}$ in einem Endzustand Q_{wahr} endet, und für $x \notin M_p$ in einem Endzustand Q_{falsch} endet.
- „Zeit" wird für Turing-Maschinen in der Anzahl der Zustandsübergänge gemessen.
- Fakt:
  Programme, die durch Turing-Maschinen gegeben sind, lassen sich mit polynomialem Aufwand in äquivalente Programme in gleichmächtigen Rechenmodellen (z.B. RAM-Modell) übertragen.

Die Klasse **P** der deterministisch polynomialen Entscheidungsprobleme besteht aus allen (entscheidbaren) Prädikaten, die von einer Turing-Maschine in polynomialer Zeit entschieden werden können.

- T entscheidet ein Prädikat p, wenn T für Eingaben aus der Menge $M_p = \{x \mid p(x) = \text{wahr}\}$ in einem Endzustand Q_{wahr} endet, und für $x \notin M_p$ in einem Endzustand Q_{falsch} endet.
- „Zeit" wird für Turing-Maschinen in der Anzahl der Zustandsübergänge gemessen.
- Fakt:
  Programme, die durch Turing-Maschinen gegeben sind, lassen sich mit polynomialem Aufwand in äquivalente Programme in gleichmächtigen Rechenmodellen (z.B. RAM-Modell) übertragen. Somit ist diese Definition von **P** für alle gebräuchlichen deterministischen Rechenmodelle die selbe.

Nicht in **P**?

Für Entscheidungsprobleme wie HC oder TSP kennen wir keine Turing-Maschine, die sie entscheiden kann. Wir wissen also nicht, ob sie in **P** liegen.

Nicht in **P**?

Für Entscheidungsprobleme wie HC oder TSP kennen wir keine Turing-Maschine, die sie entscheiden kann. Wir wissen also nicht, ob sie in **P** liegen.

Was können wir hier tun?

Für Entscheidungsprobleme wie HC oder TSP kennen wir keine Turing-Maschine, die sie entscheiden kann. Wir wissen also nicht, ob sie in **P** liegen.

Was können wir hier tun?

- Wir haben jedoch gesehen, dass wir für TSP mit polynomialem Aufwand geprüft werden kann, ob ein Lösungsvorschlag korrekt ist.

Nicht in **P**?

Für Entscheidungsprobleme wie HC oder TSP kennen wir keine Turing-Maschine, die sie entscheiden kann. Wir wissen also nicht, ob sie in **P** liegen.

Was können wir hier tun?

- Wir haben jedoch gesehen, dass wir für TSP mit polynomialem Aufwand geprüft werden kann, ob ein Lösungsvorschlag korrekt ist.
- Dies gilt ebenso für HC, da ja HC $\preceq_p$ TSP.

Für Entscheidungsprobleme wie HC oder TSP kennen wir keine Turing-Maschine, die sie entscheiden kann. Wir wissen also nicht, ob sie in **P** liegen.

Was können wir hier tun?

- Wir haben jedoch gesehen, dass wir für TSP mit polynomialem Aufwand geprüft werden kann, ob ein Lösungsvorschlag korrekt ist.
- Dies gilt ebenso für HC, da ja HC $\preceq_p$ TSP.
- Dies liefert einen Algorithmus/eine Turing-Maschine, die diese Probleme exakt lösen kann:
  - Erzeuge einen neuen Lösungsvorschlag Z (in polynomialer Zeit möglich).
  - Teste, ob Z eine Lösung für TSP (bzw. HC) ist.
  - Wiederhole, bis alle Lösungvorschläge abgearbeitet wurden oder eine Lösung gefunden wurde.

Nicht in **P**?

Für Entscheidungsprobleme wie HC oder TSP kennen wir keine Turing-Maschine, die sie entscheiden kann. Wir wissen also nicht, ob sie in **P** liegen.

Was können wir hier tun?

- Wir haben jedoch gesehen, dass wir für TSP mit polynomialem Aufwand geprüft werden kann, ob ein Lösungsvorschlag korrekt ist.
- Dies gilt ebenso für HC, da ja HC $\preceq_p$ TSP.
- Dies liefert einen Algorithmus/eine Turing-Maschine, die diese Probleme exakt lösen kann:
  - Erzeuge einen neuen Lösungsvorschlag Z (in polynomialer Zeit möglich).
  - Teste, ob Z eine Lösung für TSP (bzw. HC) ist.
  - Wiederhole, bis alle Lösungvorschläge abgearbeitet wurden oder eine Lösung gefunden wurde.
- Obwohl der Test effizient ist, haben wir das Problem, dass in der Regel exponentiell viele Lösungsmöglichkeiten getestet werden müssen.

Für Entscheidungsprobleme wie HC oder TSP kennen wir keine Turing-Maschine, die sie entscheiden kann. Wir wissen also nicht, ob sie in **P** liegen.

Was können wir hier tun?

- Wir haben jedoch gesehen, dass wir für TSP mit polynomialem Aufwand geprüft werden kann, ob ein Lösungsvorschlag korrekt ist.
- Dies gilt ebenso für HC, da ja HC $\preceq_p$ TSP.
- Dies liefert einen Algorithmus/eine Turing-Maschine, die diese Probleme exakt lösen kann:
  - Erzeuge einen neuen Lösungsvorschlag Z (in polynomialer Zeit möglich).
  - Teste, ob Z eine Lösung für TSP (bzw. HC) ist.
  - Wiederhole, bis alle Lösungvorschläge abgearbeitet wurden oder eine Lösung gefunden wurde.
- Obwohl der Test effizient ist, haben wir das Problem, dass in der Regel exponentiell viele Lösungsmöglichkeiten getestet werden müssen.
- Wäre es nicht toll, wenn wir eine Turing-Maschine hätten, die uns alle Möglichkeiten gleichzeitig überprüfen ließe?

Nicht in **P**?

Für Entscheidungsprobleme wie HC oder TSP kennen wir keine Turing-Maschine, die sie entscheiden kann. Wir wissen also nicht, ob sie in **P** liegen.

Was können wir hier tun?

- Wir haben jedoch gesehen, dass wir für TSP mit polynomialem Aufwand geprüft werden kann, ob ein Lösungsvorschlag korrekt ist.
- Dies gilt ebenso für HC, da ja HC $\preceq_p$ TSP.
- Dies liefert einen Algorithmus/eine Turing-Maschine, die diese Probleme exakt lösen kann:
  - Erzeuge einen neuen Lösungsvorschlag Z (in polynomialer Zeit möglich).
  - Teste, ob Z eine Lösung für TSP (bzw. HC) ist.
  - Wiederhole, bis alle Lösungvorschläge abgearbeitet wurden oder eine Lösung gefunden wurde.
- Obwohl der Test effizient ist, haben wir das Problem, dass in der Regel exponentiell viele Lösungsmöglichkeiten getestet werden müssen.
- Wäre es nicht toll, wenn wir eine Turing-Maschine hätten, die uns alle Möglichkeiten gleichzeitig überprüfen ließe? Oder ein Orakel, dass uns sagt, welche Möglichkeit wir überprüfen müssen?

Nicht-deterministische Turing-Maschinen

Bei einer nicht-deterministische Turing-Maschine NT hat jeder Übergang mehrere mögliche Ausgänge,

$$(Q, z) \mapsto \begin{cases} (Q_1, z_1) \\ (Q_2, z_2) \\ \vdots \\ (Q_k, z_k) \end{cases}$$

Nicht-deterministische Turing-Maschinen

Bei einer nicht-deterministische Turing-Maschine NT hat jeder Übergang mehrere mögliche Ausgänge,

$$(Q, z) \mapsto \begin{cases} (Q_1, z_1) \\ (Q_2, z_2) \\ \vdots \\ (Q_k, z_k) \end{cases}$$

NT entscheidet ein Prädikat p, wenn wir für eine Eingabe (Bandinschrift) x aus allen möglichen Übergängen endlich viele auswählen können, so dass wir in einen Endzustand Q_{wahr} gelangen falls $x \in M_p = \{x \mid p(x) = \text{wahr}\}$, und in einem Endzustand Q_{falsch} gelangen, falls $x \notin M_p$.

Nicht-deterministische Turing-Maschinen

Bei einer nicht-deterministische Turing-Maschine NT hat jeder Übergang mehrere mögliche Ausgänge,

$$(Q, z) \mapsto \begin{cases} (Q_1, z_1) \\ (Q_2, z_2) \\ \vdots \\ (Q_k, z_k) \end{cases}$$

NT entscheidet ein Prädikat p, wenn wir für eine Eingabe (Bandinschrift) x aus allen möglichen Übergängen endlich viele auswählen können, so dass wir in einen Endzustand Q_{wahr} gelangen falls $x \in M_p = \{x \mid p(x) = \text{wahr}\}$, und in einem Endzustand Q_{falsch} gelangen, falls $x \notin M_p$.

Dies kann man sich so vorstellen,

- als könnte man in jedem Schritt alle möglichen Übergänge gleichzeitig ausführen, um auf irgendeinem Wege zu einem Endzustand zu gelangen,
- oder als hätte man ein Orakel, das einem bei jedem Übergang den richtigen Ausgang vorhersagt.

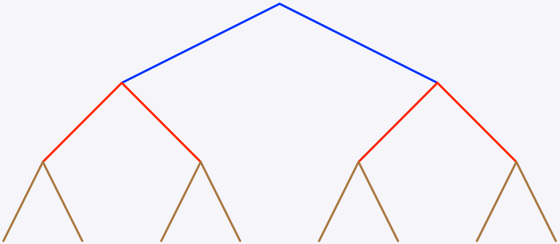
Gegeben sei ein Problem, bei dem es exponentiell viele Verzweigungen beim Finden einer Lösung geben kann.

Schematisch:

Gegeben sei ein Problem, bei dem es exponentiell viele Verzweigungen beim Finden einer Lösung geben kann.

Schematisch:

deterministisch



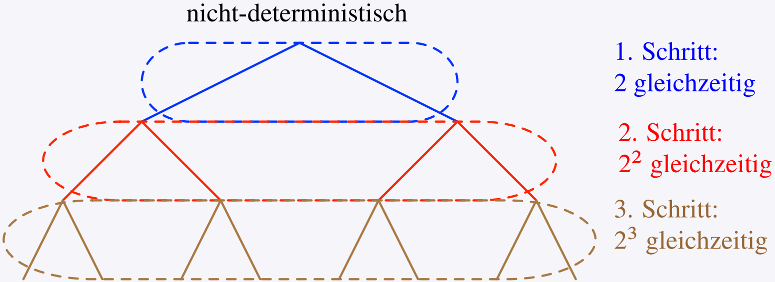
1. Schritt: 2

2. Schritt: 2^2

3. Schritt: 2^3

Gegeben sei ein Problem, bei dem es exponentiell viele Verzweigungen beim Finden einer Lösung geben kann.

Schematisch:



Gegeben sei ein Problem, bei dem es exponentiell viele Verzweigungen beim Finden einer Lösung geben kann.

Schematisch:

nicht-deterministisch

1. Schritt: 1

2. Schritt: 2

3. Schritt: 3

NP

Die Klasse **NP** der nicht-deterministisch polynomialen Entscheidungsprobleme besteht aus allen (entscheidbaren) Prädikaten, die von einer nicht-deterministischen Turing-Maschine in polynomialer Zeit entschieden werden können.

Offensichtlich ist

$$\mathbf{P} \subseteq \mathbf{NP},$$

da eine deterministische Turing-Maschine als Spezialfall einer nicht-deterministischen Turing-Maschine aufgefasst werden kann.

Offensichtlich ist

$$\mathbf{P} \subseteq \mathbf{NP},$$

da eine deterministische Turing-Maschine als Spezialfall einer nicht-deterministischen Turing-Maschine aufgefasst werden kann.

Satz

Die Probleme TSP und HC liegen in **NP**.

Offensichtlich ist

$$\mathbf{P} \subseteq \mathbf{NP},$$

da eine deterministische Turing-Maschine als Spezialfall einer nicht-deterministischen Turing-Maschine aufgefasst werden kann.

Satz

Die Probleme TSP und HC liegen in **NP**.

Heuristische Begründung (anstelle eines Beweises):

- Wir haben gesehen, dass das Überprüfen eines Lösungsvorschlags für TSP nur polynomialen Aufwand erfordert.

Offensichtlich ist

$$\mathbf{P} \subseteq \mathbf{NP},$$

da eine deterministische Turing-Maschine als Spezialfall einer nicht-deterministischen Turing-Maschine aufgefasst werden kann.

Satz

Die Probleme TSP und HC liegen in **NP**.

Heuristische Begründung (anstelle eines Beweises):

- Wir haben gesehen, dass das Überprüfen eines Lösungsvorschlags für TSP nur polynomialen Aufwand erfordert.
- Wir können also eine Lösung finden, indem wir alle möglichen Pfade aufzählen und überprüfen, ob sie Lösungen sind.

Offensichtlich ist

$$\mathbf{P} \subseteq \mathbf{NP},$$

da eine deterministische Turing-Maschine als Spezialfall einer nicht-deterministischen Turing-Maschine aufgefasst werden kann.

Satz

Die Probleme TSP und HC liegen in **NP**.

Heuristische Begründung (anstelle eines Beweises):

- Wir haben gesehen, dass das Überprüfen eines Lösungsvorschlags für TSP nur polynomialen Aufwand erfordert.
- Wir können also eine Lösung finden, indem wir alle möglichen Pfade aufzählen und überprüfen, ob sie Lösungen sind.
- Das Problem ist, dass wir dazu exponentiell viele Möglichkeiten durchgehen müssen.

Offensichtlich ist

$$\mathbf{P} \subseteq \mathbf{NP},$$

da eine deterministische Turing-Maschine als Spezialfall einer nicht-deterministischen Turing-Maschine aufgefasst werden kann.

Satz

Die Probleme TSP und HC liegen in **NP**.

Heuristische Begründung (anstelle eines Beweises):

- Wir haben gesehen, dass das Überprüfen eines Lösungsvorschlags für TSP nur polynomialen Aufwand erfordert.
- Wir können also eine Lösung finden, indem wir alle möglichen Pfade aufzählen und überprüfen, ob sie Lösungen sind.
- Das Problem ist, dass wir dazu exponentiell viele Möglichkeiten durchgehen müssen.
- Auf einer nicht-deterministischen Turing-Maschine könnten wir jedoch soviele Lösungsvorschläge gleichzeitig abarbeiten, dass wir dafür nur polynomialen Aufwand brauchen.

Offensichtlich ist

$$\mathbf{P} \subseteq \mathbf{NP},$$

da eine deterministische Turing-Maschine als Spezialfall einer nicht-deterministischen Turing-Maschine aufgefasst werden kann.

Satz

*Die Probleme TSP und HC liegen in **NP**.*

Heuristische Begründung (anstelle eines Beweises):

- Wir haben gesehen, dass das Überprüfen eines Lösungsvorschlags für TSP nur polynomialen Aufwand erfordert.
- Wir können also eine Lösung finden, indem wir alle möglichen Pfade aufzählen und überprüfen, ob sie Lösungen sind.
- Das Problem ist, dass wir dazu exponentiell viele Möglichkeiten durchgehen müssen.
- Auf einer nicht-deterministischen Turing-Maschine könnten wir jedoch soviele Lösungsvorschläge gleichzeitig abarbeiten, dass wir dafür nur polynomialen Aufwand brauchen.
- Da HC $\preceq_p$ TSP, gilt dies auch für HC. $\square$

Dass Lösungsvorschläge in polynomialer Zeit geprüft werden können, ist keine zufällige Eigenschaft von HC und TSP, sondern charakteristisch für **NP**:

Satz

Für ein Problem $\mathcal{Q}$ sind äquivalent:

1. $\mathcal{Q} \in \mathbf{NP}$.
2. *Für eine gegebene Probleminstanz q von $\mathcal{Q}$ und einen Lösungsvorschlag x dafür kann von einer deterministischen Turing-Maschine in polynomialer Zeit geprüft werden, ob x eine Lösung von q ist (etwas lax formuliert).*

Dass Lösungsvorschläge in polynomialer Zeit geprüft werden können, ist keine zufällige Eigenschaft von HC und TSP, sondern charakteristisch für **NP**:

Satz

Für ein Problem $\mathcal{Q}$ sind äquivalent:

1. $\mathcal{Q} \in \mathbf{NP}$.
2. *Für eine gegebene Probleminstanz q von $\mathcal{Q}$ und einen Lösungsvorschlag x dafür kann von einer deterministischen Turing-Maschine in polynomialer Zeit geprüft werden, ob x eine Lösung von q ist (etwas lax formuliert).*

Wir sind daran interessiert, ob HC oder TSP (oder andere Probleme in **NP**) in polynomialer Zeit exakt gelöst werden können.

Dass Lösungsvorschläge in polynomialer Zeit geprüft werden können, ist keine zufällige Eigenschaft von HC und TSP, sondern charakteristisch für **NP**:

Satz

Für ein Problem $\mathcal{Q}$ sind äquivalent:

1. $\mathcal{Q} \in \mathbf{NP}$.
2. *Für eine gegebene Probleminstanz q von $\mathcal{Q}$ und einen Lösungsvorschlag x dafür kann von einer deterministischen Turing-Maschine in polynomialer Zeit geprüft werden, ob x eine Lösung von q ist (etwas lax formuliert).*

Wir sind daran interessiert, ob HC oder TSP (oder andere Probleme in **NP**) in polynomialer Zeit exakt gelöst werden können.

Dies wäre der Fall, wenn $\mathbf{P} = \mathbf{NP}$ gilt. Wie kann man sich dieser Frage nähern?

Nicht-Determinismus simulieren

Idee

Nicht-Determinismus simulieren

Idee

- Simuliere eine gegebene nicht-deterministische Turing-Maschine NT durch eine deterministische Turing-Maschine T.

Idee

- Simuliere eine gegebene nicht-deterministische Turing-Maschine NT durch eine deterministische Turing-Maschine T.
- Die mehrfachen Übergänge von NT bilden einen Suchbaum, der von T mit Breitensuche durchsucht werden soll. Dies simuliert die gleichzeitigen Zustandsübergänge.

Idee

- Simuliere eine gegebene nicht-deterministische Turing-Maschine NT durch eine deterministische Turing-Maschine T.
- Die mehrfachen Übergänge von NT bilden einen Suchbaum, der von T mit Breitensuche durchsucht werden soll. Dies simuliert die gleichzeitigen Zustandsübergänge.

Freude! Wir können NT durch T simulieren und somit gilt $\mathbf{NP} = \mathbf{P}$!!!
All unsere Probleme sind gelöst!

Nicht-Determinismus simulieren

Idee

- Simuliere eine gegebene nicht-deterministische Turing-Maschine NT durch eine deterministische Turing-Maschine T.
- Die mehrfachen Übergänge von NT bilden einen Suchbaum, der von T mit Breitensuche durchsucht werden soll. Dies simuliert die gleichzeitigen Zustandsübergänge.

Freude! Wir können NT durch T simulieren und somit gilt $\mathbf{NP} = \mathbf{P}$!!!
All unsere Probleme sind gelöst!

Halt, nicht so schnell…

Nicht-Determinismus simulieren

Idee

- Simuliere eine gegebene nicht-deterministische Turing-Maschine NT durch eine deterministische Turing-Maschine T.
- Die mehrfachen Übergänge von NT bilden einen Suchbaum, der von T mit Breitensuche durchsucht werden soll. Dies simuliert die gleichzeitigen Zustandsübergänge.

Freude! Wir können NT durch T simulieren und somit gilt $\mathbf{NP} = \mathbf{P}$!!!
All unsere Probleme sind gelöst!

Halt, nicht so schnell…(haha)

Nicht-Determinismus simulieren

Idee

- Simuliere eine gegebene nicht-deterministische Turing-Maschine NT durch eine deterministische Turing-Maschine T.
- Die mehrfachen Übergänge von NT bilden einen Suchbaum, der von T mit Breitensuche durchsucht werden soll. Dies simuliert die gleichzeitigen Zustandsübergänge.

Freude! Wir können NT durch T simulieren und somit gilt $\mathbf{NP} = \mathbf{P}$!!!
All unsere Probleme sind gelöst!

Halt, nicht so schnell... (haha)

Satz

Eine nicht-deterministische Turing-Maschine NT *mit Rechenaufwand* $t(n)$ *im ungünstigsten Fall kann durch eine deterministische Turing-Maschine* T *mit Rechenaufwand*

$$\mathrm{O}(t(n)^2 c^{t(n)})$$

im ungünstigsten Fall simuliert werden. Hier ist $c \approx |\{Q\}| \cdot |\mathcal{B}|$.

Nicht-Determinismus simulieren

Idee

- Simuliere eine gegebene nicht-deterministische Turing-Maschine NT durch eine deterministische Turing-Maschine T.
- Die mehrfachen Übergänge von NT bilden einen Suchbaum, der von T mit Breitensuche durchsucht werden soll. Dies simuliert die gleichzeitigen Zustandsübergänge.

Freude! Wir können NT durch T simulieren und somit gilt $\mathbf{NP} = \mathbf{P}$!!!
All unsere Probleme sind gelöst!

Halt, nicht so schnell…(haha)

Satz

Eine nicht-deterministische Turing-Maschine NT *mit Rechenaufwand* $t(n)$ *im ungünstigsten Fall kann durch eine deterministische Turing-Maschine* T *mit Rechenaufwand*

$$\mathrm{O}(t(n)^2 c^{t(n)})$$

im ungünstigsten Fall simuliert werden. Hier ist $c \approx |\{Q\}| \cdot |\mathcal{B}|$.

Simulieren von Nicht-Determinismus führt also wieder auf exponentiellen Aufwand.

Idee

1. Finde ein einziges Problem $\mathscr{P}$, auf das wir alle Probleme $\mathcal{Q}$ aus **NP** effizient reduzieren können.

Idee

1. Finde ein einziges Problem $\mathscr{P}$, auf das wir alle Probleme $\mathscr{Q}$ aus **NP** effizient reduzieren können.
2. Zeige, dass $\mathscr{P} \in \mathbf{P}$ gilt.

Idee

1. Finde ein einziges Problem $\mathscr{P}$, auf das wir alle Probleme $\mathscr{Q}$ aus **NP** effizient reduzieren können.
2. Zeige, dass $\mathscr{P} \in \mathbf{P}$ gilt.
3. Da $\mathscr{Q} \preceq_{\mathrm{p}} \mathscr{P}$ für alle $\mathscr{Q}$ in **NP**, würde nun $\mathbf{P} = \mathbf{NP}$ folgen.

Idee

1. Finde ein einziges Problem $\mathscr{P}$, auf das wir alle Probleme $\mathscr{Q}$ aus **NP** effizient reduzieren können.
2. Zeige, dass $\mathscr{P} \in \mathbf{P}$ gilt.
3. Da $\mathscr{Q} \preceq_{\mathrm{p}} \mathscr{P}$ für alle $\mathscr{Q}$ in **NP**, würde nun $\mathbf{P} = \mathbf{NP}$ folgen.

Dies motiviert die folgenden Definitionen:

- Ein Problem $\mathscr{P}$ heißt **NP**-schwer (oft auch **NP**-hart, nach ungeschickter Rückübersetzung aus dem Englischen), wenn

  $$\mathscr{Q} \preceq_{\mathrm{p}} \mathscr{P}$$

  für alle Probleme $\mathscr{Q}$ in **NP**.
  Die Klasse der **NP**-schweren Probleme nennen wir **NPH**.

Idee

1. Finde ein einziges Problem $\mathscr{P}$, auf das wir alle Probleme $\mathscr{Q}$ aus **NP** effizient reduzieren können.
2. Zeige, dass $\mathscr{P} \in \mathbf{P}$ gilt.
3. Da $\mathscr{Q} \preceq_\mathrm{p} \mathscr{P}$ für alle $\mathscr{Q}$ in **NP**, würde nun $\mathbf{P} = \mathbf{NP}$ folgen.

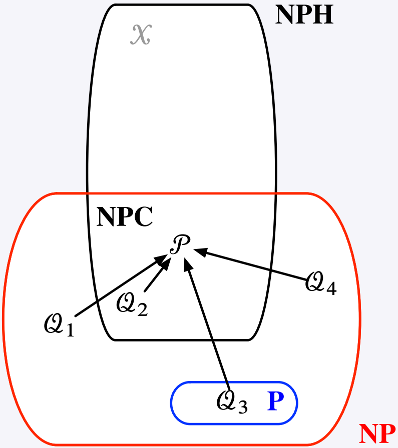
Dies motiviert die folgenden Definitionen:

- Ein Problem $\mathscr{P}$ heißt **NP**-schwer (oft auch **NP**-hart, nach ungeschickter Rückübersetzung aus dem Englischen), wenn

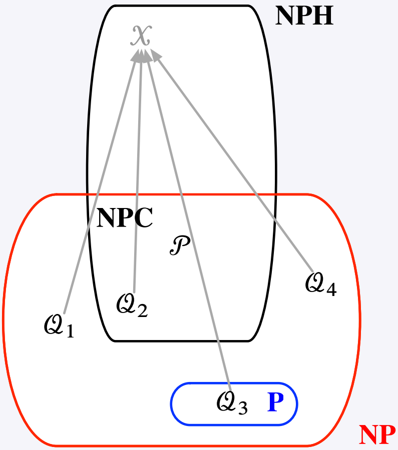
  $$\mathscr{Q} \preceq_\mathrm{p} \mathscr{P}$$

  für alle Probleme $\mathscr{Q}$ in **NP**.
  Die Klasse der **NP**-schweren Probleme nennen wir **NPH**.
- Ein Problem $\mathscr{P}$ heißt **NP**-vollständig, wenn es **NP**-schwer ist und selbst in **NP** liegt. Die Klasse der **NP**-vollständigen Probleme nennen wir **NPC**, also $\mathbf{NPC} = \mathbf{NP} \cap \mathbf{NPH}$.

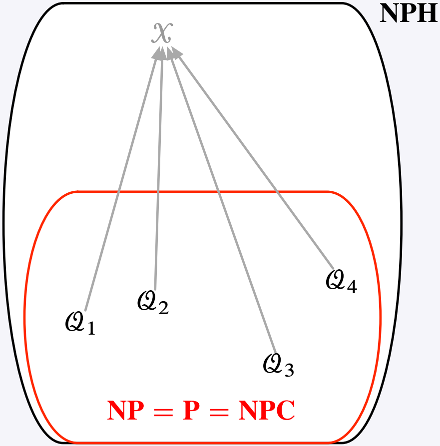
Veranschaulicht ($\mathcal{Q} \longrightarrow \mathcal{P}$ bedeutet $\mathcal{Q} \preceq_p \mathcal{P}$):



Veranschaulicht ($\mathcal{Q} \longrightarrow \mathcal{P}$ bedeutet $\mathcal{Q} \preceq_{\mathrm{p}} \mathcal{P}$):



Veranschaulicht ($\mathcal{Q} \longrightarrow \mathcal{P}$ bedeutet $\mathcal{Q} \preceq_{\mathrm{p}} \mathcal{P}$):



Satz

HC und TSP sind **NP***-vollständig*.

Satz

HC und TSP sind **NP**-*vollständig.*

Es sind derzeit über 1000 weitere **NP**-vollständige Probleme bekannt, beispielsweise

- SAT, das Problem der Erfüllbarkeit logischer Formeln in konjunktiver Normalform.
- 3SAT, wie das SAT-Problem, aber mit nur drei Variablen in jeder disjunktiven Klausel.
- Das Cliquenproblem, in einem ungerichteten Graphen einen vollständigen Teilgraphen gegebener Größe zu finden.
- Das Rucksackproblem, einen Rucksack mit begrenzter Kapazität möglichst effizient mit Gegenständen vorgegebenen Gewichts zu füllen.

Satz
*HC und TSP sind **NP**-vollständig.*

Es sind derzeit über 1000 weitere **NP**-vollständige Probleme bekannt, beispielsweise

- SAT, das Problem der Erfüllbarkeit logischer Formeln in konjunktiver Normalform.
- 3SAT, wie das SAT-Problem, aber mit nur drei Variablen in jeder disjunktiven Klausel.
- Das Cliquenproblem, in einem ungerichteten Graphen einen vollständigen Teilgraphen gegebener Größe zu finden.
- Das Rucksackproblem, einen Rucksack mit begrenzter Kapazität möglichst effizient mit Gegenständen vorgegebenen Gewichts zu füllen.

Für keines dieser Probleme ist bekannt, ob es durch einen Algorithmus mit polynomialer Laufzeit lösbar ist.

Satz

*HC und TSP sind **NP**-vollständig.*

Es sind derzeit über 1000 weitere **NP**-vollständige Probleme bekannt, beispielsweise

- SAT, das Problem der Erfüllbarkeit logischer Formeln in konjunktiver Normalform.
- 3SAT, wie das SAT-Problem, aber mit nur drei Variablen in jeder disjunktiven Klausel.
- Das Cliquenproblem, in einem ungerichteten Graphen einen vollständigen Teilgraphen gegebener Größe zu finden.
- Das Rucksackproblem, einen Rucksack mit begrenzter Kapazität möglichst effizient mit Gegenständen vorgegebenen Gewichts zu füllen.

Für keines dieser Probleme ist bekannt, ob es durch einen Algorithmus mit polynomialer Laufzeit lösbar ist.

$$\mathbf{P} \overset{?}{\neq} \mathbf{NP}.$$

Gibt es überhaupt **NP**-Probleme, von denen man nicht weiß, ob sie **NP**-vollständig sind?

Gibt es überhaupt **NP**-Probleme, von denen man nicht weiß, ob sie **NP**-vollständig sind?

Ja! Das in der Kryptographie sehr wichtige Faktorisierungsproblem (Zerlegung einer ganzen Zahl in ihre Primfaktoren). Dieses Problem liegt in **NP**, aber bisher konnte nicht bewiesen oder widerlegt werden, dass es **NP**-vollständig ist.

Gibt es überhaupt **NP**-Probleme, von denen man nicht weiß, ob sie **NP**-vollständig sind?

Ja! Das in der Kryptographie sehr wichtige Faktorisierungsproblem (Zerlegung einer ganzen Zahl in ihre Primfaktoren). Dieses Problem liegt in **NP**, aber bisher konnte nicht bewiesen oder widerlegt werden, dass es **NP**-vollständig ist.

Der beste bekannte Algorithmus zur Primfaktorzerlegung einer n-Bit Zahl hat die subexponentielle Laufzeit

$$\mathrm{O}\left(\exp\left(\sqrt[3]{\frac{64}{9} n \log(n)^2}\right)\right).$$

Gibt es überhaupt **NP**-Probleme, von denen man nicht weiß, ob sie **NP**-vollständig sind?

Ja! Das in der Kryptographie sehr wichtige Faktorisierungsproblem (Zerlegung einer ganzen Zahl in ihre Primfaktoren). Dieses Problem liegt in **NP**, aber bisher konnte nicht bewiesen oder widerlegt werden, dass es **NP**-vollständig ist.

Der beste bekannte Algorithmus zur Primfaktorzerlegung einer n-Bit Zahl hat die subexponentielle Laufzeit

$$O\left(\exp\left(\sqrt[3]{\frac{64}{9}n\log(n)^2}\right)\right).$$

Mit einem Quantencomputer könnte dieses Problem in polynomialer Zeit gelöst werden (Algorithmus von Shor).

Nachlesen

- *T.H. Cormen, C.E. Leiserson, R. Rivest, C. Stein*,
  Algorithmen – Eine Einführung,
  Abschnitte 34.2, 34.5.4, 35.2
- *G. Goos*,
  Vorlesungen über Informatik – Band 3,
  Abschnitte 13.3, 14


That's all Folks!