Inhaltsverzeichnis

Räumliche Analyse durch kombinatorische Optimierung 3
Jan-Henrik Haunert und Alexander Wolff
1 Einleitung 3
2 Grundlagen 4
2.1 Grundbegriffe 5
2.2 Graphen 6
2.3 Explizite Enumeration 7
2.4 Laufzeitangaben und Wachstum von Funktionen 8
2.5 Komplexitätstheorie 10
3 Lösungsansätze für effizient lösbare Probleme 13
3.1 Gierige Algorithmen 14
3.2 Dynamische Programmierung 17
3.3 Teile und Herrsche 19
4 Lösungsansätze für $\mathcal{NP}$-schwere Probleme 23
4.1 Mathematische Programmierung 24
4.2 Ein einfaches ganzzahliges lineares Programm zur Berechnung maximaler unabhängiger Mengen 27
4.3 Lösung ganzzahliger linearer Programme 27
4.4 Eine einfache Heuristik 31
4.5 Lokale Suche 32
4.6 Approximationsalgorithmen 34
5 Fazit und Perspektive 37
Literaturverzeichnis 38

Räumliche Analyse durch kombinatorische Optimierung

Jan-Henrik Haunert und Alexander Wolff

Zusammenfassung In diesem Beitrag geht es uns darum, an einigen wenigen Beispielen aus der räumlichen Analyse grundlegende Entwurfstechniken für Algorithmen und Werkzeuge der kombinatorischen Optimierung zu illustrieren. Außerdem wollen wir ein Minimum an theoretischem Unterbau vermitteln. Damit hoffen wir, dass es dem Leser, der Leserin gelingt, räumliche Probleme mit Methoden der Informatik bewusst und damit erfolgreich zu lösen. Wir halten es für besonders wichtig, dass man neue Probleme sorgfältig mathematisch modelliert und mittels exakter Algorithmen das eigene Modell wenigstens auf kleinen Instanzen überprüft, bevor man sich schnellen Heuristiken zuwendet, um große Instanzen zu lösen.

Schlüsselwörter: Kombinatorische Optimierung, räumliche Analyse, effiziente Algorithmen, mathematische Programmierung, dynamische Programmierung, Teile und Herrsche, Approximationsalgorithmen, Greedy-Algorithmen, Heuristiken

1 Einleitung

Wer möchte nicht gern in kurzer Zeit möglichst viel Geld verdienen? Natürlich spielen in der Wirtschaft auch andere Kriterien wie Umwelt- und Sozialverträglichkeit eine Rolle, doch Geld und Zeit sind gute Argumente, wenn es um Entscheidungen geht. Um Gewinne, zeitliche Abläufe oder andere Kriterien zu optimieren, setzen Unternehmer in den verschiedensten Bereichen Algorithmen ein, zum Beispiel für eine erfolgreiche Partnervermittlung oder den Entwurf materialsparender Ingenieur-

Jan-Henrik Haunert
Institut für Geoinformatik und Fernerkundung, Universität Osnabrück, Barbarastraße 22b, 49076 Osnabrück. E-mail: janhhaunert@uni-osnabrueck.de

Alexander Wolff
Lehrstuhl für Informatik I, Universität Würzburg, Am Hubland, 97074 Würzburg. Orchid ID: http://orcid.org/0000-0001-5872-718X

bauten. Dabei spielen oft *räumliche Kriterien* eine Rolle, etwa wenn ein Logistikunternehmer möglichst kurze Rundwege berechnet oder eine Restaurantkette einen optimalen Standort für eine neue Filiale sucht. Für viele dieser Aufgaben bieten geographische Informationssysteme (GIS) geeignete Werkzeuge. Geodäten und Geoinformatiker müssen jedoch in der Lage sein, neue Lösungen für bisher unbekannte Probleme zu entwickeln. Dies ist in der heutigen Zeit umso wichtiger, als sich Sensoren und andere technische Geräte sowie die damit verfügbaren Geodaten rapide weiterentwickeln. So hätte vor 20 Jahren wohl kaum jemand gedacht, dass heute ein Großteil der Bevölkerung über mobile Telefone verfügt, mit denen sich in kürzester Zeit optimale Routen berechnen lassen.

Viele der sehr unterschiedlichen Fragen in der Geoinformatik lassen sich mit vergleichsweise ähnlichen Mitteln lösen. Dieses Kapitel bietet einen Überblick über die wichtigsten Ansätze zur kombinatorischen Optimierung und diskutiert sie im Kontext ausgewählter Probleme der räumlichen Analyse. Die Auswahl der Probleme soll keineswegs die Vielfalt der Aufgaben in der Praxis widerspiegeln, sondern erfolgt aufgrund ihres beispielhaften Charakters.

Grundsätzlich lassen sich die Probleme, die wir in diesem Kapitel besprechen, in zwei verschiedene Klassen einteilen: Während wir für Probleme der ersten Klasse effiziente Algorithmen angeben können, würde die Existenz eines effizienten Algorithmus für ein Problem der zweiten Klasse (das ist die Menge der $\mathcal{NP}$-schweren Probleme) eine wissenschaftliche Sensation darstellen – was nicht heißen soll, dass es keine Möglichkeit gibt, ein $\mathcal{NP}$-schweres Problem zu zähmen. In der Anwendung sind Probleme beider Klassen gleichermaßen von Bedeutung.

Im Folgenden führen wir wichtige Grundlagen der kombinatorischen Optimierung ein und klären dabei genauer, was es mit der Effizienz von Algorithmen und mit $\mathcal{NP}$-schweren Problemen auf sich hat (Abschnitt 2). Danach stellen wir Ansätze für effizient lösbare Probleme (Abschnitt 3) und $\mathcal{NP}$-schwere Probleme (Abschnitt 4) vor. Zu guter Letzt schließen wir mit einem Fazit (Abschnitt 5).

2 Grundlagen

Nehmen wir an, dass ein Unternehmen eine neue Fabrik plant. Bei der Wahl des Fabrikstandorts soll berücksichtigt werden, dass die Stromversorgung sichergestellt ist. Daher bedient sich die Unternehmensleitung eines Datensatzes, in dem jedes Elektrizitätswerk als Punkt repräsentiert ist. Nun möchte sich das Management nicht zu sehr von einem einzelnen Elektrizitätswerk abhängig machen, denn dieses könnte geschlossen werden oder sein Betreiber könnte den Strompreis erhöhen. Der Standort soll daher so gewählt werden, dass der Abstand zum *zweitnächsten* Elektrizitätswerk möglichst gering ist. Um diesem Ziel nachzukommen, werden die beiden Elektrizitätswerke gesucht, die einander am nächsten sind. Ein optimaler Standort für die Fabrik liegt offensichtlich genau auf der Mitte zwischen diesen beiden Elektrizitätswerken; siehe Abb. 1. Formal lässt sich das Problem wie folgt definieren.

Problem 1 (Fabrikplanung). Gegeben sei eine Menge $P \subseteq \mathbb{R}^2$ von n Punkten in der Ebene. Finde zwei verschiedene Punkte $p = (x_p, y_p)$ und $q = (x_q, y_q)$ in P, deren euklidischer Abstand $\|p - q\| = \sqrt{(x_p - x_q)^2 + (y_p - y_q)^2}$ minimal ist.

In diesem Abschnitt diskutieren wir theoretische Grundlagen, die wir zur Lösung von Optimierungsproblemen benötigen. Wir veranschaulichen diese Grundlagen am Beispiel des Problems der Fabrikplanung. Nach der Klärung einiger zentraler Begriffe (Abschnitt 2.1) wenden wir uns Graphen zu, der mathematischen Modellierung von Netzwerken (Abschnitt 2.2). Der Grund dafür ist, dass Graphen bei der Lösung vieler Optimierungsprobleme eine Rolle spielen, auch bei solchen, bei denen es nicht um Netzwerke geht. Viele Probleme lassen sich als Graphenprobleme modellieren, und die Informatik stellt für viele Graphenprobleme effiziente Algorithmen zur Verfügung. Im Vorgriff auf Abschnitt 3 gehen wir dann kurz auf einen ersten, ganz einfachen Lösungsansatz für effizient lösbare Probleme ein: die explizite Enumeratin (Abschnitt 2.3). Dies liefert uns einen Algorithmus, dessen Laufzeit wir analysieren. Dabei machen wir uns ganz allgemein Gedanken über Laufzeitangaben und das Wachstum von Funktionen (Abschnitt 2.4). Wir beschließen diesen Grundlagenabschnitt mit einigen Überlegung zur Komplexität von Problemen (Abschnitt 2.5).

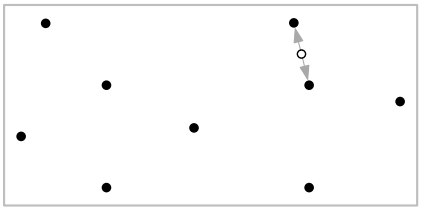

Abb. 1 Eine Menge von Elektrizitätswerken (schwarz) und ein Punkt (weiß) mit minimalem Abstand zum zweitnächsten Elektrizitätswerk.

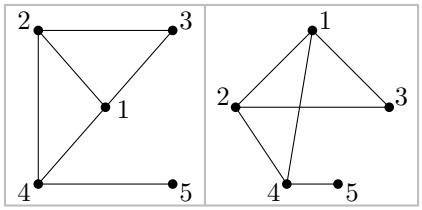

Abb. 2 Zwei Zeichnungen des Graphen $G = (V, E)$ mit $V = \{1,2,3,4,5\}$ und $E = \{\{1,2\},\{1,3\},\{1,4\},\{2,3\},\{2,4\},\{4,5\}\}$.

2.1 Grundbegriffe

Mit der formalen Definition des Problems der Fabrikplanung haben wir bereits eine der wichtigsten Voraussetzungen für die automatische Lösung geschaffen, nämlich die absolute Klarheit über das Problem. Diese Klarheit wird in der Informatik in der Regel durch mathematische Abstraktion hergestellt – so wie wir in unserem Beispiel die Elektrizitätswerke durch Punkte in der euklidischen Ebene modelliert haben. Zur Problemdefinition gehören im Allgemeinen Annahmen über die Eingabedaten und Bedingungen, die an eine Lösung geknüpft werden. Eine Eingabe, welche die Annahmen der Problemdefinition erfüllt, heißt *Instanz* des Problems. Eine *zulässige Lösung* erfüllt alle Bedingungen für eine gegebene Instanz.

Fragen wir nach einer möglichst guten zulässigen Lösung, so stehen wir vor einem *Optimierungsproblem*. Dabei wird die Güte einer Lösung durch eine *Zielfunktion* formalisiert – in unserem Beispiel ist die Zielfunktion die euklidische Distanz zwischen den beiden Punkten der Lösung. Grundsätzlich ist eine *optimale Lösung* gesucht, also eine zulässige Lösung, die den größtmöglichen oder kleinstmöglichen Zielfunktionswert hat. Entsprechend sprechen wir von einem *Maximierungsproblem* oder, wie in unserem Beispiel, von einem *Minimierungsproblem*.

Zur Optimierung werden grundsätzlich Algorithmen eingesetzt, wobei der Begriff *Algorithmus* im Allgemeinen nach einer weniger strengen (i) oder einer strengen Definition (ii) Verwendung findet:

(i) Ein Algorithmus ist eine eindeutig definierte Prozedur, die Daten einliest, eine Folge von Rechenschritten ausführt und Daten ausgibt.
(ii) Ein Algorithmus ist eine eindeutig definierte Prozedur zur Lösung eines eindeutig definierten Problems.

In jedem Fall ist ein Algorithmus unabhängig von seiner Implementierung in einer bestimmten Programmiersprache. Im Folgenden verwenden wir die strenge Definition (ii). Mit dieser Definition (und nur mit ihr) ergibt es Sinn, einen Algorithmus für ein Problem als *korrekt* zu bezeichnen, wenn er für jede Instanz eine zulässige Lösung ausgibt. Ein Algorithmus für ein Optimierungsproblem heißt *exakt*, wenn er für jede Instanz eine optimale Lösung berechnet.

2.2 Graphen

Graphen sind ein mathematisches Konzept zur Beschreibung von paarweisen Relationen zwischen den Elementen einer Menge. Formal ist ein Graph ein Paar (V,E) von zwei Mengen mit $E \subseteq \{\{u,v\} \mid u,v \in V, u \neq v\}$. Dabei wird V als *Knotenmenge* und E als *Kantenmenge* bezeichnet. Die formale Definition von E besagt, dass jede Kante ein Paar von Knoten ist. Graphen werden üblicherweise zeichnerisch dargestellt, wobei in der Regel jeder Knoten durch einen Punkt in der Zeichenebene repräsentiert wird; jede Kante $\{u,v\} \in E$ wird als Linie gezeichnet, welche die Punkte für u und v verbindet. Ein Graph, der gezeichnet werden kann, ohne dass sich zwei seiner Kanten kreuzen, heißt *planar*.

Abbildung 2 zeigt zwei Zeichnungen desselben Graphen. Grundsätzlich erlaubt ein Graph unendlich viele verschiedene Zeichnungen. Das *Graphzeichnen* ist ein umfangreiches Themengebiet der Informatik, das sich damit befasst, Algorithmen anzugeben, die für jeden Graphen (evtl. einer bestimmten Klasse von Graphen) Zeichnungen berechnen, die bezüglich bestimmter Kriterien optimal oder wenigstens zulässig sind. Es findet in der Geoinformatik zur Visualisierung geographischer Netze vielfältige Anwendungen [24].

Für die räumliche Analyse sind *geometrische Graphen* von besonderer Bedeutung; das sind Graphen, deren Knoten und Kanten geometrische Objekte repräsentieren. Geometrische Graphen können Teil der Eingabe zu einem Problem

sein, beispielsweise wenn der geometrisch kürzeste Weg zwischen zwei Punkten in einem Straßennetz (dem gegebenen Graphen) gesucht ist. Zudem kann das Problem darin bestehen, einen geometrischen Graphen bestimmter Eigenschaften zu berechnen, z. B. wenn eine gegebene Menge von Stromerzeugern und -verbrauchern mit möglichst geringem Aufwand über ein Leitungsnetz verbunden werden soll.

Selbst wenn ein Problem nicht direkt nach einem geometrischen Graphen fragt, kann es hilfreich sein, zunächst einen geometrischen Graphen bestimmter Eigenschaften zu berechnen. Der Graph dient nach seiner Berechnung als Datenstruktur, die bestimmte Nachbarschaftsrelationen vorhält und damit die Lösung des eigentlichen Problems ermöglicht. Für die räumliche Analyse werden besonders häufig *Delaunay-Triangulierungen* und *Voronoi-Diagramme* eingesetzt, deren Berechnung zu den Aufgaben der *algorithmischen Geometrie* gehört [2].

Die Graphentheorie bietet zudem eine Vielzahl an Konzepten und Algorithmen [17, 21], die wir für die räumliche Analyse einsetzen können. Dazu gehören Wege, Matchings, Spannbäume und Flüsse sowie Algorithmen zu ihrer Berechnung. Ein Konzept, das wir in diesem Kapitel mehrfach anwenden werden, ist das einer *unabhängigen Menge*: Für einen Graphen $G = (V,E)$ heißt eine Menge $V' \subseteq V$ unabhängig, wenn für jede Kante $\{u,v\} \in E$ entweder $u \notin V'$ oder $v \notin V'$ gilt. Das heißt, V' enthält keine zwei Knoten, die durch eine Kante verbunden sind. So ist $\{1,5\}$ eine unabhängige Menge des Graphen in Abb. 2, wohingegen $\{1,2,5\}$ keine unabhängige Menge dieses Graphen ist, da er die Kante $\{1,2\}$ enthält. Ein bekanntes Problem der Informatik besteht darin, für einen gegebenen Graphen eine möglichst große unabhängige Menge zu berechnen:

Problem 2 (maximale unabhängige Menge). Gegeben sei ein Graph $G = (V,E)$. Finde eine unabhängige Menge $V' \subseteq V$ von G (d. h. für jede Kante $\{u,v\} \in E$ gilt $u \notin V'$ oder $v \notin V'$), die eine maximale Anzahl von Knoten enthält.

Offensichtlich ist $\{1,5\}$ eine maximale unabhängige Menge des Graphen in Abb. 2, denn dieser Graph hat keine unabhängige Menge mit drei Knoten.

2.3 Explizite Enumeration

Eine sehr einfache, oft als *rohe Gewalt* (engl. *brute force*) bezeichnete Herangehensweise zur kombinatorischen Optimierung ist die Aufzählung und Bewertung aller möglichen Lösungen für eine gegebene Instanz. Algorithmus 1 realisiert dieses Prinzip für Problem 1, wobei zur Vereinfachung der Darstellung nur der minimale Abstand zwischen zwei Punkten berechnet und zurückgegeben wird und nicht die beiden Punkte, für die dieses Minimum erreicht wird.

Da alle möglichen Lösungen explizit in Betracht gezogen werden, ist Algorithmus 1 trivialerweise korrekt und exakt. Eine genaue Analyse der doppelten for-Schleife ergibt, dass der Algorithmus $n(n-1)/2$ Iterationen ausführt.

Theoretisch können wir die explizite Enumeration aller Lösungen auch auf Problem 2 anwenden: Wir testen einfach für jede Menge $V' \subseteq V$, ob V' eine un-

Algorithmus 1: NächstesPaarRoheGewalt(P)

Eingabe: Feld $P[1..n]$ mit Punkten in $\mathbb{R}^2$
Ausgabe: Minimaler Abstand d von zwei Punkten in P

```
d = ∞
for i = 1 to n − 1 do
    for j = i + 1 to n do
        d' = ‖P[i] − P[j]‖
        if d' < d then
            d = d'
return d
```

abhängige Menge von G ist, und merken uns zu jeder Zeit die größte unabhängige Menge, die wir bisher gefunden haben. Diese Menge geben wir am Ende zurück. Allerdings müssen wir feststellen, dass V extrem viele Teilmengen enthält, die unser Algorithmus explizit testen würde. Genauer gesagt müsste er $2^{|V|}$ Teilmengen von V testen und entsprechend viele Iterationen ausführen. Bereits für $|V| = 300$ wäre diese Zahl weit größer als die geschätzte Anzahl aller Atome im Universum (ca. 10^{80}). Daher eignet sich der Ansatz hier nur für sehr kleine Eingaben (z. B. $|V| \leq 20$).

2.4 Laufzeitangaben und Wachstum von Funktionen

Allgemein interessieren wir uns neben der Korrektheit oder der Exaktheit eines Algorithmus auch für seine Laufzeit. In der Regel werden wir einen Anstieg der Laufzeit beobachten, wenn wir einen Algorithmus auf größere Instanzen anwenden. Es ist durchaus üblich, die Laufzeit eines Algorithmus experimentell zu ermitteln, indem man Instanzen verschiedener Größe mit einer Implementierung des Algorithmus löst. Die Aussagekraft derartiger Laufzeitmessungen ist jedoch begrenzt, da wir mit Abhängigkeiten von der verwendeten Programmiersprache, der Hardware und dem Betriebssystem zu rechnen haben. Daher ist es sinnvoll, elementare Rechenoperationen (z. B. Vergleichsoperationen oder Additionen und Multiplikationen) zu zählen, statt Laufzeiten mit einer Uhr zu messen. Algorithmus 1 führt beispielsweise pro Iteration genau eine Abstandsberechnung aus (vor der if-Anweisung). Die Anzahl der Iterationen ist $\binom{n}{2} = n(n-1)/2$. Wenn wir vereinfachend davon ausgehen, dass eine Abstandsberechnung eine Zeiteinheit benötigt, so ergibt sich eine Laufzeit von $n(n-1)/2$ für Algorithmus 1.

Bei vielen Algorithmen werden wir auch Laufzeitunterschiede für verschiedene Instanzen gleicher Größe beobachten. Beispielsweise können wir uns eine Variante von Algorithmus 1 vorstellen, die zu Beginn überprüft, ob alle Punkte in P identisch sind, und gegebenenfalls 0 zurückgibt. Für sehr spezielle Instanzen benötigt diese Variante natürlich weniger als $n(n-1)/2$ Abstandsberechnungen. Allerdings

beschränken wir uns bei der Laufzeitanalyse in der Regel darauf, für jede mögliche Eingabegröße eine Instanz zu betrachten, für die der Algorithmus die meisten Rechenschritte benötigt. Das heißt, wenn wir nichts anderes sagen, analysieren wir den schlechtesten Fall. Als *Worst-Case-Laufzeit* bezeichnen wir die Funktion f, die für jede mögliche Eingabegröße n die maximal benötigte Anzahl $f(n)$ von Rechenschritten angibt. In dieser Hinsicht ist die neue Variante von Algorithmus 1 nicht im geringsten besser als die ursprüngliche Version.

Bezeichnen wir die Worst-Case-Laufzeit von Algorithmus 1 mit f_1, das heißt $f_1(n) = n(n-1)/2 = \frac{1}{2}n^2 - \frac{1}{2}n$, und nehmen wir an, dass es einen zweiten Algorithmus für dasselbe Problem mit Laufzeit $f_2(n) = n\log_2 n$ gibt. Dann wäre der zweite Algorithmus vorzuziehen, denn f_2 wächst viel langsamer als f_1. Für sehr kleine n ist der Unterschied kaum wahrnehmbar; $f_1(n)$ ist zunächst sogar kleiner als $f_2(n)$. Bereits für $n = 100$ ist der Vorteil von f_2 gegenüber f_1 jedoch riesig; siehe Abb. 3.

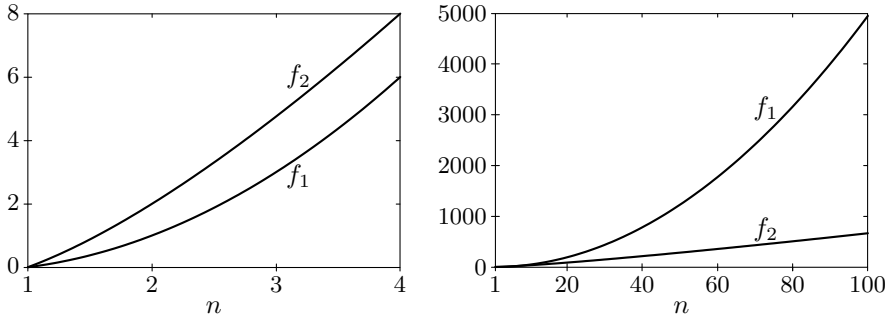

Abb. 3 Funktionsgraphen für $f_1(n) = \frac{1}{2}n^2 - \frac{1}{2}n$ und $f_2(n) = n\log_2 n$ für $n \in [1,4]$ und $n \in [1,100]$.

In der Regel ist die Worst-Case-Laufzeit f eine sehr komplizierte Funktion, die aus vielen Termen besteht; es fällt schwer, sie genau zu bestimmen. Zur Vereinfachung betrachtet man das Verhalten von f für große Werte (bzw. das *asymptotische* Verhalten von f) und schätzt f nach oben ab. Formal geschieht die Abschätzung durch die Groß-O-Notation, das heißt, durch die Angabe einer Funktion g mit $f \in O(g)$, wobei die Menge $O(g)$ alle Funktionen enthält, die für große Werte höchstens um einen konstanten Faktor schneller wachsen als g. Formal gilt

$$\begin{aligned} O(g) = \{f\colon & \text{ Es gibt positive Konstanten } c \text{ und } n_0, \\ & \text{so dass } f(n) \le cg(n) \text{ für alle } n \ge n_0\}\,. \end{aligned} \tag{1}$$

Betrachten wir beispielsweise die Laufzeit $f(n) = n(n-1)/2 = \frac{1}{2}n^2 - \frac{1}{2}n$ von Algorithmus 1. Offensichtlich ist $f(n) \le n^2$, wenn $n \ge 1$. Also ist $f \in O(n^2)$, denn es gilt $f(n) \le cn^2$ für $c = 1$ und für alle $n \ge n_0 = 1$.

Sicherlich hätten wir auch andere Werte für n_0 und c finden können, so dass $f(n) \le cn^2$ für alle $n \ge n_0$ erfüllt ist. Um $f \in O(n^2)$ zu beweisen, reicht es nach (1) aber aus, zu zeigen, dass $f(n) \le cn^2$ für alle $n \ge n_0$ bei *einer Wahl* von n_0 und c erfüllt ist. Wir bemerken weiterhin, dass nicht nur $f \in O(n^2)$, sondern z. B. auch $f \in$

$O(n^3)$ gilt, denn es gilt auch $f(n) \leq cn^3$ für $c = 1$ und für alle $n \geq n_0 = 1$. Die Groß-O-Notation drückt formal also nur eine Abschätzung nach oben aus, wenngleich wir in der Regel bemüht sind, eine möglichst genaue Abschätzung in Form einer möglichst langsam wachsenden oberen Schranke zu finden.

Eine Abschätzung einer Funktion f nach unten ist formal durch Angabe einer Funktion g mit $f \in \Omega(g)$ möglich. Dabei ist

$$\begin{aligned}\Omega(g) = \{f\colon &\text{Es gibt positive Konstanten } c \text{ und } n_0,\\ &\text{so dass } cg(n) \leq f(n) \text{ für alle } n \geq n_0\}\,.\end{aligned} \tag{2}$$

Beispielsweise ist $f(n) = \frac{1}{2}n^2 - \frac{1}{2}n \geq \frac{1}{4}n^2$ für $n \geq 2$. Also ist $f \in \Omega(n^2)$, denn $cn^2 \leq f(n)$ für alle $n \geq n_0 = 2$ und $c = \frac{1}{4}$.

Als letztes definieren wir $\Theta(g)$ für eine Funktion g als Schnittmenge von $O(g)$ und $\Omega(g)$ und ermöglichen somit, das asymptotische Verhalten einer Funktion genau abzuschätzen. Das heißt, $f \in \Theta(g)$ genau dann, wenn $f \in O(g)$ und $f \in \Omega(g)$. Für $f(n) = \frac{1}{2}n^2 - \frac{1}{2}n$ gilt $f \in \Theta(n^2)$, denn $f \in O(n^2)$ und $f \in \Omega(n^2)$.

Allgemein dienen die Definitionen von O, Ω, und Θ der Beschreibung beliebiger Funktionen. Zum Beispiel können wir sagen, dass ein Algorithmus $O(n \log n)$ zusätzlichen Speicher benötigt, die Best-Case-Laufzeit eines Algorithmus in $O(n)$ ist oder die Worst-Case-Laufzeit eines Algorithmus sowohl in $\Omega(n \log n)$ als auch in $O(n^2)$ ist. Auch Aussagen, die sich nicht auf einen Algorithmus beziehen, sind möglich. Beispielsweise hat jeder Graph mit n Knoten $O(n^2)$ Kanten, denn jeder Knoten kann höchsten mit $n-1$ anderen Knoten benachbart sein.

2.5 Komplexitätstheorie

Ein Algorithmus, dessen Worst-Case-Laufzeit f polynomiell beschränkt ist (das heißt, $f \in O(n^c)$ für die Eingabegröße n und eine Konstante $c \in \mathbb{N}$), heißt *effizient*. In der Praxis würde man so manchen effizienten Algorithmus als langsam bezeichnen, zum Beispiel, wenn $f \in \Theta(n^{100})$. Die Definition von „effizient“ ergibt dennoch Sinn, da für viele Optimierungsprobleme nur (oder noch nicht einmal) exakte Algorithmen bekannt sind, deren Laufzeit exponentiell mit n wächst. Oft lassen sich nur extrem kleine Instanzen solcher Probleme in angemessener Zeit exakt lösen.

In der Theoretischen Informatik klassifiziert man Probleme hinsichtlich ihrer Komplexität. Fällt ein Problem in eine Klasse hoher Komplexität, so gilt die Suche nach einem effizienten exakten Algorithmus als hoffnungslos. Man kann sich somit auf andere Lösungsstrategien konzentrieren – zum Beispiel auf die Suche nach einem effizienten, aber nicht exakten Algorithmus – und sich somit unnötige Entwicklungsarbeit sparen.

Viele Komplexitätsklassen der Theoretischen Informatik sind nur für *Entscheidungsprobleme* definiert. Das sind Probleme, in denen eine Frage mit „ja“ oder „nein“ zu beantworten ist. Wir können die Definitionen nicht direkt auf Optimierungsprobleme anwenden, die nach einer optimalen Lösung fragen. Grundsätzlich

lässt sich jedoch für jedes Optimierungsproblem π ein zugehöriges Entscheidungsproblem π' definieren. Wenn π ein Minimierungsproblem (bzw. Maximierungsproblem) ist, so fragt π' danach, ob es für eine gegebene Instanz von π eine Lösung gibt, deren Zielfunktionswert kleiner oder gleich (bzw. größer oder gleich) einer gegebenen Zahl ist. Natürlich ist es nicht schwerer, das Entscheidungsproblem zu lösen als das ursprüngliche Optimierungsproblem. Daher gibt die Analyse der Komplexität von π' durchaus Aufschluss über die Komplexität von π.

Für unsere weitere Diskussion sind die Komplexitätsklassen $\mathscr{P}$ und $\mathscr{NP}$ von besonderer Bedeutung.

Mit $\mathscr{P}$ bezeichnen wir die Menge aller Entscheidungsprobleme, für die effiziente Algorithmen existieren. Natürlich ist das zu Problem 1 zugehörige Entscheidungsproblem in $\mathscr{P}$, denn mit Algorithmus 1 können wir es effizient lösen. Für Problem 2 stellt die explizite Enumeration keinen effizienten Algorithmus dar, doch damit ist nicht ausgeschlossen, dass es nicht doch einen (intelligenteren) effizienten Algorithmus gibt. Wir können also nicht sagen, ob das Entscheidungsproblem zu Problem 2 in $\mathscr{P}$ ist. Tatsächlich ist diese Frage bis heute ungeklärt.

Wir verzichten auf eine genaue Definition der Klasse $\mathscr{NP}$ und verweisen für die Details auf das Standardwerk [10]. Stattdessen nennen wir zwei Bedingungen, die zusammen hinreichend sind, damit das zu einem Optimierungsproblem π zugehörige Entscheidungsproblem π' in $\mathscr{NP}$ ist:

(a) Eine optimale Lösung s zu einer Instanz I von π braucht nicht wesentlich mehr Speicher als I. (Genauer gesagt, die Größe von s ist polynomiell in der Größe von I.)
(b) Es existiert ein effizienter Algorithmus, der für eine gegebene Instanz I von π und eine beliebige Eingabe e überprüft, ob e eine zulässige Lösung für I ist und, falls dem so ist, den Zielfunktionswert von e berechnet.

Offensichtlich sind diese beiden Bedingungen in der Praxis für die meisten Optimierungsprobleme erfüllt. Beispielsweise ist das zu Problem 1 zugehörige Entscheidungsproblem in $\mathscr{NP}$, denn (a) eine optimale Lösung zum Optimierungsproblem lässt sich durch zwei Punkte repräsentieren und (b) für zwei gegebene Punkte $p,q \in \mathbb{R}^2$ können wir effizient entscheiden, ob $p \neq q$, ob $p \in P$ und $q \in P$, und wir können den euklidischen Abstand $\|p-q\|$ effizient berechnen. (Aus Sicht der Genauigkeit wäre es übrigens besser, auf die Anwendung der Wurzelfunktion zu verzichten und stattdessen quadrierte Abstände zu vergleichen.) Ebenso ist Problem 2 in $\mathscr{NP}$, denn (a) eine optimale Lösung zum Optimierungsproblem enthält höchstens $|V|$ Knoten und (b) für eine Knotenmenge V' können wir effizient entscheiden, ob sie die Eigenschaften einer unabhängigen Menge hat (indem wir testen, ob $V' \subseteq V$ und ob für jede Kante $\{u,v\} \in E$ gilt, dass $u \notin V'$ oder $v \notin V'$), und wir können die Anzahl der Elemente in V' effizient berechnen (durch simples Zählen).

Tatsächlich ist die Definition von $\mathscr{NP}$ so allgemein, dass $\mathscr{P} \subseteq \mathscr{NP}$ gilt. Das heißt, jedes Problem in $\mathscr{P}$ liegt auch in $\mathscr{NP}$. Dies entspricht der Intuition, denn es ist sicherlich nicht schwieriger, die Zulässigkeit einer gegebenen Lösung zu verifizieren und ihren Zielfunktionswert zu berechnen als eine optimale Lösung zu finden. Eine der größten offenen Fragen der Theoretischen Informatik ist jedoch,

ob $\mathscr{P} \subsetneq \mathscr{NP}$ oder $\mathscr{P} = \mathscr{NP}$. Die scheinbar sehr allgemeine Definition von $\mathscr{NP}$ könnte also tatsächlich äquivalent zur Definition von $\mathscr{P}$ sein (was allerdings kaum jemand annimmt).

Es ist wichtig festzuhalten, dass es keineswegs negativ zu bewerten ist, wenn ein Problem in der Klasse $\mathscr{NP}$ liegt, denn $\mathscr{NP}$ enthält nach Definition (wegen $\mathscr{P} \subseteq \mathscr{NP}$) auch sehr einfache Probleme. Einige Probleme (in $\mathscr{NP}$ und darüber hinaus) haben sich als besonders hartnäckig herausgestellt. Für sie gilt, dass nur dann ein effizienter Algorithmus für sie existieren kann, wenn ein effizienter Algorithmus für *jedes* Problem in $\mathscr{NP}$ existiert. Solche Probleme heißen *NP-schwer*. Fände man also einen effizienten Algorithmus für ein einziges $\mathscr{NP}$-schweres Problem, so hätte man bewiesen, dass $\mathscr{P} = \mathscr{NP}$ gilt. Da an diesem Beweis schon viele ausgezeichnete Theoretiker gescheitert sind, wird ein Anwender an der Suche nach einem effizienten Algorithmus für ein $\mathscr{NP}$-schweres Problem scheitern.

Aus Sicht der Komplexitätstheorie ist es eine sehr gute Nachricht, wenn ein Problem in $\mathscr{P}$ liegt. Wenn ein Problem in $\mathscr{NP}$ liegt, ist das immer noch eine gute Nachricht – eine obere Schranke für die Komplexität des Problems. Wenn ein Problem jedoch $\mathscr{NP}$-schwer ist, so ist das eine schlechte Nachricht – eine untere Schranke für seine Komplexität.

Wenn man eine Zeitlang erfolglos versucht hat, einen effizienten Algorithmus für ein Problem zu finden, sollte man für eine gewisse Zeit den Standpunkt wechseln und versuchen zu zeigen, dass das Problem $\mathscr{NP}$-schwer ist. Wenn einem das gelingt, kann man (nach heutigem Kenntnisstand) die Suche nach einer effizienten Lösung einstellen. Wie zeigt man aber, dass ein Problem Π $\mathscr{NP}$-schwer ist? Üblicherweise sucht man ein mehr oder weniger ähnliches Problem Π', das schon als $\mathscr{NP}$-schwer bekannt ist, und zeigt, dass man Π' effizient lösen könnte, wenn man einen effizienten Algorithmus für Π hätte. Man sagt, dass man Π' auf Π *reduziert*. Eine Reduktion ist ein Algorithmus, der aus jeder Instanz I' von Π' (in Polynomialzeit) eine Instanz I von Π konstruiert, so dass I' eine Ja-Instanz ist genau dann, wenn I eine Ja-Instanz ist. Es hat sich herausgestellt, dass das $\mathscr{NP}$-schwere Problem *planares 3-SAT* [18] ein guter Startpunkt für Reduktionen ist, um die $\mathscr{NP}$-Schwere von räumlichen Problemen zu beweisen; siehe z. B. die optimale Vereinfachung von Gebäudegrundrissen [12].

Es gibt viele Probleme, für die sowohl eine gute als auch eine schlechte Nachricht bekannt ist; sie liegen einerseits in $\mathscr{NP}$, gehören andererseits dort zu den schwierigsten Problemen, sind also $\mathscr{NP}$-schwer. Solche Probleme nennt man $\mathscr{NP}$-*vollständig*. Da die Klasse $\mathscr{NP}$ nur für Entscheidungsprobleme definiert ist, kann ein Optimierungsproblem $\mathscr{NP}$-schwer, aber nicht $\mathscr{NP}$-vollständig sein.

Die Berechnung einer maximalen unabhängigen Menge für einen gegebenen Graphen (Problem 2) ist ein klassisches $\mathscr{NP}$-schweres Problem [10]. Die Existenz eines effizienten Algorithmus für dieses Problem würde also einen effizienten Algorithmus für jedes Problem in $\mathscr{NP}$ implizieren.

3 Lösungsansätze für effizient lösbare Probleme

Nehmen wir an, dass ein Autofahrer eine Fahrt von München nach Hamburg plant und sich für eine Route entschieden hat. Nun hat er noch einen freien Platz im Auto, den er Interessenten für Geld über eine Mitfahrzentrale anbieten möchte. Die Mitfahrzentrale betreibt eine ausgefeilte Internet-Plattform, auf der Interessenten auch für Teilstrecken einer Route Gebote abgeben können. So kann unser Autofahrer z. B. insgesamt 70 € verdienen, indem er ein Gebot von 30 € für die Teilstrecke München–Würzburg und ein Gebot von 40 € für die Teilstrecke Kassel–Hamburg annimmt. Ein Interessent, der für die Gesamtstrecke München–Hamburg 60 € geboten hat, ginge dann leer aus.

Grundsätzlich können wir jedes Gebot als ein Intervall $I = (a,b)$ repräsentieren, wobei $a \in \mathbb{R}$ den gewünschten Einstiegsort und $b \in \mathbb{R}$ den Ausstiegsort angibt (gemessen in km ab Startpunkt der Route). Zudem ist für jedes Gebot der Preis gegeben, den der Interessent bereit ist zu zahlen. Um seinen freien Sitz nicht doppelt zu vergeben, muss der Autofahrer aus der Menge aller Gebote eine unabhängige Teilmenge auswählen, also eine Teilmenge, in der die Intervalle paarweise disjunkt sind. Unter allen unabhängigen Teilmengen möchte er selbstverständlich eine finden, die die Einnahmen maximiert. Formal steht er also vor folgendem Problem.

Problem 3 (Mitfahrzentrale). Gegeben sei eine Menge I von n offenen Intervallen $I_1 = (a_1,b_1), I_2 = (a_2,b_2), \ldots, I_n = (a_n,b_n)$, aufsteigend sortiert nach Endpunkt (d. h. $b_1 \leq b_2 \leq \cdots \leq b_n$), und Gewichte $w_1, w_2, \ldots, w_n \geq 0$. Finde eine unabhängige Teilmenge I' von I mit maximalem Gewicht.

Wir benutzen *offene* Intervalle, d. h. solche, die ihren Rand (das sind die zwei Endpunkte) nicht enthalten. Das hat den Vorteil, dass zwei Intervalle (a,b) und (c,d) mit $b = c$ disjunkt sind, also gemeinsam in eine Lösung gesteckt werden dürfen. Bei zwei abgeschlossenen Intervallen $[a,b]$ und $[c,d]$ mit $b = c$ wäre dies nicht möglich.

Das Problem der Mitfahrzentrale ähnelt augenscheinlich Problem 2, das nach einer maximalen unabhängigen Menge in einem Graphen fragt. In unserem Fall können wir einen Graphen G konstruieren, der einen Knoten für jedes Intervall enthält und in dem zwei Knoten genau dann mit einer Kante verbunden sind, wenn sich die entsprechenden Intervalle schneiden. Jede zulässige Auswahl von Intervallen entspricht nun einer unabhängigen Menge von G.

Es gibt jedoch zwei Unterschiede zwischen dem Problem der Mitfahrzentrale und Problem 2: Zum einen wollen wir das Gesamtgewicht der ausgewählten Intervalle maximieren, nicht ihre schiere Anzahl. Zum anderen haben unsere Instanzen eine besondere Struktur. So ist es z. B. unmöglich, vier Intervalle I_1, I_2, I_3, I_4 anzugeben, deren Schnittbeziehungen durch den Graphen $G = (V,E)$ mit $V = \{I_1, I_2, I_3, I_4\}$ und $E = \{\{I_1, I_2\}, \{I_2, I_3\}, \{I_3, I_4\}, \{I_4, I_1\}\}$ repräsentiert werden. Wie wir in diesem Abschnitt sehen werden, ermöglicht dies einen effizienten Algorithmus für Problem 3 (bei beliebigen Gewichten), obwohl für allgemeine Graphen die Berechnung einer (ungewichteten) maximalen unabhängigen Menge $\mathcal{NP}$-schwer ist.

In diesem Abschnitt wenden wir zwei der wichtigsten Entwurfstechniken für effiziente Algorithmen (für Optimierungsprobleme) auf das Problem der Mitfahrzentra-

le an, nämlich *gierige Algorithmen* (Abschnitt 3.1) für den Fall, dass alle Gewichte 1 sind, und *dynamische Programmierung* (Abschnitt 3.2) für die gewichtete Variante des Problems. Im Anschluss daran (Abschnitt 3.3) kommen wir noch einmal auf das Problem der Fabrikplanung (Problem 1) zurück und unterbieten mithilfe der Entwurfstechnik *Teile und Herrsche* die quadratische Laufzeit von Algorithmus 1. Diese Entwurfstechniken werden in Algorithmik-Lehrbüchern sehr ausführlich behandelt; von Cormen et al. [6] haben wir einige Aspekte des Problems der Mitfahrzentrale aufgegriffen.

3.1 Gierige Algorithmen

Betrachten wir erst den ungewichteten Fall, d. h. dass alle Mitfahrer die gleiche Gebühr bezahlen müssen, unabhängig davon wie lang die Strecke ist, die sie mitfahren. Wir können also einfach davon ausgehen, dass alle Gewichte gleich 1 sind und dass wir eine möglichst große unabhängige Teilmenge der gegebenen Menge I von Intervallen suchen.

Es ist verlockend, das Problem dadurch zu lösen, dass man gierig (engl. *greedy*) immer wieder das kürzeste Intervall auswählt, das zur bisherigen Lösung passt. Es ist jedoch nicht schwer sich zu diesem Ansatz ein Gegenbeispiel (mit nur drei Intervallen!) auszudenken, das zeigt, dass der Ansatz nicht immer eine maximale unabhängige Teilmenge liefert. (Eine interessante Frage an dieser Stelle ist, um wie viel schlechter dieser Greedy-Algorithmus im Vergleich zu einer optimalen Lösung ist. Mit diesem Typ von Fragen beschäftigen wir uns in Abschnitt 4.6.)

Wir können trotzdem etwas von dem gerade verworfenen Ansatz lernen: da alle Intervalle gleichen Ertrag liefern, haben wir mit dem kürzesten Intervall versucht eines auszuwählen, das den Rest der Instanz am wenigsten einschränkt. Wenn wir diese Strategie kombinieren mit der Idee die Ordnung der Intervalle von links nach rechts auszunutzen, dann bekommen wir einen exakten Greedy-Algorithmus.

Unser zweiter Algorithmus für das ungewichtete Intervallauswahlproblem wählt immer wieder das „linkeste“ Intervall I_j aus, d. h. das Intervall, dessen rechter Endpunkt unter denen der verbliebenen Intervalle am weitesten links ist (also mit minimalem b_j-Wert); siehe Algorithmus 2 und Abb. 4. Dann verwerfen wir (wie oben auch) alle Intervalle, die das ausgewählte Intervall I_j schneiden. Hier geht das ganz einfach: wir brauchen nur so lange weiter nach rechts zu gehen (d. h. einen mit $k = j+1$ initialisierten Zähler erhöhen), bis das aktuelle Intervall I_k disjunkt zu I_j ist (d. h. sobald $a_k \geq b_j$ gilt). Wenn wir die Intervalle also schon aufsteigend sortiert nach rechtem Endpunkt bekommen, so läuft unser zweiter Greedy-Algorithmus in Linearzeit (d. h. in $O(n)$ Zeit) – obwohl wir in Algorithmus 2 zwei ineinandergeschachtelte while-Schleifen verwendet haben.

Doch warum sollte dieser Algorithmus exakt sein? Um uns davon zu überzeugen, betrachten wir eine beliebige Instanz $I = \{I_1, I_2, \ldots, I_n\}$ unseres Problems und vergleichen eine optimale Lösung $I^\star \subseteq I$ mit der Lösung $I' \subseteq I$, die unser zweiter

Algorithmus 2: UngewichteteAblaufplanung(I)

Eingabe: Menge I von Intervallen $I_1 = (a_1, b_1), I_2 = (a_2, b_2), \ldots, I_n = (a_n, b_n)$ mit $b_1 \leq b_2 \leq \cdots \leq b_n$
Ausgabe: Maximale unabhängige Menge $I' \subseteq I$

```
I' = ∅
j = 1
while j ≤ n do
    I' = I' ∪ {I_j}
    k = j + 1
    while k ≤ n and a_k < b_j do
        k = k + 1
    j = k
return I'
```

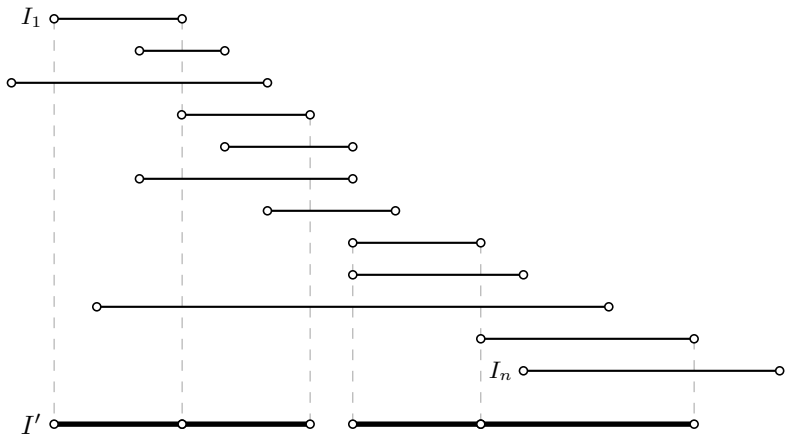

Abb. 4 Eine Instanz I des ungewichteten Ablaufproblems (aufsteigend sortiert nach rechtem Endpunkt) und die Greedy-Lösung I' von Algorithmus 2 (unten, fett).

Greedy-Algorithmus zurückgibt. Wir behaupten, dass beide Lösungen gleiche Kardinalität haben, d. h. $|I^\star| = |I'|$.

Um das zu sehen, bauen wir die optimale Lösung $I^\star$ sukzessive zur Greedy-Lösung I' um; siehe Abb. 5. Wir gehen von links nach rechts parallel durch beide Lösungen. Intervalle, die in beiden Lösungen vorkommen, überspringen wir. Angenommen, wir stoßen auf ein Intervall $I_j^\star$ der optimalen Lösung, das verschieden vom aktuellen Intervall I'_j der Greedy-Lösung ist; zum Beispiel in Schritt 3 (also für $j = 3$) in Abb. 5. Nun behaupten wir, dass der rechte Endpunkt von $I_j^\star$ nicht links vom rechten Endpunkt von I'_j liegen kann, d. h. wir behaupten, dass $b_j^\star \geq b'_j$.

Falls $j = 1$, so ist I'_j das erste Intervall der Greedy-Lösung und nach Wahl das linkeste, d. h. b'_1 ist minimal unter allen rechten Intervallendpunkten. Also ist unsere Behauptung $b_j^\star \geq b'_j$ für $j = 1$ erfüllt. Ansonsten haben wir alle bisherigen Intervalle

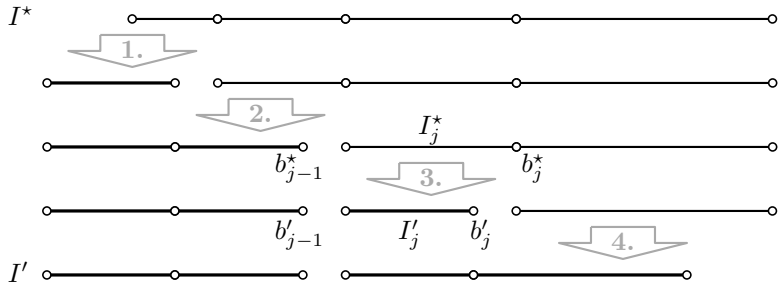

Abb. 5 Schrittweiser Umbau einer optimalen Lösung $I^\star$ in die Greedy-Lösung I' aus Abb. 4.

der beiden Lösungen übersprungen, d. h. sie stimmen überein. Nun ist I_j' unter allen Intervallen rechts von I_{j-1}' nach Wahl wieder das linkeste, d. h. es gilt auch in diesem Fall, dass $b_j^\star \geq b_j'$.

Also können wir nun die optimale Lösung $I^\star$ dahingehend abändern, dass ein Intervall mehr mit der Greedy-Lösung übereinstimmt. Dazu tauschen wir einfach $I_j^\star$ durch I_j' aus. Die geänderte Menge $I^\star = (I^\star \setminus \{I_j^\star\}) \cup \{I_j'\}$ ist immer noch unabhängig, da I_j' in die entstandene Lücke von $b_{j-1}^\star$ bis $b_j^\star$ passt; siehe noch einmal Schritt 3 in Abb. 5. (Die Lücke ist groß genug, da $b_{j-1}^\star = b_{j-1}'$ und $b_j^\star \geq b_j'$, wie wir gerade gezeigt haben. Damit unsere Aussage auch für $j = 1$ stimmt, setzen wir $b_0^\star = b_0' = -\infty$.)

Nun gehen wir weiter parallel durch die beiden Lösungen und bauen sukzessive $I^\star$ zu I_j' um. Unsere *Invariante* ist dabei, dass der schon betrachtete Teil der beiden Lösungen übereinstimmt. (Das brauchen wir oben bei unserer Argumentation, dass für die beiden aktuellen Intervalle $I_j^\star$ und I_j' immer $b_j^\star \geq b_j'$ gilt.) Aus der Aufrechterhaltung dieser Invarianten folgt, dass am Ende des Umbauprozesses $|I^\star| = |I'|$ gilt. Da wir immer *ein* Intervall durch *ein* anderes austauschen, muss die Gleichheit auch vor Beginn des Umbaus von $I^\star$ zu I' gegolten haben. Das ist es, was wir zeigen wollten.

Der hier vorgestellte Greedy-Algorithmus ist ein Beispiel für einen *iterativen* Algorithmus, also einen Algorithmus, der die Lösung schrittweise aufbaut und für den Teil der Eingabe, den er bislang gesehen hat, schon eine korrekte Lösung berechnet hat. Das ist neben Teile und Herrsche (siehe Abschnitt 3.3) eine sehr übliche Algorithmenentwurfstechnik. Als Kniff, mit dem wir uns von der Korrektheit eines solchen iterativen Algorithmus überzeugen können, sind Invarianten (wie die obige) sehr hilfreich. Eine Invariante muss man so wählen, dass bei Abbruch des Algorithmus aus der Abbruchbedingung der Schleife und der Invarianten die gewünschte Korrektheit des Algorithmus folgt. Außerdem muss die Invariante bei Beginn des Algorithmus (typischerweise vor dem Start der Hauptschleife) erfüllt sein. Und schließlich muss man aus der Annahme, dass die Invariante zu Beginn der j-ten Iteration gegolten hat, folgern können, dass sie auch zu Beginn der $(j+1)$-ten Iteration noch gilt (falls die Abbruchbedingung der Schleife nicht erfüllt ist).

Allgemein gilt für Greedy-Algorithmen, dass sie sehr naheliegend sind; die ersten Ideen, die man zur Lösung eines neuen Problems formuliert sind typischerweise

Greedy-Ansätze. Meistens liefern Greedy-Algorithmen (anders als Algorithmus 2 für die ungewichtete Variante von Problem 3) jedoch keine optimalen Lösungen. Trotzdem lohnt es sich über Greedy-Ansätze nachzudenken – und sei es nur um zu verstehen, woran sie scheitern: Dort versteckt sich die eigentliche Schwierigkeit eines Problems!

3.2 Dynamische Programmierung

Betrachten wir jetzt die gewichtete Version des Problems der Mitfahrzentrale, also die Version, die wir zu Beginn von Abschnitt 3 als Problem 3 definiert haben. Es ist klar, dass beide Greedy-Algorithmen für die ungewichtete Version im Allgemeinen keine guten Lösungen liefern werden, da sie die Gewichte der Intervalle nicht berücksichtigen. Genauer gesagt: es ist leicht Instanzen zu konstruieren, bei denen die Greedy-Algorithmen Lösungen liefern, deren Gewicht im Vergleich zum maximalen Gewicht einer unabhängigen Menge beliebig klein ist.

Aber auch hier hilft uns die Eindimensionalität des Problems weiter. Würden wir nämlich *ein* Intervall einer festen optimalen Lösung $I^\star$ kennen, dann würde dieses Intervall die gegebene Instanz in zwei Teilinstanzen zerlegen, die wir *unabhängig* voneinander lösen könnten. Noch besser wäre es, wir würden das erste oder letzte Intervall einer optimalen Lösung kennen, denn dann wäre nur *eine* Teilinstanz übrig, die wir noch lösen müssten. Genau das ist unser Plan: wir erraten das letzte Intervall, indem wir alle Intervalle ausprobieren und jedes mit einer optimalen Lösung für die verbliebene, kleinere Teilinstanz kombinieren. Die Kombination mit maximalem Gewicht ist dann eine optimale Lösung für die ursprüngliche Instanz – denn ein Intervall muss ja in der vorher fixierten optimalen Lösung $I^\star$ das letzte sein.

Diese einfache Beobachtung zeigt, dass unser Problem *optimale Teilstruktur* hat. Ein griffiges Beispiel für ein anderes Problem mit optimaler Teilstruktur ist das Kürzeste-Wege-Problem: jedes Teilstück eines kürzesten Weges von a nach b ist selbst wieder ein kürzester Weg – sonst gäbe es einen kürzeren Weg von a nach b. Optimale Teilstruktur ist die Voraussetzung dafür, dass man dynamische Programmierung anwenden kann. Das Wort *Programmierung* (engl. *programming*) bezieht sich hier (wie bei der *mathematischen Programmierung* in Abschnitt 4.1) nicht auf einen Programmierstil, sondern auf das Ausfüllen von Tabellen. In der Tabelle werden Werte optimaler Lösungen von Teilinstanzen verwaltet, aus denen man sich dann den Wert einer optimalen Lösung für die ganze Instanz „zusammenbaut“.

Nachdem man sich davon überzeugt hat, dass das vorliegende Problem optimale Teilstruktur aufweist, besteht der nächste Schritt zur Erstellung eines dynamischen Programms darin, den Wert einer optimalen Lösung (einer Teilinstanz) *rekursiv* zu definieren. Daraus ergibt sich dann im nächsten Schritt ein rekursiver Algorithmus.

Wir benötigen noch etwas Notation. Sei $I^{(k)}$ die Teilmenge der gegebenen Intervallmenge I, die jedes Intervall I_j aus I enthält, das vor dem Startpunkt a_k von I_k endet, d. h. das die Bedingung $b_j \leq a_k$ erfüllt. Außerdem sei $I^{(n+1)} = I$, und für $k = 1, 2, \ldots, n+1$ sei W_k das Gewicht einer optimalen Lösung für $I^{(k)}$. Dann gilt

nach unserer obigen Diskussion

$$W_k = \max_{I_j \in I^{(k)}} \{w_j + W_j\}. \tag{3}$$

Das ist die gesuchte rekursive Definition von Werten optimaler Lösungen für Teilinstanzen: wir definieren W_k mithilfe von W_j für Werte von j, die kleiner als k sind.

Dies liefert uns einen (ersten) rekursiven Algorithmus für die Berechnung von W_{n+1}, dem Gewicht einer optimalen Lösung für I. Wir kapseln einfach Gleichung 3 in eine Methode und ersetzen dabei W_j durch einen rekursiven Aufruf derselben Methode. Die Rekursion bricht ab, wenn die übergebene Intervallmenge leer ist; dann wird Null zurückgegeben. (Da $I^{(1)}$ leer ist, gilt $W_1 = 0$.)

Analysieren wir jetzt die Laufzeit des rekursiven Algorithmus. Sei $T(n+1)$ die Anzahl der Additionen, die bei der Berechnung von W_{n+1} ausgeführt wird. Da unsere Methode rekursiv ist, ist es einfach, die Laufzeit rekursiv zu beschreiben. Es gilt $T(1) = 0$, da $I^{(1)}$ leer ist. Für $n \geq 1$ gilt

$$T(n+1) = T(1) + T(2) + \cdots + T(n) + n,$$

da $I^{(n+1)} = I$ aus n Elementen besteht und daher auf der obersten Ebene der Rekursion n Additionen ausgeführt werden. Durch Einsetzen (also $T(2) = 1$, $T(3) = 3$, $T(4) = 7$, $T(5) = 15$, $\ldots$) gelangt man schnell zur Vermutung, dass $T(n+1) = 2^n - 1$ ist; dies lässt sich mit vollständiger Induktion über n auch beweisen. Mit anderen Worten: unser rekursiver Algorithmus benötigt exponentielle Zeit! (Man beachte, dass wir beim Aufstellen der Rekursionsgleichung für T stillschweigend vom Worst-Case ausgegangen sind, der darin besteht, dass für $k = 1, \ldots, n-1$ die Menge $I^{(k+1)}$ aus k Elementen besteht.)

Das Problem ist offensichtlich, dass wir bei der rekursiven Berechnung von W_{n+1} die Werte von $W_1, W_2, \ldots, W_n$ immer wieder von neuem berechnen. Es gibt zwei Möglichkeiten, das zu verhindern. Die erste Möglichkeit ist, die rekursive Berechnung so abzuändern, dass man eine Tabelle verwendet, in der man schon bekannte Werte von $W_1, W_2, \ldots, W_n$ ablegt und in der man bei jedem rekursiven Aufruf zuerst nachschaut, ob der Wert schon bekannt ist. Die zweite Möglichkeit ist, die Werte nicht rekursiv (also top-down, beginnend mit W_{n+1}), sondern iterativ (also bottom-up, beginnend mit W_1) zu berechnen. Dieser iterative Ansatz wurde in Algorithmus 3 umgesetzt. Es ist offensichtlich, dass der iterative Algorithmus eine Laufzeit von $O(n^2)$ hat. (Das selbe gilt übrigens für den rekursiven Algorithmus, der immer erst in der Tabelle nachschaut.) Das Problem kann sogar in Linearzeit gelöst werden, falls die Intervalle nach Endpunkt sortiert gegeben sind [14].

Zusammenfassend gilt für Probleme, die man mittels dynamischer Programmierung optimal lösen möchte, dass die wichtige Voraussetzung der optimalen Teilstruktur gegeben sein muss. Man muss also in der Lage sein, den Wert der optimalen Lösung für eine Instanz zu berechnen, indem man auf Werte optimaler Lösungen von Teilinstanzen zurückgreift. Meistens nützt man für die Zwischenspeicherung dieser Werte ein (gegebenenfalls mehrdimensionales) Array.

Algorithmus 3: DynamischesProgrammIterativ(int[] a, int[] b, int[] w)

Eingabe: Menge I von offenen Intervallen $\{(a[1],b[1]),\ldots,(a[n],b[n])\}$ mit Gewichten $w[1],\ldots,w[n]$
Ausgabe: Maximales Gewicht einer unabhängigen Menge in I

```
n = w.length
W = new int[n+1]
W[1] = 0
for k = 2 to n+1 do
    // Berechne W[k] = max_{I_j ∈ I^(k)} {w[j] + W[j]} iterativ:
    W[k] = 0
    for j = 1 to n do
        if b[j] ≤ a[k] then
            if W[k] < w[j] + W[j] then
                W[k] = w[j] + W[j]
return W[n+1]
```

3.3 Teile und Herrsche

Um die quadratische Laufzeit von Algorithmus 1 für das Problem der Fabrikplanung zu unterbieten, wenden wir uns der Strategie *Teile und Herrsche* zu. Grundsätzlich teilen wir dafür eine Probleminstanz in zwei oder mehr Teilinstanzen, die wir unabhängig voneinander lösen. Die Lösungen für die Teilinstanzen kombinieren wir zu einer Lösung für die ursprüngliche Instanz. Oft ist es möglich, diesen Kombinations- bzw. Herrsche-Schritt so zu entwerfen, dass die Exaktheit des Algorithmus gewährleistet ist.

Algorithmus 4 setzt dieses Prinzip um, indem er die gegebene Instanz *räumlich* entlang einer vertikalen Linie teilt. Nach der Lösung der Teilinstanzen erfolgt eine genaue Behandlung des Grenzbereichs in der Umgebung der Trennlinie. Neben dem Feld P mit allen n Punkten erwartet die Methode NächstesPaarTeileUndHerrsche zwei ganze Zahlen ℓ, r. Diese grenzen den Bereich von P ein, der bei der Suche nach den zwei nächsten Punkten Berücksichtigung finden soll. Beim ersten Aufruf ist $\ell = 1$ und $r = P.length$ (womit wir die Länge des Feldes P bezeichnen). Das heißt, es werden zunächst alle Punkte in P berücksichtigt. Dieses ändert sich jedoch nach dem Teile-Schritt und den erneuten (rekursiven) Aufrufen der Methode.

Grundsätzlich verfolgt Algorithmus 4 die Teile-und-Herrsche-Strategie erst für Instanzen von mehr als drei Punkten. Kleinere Instanzen (das heißt, $r - \ell \leq 2$) werden mit roher Gewalt (Algorithmus 1) gelöst; siehe Zeile 2 in Algorithmus 4. Dadurch ist der Abbruch der Rekursion gewährleistet.

Für $r - \ell > 2$ wird das Teilfeld $P[\ell..r]$ von P zunächst aufsteigend nach x-Koordinate sortiert. Danach erfolgt die Teilung der Instanz entlang einer vertikalen

Geraden durch den Punkt $P[m]$ mit $m = \lfloor(\ell+r)/2\rfloor$; siehe Abb. 6. Somit ist sichergestellt, dass beide Teilinstanzen (die Instanzen links und rechts der Teilungsgeraden) ungefähr gleich groß sind. Der minimale Abstand zwischen zwei Punkten in der linken Teilinstanz wird in Zeile 5 rekursiv berechnet, für die rechte Teilinstanz in Zeile 6. Das Minimum $d^\star$ aus diesen beiden Abständen wird in Zeile 7 berechnet.

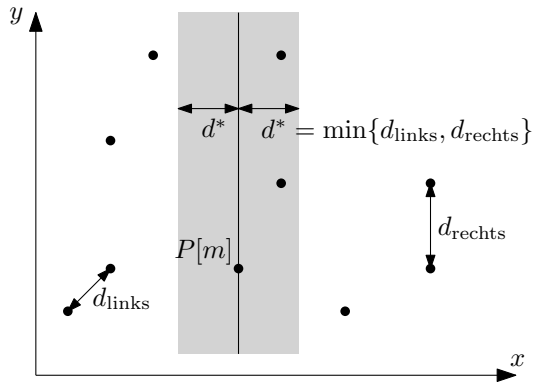

Abb. 6 Algorithmus 4 teilt eine Instanz durch eine vertikale Gerade in zwei Teilinstanzen. In beiden Teilinstanzen wird der minimale Abstand zweier Punkte ermittelt (d_{links} und d_{rechts}). Es bleibt zu überprüfen, ob zwei Punkte auf verschiedenen Seiten der Trenngeraden näher zueinander sind. Dafür kommen nur Punkte nahe der Trenngeraden (d. h. in der grauen Region) infrage.

Es bleibt zu prüfen, ob zwei Punkte $p = P[i]$ und $q = P[j]$, mit $\ell \le i \le m < j \le r$ Abstand $\|p-q\| < d^\star$ haben. Das heißt, p liegt in der linken und q in der rechten Teilinstanz. Diese Überprüfung geschieht in Zeilen 11–18. Die grobe Vorgehensweise entspricht einem Durchlauf durch die linke Teilinstanz und einem darauf abgestimmten Durchlauf durch die rechte Teilinstanz. Dabei werden jedem Punkt p der linken Teilinstanz die Punkte der rechten Teilinstanz gegenübergestellt, die möglicherweise Abstand kleiner $d^\star$ zu p haben. Um diese Abstimmung möglichst effizient zu gestalten, werden die Teilfelder $P[\ell..m]$ und $P[(m+1)..r]$ aufsteigend nach y-Koordinate sortiert (Zeilen 8 und 9); die Durchläufe durch $P[\ell..m]$ und $P[(m+1)..r]$ erfolgen von unten nach oben.

Die äußere Schleife iteriert durch alle Punkte der linken Teilinstanz. Durch die if-Abfrage in Zeile 12 wird ein Punkt p nur dann weiter betrachtet, wenn sein Abstand zur Trenngeraden kleiner $d^\star$ ist. Nur dann kann der Abstand von p zu einem Punkt der rechten Teilinstanz kleiner $d^\star$ sein, und somit bleiben alle möglichen Optima berücksichtigt.

Die erste while-Schleife (Zeilen 13–14) durchläuft die Punkte der rechten Teilinstanz, bis der erste Punkt q erreicht ist, für den $p.y - q.y < d^\star$. Für frühere Punkte kann der Abstand zu p nicht kleiner $d^\star$ sein und muss daher nicht berechnet werden. Zu beachten ist, dass die Suche nach q nicht in jeder Iteration der for-Schleife vom untersten Punkt der rechten Teilinstanz beginnen muss. Durch die Sortierungen der Felder nach y-Koordinate ist gewährleistet, dass der Index k des ersten relevanten

Punktes q mit größer werdendem Index i von p nicht kleiner werden kann. Somit muss k nur einmal initialisiert werden (Zeile 10) und wird danach nur noch inkrementiert (Zeile 14).

Ausgehend von q wird in der zweiten while-Schleife (Zeilen 16–18) jeder Punkt der rechten Teilinstanz untersucht (das heißt, sein Abstand zu p wird mit dem aktuellen Optimum d verglichen und gegebenenfalls als neues Optimum festgehalten), bis ein Punkt r mit $r.y - p.y > d^\star$ erreicht ist. Spätere Punkte können aufgrund ihrer y-Abstände zu p keine optimale Lösung mit p bilden. Somit werden alle möglichen Optima berücksichtigt. Der Algorithmus ist exakt.

Algorithmus 4: NächstesPaarTeileUndHerrsche($P, \ell = 1, r = P.length$)

Eingabe: Feld $P[1..n]$ mit Punkten in $\mathbb{R}^2$ (Punkt p hat Koordinaten $(p.x, p.y)$); Zahlen $\ell, r \in \mathbb{Z}_{>0}$

Ausgabe: Minimaler Abstand d von zwei Punkten im Teilfeld $P[\ell..r]$

```
1  if r − ℓ ≤ 2 then
2      return NächstesPaarRoheGewalt(P[ℓ..r])
3  Sortiere P[ℓ..r] aufsteigend nach x-Koordinate.
4  m = ⌊(ℓ + r)/2⌋
5  x_m = P[m].x
6  d_links = NächstesPaarTeileUndHerrsche(P, ℓ, m)
7  d_rechts = NächstesPaarTeileUndHerrsche(P, m + 1, r)
8  d = d* = min{d_links, d_rechts}
9  Sortiere P[ℓ..m] aufsteigend nach y-Koordinate.
10 Sortiere P[(m + 1)..r] aufsteigend nach y-Koordinate.
11 k = m + 1
12 for i = ℓ to m do
13     if x_m − P[i].x < d* then
14         while k ≤ r and P[i].y − P[k].y > d* do
15             k = k + 1
16         j = k
17         while j ≤ r and P[j].y − P[i].y < d* do
18             d = min{d, ‖P[i] − P[j]‖}
19             j = j + 1
20 return d
```

Zur Abschätzung der Laufzeit von Algorithmus 4 unterscheiden wir den Aufwand für die rekursiven Aufrufe und den restlichen Aufwand. Sei $T(n)$ der Gesamtaufwand, wobei $n = P.length = r - \ell + 1 > 3$. Dann ist der Aufwand für die zwei rekursiven Aufrufe (aufgrund der Teilung der Instanz in zwei gleich große Instanzen) ungefähr $2 \cdot T(n/2)$.

Zur Abschätzung des Restaufwands stellen wir fest, dass der Laufindex i und der Laufindex k höchstens $O(n)$-mal inkrementiert werden. Ferner gilt, dass für jedes $j \in \{m+1,\ldots,r\}$ der Punkt $P[j]$ in Abstandsmessungen zu höchstens sechs verschiedenen Punkten der linken Teilinstanz Berücksichtigung findet. Der Grund dafür ist, dass der Abstand zwischen jedem Paar dieser Punkte mindestens $d^\star$ ist, jeder der Punkte höchstens Abstand $d^\star$ von der Trenngeraden hat und jeder der Punkte höchstens vertikalen Abstand $d^\star$ von $P[j]$ hat; siehe Abb. 7. Es ist einfach zu sehen, dass diese Bedingungen von höchstens sechs Punkten der linken Teilinstanz erfüllt sein können. Somit wird auch die zweite while-Schleife insgesamt $O(n)$-mal ausgeführt. Zusammengefasst bedeutet dies, dass der restliche Aufwand durch die drei Aufrufe des Sortieralgorithmus dominiert wird und sich zu $\Theta(n \log n)$ ergibt, zum Beispiel wenn die Sortierung mit MergeSort [6] erfolgt – das ist ein Sortieralgorithmus, der wie Algorithmus 4 auf Teile und Herrsche beruht.

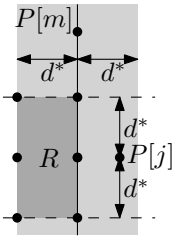

Abb. 7 Nur für Punkte im Rechteck $R = [P[m].x - d^\star, P[m].x] \times [P[j].y - d^\star, P[j].y + d^\star]$ wird die Distanz zu Punkt $P[j]$ berechnet. In R können höchstens sechs Punkte liegen, deren Abstände mindestens $d^\star$ sind: in den vier Ecken und den Mittelpunkten der vertikalen Seiten.

In der Summe aus Rekursions- und Restaufwand ergibt sich die Laufzeit zu

$$T(n) = \begin{cases} \Theta(1) & \text{für } n < 3\,, \\ 2 \cdot T(n/2) + \Theta(n \log n) & \text{sonst.} \end{cases} \tag{4}$$

Die Lösung derartiger *Rekursionsgleichungen* oder *Rekurrenzen* ist oft für die Laufzeitanalyse von rekursiven Algorithmen erforderlich. In unserem Fall ergibt sich die Lösung beispielsweise nach der Rekursionsbaummethode [6] zu $T(n) = \Theta(n \log^2 n)$. Algorithmus 4 ist also asymptotisch schneller als Algorithmus 1, denn $\log^2 n$ wächst viel langsamer als n.

Die Laufzeit von Algorithmus 4 lässt sich zu $\Theta(n \log n)$ verbessern, indem die Punkte in zwei Feldern X und Y vorgehalten werden, von denen das eine nach x-Koordinate und das andere nach y-Koordinate sortiert ist. Diese Sortierungen werden einmal zu Beginn des Algorithmus hergestellt. Im Teile-Schritt wird das Feld Y umsortiert, so dass $Y[\ell..m]$ bzw. $Y[(m+1)..r]$ die Punkte von $Y[\ell..r]$ links bzw. rechts der Trenngeraden enthält, jedoch die Sortierung nach y-Koordinaten in jedem der beiden Teilfelder gewährleistet bleibt. Da diese Umsortierung in $\Theta(n)$ Zeit vollzogen werden kann ($n = r - \ell + 1$), verändert sich die zweite Zeile der Rekur-

sionsgleichung (4) zu $T(n) = 2T(n/2) + \Theta(n)$. Die Lösung der neuen Rekursionsgleichung ist $T(n) = \Theta(n \log n)$.

Lässt sich unsere Strategie der geometrischen Zerteilung einer Probleminstanz verallgemeinern? Natürlich kann man einen geographischen Datensatz räumlich teilen, die Daten jedes Teils für sich prozessieren und (gegebenenfalls) die Nahtstellen einer speziellen Behandlung unterziehen. Ob sich dadurch ein exakter Algorithmus ergibt, muss aber in jedem Einzelfall genau untersucht werden.

4 Lösungsansätze für $\mathcal{NP}$-schwere Probleme

Stellen wir uns vor, dass eine Restaurantkette von der Insolvenz bedroht ist. Nach eingehenden Untersuchungen zur Ursache der Misere kommt die Restaurantleitung zu der Einsicht, dass ein Restaurant genau dann Gewinne bringt, wenn sich im Abstand von weniger als z. B. $D = 2\,\text{km}$ kein weiteres Restaurant befindet. Da diese Bedingung in der aktuellen Situation mehrfach verletzt ist, müssen einige Restaurants geschlossen werden. Das Ziel der Restaurantleitung besteht darin, möglichst wenige Restaurants zu schließen, so dass das Abstandskriterium erfüllt ist.

Offensichtlich ist das Abstandskriterium erfüllt, wenn wir für jedes Restaurant p einen Kreis mit Radius $D/2$ und Zentrum in p konstruieren können und sich dabei keine zwei Kreise schneiden. Zur Vereinfachung nehmen wir $D/2 = 1$ an, womit sich der Restaurantkette das folgende Problem stellt:

Problem 4 (Restaurantauswahl). Gegeben sei eine Menge C von n offenen Kreisen (d. h. wir zählen den Rand eines Kreises nicht zum Kreis dazu) mit Radius 1 in der Ebene. Finde eine maximale unabhängige Teilmenge $C' \subseteq C$, d. h. eine möglichst große Teilmenge von C, in der sich keine zwei verschiedenen Kreise schneiden.

Abbildung 8 zeigt eine Instanz dieses Problems mit einer optimalen Lösung.

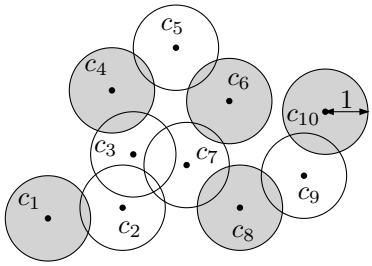

Abb. 8 Eine Instanz zu Problem 4. Die grau gefüllten Kreise bilden eine optimale Lösung.

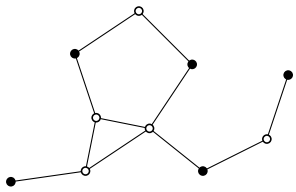

Abb. 9 Die Instanz aus Abb. 8 als Graph. Die Lösung (schwarze Knoten) entspricht einer maximalen unabhängigen Menge.

Das Problem der Restaurantkette (Problem 4) ist offensichtlich ein Spezialfall von Problem 2, wie es auch schon das Problem der Mitfahrzentrale (Problem 3)

war. So können wir einen Graphen $G = (V, E)$ berechnen, dessen Knotenmenge V ein Element für jeden Kreis in C enthält und dessen Kantenmenge E genau dann eine Kante $\{u, v\}$ zwischen zwei Knoten $u, v \in V$ enthält, wenn die Kreise für u und v einen Schnitt aufweisen; siehe Abb. 9. Eine optimale Lösung zum Problem der Restaurantkette entspricht dann einer maximalen unabhängigen Menge von G.

Unser effizienter Algorithmus für Problem 3 wirft natürlich die Frage auf, ob auch Problem 4 (z. B. aufgrund seiner geometrischen Struktur) wesentlich leichter zu lösen ist als das allgemeine Problem, eine maximale unabhängige Menge in einem Graphen zu berechnen. (Man beachte, dass Problem 3 genau die gewichtete eindimensionale Version des zweidimensionalen Problems 4 ist. Anders als im Eindimensionalen trennt hier ein ausgewähltes Objekt die Instanz jedoch *nicht* in unabhängige Teilinstanzen.)

Insofern ist es nicht verwunderlich, dass man zeigen konnte [5], dass die Berechnung einer maximalen unabhängigen Menge auch dann $\mathcal{NP}$-schwer ist, wenn der gegebene Graph, wie in unserem Fall, aus den Schnittbeziehungen zwischen Einheitskreisen herrührt. Es ist auch $\mathcal{NP}$-schwer zu entscheiden, ob sich ein gegebener Graph als Schnittgraph von Einheitskreisen darstellen lässt [3]. Und selbst, wenn die geometrische Darstellung gegeben ist, ist es $\mathcal{NP}$-schwer eine maximale unabhängige Menge zu finden [9].

Beim Problem der Restaurantkette handelt es sich also um eine wirklich harte Nuss, die besondere Mittel erfordert. In diesem Abschnitt stellen wir Lösungsansätze durch mathematische Programmierung (Abschnitte 4.1–4.3), Heuristiken (Abschnitte 4.4–4.5) und Approximationsalgorithmen (Abschnitt 4.6) vor.

4.1 Mathematische Programmierung

Ein genereller Ansatz zur Lösung von Problemen der kombinatorischen Optimierung ist die *mathematische Programmierung*. Dabei verweist der Name „Programmierung“ nicht auf die Implementierung von Computerprogrammen, sondern auf die Modellierung eines Problems in einer bestimmten Form, als *mathematisches Programm*. Im Allgemeinen besteht ein mathematisches Programm aus einer Zielfunktion und einer Menge von Bedingungen (Gleichungen oder Ungleichungen) über eine Menge von Variablen. Das Ziel besteht darin, eine Variablenbelegung zu finden, die alle Bedingungen erfüllt und hinsichtlich der Zielfunktion optimal ist.

Ein in der Geodäsie besonders häufig gewählter Ansatz, den man der mathematischen Programmierung zuordnen kann, ist die *Ausgleichungsrechnung* [20]. Dabei ist ein Vektor $\ell \in \mathbb{R}^n$ von Beobachtungen (z. B. Entfernungsmessungen) gegeben. Mit diesen soll ein Vektor $x \in \mathbb{R}^u$ von Unbekannten (z. B. Koordinaten eines Punktes) geschätzt werden. Im einfachsten Fall der Ausgleichungsrechnung lässt sich der Zusammenhang zwischen ℓ und x mithilfe einer $n \times u$-Matrix A und durch ein System von Bedingungsgleichungen in der Form $\ell = Ax$ ausdrücken, wobei die Anzahl n der Beobachtungen in der Regel größer als die Anzahl u der Unbekannten ist. Da das Gleichungssystem für $n > u$ im Allgemeinen unlösbar ist, führt man einen

Vektor $v \in \mathbb{R}^n$ mit zusätzlichen Variablen (den Verbesserungen) ein und ändert das Gleichungssystem zu $\ell + v = Ax$. (Die ursprünglichen Bedingungen werden somit aufgeweicht.) Das neue Gleichungssystem gilt es nun durch die Wahl der Vektoren $v \in \mathbb{R}^n$ und $x \in \mathbb{R}^u$ zu erfüllen. Unter allen zulässigen Wahlmöglichkeiten für v und x ist dabei eine gesucht, die das Produkt (bzw. die Zielfunktion) $v^{\mathrm{T}}v$ minimiert. Damit wird bezweckt, dass die Verbesserungen klein bleiben – die Beobachtungen werden nur so viel wie nötig *ausgeglichen*. Eine optimale Lösung für x ergibt sich durch Lösung des linearen Gleichungssystems $A^{\mathrm{T}}Ax = A^{\mathrm{T}}\ell$. Es folgt $v = Ax - \ell$.

Im Allgemeinen werden mathematische Programme nach der Art der Zielfunktion und der Bedingungen sowie nach dem Wertebereich der Variablen klassifiziert. Von besonderer Bedeutung für die kombinatorische Optimierung sind *lineare Programme* und *ganzzahlige lineare Programme*, da sie sich zur Modellierung vieler Probleme eignen. Anders als bei der Ausgleichungsrechnung lassen sich optimale Lösungen für (ganzzahlige) lineare Programme jedoch nicht durch die Lösung eines linearen Gleichungssystems berechnen. Allerdings gibt es auch für sie geeignete Lösungsalgorithmen, die in sehr hochentwickelten Computerprogrammen implementiert sind. Derartige Computerprogramme werden auch als *Problemlöser* bezeichnet.

In einem linearen Programm (linear program, LP) bestehen die Zielfunktion und die Bedingungen ausschließlich aus Termen, die linear von jeweils einer der Variablen abhängen oder konstant sind. Die Bedingungen liegen in Form von Ungleichungen vor. Die Variablen sind kontinuierlich, das heißt, sie dürfen jeden reellen Wert annehmen. Das folgende Beispiel erfüllt die Definition eines LPs:

$$\begin{aligned} \text{Minimiere} \quad & x_1 + 2x_2 && (5)\\ \text{unter den Bedingungen} \quad & 16x_1 + 12x_2 \geq 35 && (6)\\ & 16x_1 - 12x_2 \leq 17 && (7)\\ & x_1, x_2 \in \mathbb{R}\,. && (8) \end{aligned}$$

Ein ganzzahliges lineares Programm (integer linear program, ILP) unterscheidet sich von einem LP nur darin, dass die Variablen ausschließlich ganzzahlige Werte annehmen dürfen. So wird aus unserem Beispiel ein ILP, indem wir (8) durch

$$x_1, x_2 \in \mathbb{Z} \tag{9}$$

ersetzen.

Abbildung 10 stellt unser LP aus Formeln (5)–(8) geometrisch dar. Dabei wird jede Lösung als ein Punkt in der Ebene aufgefasst, indem die Werte von x_1 und x_2 auf jeweils einer Achse aufgetragen werden. Jede Bedingung des LPs beschränkt die Lösungsmenge (das ist die Menge aller zulässiger Lösungen) auf eine Halbebene, so dass die Lösungsmenge gleich der Schnittmenge S aller Halbebenen ist (schattierte Fläche in Abb. 10). Unter allen Punkten in S sind wir an einem Punkt interessiert, der die Zielfunktion minimiert. Um solch einen Punkt in der Abbildung zu ermitteln, kombinieren wir die Koeffizienten der Variablen in der Zielfunktion zu einem Vektor c, in unserem Fall $c = \binom{1}{2}$. Alle Punkte auf einer Geraden ℓ, die or-

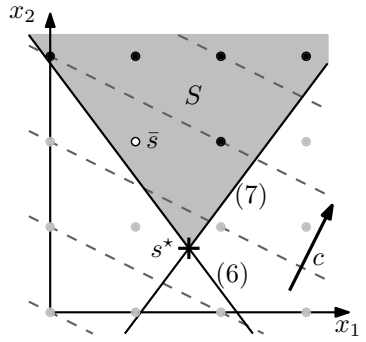

Abb. 10 Die optimale Lösung $s^\star$ des LPs aus (5)–(8) und die optimale Lösung $\bar{s}$ des ILPs aus (5)–(7) und (9).

thogonal zu c ist, sind hinsichtlich der Zielfunktion gleichwertig. Konstruieren wir eine derartige Gerade ℓ und bewegen sie über die Ebene entlang der Richtung von c, so stellt der Punkt in S, der als erstes von ℓ berührt wird, eine optimale Lösung dar. Die gestrichelten Linien in Abb. 10 veranschaulichen verschiedene Positionen von ℓ während dieser Bewegung. Wir erzielen die optimale Lösung $s^\star = \binom{1{,}625}{0{,}75}$.

Betrachten wir nun unser ILP bestehend aus Formeln (5)–(7) und (9). Durch Forderung von $x_1, x_2 \in \mathbb{Z}$ wird $s^\star$ unzulässig. Die neue optimale Lösung ist $\bar{s} = \binom{1}{2}$.

Trotz der Ähnlichkeit des LPs und des ILPs unterscheiden sich ihre Lösungen $s^\star$ und $\bar{s}$ deutlich voneinander. Insbesondere erhält man durch Runden der Komponenten von $s^\star$ keine zulässige Lösung, geschweige denn die Lösung $\bar{s}$ des ILPs. Tatsächlich hat der unterschiedliche Wertebereich der Variablen eine gravierende Konsequenz: Während LPs z. B. mit dem Algorithmus von Karmarkar [15] effizient gelöst werden können, ist die Lösung von ILPs $\mathcal{NP}$-schwer. Oft werden LPs mit dem Simplex-Algorithmus [7] gelöst, der im Gegensatz zu Karmarkars Algorithmus eine exponentielle Worst-Case-Laufzeit hat, aber in der Praxis effizient ist.

Es wird uns höchstwahrscheinlich nicht gelingen, ein $\mathcal{NP}$-schweres Problem in die Form eines LPs mit einer sinnvollen (d. h. polynomiellen) Größe zu bringen – sonst hätten wir mithilfe Karmarkars Algorithmus eine effiziente Lösung für das $\mathcal{NP}$-schwere Problem gefunden (und damit $\mathcal{P} = \mathcal{NP}$ gezeigt). Die *ganzzahlige* lineare Programmierung stellt dagegen ein durchaus vielversprechendes Mittel zur Lösung $\mathcal{NP}$-schwerer Probleme dar. Trotz der exponentiellen Worst-Case-Laufzeit, die alle bekannten Algorithmen zur Lösung von ILPs haben, lassen sich viele ILPs in der Praxis in vernünftiger Zeit lösen.

4.2 Ein einfaches ganzzahliges lineares Programm zur Berechnung maximaler unabhängiger Mengen

Bevor wir den am weitesten verbreiteten Lösungsansatz für ILPs diskutieren, soll die Anwendbarkeit der ganzzahligen linearen Programmierung anhand von Problem 4 demonstriert werden. Hierfür definieren wir eine Variable

$$x_c \in \{0,1\} \quad \text{für jeden Kreis } c \in C\,, \tag{10}$$

wobei $x_c = 1$ bedeutet, dass ein Kreis c in der Menge C' der ausgewählten Kreise enthalten ist. Das Ziel, möglichst viele Kreise auszuwählen, wird damit zu:

$$\text{Maximiere} \sum_{c \in C} x_c \tag{11}$$

Ohne weitere Bedingungen würde die Zielfunktion natürlich dazu führen, dass alle Kreise ausgewählt würden, also für jeden Kreis $c \in C$ die Variable x_c auf 1 gesetzt würde. Dies wäre jedoch im Allgemeinen keine zulässige Lösung im Sinne der Problemdefinition. Wir stellen die Zulässigkeit einer Lösung sicher, indem wir für jedes Paar (a,b) sich schneidender Kreise in C die folgende ganz einfache Bedingung aufstellen:

$$x_a + x_b \leq 1 \quad \text{für alle } a,b \in C \text{ mit } a \neq b \text{ und } a \cap b \neq \emptyset \tag{12}$$

Also dürfen a und b nicht gleichzeitig für eine Lösung ausgewählt werden. Bedingung (12) wird im schlechtesten Fall $\Theta(n^2)$-mal erzeugt, denn es gibt $\Theta(n^2)$ Paare von Kreisen und jedes Paar kann einen Schnitt aufweisen.

Binäre Variablen eignen sich inbesondere, um zwischen zwei möglichen Alternativen (z. B. Selektion oder keine Selektion eines Kreises) zu unterscheiden, und werden daher besonders oft in ILPs verwendet. Eine binäre Variable $x \in \{0,1\}$ kann jedoch als gewöhnliche ganzzahlige Variable $x \in \mathbb{Z}$ mit Bedingungen $0 \leq x$ und $x \leq 1$ interpretiert werden.

4.3 Lösung ganzzahliger linearer Programme

Zur Lösung ganzzahliger linearer Programme wird in der Regel *Branch-and-Cut* eingesetzt, welches eine Kombination von *Branch-and-Bound* und *Schnittebenenverfahren* darstellt. Wir beschreiben beide Verfahren und ihre Kombination zu Branch-and-Cut und gehen auf Implikationen für die Praxis ein.

Sowohl Branch-and-Bound als auch Schnittebenenverfahren beginnen die Lösung eines ILPs mit der Lösung seiner *LP-Relaxierung*, in der reelle Werte für die ganzzahlligen Variablen des ILPs zugelassen werden. In unserem Beispiel aus Abschnitt 4.1 heißt das, dass $x_1, x_2 \in \mathbb{Z}$ zu $x_1, x_2 \in \mathbb{R}$ relaxiert (aufgeweicht) wird. Das Wort „Branch“ (Verzweigen) in Branch-and-Bound verweist darauf, dass eine Variable x_i mit nicht ganzzahligem Wert $x_i^\star$ in der Lösung $s^\star$ der LP-Relaxierung

ausgewählt und die ursprüngliche Probleminstanz in zwei Instanzen aufgeteilt wird. Jede der beiden neuen Instanzen erbt dabei alle Bedingungen der ursprünglichen Instanz und erhält zusätzlich eine der Bedingungen $x_i \leq \lfloor x_i^\star \rfloor$ und $x_i \geq \lceil x_i^\star \rceil$. Das heißt, der Wert $x_i^\star$ wird ab- bzw. aufgerundet. In unserem Beispiel selektieren wir x_1 (mit Wert 1,625) zur Verzweigung und generieren zwei neue Instanzen, jeweils durch Hinzufügen einer der Bedingungen

$$x_1 \leq 1 \tag{13}$$

$$x_1 \geq 2, \tag{14}$$

wie es in Abb. 11 dargestellt ist. Die Verzweigung kann rekursiv auf Teilinstanzen angewandt werden. Somit entsteht ein Baum, in dem jeder Knoten einem ILP entspricht. Das Verfahren terminiert, sobald für jedes Blatt des Baums die optimale Lösung der LP-Relaxierung ganzzahlig ist oder die Lösungsmenge der LP-Relaxierung leer ist. Die beste ganzzahlige Lösung stellt eine optimale Lösung für das ursprüngliche ILP dar.

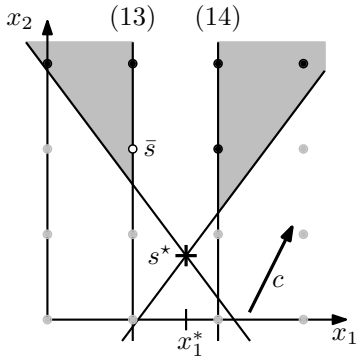

Abb. 11 Durch Verzweigung (Branching) wird ein ILP in zwei ILPs zerteilt.

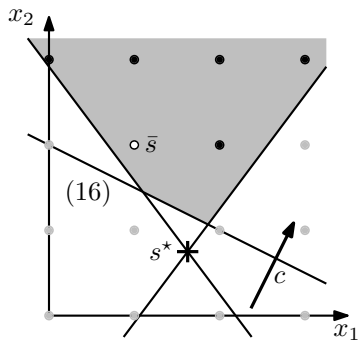

Abb. 12 Eine Schnittebene entfernt die optimale Lösung der LP-Relaxierung, ohne eine ganzzahlige Lösung auszuschließen.

Das Wort „Bound“ in Branch-and-Bound verweist darauf, dass in jedem Knoten des Branch-and-Bound-Baums für das entsprechende ILP eine Schranke berechnet wird. Bei einem Minimierungsproblem wird in jedem Knoten eine untere Schranke berechnet; das ist ein Wert, der niemals größer ist als der Zielfunktionswert einer optimalen Lösung für das entsprechende ILP. In der Regel wird die Lösung der LP-Relaxierung als Schranke verwendet. Die LP-Relaxierung eines ILPs liefert im Falle eines Minimimierungsproblems tatsächlich eine untere Schranke, denn die Lösungsmenge eines ILPs ist in der Lösungsmenge seiner LP-Relaxierung enthalten. Dies kann ausgenutzt werden, um einige Äste des Branch-and-Bound-Baums vorzeitig abzuschließen und den Baum somit klein zu halten. Nehmen wir an, dass wir in einem Knoten des Branch-and-Bound-Baums eine ganzzahlige Lösung mit Zielfunktionswert z finden. Dann kann die Rekursion in jedem Knoten v abgebro-

chen werden, für den eine untere Schranke $L \geq z$ vorliegt. Der Grund dafür ist, dass es im Teilbaum mit Wurzel v keine Lösung geben kann, die einen Zielfunktionwert besser als z erreicht. Der Erfolg dieses Verfahrens hängt offensichtlich davon ab, wie ähnlich sich die Zielfunktionswerte der optimalen Lösung eines ILPs und der optimalen Lösung seiner LP-Relaxierung sind. Viele ILP-Formulierungen sind leider sehr *schwach*, d. h., die LP-Relaxierung erlaubt (gemessen an der Zielfunktion) oft wesentlich bessere Lösungen als das zugehörige ILP.

Betrachten wir zum Beispiel unser ILP aus Formeln (10)–(12) für das Problem der Restaurantkette. Nehmen wir eine Instanz der Größe $n > 1$, in der jeder Kreis jeden anderen Kreis schneidet. In diesem Fall können wir höchstens einen Kreis für die Lösung auswählen; entsprechend hat die Zielfunktion f für eine optimale Lösung $\bar{s}$ des ILPs den Wert $f(\bar{s}) = 1$. In der LP-Relaxierung, die kontinuierliche (also nicht-ganzzahlige) Werte erlaubt, können wir jedoch jede der n Variablen auf 0,5 setzen, ohne Bedingung (12) zu verletzten. Somit hat das Optimum $s^\star$ der LP-Relaxierung den Zielfunktionswert $f(s^\star) = n/2$, was für große n darauf hinausläuft, dass $f(s^\star)$ im Verhältnis zu $f(\bar{s})$ extrem groß wird und somit ein schlechter Schätzwert für $f(\bar{s})$ ist.

Für Problem 4 können wir eine stärkere ILP-Formulierung gewinnen, indem wir das Arrangement A betrachten, das durch die Überlagerung aller Kreise einer gegebenen Instanz gebildet wird. Beispielsweise bilden in Abb. 8 alle Punkte der Ebene, die sowohl in c_2, c_3 und c_7, aber in keinem anderen Kreis enthalten sind, eine Facette. Diese wird durch drei Kreisbögen begrenzt, nämlich durch jeweils einen Teil des Rands von c_2, c_3 und c_7. Für jede Facette F von A definieren wir eine Bedingung, die dafür sorgt, dass höchstens ein Kreis selektiert wird, der F enthält:

$$\sum_{c \in C_F} x_c \leq 1 \quad \text{für jede Facette } F \text{ von } A, \tag{15}$$

wobei C_F die Menge der Kreise ist, die F enthalten. Für die Facette, die der Schnittmenge von c_2, c_3 und c_7 entspricht, wird Bedingung (15) zu $x_{c_2} + x_{c_3} + x_{c_7} \leq 1$. Es ist damit nicht mehr möglich, in der LP-Relaxierung alle Variablen auf 0,5 zu setzen. Für realistische Instanzen liefert die LP-Relaxierung somit tatsächlich bessere Schätzungen des ganzzahligen Optimums. Für die Instanz in Abb. 13 mit $n = 2500$ hatte die LP-Relaxierung mit Bedingung (15) eine Lösung mit Zielfunktionswert 467,28. Dieses kommt dem Optimum des ILPs in Höhe von 453 sehr nahe, wenn man bedenkt, dass die LP-Relaxierung ohne Bedingung (15) eine Lösung mit Zielfunktionswert $2500/2 = 1250$ hat.

Übrigens lassen sich die Facetten eines Arrangements von n Kreisen effizient mit einem Sweep-Line-Algorithmus [2] berechnen, und es lässt sich leicht zeigen, dass ein derartiges Arrangement nur $O(n^2)$ Facetten hat: Der Rand eines Kreises wird durch Schnitte mit anderen Kreisen in höchstens $2(n-1)$ Kreisbögen zerteilt. Es gibt also nicht mehr als $n \cdot 2(n-1)$ bzw. $O(n^2)$ Grenzen zwischen verschiedenen Facetten – und somit auch nur $O(n^2)$ Facetten. Das ILP mit Bedingung (15) hat demnach eine überschaubare Größe; wie das ILP aus (5)–(7) und (9) hat es $\Theta(n^2)$ Bedingungen. Zudem können einige Bedingungen weggelassen werden, da sie von

anderen Bedingungen dominiert werden. Die Bedingung $x_{c_2} + x_{c_3} \leq 1$ ist z. B. durch $x_{c_2} + x_{c_3} + x_{c_7} \leq 1$ sichergestellt und muss daher nicht zusätzlich bedacht werden.

Bei der Lösung mit Branch-and-Bound (ohne Kombination mit Schnittebenenverfahren) führt die stärkere ILP-Formulierung zu einer deutlichen Reduktion der Laufzeit. Dies zeigte ein Test mit 50 zufällig erzeugten Instanzen mit 2500 Kreisen. Eine dieser Instanzen ist in Abb. 13 zu sehen. Die durchschnittliche Laufzeit wurde durch Ersetzen von Bedingung (12) durch (15) von 1024 auf 245 Sekunden verbessert; siehe Tabelle 1 (Spalten 2–3).

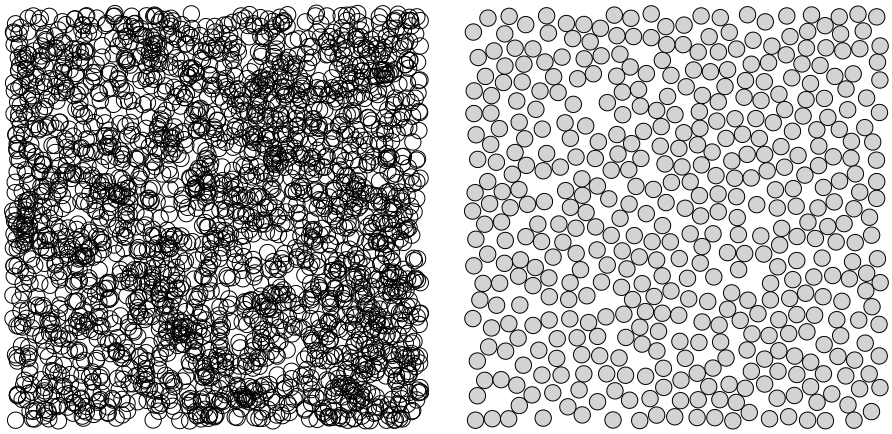

Abb. 13 Eine zufällig generierte Instanz zu Problem 4 aus 2500 Kreisen (links) und eine optimale Lösung mit 453 Kreisen (rechts).

Erfreulicherweise können gute Problemlöser schwache ILP-Formulierungen oft automatisch verstärken. Dazu wird Branch-and-Bound in der Regel mit Schnittebenenverfahren kombiniert. Eine Schnittebene ist eine zusätzliche Bedingung in Form einer Ungleichung, welche die optimale Lösung der LP-Relaxierung eines ILPs ausschließt, ohne eine zulässige ganzzahlige Lösung auszuschließen. In unserem ILP aus (5)–(7) und (9) stellt die Ungleichung

$$x_1 + 2x_2 \geq 4, \tag{16}$$

eine solche Schnittebene dar; siehe Abb. 12. Diese lässt sich mit dem Verfahren von Gomory [11] berechnen.

Schnittebenenverfahren können ILPs ganz allein (ohne Branch-and-Bound) lösen, indem sie iterativ die LP-Relaxierung lösen, eine Schnittebene finden und sie dem ILP hinzufügen, bis die LP-Relaxierung die Ganzzahligkeitsbedingung erfüllt. In diesem Fall ist eine optimale Lösung für das ursprüngliche ILP gefunden. Dieser Ansatz führt in der Regel jedoch erst nach sehr vielen Iteration zu einer optimalen ganzzahligen Lösung. Durch das Hinzufügen einer vergleichsweise kleinen Zahl von Schnittebenen kann eine ILP-Formulierung jedoch oft bereits entschieden verstärkt werden. Daher werden Schnittebenenverfahren und Branch-and-Bound oft

kombiniert, zu Branch-and-Cut. Die heutzutage erfolgreichsten Problemlöser für ganzzahlige Programme wie Gurobi und CPLEX verfolgen diese Strategie.

In unserem Test ergab sich mit Branch-und-Cut eine deutliche Reduktion der Laufzeiten im Vergleich zu Branch-and-Bound. Instanzen von 2500 Kreisen konnten wir im Mittel in weniger als 10 Sekunden lösen; siehe Tabelle 1 (Spalten 4–5). Durch die Verstärkung der ILPs mit automatisch erzeugten Schnittebenen fällt der Vorteil unseres starken ILPs (mit Bedingung (15) statt (12)) allerdings kaum noch ins Gewicht. Nichtsdestoweniger ist es in der Praxis oft wichtig, eine starke ILP-Formulierung zu finden. Außerdem sind ILP-Formulierungen vorzuziehen, die mit wenigen Bedingungen und Variablen auskommen. In der Regel ist die Stärke eines ILPs jedoch wichtiger als seine geringe Größe.

Tabelle 1 Durchschnittliche Laufzeiten (in Sekunden) des Problemlösers Gurobi für 50 zufällig erzeugte Instanzen, ähnlich zu der in Abb. 13. Jede Instanz enthält 2500 Einheitskreise, deren Mittelpunkte im Quadrat $[0,50] \times [0,50]$ liegen. Die Tests erfolgten auf einem Windows PC mit 8 GB Arbeitsspeicher und einem Intel-Prozessor mit einer Taktfrequenz von 3,40 GHz.

Algorithmus	Branch+Bound (ohne Schnittebenen)		Branch+Cut (mit Schnittebenen)	
ILP: (10) & (11) &	(12)	(15)	(12)	(15)
Laufzeit (in Sek.)	1024	245	9,8	9,6

4.4 Eine einfache Heuristik

Nicht jeder Benutzer hat Zugriff auf einen kommerziellen Problemlöser für ILPs – oder nicht die Zeit und den Speicherplatz, die nötig wären, um auch größere Instanzen exakt zu lösen. Dann liegt es nahe sein Glück mit einer *Heuristik* zu versuchen, also mit einem Algorithmus, der keine Aussage über die Qualität seiner Lösungen trifft. In diesem Fall empfiehlt es sich jedoch, parallel zur Heuristik eben doch einen langsamen, aber exakten Algorithmus (etwa über eine ILP-Formulierung) zu implementieren. So bekommt man wenigstens auf kleinen Probleminstanzen ein Gefühl dafür, wie gut oder schlecht die Heuristik tatsächlich ist.

Beim Problem der Restaurantkette (Problem 4) würde eine ganz einfache Heuristik darin bestehen, dass man wiederholt einen beliebigen Kreis $c \in C$ auswählt, aus C entfernt und in die Ausgabemenge C' steckt. Das funktioniert natürlich nur, wenn man alle Kreise $c' \in C$, die c schneiden, ebenfalls aus C entfernt. Dadurch wird vermieden, dass die Ausgabe (siehe Algorithmus 5) ein Paar sich schneidender Kreise enthält. Also ist die Ausgabe unserer Heuristik zulässig.

Dieser Algorithmus ist übrigens eine gierige Heuristik. Das ist eine gieriger Algorithmus, der anders als z. B. der gierige Algorithmus aus Abschnitt 3.1 keine optimale Lösung garantiert. Wie jeder gierige Algorithmus trifft allerdings auch Algorithmus 5 immer wieder Entscheidungen, die in der momentanen Situation richtig

Algorithmus 5: KreisAuswahlGierig(P)

Eingabe: Menge C mit Einheitskreisen gegeben durch ihre Zentren
Ausgabe: Nicht-erweiterbare unabhängige Menge C' von C
$C' = \emptyset$
while $C \neq \emptyset$ **do**
  Wähle ein beliebiges $c \in C$
  $C' = C' \cup \{c\}$
  foreach $\bar{c} \in C$ **do**
    if $\bar{c} \cap c \neq \emptyset$ **then**
      $C = C \setminus \{\bar{c}\}$
return C'

erscheinen, ohne sich jedoch über die globale Richtigkeit dieser Entscheidungen Gedanken zu machen.

4.5 Lokale Suche

Ein allgemeiner heuristischer Ansatz zur Lösung $\mathcal{NP}$-schwerer Probleme besteht darin, statt einer global optimalen Lösung eine *lokal* optimale Lösung zu berechnen. Grundsätzlich bezeichnet man damit eine Lösung, die in einer festgelegten Nachbarschaft von keiner anderen Lösung hinsichtlich der Zielfunktion übertroffen wird. Betrachten wir beispielsweise eine Lösung s zu einer Instanz C von Problem 4; siehe Abb. 14. Wir definieren die Nachbarschaft $N_1(s)$ als die Menge aller Lösungen für C, die man ausgehend von s erreichen kann, indem man *einen* zusätzlichen Kreis $c \in C$ in s aufnimmt und alle Kreise in s ausschließt, die von c geschnitten werden. Wenn es in $N_1(s)$ keine Lösung $s^\star$ gibt, die mehr Kreise enthält als s, dann heißt s *lokal optimal*. Demnach ist die Lösung von Algorithmus 5 lokal optimal, denn sie kann keinen weiteren Kreis aufnehmen, ohne mindestens einen anderen Kreis auszuschließen.

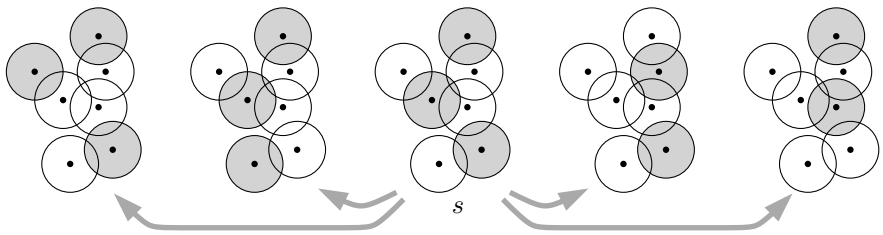

Abb. 14 Eine Instanz von Problem 4 mit einer Lösung s (mittig) und ihren vier Nachbarn in $N_1(s)$. Da keiner der Nachbarn mehr Kreise enthält als s, ist s hinsichtlich N_1 lokal optimal.

Ob wir eine Lösung als lokal optimal bezeichnen können, hängt natürlich stark von der zugrunde gelegten Nachbarschaft ab. Die Definition unserer Nachbarschaft N_1 für Problem 4 lässt sich zum Beispiel folgendermaßen verallgemeinern: $N_k(s)$ ist die Menge aller Lösungen, die man ausgehend von s erreichen kann, indem man höchstens k zusätzliche Kreise in s aufnimmt und alle Kreise in s, die von den zusätzlichen Kreisen geschnitten werden, ausschließt. Algorithmus 5 berechnet zwar eine Lösung, die hinsichtlich der Nachbarschaft N_1 lokal optimal ist, aber für $k > 1$ ist die lokale Optimalität hinsichtlich der (größeren) Nachbarschaft N_k nicht gewährleistet. Es ist durchaus üblich, die Größe einer Nachbarschaft durch einen Parameter k zu steuern. Allgemein ist der Begriff *k-opt-Nachbarschaft* für eine so parametrisierte Nachbarschaft gebräuchlich. Eine Lösung, die hinsichtlich der k-opt-Nachbarschaft lokal optimal ist, bezeichnet man auch als *k-optimal*.

Grundsätzlich lässt sich eine Lösung, die hinsichtlich einer Nachbarschaft N lokal optimal ist, mit Algorithmus 6 berechnen. Dieser sucht ausgehend von einer zulässigen Startlösung s nach der besten Lösung s' in der Nachbarschaft $N(s)$. Ist s' besser als s, so setzt der Algorithmus die Suche ausgehend von s' fort; ansonsten gibt er die aktuelle Lösung s zurück. Im Beispiel von Problem 4 könnte man die leere Menge als Startlösung nehmen. Eine bessere Startlösung könnte man dagegen mit Algorithmus 5 berechnen – die lokale Suche würde dann der Nachbesserung dienen. Die Nachbarschaft $N_k(s)$ einer Lösung s hat in diesem Beispiel Größe $O(n^k)$, daher kann die beste Lösung in der Nachbarschaft von s für kleine k effizient durch explizite Enumeration berechnet werden. Die while-Schleife wird $O(n)$-mal durchlaufen, denn mit jeder Iteration stellt sich eine Verbesserung um mindestens 1 ein und eine Lösung kann höchstens alle n Kreise der Eingabe enthalten.

Algorithmus 6: Lokale Suche

```
s = eine zulässige Lösung
while true do
    s' = beste Lösung in N(s)
    if s' ist besser als s then
        s = s'
    else
        return s
```

Algorithmus 6 lässt sich auf vielfältige Weise anpassen und variieren, um für ein spezielles Problem eine gute Heuristik zu gewinnen. Beispielsweise wird die Suche nach der besten Lösung in der Nachbarschaft $N(s)$ der aktuellen Lösung s oft vermieden. Stattdessen wird eine zufällige Lösung s' in $N(s)$ generiert und mit der aktuellen Lösung s verglichen. Ist s' besser als s, so wird s' als neue aktuelle Lösung akzeptiert. Ansonsten wird in der Nachbarschaft von s weiter nach einer besseren Lösung gesucht, bis ein Abbruchkriterium (z. B. eine festgelegte Anzahl an Iterationen) erreicht ist.

Eine besonders häufig verwende Variante von Algorithmus 6 bezeichnet man als *Simulated Annealing* [16], was simuliertes Ausglühen bedeutet und mit der Inspiration des Algorithmus aus der Metallverarbeitung zusammenhängt, in der Stähle durch langsame Abkühlungsprozesse erzeugt werden. Beim Simulated Annealing wird die Suche unter bestimmten Umständen und abhängig vom Zufall auch dann von s' aus fortsetzt, wenn s' schlechter als s ist. Damit soll vermieden werden, dass der Algorithmus in einem schlechten lokalen Optimum stecken bleibt. Die Wahrscheinlichkeit der Akzeptanz von s' als neue aktuelle Lösung wird vom Ausmaß der Verschlechterung sowie einem Temperaturparamter T abhängig gemacht. Analog zur Metallverarbeitung wird T mit zunehmender Anzahl an Iterationen verringert. Dadurch nimmt im Lauf der Zeit die Wahrscheinlichkeit ab, dass Verschlechterungen akzeptiert werden. Der Algorithmus wird in der Regel nach einer vorgegebenen Anzahl an Iterationen terminiert. Michiels et al. [19] diskutieren theoretische Resultate zu Simulated Annealing und anderen lokalen Suchverfahren.

4.6 Approximationsalgorithmen

Auch wenn wir im letzten Abschnitt gezeigt haben, wie man durch geeignetes Nacharbeiten die Lösung einer Heuristik möglicherweise verbessert, bleibt die Frage, wie gut oder schlecht Algorithmus 5 eigentlich ist. Dazu wollen wir die Ausgabe unseres Algorithmus, der ja immer wieder willkürlich einen Kreis auswählt, mit einer optimalen Lösung vergleichen, und zwar für eine beliebige Instanz unseres Problems, also eine Menge von Einheitskreisen in der Ebene. Sei I eine solche Instanz, $\mathsf{ALG}(I)$ die Lösung unseres Algorithmus, $\mathsf{OPT}(I)$ eine optimale Lösung von I und ζ unsere Zielfunktion, die eine Lösung auf eine Zahl abbildet, nämlich die Anzahl der Kreise, aus der die Lösung besteht. Dann interessiert uns das Verhältnis

$$\frac{\zeta(\mathsf{ALG}(I))}{\zeta(\mathsf{OPT}(I))}\,.$$

Natürlich wird dieses Verhältnis – eine Zahl zwischen 0 und 1 – von Instanz zu Instanz verschieden sein. Im besten Fall findet unsere Heuristik das Optimum, z. B. wenn alle Kreise disjunkt sind oder wenn sich alle Kreise paarweise schneiden. Für solche Instanzen ist das Verhältnis genau 1, ansonsten ist es echt kleiner 1.

Wir wollen die Qualität eines Algorithmus durch eine einzige Zahl beschreiben. Um das zu erreichen, nehmen wir (wie bei der Laufzeitanalyse) wieder eine pessimistische Haltung ein und sagen, dass ein Algorithmus die *Approximationsgüte* (oder den *Approximationsfaktor*) γ hat, wenn für *jede* Instanz I unseres Problems gilt, dass

$$\frac{\zeta(\mathsf{ALG}(I))}{\zeta(\mathsf{OPT}(I))} \geq \gamma\,.$$

Wir sind am größtmöglichen Wert von γ interessiert, so dass die Aussage gerade noch zutrifft, also an einer möglichst großen unteren Schranke für das Verhältnis $\zeta(\mathsf{ALG}(I))/\zeta(\mathsf{OPT}(I))$.

Gibt es für unser konkretes Restaurantketten-Problem eine solche Schranke oder kann das Verhältnis für eine Folge „gemein“ gewählter Instanzen beliebig schlecht werden?

Dazu betrachten wir eine beliebige Instanz I und vergleichen eine feste optimale Lösung $\mathsf{OPT}(I)$ mit der Lösung $\mathsf{ALG}(I)$ unserer Heuristik – das ist die Menge C', die Algorithmus 5 schließlich zurückgibt. Jedes Mal, wenn unsere Heuristik einen Kreis c auswählt, gibt es zwei Möglichkeiten. Entweder c liegt auch in der optimalen Lösung; dann haben wir in diesem Schritt alles richtig gemacht. Oder c ist nicht in der optimalen Lösung; dann interessiert uns, wie viele Kreise in $\mathsf{OPT}(I)$ durch unsere Entscheidung für c „blockiert“ werden.

Man beachte, dass es mindestens einen Kreis in $\mathsf{OPT}(I)$ geben muss, der c schneidet (sonst könnten wir $\mathsf{OPT}(I)$ durch c erweitern, was der Optimalität von $\mathsf{OPT}(I)$ widersprechen würde). Insgesamt sind die von c blockierten Kreise eine Teilmenge der Kreise, die in der foreach-Schleife von Algorithmus 5 aus der verbliebenen Kandidatenmenge C entfernt werden, weil sie c schneiden. Die Frage ist also: Wie viele Kreise können gleichzeitig c schneiden und untereinander paarweise disjunkt sein – da sie ja zur optimalen Lösung $\mathsf{OPT}(I)$ gehören. Ausgehend von einer dichten Kreispackung (siehe Abb. 15(a)) macht man sich klar, dass es maximal fünf Kreise gibt, die einerseits alle c schneiden und andererseits paarweise disjunkt sind (wie in dem Beispiel in Abb. 15(b)).

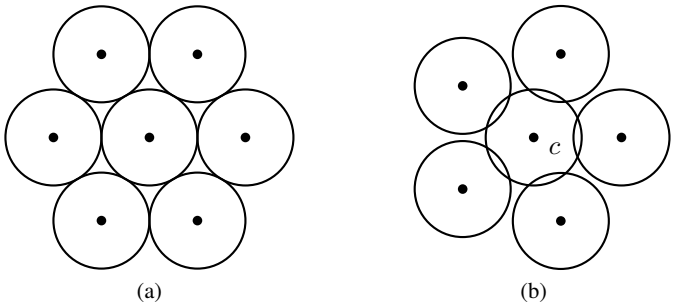

Abb. 15 Dichte Kreispackung (a) und fünf paarweise disjunkte Einheitskreise, die jeweils einen sechsten Einheitskreis schneiden (b).

Jeder Kreis in $\mathsf{ALG}(I)$ „bezahlt“ also für bis zu fünf Kreise in $\mathsf{OPT}(I)$. Wichtig ist, dass wir damit alle Kreise in $\mathsf{OPT}(I)$ bezahlt haben. Am Ende bleibt kein unbezahlter Kreis mehr übrig, denn ein solcher Kreis würde keinen Kreis der Heuristik schneiden und dann hätte unsere Heuristik ihn ebenfalls in ihre Lösung C' gesteckt. Zur Erinnerung: bei unserem Problem gibt die Zielfunktion ζ die Kardinalität einer Lösung an. Es gilt also

$$5\zeta(\mathsf{ALG}(I)) = 5|\mathsf{ALG}(I)| \;\geq\; |\mathsf{OPT}(I)| = \zeta(\mathsf{OPT}(I)).$$

Diese Rechnung können wir für jede Instanz unseres Problems wiederholen; obige Ungleichung ist immer erfüllt. Also gilt für jede Instanz I

$$\frac{\zeta(\mathsf{ALG}(I))}{\zeta(\mathsf{OPT}(I))} \geq 1/5\,.$$

Somit hat unsere Heuristik Approximationsgüte 1/5; sie wählt immer mindestens 1/5 mal so viele Kreise aus wie eine optimale Lösung enthält. Man kann die Heuristik leicht so implementieren, dass sie eine Menge von n Kreisen in $O(n^2)$ Zeit und $O(n)$ Speicher verarbeitet.

Was weiß man noch über die *Approximierbarkeit* unseres Restaurantauswahlproblems? Vor kurzem haben Das et al. [8] einen Approximationsalgorithmus mit Güte 1/2 angegeben, der in $O(n^3)$ Zeit läuft und $O(n^2)$ Speicherplatz benötigt. Außerdem haben sie ein sogenanntes *Polynomialzeit-Approximationsschema* (engl. *polynomial-time approximation scheme*, kurz PTAS) angegeben. Das ist im Allgemeinen eine Schar von Approximationsalgorithmen $(\mathscr{A}_k)_{k\in\mathbb{N}}$, wobei $\mathscr{A}_k$ ein Approximationsalgorithmus mit Güte $(1-1/k)$ ist. Das heißt, man kommt beliebig nah an eine exakte Lösung heran, indem man k entsprechend groß wählt, bezahlt aber dafür mit einer Laufzeit, deren *Exponent* üblicherweise mit k wächst. Daher liefert ein PTAS im Allgemeinen keine praktisch einsetzbaren Algorithmen, aber die Antwort auf eine wichtige theoretische Frage: wie gut lässt sich ein gegebenes Problem approximieren? (Es gibt durchaus Optimierungsprobleme, die gar keine Approximation zulassen – davon ist gleich im nächsten Absatz die Rede – aber es gibt auch Probleme, von denen man weiß, dass es einen Approximationsalgorithmus mit Güte $1/2$ gibt, aber keinen, dessen Güte auch nur ein Haar besser wäre.) Das PTAS von Das et al. hat Güte $1/(1+1/k)^2$ (ein Ausdruck, den man mit entsprechend großem k beliebig nah an 1 bringen kann), Laufzeit $k^4 n^{O(k\log k)}$ und Speicherplatzbedarf $O(n+k\log k)$.

Bei *Minimierungsproblemen* (wie dem berühmten Problem des Handlungsreisenden oder Traveling Salesman Problem, kurz TSP) interessieren wir uns ebenfalls für das Verhältnis

$$\frac{\zeta(\mathsf{ALG}(I))}{\zeta(\mathsf{OPT}(I))}\,,$$

das hier jedoch immer mindestens 1 ist. Also hätten wir gerne eine möglichst kleine obere Schranke γ für dieses Verhältnis. (Zum Beispiel hat der Algorithmus von Christofides [4] für das *metrische* TSP, einen Spezialfall von TSP, eine Güte von $3/2$, während es für das allgemeine TSP $\mathscr{NP}$-schwer ist, irgendeinen Approximationsalgorithmus anzugeben.) In beiden Fällen – bei Minimierungs- und Maximierungsproblemen – geht es darum, Approximationsalgorithmen mit einer Güte möglichst nahe 1 zu finden; eine Güte von 1 haben gerade die exakten Algorithmen.

Zusammenfassend lässt sich zu Approximationsalgorithmen sagen, dass es oft schwer bis unmöglich ist eine bestimmte Güte zu beweisen, besonders bei echten Problemen mit vielen Randbedingungen. Außerdem ist die Güte, die man be-

weisen kann, das Ergebnis einer sehr pessimistischen Haltung; auf realen Daten sind die meisten Approximationsalgorithmen deutlich besser als ihre Güte vermuten lässt. Trotzdem sind Gütegarantien wertvoll: sie gelten im Gegensatz zu experimentell gewonnenen Messdaten immer, für alle Eingaben. Vazirani [22] hat ein sehr schönes Lehrbuch über Approximationsalgorithmen geschrieben; Williamson und Shmoy [23] beschäftigen sich besonders mit Approximationsalgorithmen für Standortprobleme.

5 Fazit und Perspektive

Die kombinatorische Optimierung lässt sich vielfältig in der Geoinformatik anwenden – nicht nur auf Probleme der Standortanalyse. Beispielsweise erfordert die Integration verschiedener Geodatensätze oft eine automatische Objektzuordnung, die durch geometrische Abweichungen zwischen den Datensätzen erschwert wird. Durch Optimierung können Zuordnungen so gebildet werden, dass ein globales Fehlermaß minimiert wird [1]. Optimierung wurde auch zur automatischen Generalisierung angewendet, also zur Gewinnung einer Karte kleineren Maßstabs aus einer gegebenen Karte [13]. Dabei gilt es, einen guten Kompromiss zwischen der Bewahrung wichtiger Karteninhalte und der Übersichtlichkeit der Ausgabekarte zu finden.

In der Regel bedeutet die Modellierung eines Problems die erste, vielleicht sogar die größte Hürde für einen Anwender. Welches Fehlermaß ist das richtige für eine bestimmte Zuordnungsaufgabe? Wie lassen sich die teilweise nur vage bekannten Kriterien der kartographischen Generalisierung mathematisch formalisieren und in eine einzige Problemdefinition fassen? Es kann nicht genug betont werden, dass die sorgfältige Analyse und Beantwortung dieser Fragen von großem Wert ist, denn ist eine mathematische Beschreibung eines Problems erst einmal geschaffen, so steht ein breites Spektrum algorithmischer Werkzeuge zur Lösung bereit. Oft werden heuristische Verfahren entworfen, nach dem Motto „Alles ist erlaubt, sofern es in der Praxis schnell ist und gute Lösungen liefert.“ Gegen eine gute Heuristik ist nichts einzuwenden, doch zunächst sollte die Möglichkeit eines exakten effizienten Algorithmus in Betracht gezogen werden.

Auch wenn die abschließende Entscheidung oft nicht zugunsten eines exakten Algorithmus ausfällt, so gibt es wichtige Gründe, einen exakten Ansatz (z. B. durch ganzzahlige Programmierung) zu versuchen. So lassen sich zumindest für kleine Instanzen optimale Lösungen berechnen. Diese haben zweierlei Wert: Erstens können sie mit Lösungen einer Heuristik verglichen werden – z. B. wäre es interessant zu erfahren, dass eine Heuristik in der Regel nur 5% vom Optimum abweicht. Zweitens kann ein Modell für ein Problem anhand optimaler Lösungen verifiziert werden: Eine optimale Lösung, die nicht den Ansprüchen eines Anwenders genügt, deutet zweifelsfrei auf ein unzureichendes Modell hin – zum Beispiel wurde ein wichtiges Kriterium in der Problemdefinition vergessen. Nicht zuletzt da sich ganzzahlige Programme mit heutigen Problemlösern ohne großen Programmieraufwand lösen

lassen, messen wir der exakten Optimierung eine hohe Relevanz sowie Potenzial für zukünftige algorithmische Entwicklungen in der Geoinformatik zu.

Literaturverzeichnis

1. Alt H., Efrat A., Rote G., Wenk C.: Matching planar maps. J. Algorithms 49(2), 262–283 (2003). doi:10.1016/S0196-6774(03)00085-3. URL http://dx.doi.org/10.1016/S0196-6774(03)00085-3
2. de Berg M., Cheong O., van Kreveld M., Overmars M.: Computational Geometry: Algorithms and Applications. Springer-Verlag, 3. Aufl. (2008). doi:10.1007/978-3-540-77974-2. URL http://dx.doi.org/10.1007/978-3-540-77974-2
3. Breu H., Kirkpatrick D.G.: Unit disk graph recognition is NP-hard. Comput. Geom. Theory Appl. 9(1–2), 3–24 (1998). doi:10.1016/S0925-7721(97)00014-X. URL http://dx.doi.org/10.1016/S0925-7721(97)00014-X
4. Christofides N.: Worst-case analysis of a new heuristic for the travelling salesman problem. Techn. Ber. 388, Graduate School of Industrial Administration, Carnegie Mellon University (1976)
5. Clark B.N., Colbourn C.J., Johnson D.S.: Unit disk graphs. Discrete Math. 86(1–3), 165–177 (1990). doi:10.1016/0012-365X(90)90358-O. URL http://dx.doi.org/10.1016/0012-365X(90)90358-O
6. Cormen T.H., Leiserson C.E., Rivest R.L., Stein C.: Algorithmen – Eine Einführung. Oldenbourg, München, 4. Aufl. (2013)
7. Dantzig G.B.: Linear Programming and Extensions. Princeton University Press (1963)
8. Das G.K., De M., Kolay S., Nandy S.C., Sur-Kolay S.: Approximation algorithms for maximum independent set of a unit disk graph. Inform. Process. Lett. 115(3), 439–446 (2015). doi:10.1016/j.ipl.2014.11.002. URL http://dx.doi.org/10.1016/j.ipl.2014.11.002
9. Fowler R.J., Paterson M.S., Tanimoto S.L.: Optimal packing and covering in the plane are NP-complete. Inform. Process. Lett. 12(3), 133–137 (1981). doi:10.1016/0020-0190(81)90111-3. URL http://dx.doi.org/10.1016/0020-0190(81)90111-3
10. Garey M.R., Johnson D.S.: Computers and Intractability; A Guide to the Theory of NP-Completeness. W. H. Freeman & Co. (1990)
11. Gomory R.E.: Outline of an algorithm for integer solutions to linear programs. Bull. Amer. Math. Soc. 64, 275–278 (1958). doi:10.1007/978-3-540-68279-0_4. URL http://dx.doi.org/10.1007/978-3-540-68279-0_4
12. Haunert J.H., Wolff A.: Optimal simplification of building ground plans. In: Proc. 21st Congress Int. Society Photogrammetry Remote Sensing (ISPRS'08), Technical Commision II/3, *Int. Archives of Photogrammetry, Remote Sensing and Spatial Inform. Sci.*, Bd. XXXVII, Part B2, S. 373–378. Beijing (2008). URL http://www.isprs.org/proceedings/XXXVII/congress/2_pdf/3_WG-II-3/01.pdf
13. Haunert J.H., Wolff A.: Area aggregation in map generalisation by mixed-integer programming. Int. J. Geogr. Inform. Sci. 24(12), 1871–1897 (2010). doi:10.1080/13658810903401008. URL http://dx.doi.org/10.1080/13658810903401008
14. Hsiao J.Y., Tang C.Y., Chang R.S.: An efficient algorithm for finding a maximum weight 2-independent set on interval graphs. Inform. Process. Lett. 43(5), 229–235 (1992). doi:10.1016/0020-0190(92)90216-I. URL http://dx.doi.org/10.1016/0020-0190(92)90216-I
15. Karmarkar N.: A new polynomial-time algorithm for linear programming. Combinatorica 4(4), 373–395 (1984). doi:10.1007/BF02579150. URL http://dx.doi.org/10.1007/BF02579150

16. Kirkpatrick S., Gelatt Jr. C.D., Vecchi M.P.: Optimization by simulated annealing. Science 220(4598), 671–680 (1983). doi:10.2307/1690046. URL http://dx.doi.org/10.2307/1690046
17. Krumke S.O., Noltemeier H.: Graphentheoretische Konzepte und Algorithmen. Vieweg+Teubner, 3. Aufl. (2012). doi:10.1007/978-3-8348-2264-2. URL http://dx.doi.org/10.1007/978-3-8348-2264-2
18. Lichtenstein D.: Planar formulae and their uses. SIAM J. Comput. 11(2), 329–343 (1982). doi:10.1137/0211025. URL http://dx.doi.org/10.1137/0211025
19. Michiels W., Aarts E., Korst J.: Theoretical Aspects of Local Search. Springer-Verlag (2007). doi:10.1007/978-3-540-35854-1. URL http://dx.doi.org/10.1007/978-3-540-35854-1
20. Niemeier W.: Ausgleichungsrechnung. De Gruyter, Berlin, 2. Aufl. (2008)
21. Papadimitriou C.H., Steiglitz K.: Combinatorial optimization: Algorithms and complexity. Dover (1998)
22. Vazirani V.V.: Approximation Algorithms. Springer-Verlag, Berlin (2001). doi:10.1007/978-3-662-04565-7. URL http://dx.doi.org/10.1007/978-3-662-04565-7
23. Williamson D.P., Shmoys D.B.: The Design of Approximation Algorithms. Cambridge University Press (2011)
24. Wolff A.: Graph drawing and cartography. In: R. Tamassia (Hg.) Handbook of Graph Drawing and Visualization, Kap. 23, S. 697–736. CRC Press, Boca Raton, FL (2013)