

Grundbegriffe der Theoretischen Informatik

Sommersemester 2018 - Thomas Schwentick

Teil D: Komplexitätstheorie

17: Polynomielle Zeit

Version von: 21. Juni 2018 (14:11)

Inhalt

▷ **17.1 Zwei algorithmische Probleme**

17.2 Berechnungsaufwand und Komplexitätstheorie

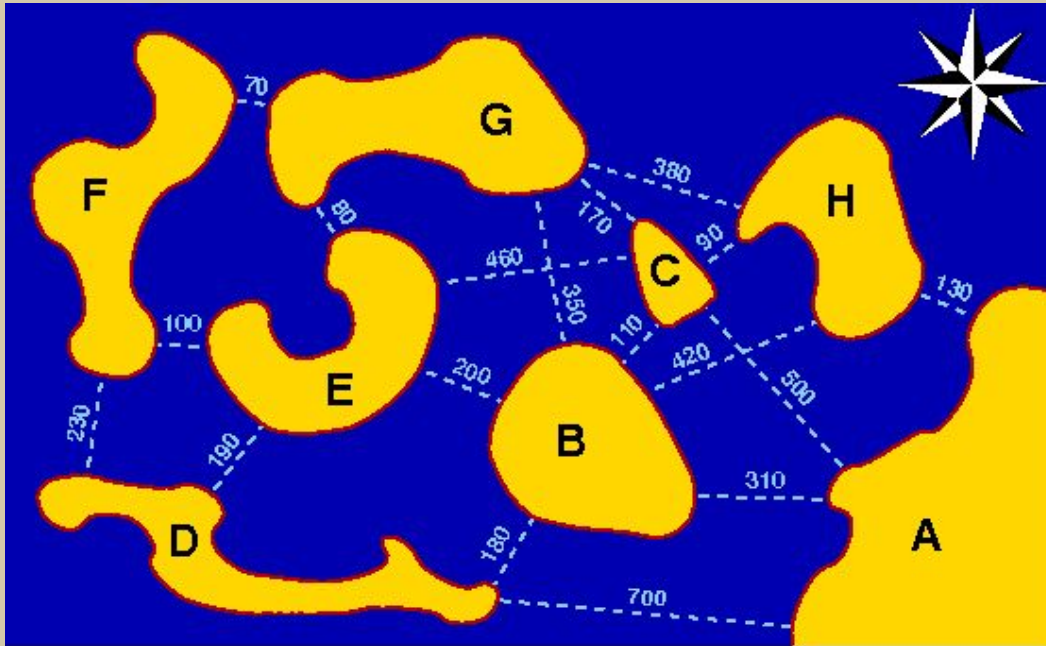
17.3 Laufzeit und erweiterte Church-Turing-These

17.4 Effizient lösbare Entscheidungsprobleme


17.5 Optimierungs- vs. Entscheidungsprobleme

Sparsamer Brückenbau

Beispiel



- Einst lebten in einem fernen Inselreich die Algolaner
- Sie wohnten verstreut auf allen sieben Inseln
- Zwischen den sieben Inseln und dem Festland verkehrten mehrere Fähren, die gegenseitige Besuche und Ausflüge auf das Festland ermöglichten
- Die Fährverbindungen sind in der Karte gestrichelt eingezeichnet

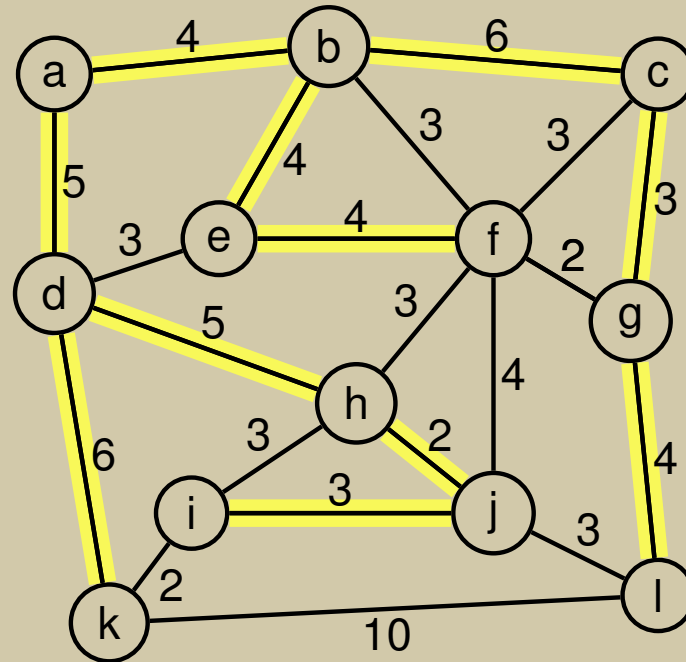
 Die Zahlen geben die Länge der Fährverbindungen in Metern an

Beispiel (Forts.)

- Bei stürmischem Wetter kam es regelmäßig vor, dass eine Fähre kenterte
 - Deshalb beschlossen die Algolaner, einige Fährverbindungen durch Brücken zu ersetzen
 - Natürlich sollte der Bauaufwand dafür möglichst gering sein
-
- Das Beispiel führt uns zu dem Problem der *Minimalen Spannbäume*
 - Das Beispiel stammt von Katharina Langkau und Martin Skutella (TU Berlin)
 - Die Quelle zu diesem Beispiel und viel mehr über Algorithmen finden Sie unter „*Algorithmus der Woche*“

Minimale Spannbäume

Beispiel: ein Spannbaum



Gewicht: 46

Definition (MINSPANNINGTREEO)

Gegeben: Graph $G = (V, E)$

👉 ungerichtet, zusammenhängend

Gewichtsfunktion $\ell : E \rightarrow \mathbb{N}$

Gesucht: Aufspannender Baum $T \subseteq E$ von G
mit minimalem Gesamtgewicht $\sum_{e \in T} \ell(e)$

Minimale Spannbäume: Algorithmus

Algorithmus von Prim

Eingabe: Graph $G = (V, E)$,
Gewichtsfunktion ℓ

Ausgabe: Spannbaum (V, T) , $T \subseteq E$,
minimalen Gewichts

$r :=$ beliebiger Knoten aus V

$R := \{r\}; Q := V - \{r\}$

$T := \emptyset$

WHILE $Q \neq \emptyset$ DO

$(u, v) :=$ Kante minimalen Gewichts
 mit $u \in R, v \in Q$

$T := T \cup \{(u, v)\}$

$R := R \cup \{v\}$

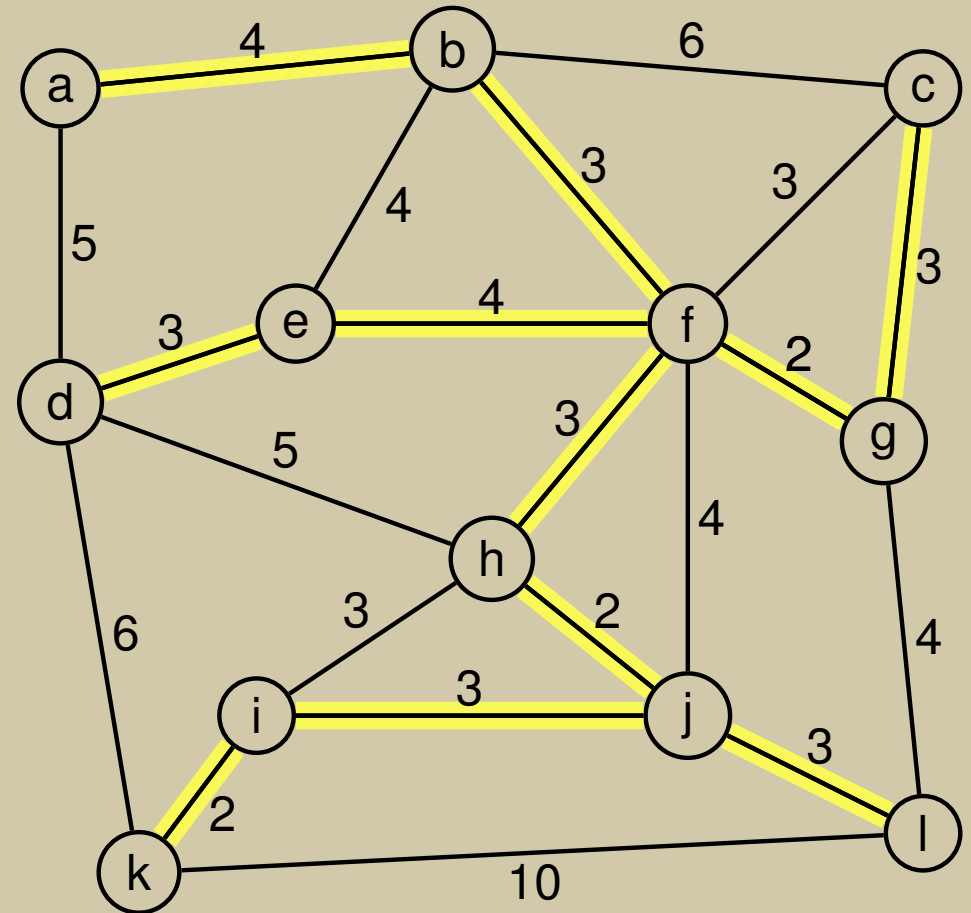
$Q := Q - \{v\}$

END

Ausgabe T

- Aufwand, bei geschickter Implementierung:
 $\mathcal{O}(|V| \log(|V|) + |E|)$ Schritte

Beispiel



Gewicht: 32

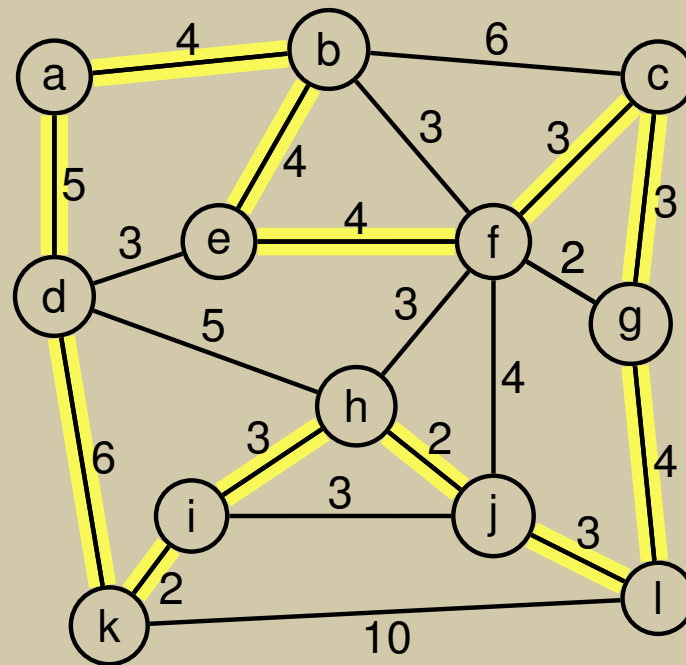
Komfortabler Brückenbau

Beispiel

- Nach reiflicher Überlegung beschloss das Oberhaupt des Inselstaates, für den Brückenbau ein anderes Optimierungskriterium zu verwenden:
 - Die Brücken sollten so konstruiert werden, dass er auf seiner wöchentlichen Rundreise durch sein Reich einen möglichst kurzen Brückensamtweg zurücklegen muss

Minimale Kreise

Beispiel



Gesamtstrecke: 41

Definition (MINCYCLEO)

Gegeben: Graph $G = (V, E)$

☞ ungerichtet, zusammenhängend

Entfernungsfunktion $\ell : E \rightarrow \mathbb{N}$

Gesucht: Kreis $K \subseteq E$ durch alle Knoten mit minimalem Gesamtgewicht $\sum_{e \in K} \ell(e)$ (oder \perp)

Minimale Kreise: Algorithmus


- Algorithmus für MINCYCLEO:
 - Zähle alle Folgen v_{i_1}, \dots, v_{i_n} von Knoten von G auf, die jeden Knoten genau einmal enthalten
 - Teste jeweils, ob $(v_{i_1}, v_{i_2}) \dots, (v_{i_n}, v_{i_1})$ ein Kreis ist
 - Wähle den Kreis mit minimalem Kantengewicht aus

- Aufwand im schlimmsten Fall:
mindestens $n! \sim \left(\frac{n}{e}\right)^n \sqrt{2\pi n}$

- Ein wesentlich besserer Algorithmus ist nicht bekannt

- MINCYCLEO ist eine Variante des Problems des Handlungsreisenden („Traveling Salesperson“):
 - Ein Handlungsreisender soll eine gegebene Menge von Städten auf einer möglichst kurzen Rundreise besuchen
 - Viele Anwendungen:
 - * Transport- und Logistikprobleme (Paketdienst, Schulbus, Versand,...)
 - * Maschinensteuerung

Effizient lösbare algorithmische Probleme: Vorbemerkungen

- In Teil C der Vorlesung ging es um die Frage, welche algorithmischen Probleme überhaupt mit Computern gelöst werden können
 - Dabei haben wir uns dann vor allem mit Problemen beschäftigt, die **nicht** lösbar sind
- Wenn ein Problem wirklich mit Computern gelöst werden soll, genügt es aber nicht, dass es prinzipiell lösbar ist
- Es sollte auch einigermaßen „effizient“ lösbar sein
 -  Wenn das Ergebnis einer Berechnung erst nach einigen Menschengenerationen vorliegt, könnte es sein, dass die Frage schon in Vergessenheit geraten ist: **42**
- In Teil D der Vorlesung geht es um die Grenze zwischen algorithmischen Problemen, die *effizient* mit Computern gelöst werden können, und solchen, die (scheinbar) **nicht** effizient lösbar sind
- In diesem Kapitel beschäftigen wir uns mit folgenden Fragen
- Wie wird der Berechnungsaufwand eines Algorithmus bzw. eines algorithmischen Problems definiert?
 - Welche Rolle spielt dabei die Kodierung der Eingabe?
 - Welche Rolle spielt das zugrunde gelegte Berechnungsmodell?
- Welche Arten von Ressourcenbeschränkungen von Berechnungen werden (üblicherweise) betrachtet?
- Wann wird ein algorithmisches Problem als effizient lösbar betrachtet?

Inhalt

17.1 Zwei algorithmische Probleme

▷ **17.2 Berechnungsaufwand und Komplexitätstheorie**

17.3 Laufzeit und erweiterte Church-Turing-These



17.4 Effizient lösbare Entscheidungsprobleme

17.5 Optimierungs- vs. Entscheidungsprobleme

Ressourcen

- Der Aufwand einer Berechnung lässt sich hinsichtlich verschiedener Ressourcen messen:
 - Laufzeit
 - Benötigter Speicherplatz
 - Energieverbrauch
 - Anzahl Prozessoren
 - ...
- Die mit Abstand am meisten betrachteten Ressourcen sind dabei Laufzeit und Speicherplatz
- Speicherplatz kann eine wesentlich kritischere Resource sein als Laufzeit, da er nicht alleine durch Geduld vergrößert werden kann
- In dieser Vorlesung werden wir uns aber fast ausschließlich mit Laufzeit beschäftigen

Laufzeit: asymptotischer Worst-Case-Aufwand

- Wie schon aus DAP 2 bekannt ist, wird meist das *asymptotische* Laufzeitverhalten von Algorithmen betrachtet:
 - Der Aufwand wird als Funktion in der Größe der Eingabe für wachsende Eingabegrößen untersucht
- Zumeist wird der *Worst-Case*-Aufwand betrachtet:
 - Wenn ein Algorithmus eine Worst-Case-Laufzeit $\mathcal{O}(n^2)$ hat, so gibt es Konstanten c und n_0 , so dass der Algorithmus für *jede* Eingabe der Größe $n > n_0$ maximal cn^2 Schritte benötigt
- Worst-Case Aufwand bietet eine Garantie, dass der Algorithmus in jedem Fall nach einer bestimmten Schrittzahl zum Ende kommt
 abhängig von der Eingabegröße
- Eine Alternative wäre beispielsweise die Betrachtung der durchschnittlichen Laufzeit
 - Sie führt jedoch zu einer Reihe von Schwierigkeiten
 siehe später in diesem Kapitel

Komplexitätstheorie: Vorbemerkungen (1/2)

- Die Komplexitätstheorie will nicht nur die Laufzeit einzelner *Algorithmen* untersuchen
 - Sie interessiert sich vielmehr für die prinzipielle algorithmische Schwierigkeit von *algorithmischen Problemen*
 - Gibt es für ein bestimmtes Problem überhaupt einen Algorithmus mit einem bestimmten Worst-Case Aufwand?
 - Das Ziel sind dabei nicht möglichst präzise Schranken, wie z.B. $\mathcal{O}(n \log n)$ vs. $\mathcal{O}(n^2)$
 - Angestrebt wird stattdessen eine grobe Kategorisierung der Probleme nach ihrer prinzipiellen algorithmischen Schwierigkeit
 - z.B.: „effizient lösbar“ vs. „nicht effizient lösbar“
 - Die Komplexitätstheorie fasst Probleme mit ähnlichem Ressourcenverbrauch in *Komplexitätsklassen* zusammen
 - Komplexitätsklassen werden üblicherweise durch drei Komponenten beschrieben:
 - Modus der Berechnung:
 - * z.B., deterministisch, nicht-deterministisch, probabilistisch, parallel
 - Art der betrachteten Ressource:
 - * z.B.: Laufzeit, Speicherbedarf, Anzahl Zufallsbits, Prozessorenzahl
 - Wachstumsverhalten bzgl. der betrachteten Ressource:
 - * z.B.: logarithmisch, polynomiell, exponentiell
 - Bekannteste offene Frage: Ist $\mathbf{P} \neq \mathbf{NP}$?
 - D.h.: können in polynomieller Zeit mehr Probleme nichtdeterministisch (**NP**) als deterministisch (**P**) gelöst werden?
- Hauptthema in diesem Teil der Vorlesung

Komplexitätstheorie: Vorbemerkungen (2/2)

- Verhältnis zwischen verschiedenen Teilgebieten der **Algorithmentheorie**:
 - **Berechenbarkeitstheorie**: Klassifikation von Problemen nach entscheidbar und (verschiedenen Graden von) unentscheidbar
 - **Komplexitätstheorie**: Klassifikation von Problemen nach algorithmischer Schwierigkeit
 - **Effiziente Algorithmen**: Konstruktion möglichst effizienter Algorithmen

Inhalt

17.1 Zwei algorithmische Probleme

17.2 Berechnungsaufwand und Komplexitätstheorie

▷ **17.3 Laufzeit und erweiterte Church-Turing-These**

17.4 Effizient lösbare Entscheidungsprobleme

17.5 Optimierungs- vs. Entscheidungsprobleme

Laufzeit: Vorbemerkungen


- Wir beschäftigen uns jetzt mit der Laufzeit von Algorithmen und insbesondere mit den schon genannten Fragen
- Welche Rolle spielt die Kodierung der Eingabe?
- Welche Rolle spielt das zugrunde gelegte Berechnungsmodell?

Laufzeit: Definitionen (1/2)


- Wir definieren jetzt formal die Laufzeit von Algorithmen
- Es stellt sich die Frage, inwieweit die Laufzeit eines Algorithmus vom zugrunde liegenden Berechnungsmodell abhängt
- Wir betrachten zunächst die Laufzeit von Turingmaschinen und GOTO-Programmen
- Wir müssen uns jeweils überlegen, was wir als einzelnen Schritt einer Berechnung zählen wollen
- Bei der formalen Definition der Laufzeit gehen wir davon aus, dass die Eingabe kodiert als String (bei TMs) oder Zahl (bei GOTO-Programmen) vorliegt

Definition (Laufzeit)


• Turingmaschinen:

- Ist $K_0(x) \vdash_M K_1 \vdash_M K_2 \vdash \dots \vdash_M K_m$ eine Berechnung einer TM M bei Eingabe x und ist K_m Halte-Konfiguration, so definieren wir
 - * $\underline{t_M(x)} \stackrel{\text{def}}{=} m$  Laufzeit von M bei Eingabe x
- Falls keine solche Folge existiert:
 - * $\underline{t_M(x)} \stackrel{\text{def}}{=} \perp$

• GOTO-Programme:

- Analog wie bei TMs definieren wir für GOTO-Programme P :
 - * $\underline{t_P(n)} \stackrel{\text{def}}{=} m-1$  Laufzeit von P bei Eingabe n
falls $(M_1, X_1) \vdash_P \dots \vdash_P (M_m, X_m)$ mit Haltekonfiguration (M_m, X_m) und $X_1 \stackrel{\text{def}}{=} X_{\text{Init}}^n$

 \vdash_P bezeichnet die Nachfolgekonnfigurationsrelation

-  Bei LOOP-/WHILE-Programmen verzichten wir auf die (etwas kompliziertere) formale Definition des Aufwandes
- Intuitiv: je Zuweisung oder Schleifen-Test 1 Schritt

Laufzeit: Definitionen (2/2)

- Die genaue Schrittzahl einzelner Berechnungen ist für unsere zukünftigen Untersuchungen meistens nicht interessant
- Stattdessen interessieren uns *asymptotische obere Schranken* des Aufwandes bei größer werdenden Eingaben
- Deshalb definieren wir zunächst die Größe einer Eingabe

Definition (Eingabegröße)

Turingmaschinen: Die Eingabegröße ist die Länge des Eingabestrings

LOOP-/WHILE-/GOTO-Programme:

Die Eingabegröße ist die Länge der Kodierung der Eingabezahl, also $|\text{N2Str}(n)| = \lfloor \log(n + 1) \rfloor$ bei Eingabe n

Definition (Zeitschranke)

Turingmaschinen: Eine Funktion


$T : \mathbb{N} \rightarrow \mathbb{R}$ heißt Zeitschranke für eine TM M , falls es ein $n_0 \in \mathbb{N}$ gibt, so dass für alle $x \in \Sigma^*$ mit $|x| > n_0$ gilt:

$$t_M(x) \leq T(|x|)$$

LOOP-/WHILE-/GOTO-Programme: Eine

Funktion $T : \mathbb{N} \rightarrow \mathbb{R}$ heißt Zeitschranke für ein LOOP-, WHILE- oder GOTO-Programm P , falls es ein $n_0 \in \mathbb{N}$ gibt, so dass für alle $n \in \mathbb{N}$ mit $n > n_0$ gilt:

$$t_P(n) \leq T(\lfloor \log(n + 1) \rfloor)$$

 Wir gehen hier davon aus, dass GOTO- und WHILE-Programme Anweisungen der Art $x_i := x_j + x_k$ verwenden dürfen

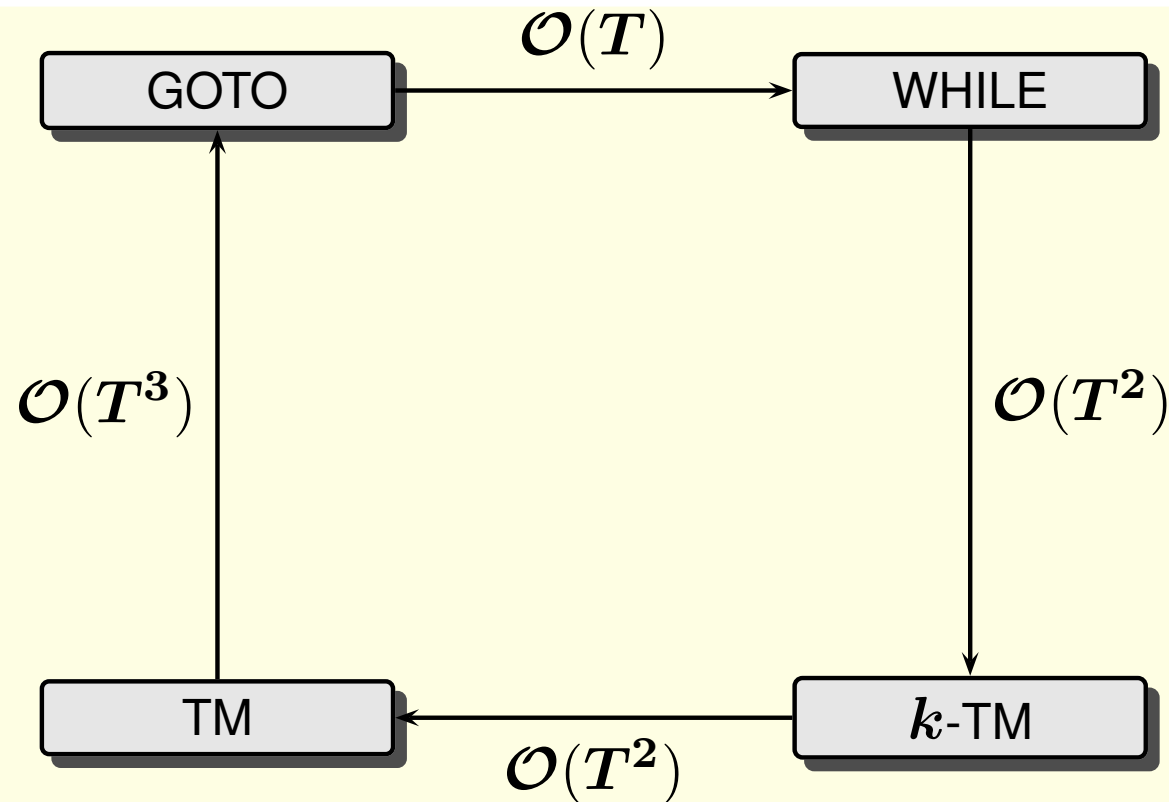
- Die „Zeitschranken-Eigenschaft“ muss nur für genügend große Eingaben gelten
- Wir betrachten hier nur TMs und Programme, die immer terminieren

Erweiterte Church-Turing-These (1/2)

Lemma 17.1

- Seien $f : \Sigma^* \rightarrow \Sigma^*$, $g : \mathbb{N} \rightarrow \mathbb{N}$ und sei $T : \mathbb{N} \rightarrow \mathbb{N}$, wobei für alle $n \in \mathbb{N}$ gilt: $T(n) \geq n$
- Dann gelten die folgenden Aussagen:
 - (a) Ist P ein GOTO-Programm, das g mit Zeitschranke T berechnet, so gibt es ein WHILE-Programm P' , das g mit Zeitschranke $\mathcal{O}(T)$ berechnet
 - (b) Ist P ein WHILE-Programm, das g mit Zeitschranke T berechnet, so gibt es eine k -String-TM M , die g im Sinne von Satz 13.4 mit Zeitschranke $\mathcal{O}(T^2)$ berechnet
 - (c) Ist M eine k -String-TM, die f mit Zeitschranke T berechnet, so gibt es eine 1-String-TM M' , die f mit Zeitschranke $\mathcal{O}(T^2)$ berechnet
 - (d) Ist M eine 1-String-TM, die f mit Zeitschranke T berechnet, so gibt es ein GOTO-Programm P , das g im Sinne von Satz 13.5 mit Zeitschranke $\mathcal{O}(T^3)$ berechnet
- Also: alle hier genannten Berechnungsmodelle sind bezüglich des Zeitaufwandes „polynomiell äquivalent“
- Beweis durch genaue Analyse der jeweiligen Simulationen

Erweiterte Church-Turing-These (2/2)



- Die Äquivalenz aus Lemma 17.1 lässt sich auf alle in Kapitel 13 genannten Berechnungsmodelle ausdehnen
- Die **erweiterte Church-Turing-These** besagt, dass sie sich auf *alle* „vernünftigen“ Berechnungsmodelle erweitern lässt:
 - Also: „vernünftige“ Berechnungsmodelle unterscheiden sich hinsichtlich ihrer Laufzeit nur um polynomielle Faktoren


Zwei Sichtweisen: Formal vs. informell

- Bei der Analyse der Komplexität algorithmischer Probleme werden wir, je nach Kontext, eine formale oder eine informelle Sichtweise einnehmen:

Formale Sichtweise:

- Algorithmische Probleme sind Sprachen oder Funktionen über Binärstrings
- Algorithmen sind Turingmaschinen
- Diese Sichtweise ist geeignet um Aussagen zu *beweisen*
 - * insbesondere für untere Schranken
- Wir werden sie nur anwenden, wenn nötig

Informelle Sichtweise:

- Algorithmische Probleme haben komplexe Eingaben  z.B.: Graphen, Automaten
- Algorithmen werden in Pseudocode oder noch informeller beschrieben
- Aufwandanalyse ist grob und „handwaving“
- Wir werden diese Sichtweise einnehmen, wann immer es möglich ist
 - * insbesondere für obere Schranken

- Bei der informellen Sichtweise stellt sich die Frage, wie die Größe der Eingabe „gemessen“ wird

- Hier werden wir recht flexibel sein
 - Bei Graphen mit n Knoten wäre die Länge der Kodierung als String beispielsweise n^2 , wir werden aber als Eingabegröße die Anzahl der Knoten (also: n) nehmen

- Wichtig ist nur, dass
 - die „formale Eingabegröße“ höchstens polynomiell größer ist als die „informelle Eingabegröße“ (im Beispiel: quadratisch)
 - und umgekehrt (in der Regel ist die „informelle Eingabegröße“ aber sowieso kleiner als die „formale Eingabegröße“)

Inhalt

17.1 Zwei algorithmische Probleme

17.2 Berechnungsaufwand und Komplexitätstheorie

17.3 Laufzeit und erweiterte Church-Turing-These

▷ **17.4 Effizient lösbare Entscheidungsprobleme**

17.5 Optimierungs- vs. Entscheidungsprobleme

Effizient lösbar Probleme

- Nach diesen Vorbereitungen können wir uns nun endlich der Frage zuwenden, wann wir ein algorithmisches Problem als effizient lösbar ansehen wollen
- Wir werfen dazu nochmals einen Blick auf das Laufzeitverhalten der beiden Algorithmen aus der Einleitung
- Dabei nehmen wir an, dass die genaue Laufzeit der Implementierungen wie folgt ist:
 - Für Prim: $\frac{1}{10.000} n^2$ Sekunden
 - Für MINCYCLEO: $\frac{1}{1.000.000.000} n!$ Sekunden

Eingabegröße	Prims Algorithmus	MINCYCLEO-Alg.
10	0,01 Sekunden	0,03 Sekunden
15	0,02 Sekunden	20 Minuten
20	0,04 Sekunden	>1000 Jahre
30	0,09 Sekunden	
40	0,16 Sekunden	
100	1 Sekunden	
1000	1,6 Minuten	


Polynomielle vs. exponentielle Laufzeit

- Der Prim-Algorithmus hat *polynomielle Laufzeit*, da er ein Polynom (cn^2) als Laufzeitschranke hat
- Der naive Algorithmus für MINCYCLEO hat hingegen *exponentielle Laufzeit*

- Der prinzipielle Unterschied zwischen polynomieller und exponentieller Laufzeit lässt sich wie folgt beschreiben

- **Wenn** Eingaben der Größe n bisher in Zeit t bearbeitet werden können,
- **dann** können mit einem doppelt so schnellen Rechner in der selben Zeit bearbeitet werden:
 - bei **polynomiell**em Aufwand:
Eingaben der Größe cn , für ein $c > 1$
 - bei **exponentiell**em Aufwand:
Eingaben der Größe $n + d$, für ein $d > 0$

- Bei Laufzeit n^2 : Eingaben der Größe $\sqrt{2}n$
- Bei Laufzeit 2^n : Eingaben der Größe $n + 1$

 Die obigen Aussagen beziehen sich auf die Laufzeit von Algorithmen, nicht auf die allgemeine Komplexität der betrachteten Probleme


Zeitbasierte Komplexitätsklassen

- Es besteht ein weitgehender Konsens darüber, dass ein Problem nur dann als effizient lösbar bezeichnet werden kann, wenn es in polynomieller Zeit lösbar ist
 - Die Frage, ob die Umkehrung auch gilt, diskutieren wir gleich
- Die erweiterte Church-Turing-These rechtfertigt nun die folgende Definition der Komplexitätsklasse **P**
 - Denn es ist egal, ob wir für die Definition von **P** Turingmaschinen oder ein anderes Berechnungsmodell zugrunde legen

Definition (**TIME**(T), **P**)

(a) Für $T : \mathbb{N} \rightarrow \mathbb{R}$ sei **TIME**(T) die Menge aller Sprachen L , für die es eine k -String TM M mit Zeitschranke T gibt, so dass $L = L(M)$

(b) $\mathbf{P} \stackrel{\text{def}}{=} \bigcup_{p \text{ Polynom}} \mathbf{TIME}(p)$

 Die Komplexitätsklasse **P** wurde übrigens erst in den 60er Jahren definiert — lange Zeit nach den ersten Untersuchungen der entscheidbaren Sprachen...

P \equiv effizient lösbar Probleme? (1/3)

- Wir gehen in den folgenden Kapiteln davon aus, dass die Komplexitätsklasse **P** eine vernünftige Formalisierung des informellen Begriffes der „effizient lösbar Probleme“ ist
- Diese Sichtweise ist sehr weit verbreitet, aber durchaus nicht unumstritten
- Einige Einwände und mögliche Er widerungen darauf betrachten wir auf den nächsten beiden Folien

P \equiv effizient lösbare Probleme? (2/3)

- **Einwand:** Polynome mit großen Exponenten haben mit „effizient“ nichts zu tun

☞ z.B.: n^{1000}

- **Aber:**

- Wenn für ein „natürliches“ Problem überhaupt ein polynomieller Algorithmus gefunden wird, gibt es (für relevante algorithmische Probleme) meistens auch einen mit kleinem Exponenten

☞ z.B.: 2 oder 3

- Außerdem ist es vorteilhaft, dass die Klasse der Polynome unter Komposition abgeschlossen ist
→ Programme und Unterprogramme

- **Einwand:** Worst-Case-Komplexität ist ungeeignet, der Durchschnittsfall wäre interessanter

- **Aber:**

- In vielen Fällen ist eine Laufzeit-Garantie wichtig
- Durchschnittskomplexität ist viel komplizierter zu handhaben:
 - * Es müsste zum Beispiel die Frage beantwortet werden, wie die Wahrscheinlichkeitsverteilung der Eingaben ist
 - * Es ist viel schwieriger die Durchschnittskomplexität eines Problems zu analysieren


P \equiv effizient lösbare Probleme? (3/3)

- **Einwand:** Entscheidungsprobleme (Sprachen) sind zu eingeschränkt

- **Aber:**

- Im nächsten Abschnitt werden wir sehen, dass sich die Frage nach der effizienten Lösbarkeit von Optimierungsproblemen auf die Frage nach der effizienten Lösbarkeit von Entscheidungsproblemen zurückführen lässt

- **Einwand:** Die Definition von **P** hängt von der Wahl des Berechnungsmodells ab

 hier: TMs

- **Aber:**

- Die erweiterte Church-Turing-These sagt, dass alle „vernünftigen“ Modelle sich nur polynomiell bzgl. Zeitaufwand unterscheiden

Nicht in polynomieller Zeit lösbare Probleme

Definition (**EXPTIME**)

$$\bullet \text{ EXPTIME} \stackrel{\text{def}}{=} \bigcup_{p \text{ Polynom}} \text{TIME}(2^p)$$

- Die Klasse **EXPTIME** ist eine echte Oberklasse von **P**, d.h., sie umfasst **P**, enthält aber auch Probleme, die sich nicht in polynomieller Zeit lösen lassen
 - Und das lässt sich auch beweisen
- Beispiel: Das Problem zu entscheiden, ob beim Brettspiel GO auf einem $n \times n$ -Brett der erste Spieler eine Gewinnstrategie hat, ist in **EXPTIME** aber nicht in **P**
 - Dies gilt für die Ko-Regel, die Stellungswiederholungen verbietet

Polynomielle Zeitschranken: n^k

- Wir nutzen die folgende **Beobachtung**, um uns den Umgang mit komplizierten Polynomen als Zeitschranken zu ersparen
- Wenn eine Turingmaschine M eine polynomielle Zeitschranke hat, dann hat sie auch eine Zeitschranke der Form n^k , für ein geeignetes k

• Denn:

– Sei $p(n) = \sum_{i=0}^{\ell} c_i n^i$ ein Polynom,

das Zeitschranke für M ist

* Es gibt also ein n_0 , so dass für alle Strings x mit $|x| > n_0$ gilt:

$$t_M(x) \leq p(|x|)$$

– Wir wählen

* $n'_0 \stackrel{\text{def}}{=} \max\{n_0, c_0, \dots, c_\ell\}$ und

* $k \stackrel{\text{def}}{=} \ell + 2$

– Dann gilt für alle $n > n'_0$:

$$p(n) \leq n'_0 \sum_{i=0}^{\ell} n^i \leq n'_0 n^{\ell+1} \leq n^{\ell+2}$$

→ Wir werden also bei Problemen aus **P** zukünftig davon ausgehen, dass sie eine Zeitschranke der Form n^k haben

Inhalt

17.1 Zwei algorithmische Probleme



17.2 Berechnungsaufwand und Komplexitätstheorie

17.3 Laufzeit und erweiterte Church-Turing-These

17.4 Effizient lösbare Entscheidungsprobleme

▷ **17.5 Optimierungs- vs. Entscheidungsprobleme**

Optimierungsprobleme vs. Entscheidungsprobleme (1/6)

- Einige algorithmische Probleme haben „ja“ oder „nein“ als Antwort,  Entscheidungsprobleme
andere suchen eine optimale Lösung
 Optimierungsprobleme
- Eine dritte Variante ist die Berechnung des Wertes einer optimalen Lösung
- Wir werden sehen: hinsichtlich polynomieller Lösbarkeit können wir uns auf Entscheidungsprobleme beschränken
- Wir betrachten die drei Varianten am Beispiel des Traveling-Salesperson-Problems

Optimierungsprobleme vs. Entscheidungsprobleme (2/6)

- Das TSP-Problem (*Traveling Salesperson*) sucht nach der kürzesten Rundreise durch eine gegebene Menge von Städten, die jede Stadt genau einmal besucht
- Formal besteht die Eingabe zum TSP-Problem aus einer Folge s_1, \dots, s_n von Städten und einer Entfernungsfunktion d
 - $d(s_i, s_j)$ ist die Entfernung von s_i nach s_j
- Wir betrachten hier nur den symmetrischen Fall: für alle i, j ist $d(s_i, s_j) = d(s_j, s_i)$
- Formal ist eine **TSP-Reise** eine bijektive Funktion $f : \{1, \dots, n\} \rightarrow \{s_1, \dots, s_n\}$
 - $f(i)$ ist die i -te besuchte Stadt
- Die **Gesamtstrecke** $d(f)$ einer solchen TSP-Reise f ist
$$d(f) \stackrel{\text{def}}{=} d(f(n), f(1)) + \sum_{i=1}^{n-1} d(f(i), f(i+1))$$

Definition (TSP)

Gegeben: Entfernungsfunktion d , Zielwert $k \in \mathbb{N}$

Frage: Gibt es eine TSP-Reise f zu d mit $d(f) \leq k$?

Definition (TSPO)


Gegeben: Entfernungsfunktion d

Gesucht: TSP-Reise f zu d mit minimaler Gesamtstrecke

Definition (TSPV)

Gegeben: Entfernungsfunktion d

Gesucht: Minimale Gesamtstrecke $d(f)$ einer TSP-Reise zu d

 Da die Entfernungsfunktion d implizit auch die Städte repräsentiert, geben wir sie nicht explizit als Eingabe der drei obigen Probleme an

Optimierungsprobleme vs. Entscheidungsprobleme (3/6)


Lemma 17.2

- (a) Falls TSP in polynomieller Zeit lösbar ist, dann auch TSPV
- (b) Falls TSPV in polynomieller Zeit lösbar ist, dann auch TSPO

Beweisskizze für (a)

- Idee: *Binäre Suche*
- Annahme: A ist ein Algorithmus für TSP mit polynomieller Laufzeit
- Sei d eine Entfernungsfunktion für TSPV mit n Städten
- Sei m der maximale in d vorkommende Funktionswert
 - ➔ Die optimale Lösung hat höchstens den Wert $N \stackrel{\text{def}}{=} nm$
- Zu beachten: die Kodierung von d als Eingabestring benötigt nur $\mathcal{O}(n^2 \log m)$ Bits

Beweisskizze (Forts.)

- Der Algorithmus arbeitet wie folgt:
 - 1: $i := 0; j := N$
 - 2: **repeat**
 - 3: $k := \left\lfloor \frac{i+j}{2} \right\rfloor$
 - 4: **if** A sagt, dass Lösung $\leq k$ existiert **then**
 - 5: $j := k$
 - 6: **else**
 - 7: $i := k + 1$
 - 8: **until** $i = j$
 - 9: Ausgabe j  oder \perp , wenn $j = N + 1$
- Korrektheit: Durch Induktion nach der Anzahl der Schleifendurchläufe ist leicht zu zeigen, dass der optimale Wert immer in $[i, j]$ liegt
- Laufzeit:
 - In jedem Durchlauf wird $j - i$ ungefähr halbiert
 - ➔ $\mathcal{O}(\log N)$ Schleifendurchläufe
 - Jeder Durchlauf benötigt nur polynomielle Zeit
 - ➔ Insgesamt polynomielle Laufzeit

Optimierungsprobleme vs. Entscheidungsprobleme (4/6)

Beweisskizze für (b)

- Sei d eine TSPO-Eingabe mit n Städten und sei m wieder der maximal vorkommende Funktionswert von d
- Für zwei Indizes k, ℓ bezeichne $d_{(k,\ell)}$ die Entfernungsfunktion definiert durch:
$$d_{(k,\ell)}(s_i, s_j) \stackrel{\text{def}}{=} \begin{cases} m + 1 & \text{falls } i = k \text{ und } j = \ell \text{ (oder umgekehrt)} \\ d(s_i, s_j) & \text{sonst} \end{cases}$$
- Die beiden folgenden Aussagen sind für jedes Paar k, ℓ mit $k \neq \ell$ äquivalent:
 - die minimale Gesamtstrecke zu $d_{(k,\ell)}$ ist gleich der minimalen Gesamtstrecke zu d
 - es gibt zu d eine minimale TSP-Reise f , die *nicht* direkt von s_k nach s_ℓ (oder umgekehrt) geht

Optimierungsprobleme vs. Entscheidungsprobleme (5/6)

Beweisskizze (Forts.)


- Algorithmus:
 - 1: $m :=$ maximaler Funktionswert von d
 - 2: $d' := d$
 - 3: **for** jedes Index-Paar $k \neq \ell$ **do**
 - 4: **if** optimaler Wert für $d'_{(k,\ell)} =$
 optimaler Wert für d **then**
 - 5: $d' := d'_{(k,\ell)}$
- Behauptung: nach Ende der Berechnung induzieren die Paare s_i, s_j mit $d'(s_i, s_j) \leq m$ eine TSP-Reise zu d mit minimaler Gesamtentfernung
- Dazu lässt sich durch Induktion nach der Anzahl der Schleifendurchläufe zeigen
 - Der Wert der optimalen Lösung für d' ändert sich nicht
- Und: am Ende ist nur noch eine TSP-Reise übrig

Optimierungsprobleme vs. Entscheidungsprobleme (6/6)

- Das TSP-Optimierungsproblem kann also (in polynomieller Zeit) auf das TSP-Entscheidungsproblem zurückgeführt (reduziert) werden

- Eine Lemma 17.2 entsprechende Aussage gilt für die meisten uns interessierenden Optimierungsprobleme:

- (a) funktioniert immer wie bei TSP

 Wenn es nur polynomiell viele Lösungswerte gibt, ist binäre Suche nicht nötig

- Für (b) muss jeweils ein individueller Ansatz gefunden werden

* Das ist in der Regel ähnlich wie bei TSPO möglich

→ Wir beschränken uns im Folgenden deshalb der Einfachheit halber auf Entscheidungsprobleme

- Notation: Optimierungsprobleme haben am Ende Ihres Namens ein O, die zugehörigen Entscheidungsprobleme nicht

Zusammenfassung

- Wir konzentrieren uns bei der Betrachtung effizient lösbarer algorithmischer Probleme auf die asymptotische Laufzeit im *worst case*
- Die in Teil C betrachteten Berechnungsmodelle ergeben eine robuste Definition der Klasse der in polynomieller Zeit lösbaren Entscheidungsprobleme
 - Formal basieren unsere Definitionen auf Berechnungen von Turingmaschinen
- Wir betrachten die Begriffe „effizient lösbar“ und „in polynomieller Zeit lösbar“ im Folgenden als gleichbedeutend
- Es gibt auch Probleme, die sich nicht in polynomieller sondern nur in (mindestens) exponentieller Zeit lösen lassen