Computatietheorie

Lucien Sina

12.6.2025

ISBN: 9789403843087

Gedrukt door Bookmundo

Inhoudsopgave

Voorwoord

Dit boek *Computatietheorie* is bedoeld als een compacte, doch rigoureuze inleiding in de fundamentele concepten van berekenbaarheid, formele talen en automatenleer. Het doel is u — studenten van informatica, wiskunde en aanverwante vakken, evenals alle technisch geïnteresseerden — de belangrijkste modellen, bewijstechnieken en grenzen van het berekenbare op een begrijpelijke manier na te brengen.

Centraal staan de klassieke rekenmodellen (eindige automaten, kellerautomaten (pushdownautomaten), Turingmachines), de bijbehorende taalklassen (regulier, contextvrij, beslisbaar / opsommbaar) en de kernstellingen die ons laten zien wat berekenbaar is en waar fundamentele grenzen beginnen (onder andere Myhill–Nerode, pumping-lemma, Kleene-stelling, diagonalisatie,

Rice-stelling). Naast formele definities en bewijzen vindt u talrijke voorbeelden, oefenopgaven en modeloplossingen — de nadruk ligt op bewijstechniek en intuïtie, niet op formele pedanterie.

Voor het begrijpen van dit boek zijn basiskennis van discrete wiskunde, propositieslogica en predikaatlogica evenals grondbeginselen van algoritmetheorie nuttig; deze kunt u, indien nodig, via mijn andere boeken over deze onderwerpen verwerven. Voorafgaande kennis van formele talen en automaten is handig maar niet verplicht: de behandeling is zodanig opgezet dat ook lezers die deze onderwerpen voor het eerst grondig bestuderen, stapsgewijs worden meegenomen.

De opbouw van het boek is bewust modulair: eerst behandelen we reguliere talen en automaten, daarna contextvrije grammatica's en kellerautomaten, vervolgens Turingmachines en de klassen van beslisbare respectievelijk opsommbare talen. Tenslotte onderzoeken we onvertaalbaarheid (onbeslisbaarheid), het reductieprincipe en de stelling van Rice. Elk hoofdstuk bevat motiverende voorbeelden, grafische weergaven, volledige bewijzen en oefenopgaven met modeloplossingen — hierdoor leent het boek zich zowel voor zelfstudie als als begeleidend tekstboek bij colleges.

Tijdens het schrijven vond ik het belangrijk een

balans te bewaren tussen wiskundige strengheid en een heldere, intuïtieve presentatie. Veel bewijzen zijn volledig uitgewerkt; waar technische details de leesbaarheid sterk zouden schaden, heb ik bewust teruggehouden en de intuïtie benadrukt.

Ik ontvang graag opmerkingen, correcties en suggesties — die helpen de volgende druk nog duidelijker en nuttiger te maken.

Veel lees- en bewijsplezier!

Hoofdstuk 1

Inleiding

Een *algoritme* is een precieze en eindige reeks instructies die een gegeven probleem voor willekeurige geldige invoer in een eindig aantal stappen oplost. Het kenmerkt zich door de volgende eigenschappen:

- **Eenduidigheid:** Elke stap is duidelijk gedefinieerd en eenduidig uitvoerbaar.
- **Eindigheid:** Het algoritme bestaat uit een eindig aantal instructies en beëindigt na een eindig aantal stappen.
- **Determinisme:** Bij gelijke toestand en gelijke invoer leidt dezelfde stap altijd tot dezelfde volgende actie (bij

niet-deterministische modellen wordt dit overeenkomstig uitgebreid).

- **Invoer en uitvoer:** Het neemt een goed gedefinieerde invoer aan en levert na voltooiing een goed bepaalde uitvoer.
- **Effectiviteit:** Elke stap kan mechanisch worden uitgevoerd (met pen en papier of door een geïdealiseerde machine).

Deze formele opvatting van een algoritme vormt de basis voor alle volgende beschouwingen over berekenbaarheid en de analyse van de benodigde hulpbronnen.

De algoritmiek houdt zich bezig met de vraag welke problemen überhaupt met behulp van algoritmen op te lossen zijn en hoeveel rekeninspanning daarvoor nodig is. In dit boek volgen we twee centrale leidende vragen:

1. **Berekenbaarheid:** Welke problemen zijn principieel algoritmisch oplosbaar, en welke niet?
2. **Complexiteit:** Voor die problemen die berekenbaar zijn, hoe kunnen de daarvoor benodigde hulpbronnen — in het bijzonder tijd en geheugen — worden gekwantificeerd en vergeleken?

Om deze vragen systematisch te beantwoorden, voeren we eerst formele rekenmodellen in, waarmee algoritmische processen nauwkeurig beschreven en geanalyseerd kunnen worden. We behandelen onder andere:

- Turingmachines als standaardsmodel van berekenbaarheid,
- registermachines en RAM-modellen om praktische kosten in termen van hulpbronnen te beoordelen,
- evenals de basisbegrippen van formele talen en grammatica's.

Een centraal resultaat van de berekenbaarheidstheorie is dat er problemen bestaan die zelfs met willekeurig krachtige (theoretische) rekenmodellen niet oplosbaar zijn. Een klassiek voorbeeld is het halteprobleem (stopprobleem): er bestaat geen algoritme dat voor elke willekeurige combinatie van programma en invoer beslist of het programma stopt of oneindig blijft doorgaan.

Zelfs wanneer een probleem berekenbaar is, kan de benodigde inspanning zo groot zijn dat een praktische oplossing onhaalbaar blijft. In de complexiteitstheorie classificeren we problemen

daarom naar hun moeilijkheidsgraad in complexiteitsklassen zoals P, NP, PSPACE en verder.

Samen met mijn vervolgboek over complexiteitstheorie vormt dit een tweedelige boekenreeks:

Deel I Berekenbaarheidstheorie: grondslagen, rekenmodellen en onbeslisbaarheidsbewijzen.

Deel II Complexiteitstheorie: analyse van hulpbronnen, inleiding in complexiteitsklassen en centrale onopgeloste vragen.

Het doel van dit boek is u zowel het theoretische fundament als de methodische instrumenten te bieden om zelf berekenbaarheids- en complexiteitsvraagstukken nauwkeurig te formuleren en te beantwoorden.

Hoofdstuk 2

Formele talen

2.1 Inleiding

In de informatica dienen formele talen als een krachtig en precies middel om complexe structuren, regels en feiten om te zetten in eenduidig gedefinieerde tekenreeksen. In plaats van informele beschrijvingen in natuurlijke taal bieden formele talen en de bijbehorende formaliseringen de volgende voordelen:

- **Eenduidigheid:** Elke uitspraak en elke expressie is goed gedefinieerd en vrij van ambiguïteit.
- **Abstractie:** Essentiële eigenschappen wor-

den benadrukt, onnodige details worden weggelaten.

- **Analyseerbaarheid:** Wiskundige en algorithmische procedures kunnen direct op formele beschrijvingen worden toegepast.
- **Automatisering:** Hulpmiddelen zoals parsers, compilers en bewijssystemen kunnen formele specificaties machinaal verwerken.
- **Controleerbaarheid:** Correctheidsbewijzen en complexiteitsanalyses kunnen systematisch worden uitgevoerd.

Een formele taal L over een alfabet Σ is een verzameling woorden, waarbij elk woord een eindige opeenvolging van symbolen uit Σ voorstelt. Formeel schrijven we

$$L \subseteq \Sigma^*,$$

waarin Σ^* alle mogelijke eindige combinaties van symbolen uit Σ omvat. Een formele taal bepaalt daarmee welke tekenreeksen als 'geldig' gelden en vormt de basis voor precieze definities van syntaxis, herkennings- en beoordelingsprocedures.

Met formele talen kunnen we algoritmische problemen zo op- en beschrijven dat we

1. problemen precies formuleren en vertalen naar bewerkingen op tekens en structuren,
2. de syntaxis en semantiek van programmeertalen, dataformaten en protocollen modelleren,
3. bereikbaarheids-, bevredigbaarheids- en beslissingsvragen systematisch onderzoeken.

In het vervolg van dit hoofdstuk verdelen we de inhoud in de onderstaande secties, voordat we in het volgende hoofdstuk beginnen met de gedetailleerde codering van individuele problemen:

§ **Probleemcodering** Weergave van problemen via alfabetten, unaire en binaire codering, adjacency-lijsten, adjacency-matrices en Booleaanse functies.

§ **Beslissingsproblemen** Formele definitie van ja-/nee-vragen en de talen die deze antwoorden karakteriseren.

§ **Woord- en taalkundige operaties** Onderzoek naar operaties zoals vereniging, concatenatie en Kleene-ster en hun effect op taalklassen.

§ **Rekenmodellen** Invoering van geïdealiseerde rekenaars – eindige automaten, pushdownautomaten, Turingmachines.

§ Looptijdanalyse Methoden om de tijdscomplexiteit bij de verwerking van formele talen te schatten en te vergelijken.

2.2 Probleemcodering

2.2.1 Inleiding

Om algorithmische problemen in de praktijk op te lossen, moeten we reële vraagstukken eerst omzetten in wiskundige modellen. Pas op dit abstracte niveau kunnen we algoritmen ontwikkelen en hun correctheid en efficiëntie analyseren.

Voorbeeld (navigatie in een openbaar vervoersnet). We willen de kortste reis (zonder wachttijden) tussen twee haltes in een openbaar vervoersnet berekenen. In het wiskundige model representeren we het net door een *gewogen graaf* $G = (V, E, w)$, waarbij

- V de verzameling stations (knooppunten) is,
- $E \subseteq V \times V$ de lijnen/verbindingen (kanten) zijn, en
- $w \colon E \to \mathbb{N}$ het gewicht van elke kant aangeeft als gemiddelde reistijd in minuten.

Het oorspronkelijke vraagstuk reduceert hiermee tot het klassieke kortste-padprobleem.

Maar hoe "begrijptëen geïdealiseerde rekenaar deze graaf? Op het laagste niveau kan een rekenaar slechts bits (0 en 1) verwerken. Daarom hebben we een *codering* nodig die de gewogen graaf als een eindige tekenreeks (woord) voorstelt.

1. **Keuze van het alfabet:** We gebruiken de tekenvoorraad
$$\Sigma = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, \#\}.$$
2. **Benaming van knopen:** De knopen $V = \{v_1, v_2, \ldots, v_n\}$ nummeren we van 1 tot n.
3. **Codering van kanten:** Elke kant (v_i, v_j) met gewicht w_{ij} schrijven we als
$$i \# j \# w_{ij}.$$
4. **Scheidingssymbol:** Enkelvoudige kantbeschrijvingen worden door het woord '##' van elkaar gescheiden.

Het resulterende woord

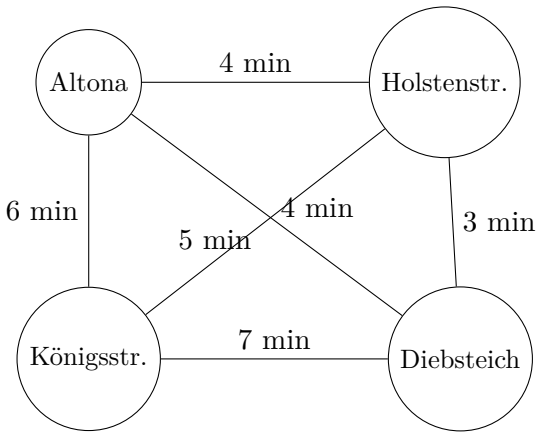
$$i_1\#j_1\#w_{i_1j_1} \,\#\#\, i_2\#j_2\#w_{i_2j_2} \,\#\#\, \ldots \,\#\#\, i_m\#j_m\#w_{i_mj_m}$$

is een geldige codering van de graaf G. Dezelfde structuur kan in verschillende volgordes gecodeerd worden zonder dat de informatie verandert.

Met deze gestandaardiseerde voorstelling kunnen we later geschikte rekenmodellen (bijv. Turingmachines) met het woord voeden en de berekenbaarheid en looptijdeigenschappen van het probleem onderzoeken.

2.3 Métro-graaf en modellering als voorbeeld

Uitsnede van de Métro-kaart: Hamburg



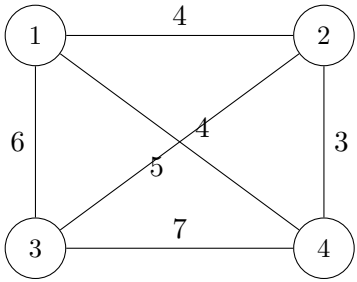
Figuur 2.1: Uitsnede van de Métro-kaart: stations en reistijden

2.3.1 Geabstraheerde gewogen graaf

We wijzen de vier stations de natuurlijke getallen $1, \ldots, 4$ toe:

$$\{\text{Altona}, \text{Holstenstr.}, \text{Königsstr.}, \text{Diebsteich}\} \mapsto \{1, 2, 3, 4\}.$$

Elke directe verbinding krijgt haar reistijd als kantgewicht toegewezen.



Figuur 2.2: Geabstraheerde gewogen graaf met knopen 1–4 en kantgewichten

2.3.2 Codering van de geabstraheerde graaf

In deze sectie tonen we hoe de gewogen graaf $G = (V, E, w)$ als woord over het alfabet

$$\Sigma = \{0, 1, \ldots, 9, \#\}$$

aan de abstracte rekenaar wordt doorgegeven. Elke deelreeks

$$i\#j\#w$$

representeert een kant (i, j) met het gewicht w, en het dubbele scheidingsteken '##' scheidt opeenvolgende kantenbeschrijvingen van elkaar.

Voor de hier beschouwde graaf met de knopen $\{1, 2, 3, 4\}$ en de kanten

$$(1, 2, 4),\ (2, 4, 3),\ (4, 3, 7),\ (3, 1, 6),\ (1, 4, 4),\ (2, 3, 5)$$

krijgt men de volgende codering:

$$1\#2\#4\ \#\#\ 2\#4\#3$$
$$4\#3\#7\ \#\#\ 3\#1\#6$$
$$1\#4\#4\ \#\#\ 2\#3\#5.$$

Met deze tekenreeks kan de rekenaar alle kanten en hun gewichten eenduidig uitlezen en het probleem verder verwerken.

2.4 Codering door middel van een adjacentielijst

Om de gewogen graaf $G = (V, E, w)$ eveneens in een voor de ge"ıdealiseerde rekenaar geschikte tekenreeks om te zetten, gebruiken we een *adjacentielijst*-representatie. We gaan als volgt te werk:

1. We nummeren de knopen $V = \{v_1, \ldots, v_n\}$ achtereenvolgens met de natuurlijke getallen $\{1, \ldots, n\}$.

2. Voor iedere knoop u definiëren we het woord

$$\begin{aligned} N_u = u \ \# \ v_1 \ \# \ w(u, v_1) \\ \# \ v_2 \ \# \ w(u, v_2) \\ \# \ \ldots \\ \# \ v_{k_u} \ \# \ w(u, v_{k_u}). \end{aligned}$$

   waarbij $v_1, \ldots, v_{k_u}$ de directe buren van u zijn en $w(u, v_i)$ het gewicht van de kant (u, v_i) aangeeft.

3. De totale codering van de graaf verkrijgt men door alle deelwoorden $N_1, \ldots, N_n$ met het scheidingsteken '##' aan elkaar te plakken:

$$N_1 \ \#\# \ N_2 \ \#\# \ \ldots \ \#\# \ N_n.$$

Deze voorstelling bevat weliswaar elke kant tweemaal (eens in de lijst van u, eens in die van v), maar biedt een eenvoudige en systematische manier om door de rekenaar direct toegang te krijgen tot de buurinformatie van elke knoop.

Voorbeeld (Métro-Plan Hamburg): Voor onze graaf met de knopen $\{1, 2, 3, 4\}$ en de kanten

$$(1, 2, 4),\ (1, 3, 6),\ (1, 4, 5),\ (2, 3, 4),\ (2, 4, 3),\ (3, 4, 7)$$

ontstaan de volgende adjacentielijsten:

$$N_1 = 1\#2\#4\#3\#6\#4\#5,$$
$$N_2 = 2\#1\#4\#3\#4\#4\#3,$$
$$N_3 = 3\#1\#6\#2\#4\#4\#7,$$
$$N_4 = 4\#1\#5\#2\#3\#3\#7.$$

De volledige codering luidt daarmee:

$$1\#2\#4\#3\#6\#4\#5 \;\#\#\; 2\#1\#4\#3\#4\#4\#3$$
$$3\#1\#6\#2\#4\#4\#7 \;\#\#\; 4\#1\#5\#2\#3\#3\#7.$$

2.5 Codering door middel van een adjacentiematrix

We nummeren de knopen $V = \{v_1, \ldots, v_n\}$ met de natuurlijke getallen $\{1, \ldots, n\}$. De *adjacentiematrix* van een gewogen graaf $G = (V, E, w)$ is de matrix $A = (a_{ij})_{1 \leq i,j \leq n}$ met

$$a_{ij} = \begin{cases} w(i,j), & \text{als } (i,j) \in E, \\ 0, & \text{anders.} \end{cases}$$

We gaan ervan uit dat geen enkele kant in de graaf het gewicht 0 heeft.

Om de matrix A als woord te coderen, schrijven we elke rij als een reeks van zijn elementen, geschei-

den door '#', en voegen we de rijen vervolgens samen met het scheidingsteken '##'. Formeel:

$$\underbrace{a_{11}\#a_{12}\#\ldots\#a_{1n}}_{1.\,\text{rij}} \#\# \underbrace{a_{21}\#a_{22}\#\ldots\#a_{2n}}_{2.\,\text{rij}} \#\# \ \ldots\ \#\# \underbrace{a_{n1}\#a_{n2}\#\ldots\#a_{nn}}_{n.\,\text{rij}}.$$

Voorbeeld (Métro-Plan Hamburg): Voor onze graaf met $n = 4$ en de kanten $(1, 2, 4)$, $(1, 3, 6)$, $(1, 4, 5)$, $(2, 3, 4)$, $(2, 4, 3)$, $(3, 4, 7)$ krijgen we de symmetrische adjacentiematrix

$$A = \begin{pmatrix} 0 & 4 & 6 & 5 \\ 4 & 0 & 4 & 3 \\ 6 & 4 & 0 & 7 \\ 5 & 3 & 7 & 0 \end{pmatrix}.$$

De codering van de matrix luidt dan

$$0\#4\#6\#5 \ \#\# \ 4\#0\#4\#3$$
$$6\#4\#0\#7 \ \#\# \ 5\#3\#7\#0.$$

2.6 Voor- en nadelen

Voor de keuze van een geschikte codering stellen we ten minste twee eisen:

1. **Efficiënte opvragingen:** De gecodeerde gegevens moeten zich zo gemakkelijk en snel mogelijk laten verwerken. Bijvoorbeeld maakt de adjacentielijst-codering het mogelijk om alle buurknopen van een gegeven knoop in $O(d)$-tijd (met d diens graad) te vinden.

2. **Compacte voorstelling:** De totale lengte van de codering moet zo klein mogelijk zijn, om opslag- en communicatiekosten te minimaliseren.

Hieronder vergelijken we de lengtes van de drie coderingen voor ons S-Bahn-planfragment met knopen $\{1, 2, 3, 4\}$ en kanten

$$(1, 2, 4), (1, 3, 6), (1, 4, 5),$$
$$(2, 3, 4), (2, 4, 3), (3, 4, 7).$$

1. Kantenlijst Codeert elke kant precies éénmaal als $i\#j\#w$ en scheidt ze met '##'.

1#2#4## 2#4#3 ## 4#3#7
3#1#6 ## 1#4#4## 2#3#5

Lengte: $6 \times 5 + 5 \times 2 = 40$ tekens.