8

Exponentialzeit-Algorithmen für TSP und DNP

Im vorigen Kapitel haben wir eine Technik kennen gelernt, mit der sich die naiven Exponentialzeit-Algorithmen für eine Vielzahl von Problemen deutlich verbessern lassen, nämlich alle die Probleme, die sich für positive natürliche Zahlen a und b als (a,b)-CSP darstellen lassen. Dies gelingt für ganz unterschiedliche Probleme, etwa das Erfüllbarkeitsproblem der Aussagenlogik und Färbbarkeitsprobleme auf Graphen, aber nicht für alle Probleme.

In diesem Kapitel befassen wir uns mit dem Problem DOMATISCHE ZAHL und dem Problem des Handelsreisenden (kurz TSP genannt), die in den Beispielen 3.56 und 3.58 in Abschnitt 3.4 eingeführt wurden, und stellen für diese Probleme Exponentialzeit-Algorithmen vor, die deren naive Algorithmen deutlich verbessern.

8.1 Das Problem des Handelsreisenden

Ein Händler muss 21 Städte in Deutschland bereisen. Da er so schnell wie möglich damit fertig sein möchte, interessiert er sich für eine kürzeste Rundreise. Das in Abschnitt 3.4 definierte Problem des Handelsreisenden ist eines der klassischen NP-vollständigen Probleme und kann deshalb vermutlich nicht in (deterministischer) Polynomialzeit gelöst werden.

8.1.1 Pseudo-Polynomialzeit und starke NP-Vollständigkeit

Anders als NP-vollständige Probleme wie 3-SAT und 3-FÄRBBARKEIT, bei denen keine Zahlen, sondern boolesche Ausdrücke bzw. ungerichtete Graphen gegeben sind, können Probleme wie TSP, deren Instanzen Zahlen enthalten,[47] oft mit dynamischer Programmierung in *Pseudo-Polynomialzeit* gelöst werden. Betrachtet man die Zeitkomplexität solcher Probleme als eine Funktion zweier Variabler (nämlich als eine Funktion, die sowohl von der Instanzengröße als auch dem größten Wert

Pseudo-Polynomialzeit

[47] Bei TSP-Instanzen sind dies die Entfernungen zwischen den gegebenen Städten.

F. Gurski et al., *Exakte Algorithmen für schwere Graphenprobleme*, eXamen.press,
DOI 10.1007/978-3-642-04500-4_8,

der gegebenen Zahlen abhängt), so kann man eine polynomielle Zeitschranke möglicherweise einfach dadurch erhalten, dass die binär dargestellten Werte der gegebenen Zahlen relativ zur eigentlichen Instanzengröße exponentiell groß sein können. Misst man die Zeitkomplexität dieser Probleme jedoch wie üblich nur in der Instanzengröße (siehe Abschnitt 2.2), so ist ein solcher Pseudo-Polynomialzeit-Algorithmus in der Regel nichts anderes als ein Exponentialzeit-Algorithmus.

Ist ein Problem, dessen Instanzen Zahlen enthält, selbst dann NP-vollständig, wenn alle diese Zahlen polynomiell in der Instanzengröße beschränkt sind (d. h., dieses Problem ist nicht in Pseudo-Polynomialzeit lösbar, außer wenn P = NP gelten würde), so nennt man es *stark* NP-*vollständig*. Die starke NP-Vollständigkeit stellt einen Spezialfall der NP-Vollständigkeit gemäß Definition 5.18 dar (siehe z. B. Garey und Johnson [GJ79]).

stark NP-vollständig

Satz 8.1. TSP *ist stark* NP-*vollständig.* **ohne Beweis**

Übung 8.2. (a) Sei Π ein NP-vollständiges Problem, dessen Instanzen Zahlen enthält. Betrachten Sie die folgenden beiden Varianten von Π:

1. $\Pi_{\text{unär}}$ ist genau wie Π definiert, mit dem einzigen Unterschied, dass alle Zahlen in den Instanzen von $\Pi_{\text{unär}}$ unär (und nicht binär) dargestellt sind.
2. Π_{poly} ist genau wie Π definiert, mit dem einzigen Unterschied, dass alle Zahlen in den Instanzen von Π_{poly} polynomiell in der Instanzengröße beschränkt sind.

Was können Sie über die Komplexität (als Funktion der Instanzengröße) der Probleme $\Pi_{\text{unär}}$ und Π_{poly} sagen? Diskutieren Sie diese Komplexität jeweils für solche Probleme Π, für die es einen Pseudo-Polynomialzeit-Algorithmus gibt, und für solche, die stark NP-vollständig sind.

(b) Beweisen Sie Satz 8.1. **Hinweis:** [GJ79].

PARTITION (c) Betrachten Sie die folgenden beiden Probleme:

PARTITION	
Gegeben:	Eine Folge $a_1, a_2, \ldots, a_m$ von natürlichen Zahlen.
Frage:	Gibt es eine Partition der Indizes in zwei Mengen $\{A, B\}$, $A \cup B = \{1, 2, \ldots, m\}$, $A \cap B = \emptyset$, sodass $\sum_{i \in A} a_i = \sum_{i \in B} a_i$ gilt?

BIN PACKING

BIN PACKING	
Gegeben:	Eine Folge $a_1, a_2, \ldots, a_m$ von natürlichen Zahlen, eine natürliche Zahl $k \geq \max_{1 \leq i \leq m} a_i$ und eine natürliche Zahl $b < m$.
Frage:	Gibt es eine Partition der Indizes in b Mengen $\{A_1, A_2, \ldots, A_b\}$, $A_1 \cup A_2 \cup \cdots \cup A_b = \{1, 2, \ldots, m\}$, $A_i \cap A_j = \emptyset$, $1 \leq i < j \leq b$, sodass $\sum_{i \in A_j} a_i \leq k$ für alle j, $1 \leq j \leq b$, gilt?

Der Name BIN PACKING rührt daher, dass man bei diesem Problem testet, ob die m Gewichte $a_1, a_2, \ldots, a_m$ so auf die b Behälter (englisch: *bins*) $A_1, A_2, \ldots, A_b$ verteilt (bzw. so in diese Behälter „gepackt“) werden können, dass das Fassungsvermögen k keines der Behälter überschritten wird.
Stellen Sie eine Vermutung auf, welches bzw. welche dieser Probleme stark NP-vollständig ist bzw. sind, und versuchen Sie, Ihre Vermutung zu beweisen.

8.1.2 Naiver Algorithmus

In Abschnitt 8.1.3 wollen wir einen Exponentialzeit-Algorithmus für TSP vorstellen, der den naiven Algorithmus verbessert. Zunächst betrachten wir hier den naiven Exponentialzeit-Algorithmus für TSP selbst. Dieser inspiziert für eine gegebene TSP-Instanz nacheinander in systematischer Weise alle möglichen Rundreisen, berechnet die jeweilige Länge einer jeden Rundreise (also die aufsummierten Entfernungen zwischen je zwei Städten auf dieser Tour) und bestimmt schließlich eine kürzeste dieser Rundreisen.

Wie viele verschiedene Touren müssen beim naiven TSP-Algorithmus inspiziert werden? Sind n Städte gegeben, so ist jede Tour eindeutig durch eine Anordnung von $n-1$ Städten festgelegt. Außerdem genügt es, jede Rundreise in einer der beiden möglichen Richtungen zu durchlaufen. Folglich ergeben sich

$$\frac{(n-1)!}{2}$$

zu inspizierende Touren. Bei einer TSP-Instanz mit 21 Städten gibt es demnach

$$\frac{20!}{2} = 20 \cdot 19 \cdot 18 \cdot \ldots \cdot 3$$

verschiedene Rundreisen, d. h., es sind genau 1 216 451 004 088 320 000 Rundreisen zu inspizieren. Nehmen wir an, dass wir einen Rechner haben, der die Länge einer Rundreise in 10^{-9} Sekunden berechnet, dann würde dieser Rechner für alle Rundreisen insgesamt 1 216 451 004 Sekunden brauchen, was ungefähr 20 274 183 Minuten bzw. 337 903 Stunden bzw. 14 079 Tage bzw. 38 Jahre sind. Für die Berechnung einer kürzesten Rundreise durch gerade einmal 21 Städte ist das ziemlich lang – schon lange vor seiner Abreise wären alle Verkaufsprodukte des Händlers verdorben oder veraltet, und seine Tochter hätte wohl inzwischen die Geschäfte übernommen.

Anmerkung 8.3. 1. Natürlich gibt es heutzutage viel schnellere Rechner, und künftige Rechnergenerationen werden noch schneller sein. Doch ist das für die Abschätzung des Zeitbedarfs eines Exponentialzeit-Algorithmus wirklich relevant? Angenommen, man hat einen Algorithmus, der in der Zeit 3^n arbeitet, und N sei die maximale Eingabegröße, die dieser in einem gerade noch vertretbaren Zeitaufwand (von, sagen wir, 24 Stunden) auf unserem Rechner bearbeiten kann. Nun wird ein neuer Rechner geliefert, der tausendmal so schnell wie der alte ist. Die maximale Größe der Eingaben, die dieser neue, schnellere Rechner in unserem gerade noch vertretbaren Zeitaufwand von 24 Stunden bearbeiten kann, ist dann etwa $N + 6.29$.

Zwar führt die Vertausendfachung der Rechengeschwindigkeit bei einem Linearzeit-Algorithmus zu einer Vertausendfachung der Größe der in vertretbarem Aufwand lösbaren Instanzen: Statt Graphen mit 100 Knoten kann man nun Graphen mit 100 000 Knoten in derselben Zeit bearbeiten. Bei einem 3^n-Algorithmus ist dieselbe Vertausendfachung der Rechengeschwindigkeit jedoch nahezu wirkungslos: Statt Graphen mit 100 Knoten kann man nun Graphen mit 106 Knoten in derselben Zeit bearbeiten.

2. Das Problem des Handelsreisenden ist in der Praxis nicht nur für Logistik-Unternehmen von großer Bedeutung. Um nur ein anderes Beispiel zu nennen: Die Berechnung kürzester Rundreisen spielt auch beim Herstellen von Leiterplatten eine wichtige Rolle. Angenommen, es sind Leiterplatten mit 100 Bohrlöchern an vorgegebenen Positionen zu versehen. Dazu muss man eine möglichst kurze Rundreise von Bohrloch zu Bohrloch berechnen. Je kürzer die Strecke ist, die ein Bohrer pro Platte insgesamt zurücklegen muss, umso schneller kann er alle Löcher bohren und umso mehr Geld verdient der Leiterplattenhersteller an diesem Tag. Da vielleicht täglich andere Muster von Löchern gebohrt werden sollen, lässt sich eine kürzeste Tour nicht einfach ein für alle Mal vorberechnen, sondern sie muss jeden Morgen neu bestimmt werden.

Beispiel 8.4 (naiver Algorithmus für TSP). Um den naiven Algorithmus für das Problem des Handelsreisenden anhand der konkreten TSP-Instanz $(K_5, c, 9)$ vorzuführen, wobei $K_5 = (V, E)$ der vollständige Graph mit fünf Knoten und $c : E \to \mathbb{N}$ die zugehörige Kostenfunktion ist, betrachten wir noch einmal die Karte mit fünf Städten aus Beispiel 3.58:

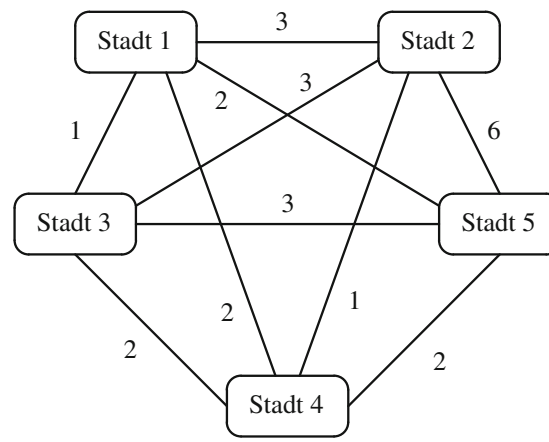

Abb. 8.1. Eine TSP-Instanz

Entfernungsmatrix

Die Entfernungen $c(\{i, j\})$ zwischen je zwei Städten i und j, mit denen die Kanten des Graphen in Abb. 8.1 beschriftet sind, können wie folgt als *Entfernungsmatrix* dargestellt werden:

$$D = \begin{pmatrix} 0 & 3 & 1 & 2 & 2 \\ 3 & 0 & 3 & 1 & 6 \\ 1 & 3 & 0 & 2 & 3 \\ 2 & 1 & 2 & 0 & 2 \\ 2 & 6 & 3 & 2 & 0 \end{pmatrix}.$$

Ein Eintrag $d(i,j) = c(\{i,j\})$ in dieser Matrix D stellt die Entfernung der Stadt i von der Stadt j dar. Da D symmetrisch ist (d. h., es gilt $d(i,j) = d(j,i)$ für alle $i, j \in \{1,2,\ldots,5\}$), hätte in diesem Beispiel auch die Angabe einer oberen Dreiecksmatrix genügt.[48] Um die Instanz $(K_5, c, 9)$ zu entscheiden, ist also die Frage zu beantworten, ob es eine Rundreise der Länge höchstens 9 gibt.

Wie oben erwähnt besteht der naive Algorithmus zur Berechnung einer kürzesten Rundreise im Durchprobieren aller möglichen Rundreisen. Es ergeben sich $4!/2 = 12$ verschiedene Rundreisen, die mit ihren jeweiligen Längen in Tabelle 8.1 dargestellt sind. Wir nehmen dabei an, dass die Reise immer bei der Stadt 1 beginnt und zu ihr zurückkehrt.

Tabelle 8.1. Vom naiven TSP-Algorithmus inspizierte Rundreisen für die Instanz in Abb. 8.1

Nr.	Rundreise	Länge der Rundreise
1	$1 \to 2 \to 3 \to 4 \to 5 \to 1$	$3+3+2+2+2 = 12$
2	$1 \to 2 \to 4 \to 3 \to 5 \to 1$	$3+1+2+3+2 = 11$
3	$1 \to 4 \to 2 \to 3 \to 5 \to 1$	$2+1+3+3+2 = 11$
4	$1 \to 2 \to 3 \to 5 \to 4 \to 1$	$3+3+3+2+2 = 13$
5	$1 \to 2 \to 4 \to 5 \to 3 \to 1$	$3+1+2+3+1 = 10$
6	$1 \to 4 \to 2 \to 5 \to 3 \to 1$	$2+1+6+3+1 = 13$
7	$1 \to 2 \to 5 \to 3 \to 4 \to 1$	$3+6+3+2+2 = 16$
8	$1 \to 2 \to 5 \to 4 \to 3 \to 1$	$3+6+2+2+1 = 14$
9	$1 \to 4 \to 5 \to 2 \to 3 \to 1$	$2+2+6+3+1 = 14$
10	$1 \to 5 \to 2 \to 3 \to 4 \to 1$	$2+6+3+2+2 = 15$
11	$1 \to 5 \to 2 \to 4 \to 3 \to 1$	$2+6+1+2+1 = 12$
12	$1 \to 5 \to 4 \to 2 \to 3 \to 1$	$2+2+1+3+1 = 9$

Und der Sieger ist: die 12. Rundreise! Diese durchläuft die Städte in der Reihenfolge

$$\text{Stadt } 1 \to \text{Stadt } 5 \to \text{Stadt } 4 \to \text{Stadt } 2 \to \text{Stadt } 3 \to \text{Stadt } 1$$

und hat eine Länge von 9. Die Frage, ob die gegebene Instanz $(K_5, c, 9)$ zu TSP gehört oder nicht, ist wegen dieser Lösung mit „ja“ zu beantworten. Mit etwas mehr Glück (d. h. in einer anderen Durchmusterungsreihenfolge) hätte man auf diese

[48] Wie bereits in Abschnitt 3.4.4 erwähnt wurde, kann man auch Varianten von TSP betrachten, bei denen die Distanzmatrix nicht symmetrisch sein muss. Bei einer anderen Variante von TSP verlangt man zum Beispiel, dass die Dreiecksungleichung erfüllt ist: $d(i,j) + d(j,k) \geq d(i,k)$ für alle Städte i, j und k. Die Instanz in Abb. 8.1 erfüllt die Dreiecksungleichung nicht, da z. B. $d(2,1) + d(1,5) = 3+2 = 5 < 6 = d(2,5)$ gilt.

Rundreise, mit der die positive Entscheidung für die gegebene Instanz gefällt werden kann, schon eher stoßen können. Doch im schlechtesten Fall muss man sämtliche Rundreisen inspizieren, ehe man diese Entscheidung treffen kann.

Natürlich kann man mit diesem Algorithmus nicht nur das Entscheidungsproblem TSP, sondern auch das Optimierungsproblem MIN TSP (das in Abschnitt 3.6 definiert wurde) in derselben exponentiellen Zeit lösen. In der Praxis benutzt man dafür auch Approximationsalgorithmen, um in „vernünftiger“ Zeit zumindest eine Näherungslösung zu berechnen. Ein Greedy-Algorithmus zum Beispiel geht wie folgt vor: Ausgehend von einem beliebigen Startpunkt, etwa der Stadt 1, wählt dieser Algorithmus eine Stadt, die am nächsten liegt, von dort wieder eine nächste Stadt und so weiter. Bereits besuchte Städte werden dabei markiert und – bis auf den Startpunkt – nicht wieder aufgesucht. So wird fortgefahren, bis man in allen Städten einmal war und zum Ausgangspunkt zurückkehrt. Gibt es unterwegs Wahlmöglichkeiten, weil zum Beispiel zwei Städte am nächsten sind, wählt man eine beliebige aus, etwa die mit dem kleinsten Namen (in der lexikographischen Ordnung).

Mit diesem Greedy-Algorithmus erhalten wir die folgende Näherungslösung für unsere Probleminstanz aus Beispiel 8.4:

$$\text{Stadt } 1 \to \text{Stadt } 3 \to \text{Stadt } 4 \to \text{Stadt } 2 \to \text{Stadt } 5 \to \text{Stadt } 1,$$

die der Tour Nr. 11 in Tabelle 8.1 (nur in umgekehrter Richtung durchlaufen) entspricht und eine Länge von $1+2+1+6+2=12$ hat. Diese Näherungslösung ist offenbar weit von einer kürzesten Rundreise entfernt. Der beschriebene Algorithmus ist auch als „Nächster-Nachbar-Heuristik“ (englisch: *nearest neighbor heuristic*) bekannt und liefert beweisbar beliebig schlechte Lösungen [ACG+03].

In diesem Buch beschäftigen wir uns jedoch nicht mit Approximations-, sondern mit *exakten* Algorithmen für schwere Graphenprobleme. Es gibt verschiedene Ideen, mit denen man ineffiziente Algorithmen verbessern kann, ohne auf den Anspruch einer *exakten* Lösung zu verzichten, und einen solchen Ansatz stellen wir nun vor. Er beruht auf dem Prinzip der dynamischen Programmierung (siehe Abschnitt 3.5.3).

8.1.3 Algorithmus mit dynamischer Programmierung

Mit dynamischer Programmierung kann das Optimierungsproblem MIN TSP nach wie vor nicht effizient, jedoch schneller als durch den naiven Algorithmus gelöst werden. Der Algorithmus funktioniert wie folgt. Wie beim naiven Algorithmus aus Abschnitt 8.1.2 sollen alle Rundreisen von der Stadt 1 aus[49] über die $n-1$ anderen Städte zurück zur Stadt 1 berechnet werden, und eine kürzeste dieser Reisen ist dann die gesuchte Tour. Anders als beim naiven Algorithmus sparen wir jetzt aber Zeit dadurch ein, dass wir bereits berechnete optimale Teillösungen speichern und nicht mehrmals berechnen müssen. Um also eine kürzeste Rundreise zwischen allen Städten in $\{1,2,\ldots,n\}$ zu finden, berechnen wir einen kürzesten Weg von der Stadt 1 aus über alle Städte in $\{2,\ldots,n\}-\{i\}$ bis zur Stadt i, wobei $i \in \{2,\ldots,n\}$.

[49] Wir nehmen weiterhin an, dass die Reise immer bei der Stadt 1 beginnt und zu ihr zurückkehrt.

Indem man diese Idee über alle Teilmengen $S \subseteq \{2,\ldots,n\}$ mit wachsender Kardinalität immer wieder anwendet, entsteht ein rekursiver Algorithmus. Formelmäßig kann die Rekursion wie folgt beschrieben werden. Sei $f(S,i)$ die Länge eines kürzesten Weges, der bei der Stadt 1 beginnt, alle Städte der Menge $S-\{i\}$ genau einmal in beliebiger Reihenfolge besucht und in der Stadt i endet. Diese Funktion erfüllt die Rekursion

$$f(S,i) = \min_{j \in S-\{i\}} \{f(S-\{j\},j) + d(j,i)\} \tag{8.1}$$

für alle $S \subseteq \{2,\ldots,n\}$ und alle $i \in \{2,\ldots,n\}$. Offenbar gilt für alle $i \in \{2,\ldots,n\}$:

$$f(\{i\},i) = d(1,i), \tag{8.2}$$

das heißt, der kürzeste Weg von der Stadt 1 zur Stadt $i \in \{2,\ldots,n\}$, ohne einen Zwischenstopp, ist gerade der Wert $d(1,i)$ der Entfernungsmatrix D.

Beispiel 8.5 (Algorithmus mit dynamischer Programmierung für Min TSP). Wir wenden diesen Algorithmus nun auf unser Beispiel aus Abb. 8.1 an.

1. Die Entfernungen von der Stadt 1 zu allen anderen Städten entsprechen den Matrixwerten der ersten Zeile von D. Sie bilden gemäß (8.2) die unterste Ebene der Rekursion f, also $f(\{i\},i) = d(1,i)$ für $i \in \{2,\ldots,5\}$:

i	2	3	4	5
$f(\{i\},i) = d(1,i)$	3	1	2	2

Tabelle 8.2. MIN TSP mit dynamischer Programmierung: Berechnung von $f(S,i)$ für $|S| = 2$

Reiseweg	$f(S,i)$ für $\lvert S\rvert = 2$
$1 \to 3 \to 2$	$f(\{2,3\},2) = f(\{3\},3) + d(3,2) = 1+3 = 4$
$1 \to 4 \to 2$	$f(\{2,4\},2) = f(\{4\},4) + d(4,2) = 2+1 = 3$
$1 \to 5 \to 2$	$f(\{2,5\},2) = f(\{5\},5) + d(5,2) = 2+6 = 8$
$1 \to 2 \to 3$	$f(\{2,3\},3) = f(\{2\},2) + d(2,3) = 3+3 = 6$
$1 \to 4 \to 3$	$f(\{3,4\},3) = f(\{4\},4) + d(4,3) = 2+2 = 4$
$1 \to 5 \to 3$	$f(\{3,5\},3) = f(\{5\},5) + d(5,3) = 2+3 = 5$
$1 \to 2 \to 4$	$f(\{2,4\},4) = f(\{2\},2) + d(2,4) = 3+1 = 4$
$1 \to 3 \to 4$	$f(\{3,4\},4) = f(\{3\},3) + d(3,4) = 1+2 = 3$
$1 \to 5 \to 4$	$f(\{4,5\},4) = f(\{5\},5) + d(5,4) = 2+2 = 4$
$1 \to 2 \to 5$	$f(\{2,5\},5) = f(\{2\},2) + d(2,5) = 3+6 = 9$
$1 \to 3 \to 5$	$f(\{3,5\},5) = f(\{3\},3) + d(3,5) = 1+3 = 4$
$1 \to 4 \to 5$	$f(\{4,5\},5) = f(\{4\},4) + d(4,5) = 2+2 = 4$

2. Aufbauend auf diesen Werten der untersten Rekursionsstufe kann nun $f(S,i)$ gemäß (8.1) für Teilmengen S mit $|S| = 2$ berechnet werden. Es werden alle Wege von der Stadt 1 in die Stadt i über eine beliebige andere Stadt berechnet.

Dabei muss man in diesem einfachen Fall noch keine Minima bestimmen, weil die Verbindungen von 1 und i über nur einen Zwischenstopp j, $1 \neq j \neq i$, keine verschiedenen Reisemöglichkeiten zulassen. Es ergeben sich die in Tabelle 8.2 angegebenen Werte für $f(S,i)$ mit $|S| = 2$.

3. Nun werden die kürzesten Wege von der Stadt 1 zu allen Städten $i \in \{2,\ldots 5\}$ mit *zwei* Zwischenstopps berechnet, d. h., $|S| = 3$. Dabei kann man die schon berechneten kürzesten Wege über einen Zwischenstopp aus Tabelle 8.2 benutzen. Es ergeben sich die in Tabelle 8.3 angegebenen Werte für $f(S,i)$ mit $|S| = 3$.

Tabelle 8.3. MIN TSP mit dynamischer Programmierung: Berechnung von $f(S,i)$ für $|S| = 3$

Reiseweg	$f(S,i)$ für $\|S\| = 3$
$1 \to 3 \to 4 \to 2$ bzw. $1 \to 4 \to 3 \to 2$	$f(\{2,3,4\},2) = \min\{f(\{3,4\},3)+d(3,2), f(\{3,4\},4)+d(4,2)\}$ $= \min\{4+3, 3+1\} = 4$
$1 \to 3 \to 5 \to 2$ bzw. $1 \to 5 \to 3 \to 2$	$f(\{2,3,5\},2) = \min\{f(\{3,5\},3)+d(3,2), f(\{3,5\},5)+d(5,2)\}$ $= \min\{5+3, 4+6\} = 8$
$1 \to 4 \to 5 \to 2$ bzw. $1 \to 5 \to 4 \to 2$	$f(\{2,4,5\},2) = \min\{f(\{4,5\},4)+d(4,2), f(\{4,5\},5)+d(5,2)\}$ $= \min\{4+1, 4+6\} = 5$
$1 \to 2 \to 4 \to 3$ bzw. $1 \to 4 \to 2 \to 3$	$f(\{2,3,4\},3) = \min\{f(\{2,4\},2)+d(2,3), f(\{2,4\},4)+d(4,3)\}$ $= \min\{3+3, 4+2\} = 6$
$1 \to 2 \to 5 \to 3$ bzw. $1 \to 5 \to 2 \to 3$	$f(\{2,3,5\},3) = \min\{f(\{2,5\},2)+d(2,3), f(\{2,5\},5)+d(5,3)\}$ $= \min\{8+3, 9+3\} = 11$
$1 \to 4 \to 5 \to 3$ bzw. $1 \to 5 \to 4 \to 3$	$f(\{3,4,5\},3) = \min\{f(\{4,5\},4)+d(4,3), f(\{4,5\},5)+d(5,3)\}$ $= \min\{4+2, 4+3\} = 6$
$1 \to 2 \to 3 \to 4$ bzw. $1 \to 3 \to 2 \to 4$	$f(\{2,3,4\},4) = \min\{f(\{2,3\},2)+d(2,4), f(\{2,3\},3)+d(3,4)\}$ $= \min\{4+1, 6+2\} = 5$
$1 \to 2 \to 5 \to 4$ bzw. $1 \to 5 \to 2 \to 4$	$f(\{2,4,5\},4) = \min\{f(\{2,5\},2)+d(2,4), f(\{2,5\},5)+d(5,4)\}$ $= \min\{8+1, 9+2\} = 9$
$1 \to 3 \to 5 \to 4$ bzw. $1 \to 5 \to 3 \to 4$	$f(\{3,4,5\},4) = \min\{f(\{3,5\},3)+d(3,4), f(\{3,5\},5)+d(5,4)\}$ $= \min\{5+2, 4+2\} = 6$
$1 \to 2 \to 3 \to 5$ bzw. $1 \to 3 \to 2 \to 5$	$f(\{2,3,5\},5) = \min\{f(\{2,3\},2)+d(2,5), f(\{2,3\},3)+d(3,5)\}$ $= \min\{4+6, 6+3\} = 9$
$1 \to 2 \to 4 \to 5$ bzw. $1 \to 4 \to 2 \to 5$	$f(\{2,4,5\},5) = \min\{f(\{2,4\},2)+d(2,5), f(\{2,4\},4)+d(4,5)\}$ $= \min\{3+6, 4+2\} = 6$
$1 \to 3 \to 4 \to 5$ bzw. $1 \to 4 \to 3 \to 5$	$f(\{3,4,5\},5) = \min\{f(\{3,4\},3)+d(3,5), f(\{3,4\},4)+d(4,5)\}$ $= \min\{4+3, 3+2\} = 5$

4. Genau wie oben berechnet man nun die kürzesten Wege von der Stadt 1 zu allen Städten $i \in \{2,\ldots 5\}$ über *drei* Zwischenstopps, d. h., $|S| = 4$. Die schon berechneten kürzesten Wege über zwei Zwischenstopps aus Tabelle 8.3 kann man dafür benutzen. Es ergeben sich die in Tabelle 8.4 angegebenen Werte für $f(S,i)$ mit $|S| = 4$. Dabei bezeichnen wir mit $1 \to \{j,k,\ell\} \to i$ alle Möglichkeiten, von der Stadt 1 in die Stadt i über die Städte j, k und ℓ zu gelangen, also die Reisewege

$$1 \to j \to k \to \ell \to i, \quad 1 \to j \to \ell \to k \to i, \quad 1 \to k \to j \to \ell \to i,$$
$$1 \to k \to \ell \to j \to i, \quad 1 \to \ell \to j \to k \to i, \quad 1 \to \ell \to k \to j \to i,$$

und versuchen, unter diesen einen kürzesten Weg zu finden.

Tabelle 8.4. MIN TSP mit dynamischer Programmierung: Berechnung von $f(S,i)$ für $|S|=4$

Reiseweg	$f(S,i)$ für $\lvert S\rvert = 4$
$1 \to \{3,4,5\} \to 2$	$f(\{2,3,4,5\},2) = \min\left\{\begin{array}{l} f(\{3,4,5\},3)+d(3,2),\\ f(\{3,4,5\},4)+d(4,2),\\ f(\{3,4,5\},5)+d(5,2)\end{array}\right\}$ $= \min\{6+3, 6+1, 5+6\} = 7$
$1 \to \{2,4,5\} \to 3$	$f(\{2,3,4,5\},3) = \min\left\{\begin{array}{l} f(\{2,4,5\},2)+d(2,3),\\ f(\{2,4,5\},4)+d(4,3),\\ f(\{2,4,5\},5)+d(5,3)\end{array}\right\}$ $= \min\{5+3, 9+2, 6+3\} = 8$
$1 \to \{2,3,5\} \to 4$	$f(\{2,3,4,5\},4) = \min\left\{\begin{array}{l} f(\{2,3,5\},2)+d(2,4),\\ f(\{2,3,5\},3)+d(3,4),\\ f(\{2,3,5\},5)+d(5,4)\end{array}\right\}$ $= \min\{8+1, 11+2, 9+2\} = 9$
$1 \to \{2,3,4\} \to 5$	$f(\{2,3,4,5\},5) = \min\left\{\begin{array}{l} f(\{2,3,4\},2)+d(2,5),\\ f(\{2,3,4\},3)+d(3,5),\\ f(\{2,3,4\},4)+d(4,5)\end{array}\right\}$ $= \min\{4+6, 6+3, 5+2\} = 7$

5. Schließlich muss man nur noch einen kürzesten Rückweg in die Stadt 1 finden und berechnet also:

$$f(\{1,2,3,4,5\},1) = \min\left\{\begin{array}{l} f(\{2,3,4,5\},2)+d(2,1),\\ f(\{2,3,4,5\},3)+d(3,1),\\ f(\{2,3,4,5\},4)+d(4,1),\\ f(\{2,3,4,5\},5)+d(5,1)\end{array}\right\}$$
$$= \min\{7+3, 8+1, 9+2, 7+2\} = 9.$$

Wenn man den Algorithmus so modifiziert, dass er sich zusätzlich zur Berechnung der jeweiligen schrittweise wachsenden kürzesten Wege merkt, *warum* er in den einzelnen Schritten schließlich zu dieser kürzesten Rundreise über alle Städte gelangt ist, dann erhält man nicht nur den optimalen Wert (in diesem Fall die Länge 9 einer kürzesten Tour), sondern kann auch eine optimale Lösung konstruieren. In unserem Beispiel ergibt sich nämlich:

- die Wegstrecke $5 \to 1$ aus $f(\{1,2,3,4,5\},1) = f(\{2,3,4,5\},5)+d(5,1) = 9$,
- die Wegstrecke $4 \to 5$ aus $f(\{2,3,4,5\},5) = f(\{2,3,4\},4)+d(4,5) = 7$,
- die Wegstrecke $2 \to 4$ aus $f(\{2,3,4\},4) = f(\{2,3\},2)+d(2,4) = 5$,
- die Wegstrecke $3 \to 2$ aus $f(\{2,3\},2) = f(\{3\},3)+d(3,2) = 4$ und
- die Wegstrecke $1 \to 3$ aus $f(\{3\},3) = d(1,3) = 1$,

insgesamt also: $1 \to 3 \to 2 \to 4 \to 5 \to 1$, die der Tour Nr. 12 in Tabelle 8.1 (nur in umgekehrter Richtung durchlaufen) entspricht.[50] Der Algorithmus löst also mittels dynamischer Programmierung nicht nur das Optimierungs-, sondern auch das Suchproblem.

Auch wenn durch die Anwendung der dynamischen Programmierung kein wirklich effizienter (also kein Polynomialzeit-)Algorithmus entsteht, wird die Rechenzeit gegenüber dem naiven Algorithmus deutlich verringert, da alle schon berechneten kleineren Teillösungen genutzt werden, wenn man sukzessive die Anzahl der Städte erhöht. Es werden also keine Wege doppelt berechnet.

Satz 8.6 (Held und Karp [HK62]). *Das Problem des Handelsreisenden (*TSP*) kann für n Städte in der Zeit*

$$\tilde{\mathcal{O}}(2^n)$$

gelöst werden. **ohne Beweis**

Übung 8.7. Analysieren Sie den oben angegebenen Algorithmus mit dynamischer Programmierung und begründen Sie die in Satz 8.6 angegebene Laufzeit.

8.2 Das Domatische-Zahl-Problem

Die domatische Zahl $d(G)$ eines ungerichteten Graphen G wurde in Definition 3.55 eingeführt und bezeichnet die Größe einer größten Partition von G in dominierende Mengen. Eine dominierende Menge von $G = (V,E)$ ist dabei eine Teilmenge $D \subseteq V$ der Knotenmenge von G, sodass jeder Knoten von G entweder zu D gehört oder mit einem Knoten in D durch eine Kante verbunden ist (siehe Definition 3.54 und die Beispiele 3.15 und 3.54, insbesondere Abb. 3.9 und Abb. 3.29).

Die in Abschnitt 3.4.3 definierten Probleme DOMINIERENDE MENGE und DOMATISCHE ZAHL sind NP-vollständig (siehe Übung 5.25(c)). Wir konzentrieren uns hier auf das Problem DOMATISCHE ZAHL: Gegeben ein Graph $G = (V,E)$ und eine natürliche Zahl $k \leq |V|$, ist $d(G) \geq k$? Genauer gesagt werden wir uns nur mit der Variante dieses Problems befassen, für die der Parameter k den Wert 3 annimmt. Das heißt, in diesem Abschnitt untersuchen wir das Problem

3-DNP

$$\text{3-DNP} = \{G \mid G \text{ ist ein ungerichteter Graph mit } d(G) \geq 3\}.$$

Wie das allgemeine Problem DOMATISCHE ZAHL ist bereits 3-DNP NP-vollständig und daher vermutlich nicht in (deterministischer) Polynomialzeit lösbar. Das analog definierte Problem 2-DNP (bei dem gefragt wird, ob die Knotenmenge eines gegebenen Graphen G in mindestens zwei dominierende Mengen partitioniert werden kann) hingegen kann deterministisch in Polynomialzeit gelöst werden.

2-DNP

[50] Genau die Tour Nr. 12 aus Tabelle 8.1 (nicht in umgekehrter Richtung durchlaufen) erhält man ebenfalls, da man für die letzte Wegstrecke statt $5 \to 1$ auch $3 \to 1$ nehmen könnte, denn es gilt ja ebenso $f(\{1,2,3,4,5\},1) = f(\{2,3,4,5\},3) + d(3,1) = 9$, usw.

Übung 8.8. Zeigen Sie, dass 2-DNP in P liegt.

3-DNP lässt sich als ein Constraint Satisfaction Problem im Sinne von Definition 7.5 formulieren, sofern der maximale Knotengrad durch eine Konstante beschränkt ist. Das heißt, schränkt man 3-DNP auf solche Graphen G ein, für die $\Delta(G) \leq b$ für eine feste Zahl $b \in \mathbb{N}$ gilt, so lässt sich dieses eingeschränkte Problem als $(3, b+1)$-CSP ausdrücken, wodurch die in den Abschnitten 7.3 bis 7.5 vorgestellten Techniken anwendbar werden. Hier werden wir jedoch eine andere Methode einsetzen, um für das (uneingeschränkte) Problem 3-DNP einen Exponentialzeit-Algorithmus zu beschreiben, der besser als der naive $\tilde{\mathcal{O}}(3^n)$-Algorithmus für dieses Problem ist.

Übung 8.9. (a) Zeigen Sie, dass 3-DNP deterministisch in der Zeit $\tilde{\mathcal{O}}(3^n)$ gelöst werden kann, d. h., beschreiben Sie den naiven Algorithmus für dieses Problem.

(b) Sei 3-DNP$_b$ die Einschränkung von 3-DNP auf Graphen mit einem durch $b \in \mathbb{N}$ beschränkten maximalen Knotengrad:

$$\text{3-DNP}_b = \{G \mid G \text{ ist ein ungerichteter Graph mit } \Delta(G) \leq b \text{ und } d(G) \geq 3\}.$$

Formulieren Sie 3-DNP$_b$ als $(3, b+1)$-CSP und wenden Sie die Methoden aus den Abschnitten 7.3 bis 7.5 darauf an. Welche Zeitschranke erhalten Sie in Abhängigkeit von b? Ermitteln Sie insbesondere die Zeitschranke von 3-DNP, eingeschränkt auf kubische Graphen.

8.2.1 Vorbereitungen

Zunächst führen wir noch eine nützliche Notation ein. Sei $G = (V, E)$ ein ungerichteter Graph. Für Knoten $v \in V$ wurde in Definition 3.3 die (offene) Nachbarschaft $N(v)$ von v als die Menge aller mit v adjazenten Knoten definiert. Die *geschlossene Nachbarschaft von v* ist definiert als $N[v] = \{v\} \cup N(v)$. Diese Begriffe erweitern wir auf Teilmengen $U \subseteq V$ der Knotenmenge von G wie folgt:

offene bzw. geschlossene Nachbarschaft eines Knotens bzw. einer Knotenmenge

- $N[U] = \bigcup_{u \in U} N[u]$ ist die *geschlossene Nachbarschaft von U* und
- $N(U) = N[U] - U$ ist die *offene Nachbarschaft von U*.

Eine Menge $D \subseteq V$ ist also genau dann eine dominierende Menge von $G = (V, E)$, wenn $N[D] = V$ gilt. Wir sind hier speziell an den *minimalen* dominierenden Mengen von Graphen interessiert. Wie in Beispiel 3.15 erläutert wurde, heißt eine dominierende Menge D von G minimal, falls es keine dominierende Menge D' von G mit $D' \subset D$ gibt. Fomin et al. [FGPS08] zeigten, dass ein Graph mit n Knoten höchstens 1.7159^n minimale dominierende Mengen haben kann. Daraus ergibt sich das folgende Resultat.

Satz 8.10 (Fomin, Grandoni, Pyatkin und Stepanov [FGPS08]). *Es gibt einen Algorithmus, der für einen gegebenen Graphen G mit n Knoten in der Zeit*

$$\tilde{\mathcal{O}}(1.7159^n)$$

sämtliche minimalen dominierenden Mengen von G auflistet. **ohne Beweis**

Aus Satz 8.10 ergibt sich ein Algorithmus zur Berechnung der domatischen Zahl eines gegebenen Graphen.

Korollar 8.11 (Fomin, Grandoni, Pyatkin und Stepanov [FGPS08]). *Die domatische Zahl eines gegebenen Graphen mit n Knoten kann in der Zeit*

$$\tilde{\mathcal{O}}(2.8718^n)$$

berechnet werden. **ohne Beweis**

Wir geben die (recht aufwändigen) Beweise von Satz 8.10 und Korollar 8.11 nicht im Einzelnen an. Stattdessen wollen wir in Abschnitt 8.2.2 das Resultat aus Satz 8.10 mit dem folgenden Resultat von Yamamoto [Yam05] kombinieren, um einen Exponentialzeit-Algorithmus für 3-DNP zu erhalten, der besser als der naive Algorithmus für dieses Problem ist. Satz 8.12 gibt eine in der Klauselanzahl einer gegebenen KNF-Formel exponentielle Zeitschranke für das Problem SAT an. Auch diesen Satz beweisen wir hier nicht.

Satz 8.12 (Yamamoto [Yam05]). *Es gibt einen Algorithmus, der das Problem* SAT *für eine gegebene Formel in KNF mit m Klauseln in der Zeit*

$$\tilde{\mathcal{O}}(1.234^m)$$

löst. **ohne Beweis**

8.2.2 Kombination zweier Algorithmen

Aus Korollar 8.11 ergibt sich unmittelbar eine $\tilde{\mathcal{O}}(2.8718^n)$-Zeitschranke für 3-DNP. Diese kann jedoch noch verbessert werden, wenn wir den Algorithmus von Fomin et al. [FGPS08] aus Satz 8.10 mit dem Algorithmus von Yamamoto [Yam05] aus Satz 8.12 kombinieren.

Satz 8.13 (Riege, Rothe, Spakowski und Yamamoto [RRSY07]). *Es gibt einen Algorithmus, der das Problem* 3-DNP *für einen gegebenen Graphen mit n Knoten in der Zeit*

$$\tilde{\mathcal{O}}(2.6129^n)$$

löst.

Beweis. Sei $G = (V,E)$ ein gegebener Graph mit n Knoten. Mit dem Algorithmus von Fomin et al. [FGPS08] aus Satz 8.10 listen wir alle minimalen dominierenden Mengen von G der Reihe nach auf.[51] Hat man eine solche minimale dominierende Menge von G erzeugt, sagen wir $D \subseteq V$, so definieren wir die boolesche Formel $\varphi_D(X,C)$ mit

[51] Es ist dabei nicht nötig, diese alle zu speichern, was exponentiellen Platz brauchen würde. Stattdessen kann man eine nach der anderen weiter bearbeiten, wie im Folgenden beschrieben wird. Wurde die aktuelle minimale dominierende Menge von G erfolglos getestet, so kann man denselben Speicherplatz für die nächste minimale dominierende Menge von G verwenden. Auf diese Weise benötigt man nur polynomiellen Speicherplatz.

- der Variablenmenge $X = \{x_v \mid v \in V - D\}$ und
- der Klauselmenge $C = \{C_v \mid v \in V\}$, wobei für jeden Knoten $v \in V$ die Klausel C_v definiert ist durch

$$C_v = (u_1 \vee u_2 \vee \cdots \vee u_k)$$

mit $\{u_1, u_2, \ldots, u_k\} = N[v] - D$.

Übung 8.14. Bestimmen Sie alle minimalen dominierenden Mengen des Graphen in Abb. 8.2 (welcher auch in Abb. 3.9 dargestellt wurde) und konstruieren Sie für jede dieser Mengen D die entsprechende boolesche Formel φ_D.

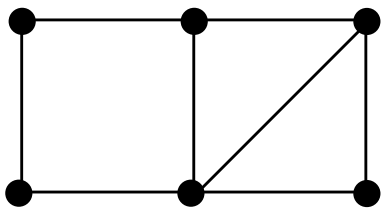

Abb. 8.2. Ein Graph für Übung 8.14

Der nächste Schritt im Beweis besteht darin, die folgende Äquivalenz zu zeigen:

$$G \in \text{3-DNP} \iff \varphi_D \in \text{NAE-SAT}, \tag{8.3}$$

wobei NAE-SAT die Variante von SAT ist, bei der getestet wird, ob es eine erfüllende Belegung für φ_D gibt, die in keiner Klausel sämtliche Literale erfüllt.[52] Wir zeigen nun beide Implikationen von (8.3). NAE-SAT

Von links nach rechts: Angenommen, G ist in 3-DNP. Dann gibt es eine Partition von V in drei dominierende Mengen, und wir dürfen annehmen, dass eine dieser drei Mengen (diese heiße D) minimal ist (siehe Übung 8.16). Seien F und T die beiden anderen dominierenden Mengen in dieser Partition von V. Nach Definition gilt $N[F] = V = N[T]$, d. h., jeder Knoten von G gehört sowohl zur abgeschlossenen Nachbarschaft von F als auch zu der von T. Konstruieren wir nun eine Belegung der Variablen in X, die einer Variablen x_v den Wert **true** zuordnet, falls v in T ist, und einer Variablen x_v den Wert **false**, falls v in F ist, so folgt $\varphi_D \in$ NAE-SAT.

Von rechts nach links: Angenommen, $\varphi_D \in$ NAE-SAT gilt für eine minimale dominierende Menge D von G. Sei $\beta : X \to \{\textbf{false}, \textbf{true}\}$ eine Belegung, die φ_D im Sinne von NAE-SAT erfüllt. Wir definieren die folgenden beiden Mengen:

$$F = \{v \in V - D \mid \beta(x_v) = \textbf{false}\};$$
$$T = \{v \in V - D \mid \beta(x_v) = \textbf{true}\}.$$

[52] Das heißt, das in Abschnitt 5.1.3 definierte Problem NAE-3-SAT ist die Einschränkung von NAE-SAT auf Formeln in 3-KNF. Wie SAT ist nach Lemma 5.24 auch NAE-3-SAT und somit ebenso NAE-SAT NP-vollständig.

Da für jeden Knoten $v \in V$ die Klausel C_v sowohl Literale enthält, die unter β wahr sind, als auch solche, die unter β falsch sind, sind F und T jeweils dominierende Mengen von G. Somit ist G in 3-DNP.

Die Äquivalenz (8.3) erlaubt es uns, die Frage „$G \in$ 3-DNP?“ auf die Frage „$\varphi_D \in$ NAE-SAT?“ zu reduzieren. Diese Frage wiederum kann leicht auf die Frage „$\psi_D \in$ SAT?“ reduziert werden, wobei ψ_D eine boolesche Formel in KNF ist, die doppelt so viele Klauseln wie φ_D hat.

Übung 8.15. Zeigen Sie, dass eine gegebene boolesche Formel φ_D in KNF so in eine boolesche Formel ψ_D in KNF umgeformt werden kann, dass ψ_D doppelt so viele Klauseln wie φ_D hat und die folgende Äquivalenz gilt:

$$\varphi_D \in \text{NAE-SAT} \iff \psi_D \in \text{SAT}. \tag{8.4}$$

Für jede der höchstens 1.7159^n minimalen dominierenden Mengen D von G (siehe Satz 8.10) können wir den $\tilde{\mathcal{O}}(1.234^m)$-Algorithmus von Yamamoto aus Satz 8.12 auf die Formel ψ_D anwenden, um ihre Erfüllbarkeit zu testen. Da ψ_D jeweils $m = 2n = |V|$ Klauseln hat, ergibt sich insgesamt eine Laufzeit von

$$\tilde{\mathcal{O}}(1.7159^n \cdot 1.234^{2n}) = \tilde{\mathcal{O}}(2.6129^n),$$

womit der Satz bewiesen ist. ❑

Übung 8.16. Sei $G = (V,E)$ ein Graph in 3-DNP. Zeigen Sie, dass es unter allen Partitionen von V in drei dominierende Mengen mindestens eine gibt, die eine minimale dominierende Menge enthält.

8.3 Literaturhinweise

Die in Abschnitt 8.1.2 skizzierte Nächster-Nachbar-Heuristik zur Approximation des Problems des Handelsreisenden kann leider nicht so verbessert werden, dass man eine abschätzbar gute Lösung für das allgemeine Problem TSP erhält. Setzt man zusätzlich die Dreiecksungleichung aus Fußnote 48 für die Instanzen des TSP voraus, so bezeichnet man diese Einschränkung des Problems als Δ-TSP. Für das Δ-TSP gibt es einen Approximationsalgorithmus, der eine optimale Lösung stets bis auf einen Faktor von 1.5 annähert [ACG+03]. Schränkt man das Δ-TSP noch weiter ein, indem man Punkte im $\mathbb{R}^2$ betrachtet und die Entfernungen zwischen diesen Punkten als ihren euklidischen Abstand definiert, so kann man das Problem beliebig gut approximieren, d. h., es gibt dann sogar ein so genanntes polynomielles Approximationsschema [ACG+03].

Δ-TSP

Der in Abschnitt 8.1.3 vorgestellte Algorithmus mit dynamischer Programmierung für das Problem des Handelsreisenden wurde von Held und Karp [HK62] entworfen. Obwohl die Idee der dynamischen Programmierung für dieses Problem seit 1962 bekannt ist und nicht sehr schwierig wirkt, ist dieser Algorithmus noch immer der beste bekannte exakte Algorithmus für dieses Problem.

Riege und Rothe [RR05] lieferten das erste Resultat, das die triviale $\tilde{\mathcal{O}}(3^n)$-Schranke für 3-DNP verbesserte. Fomin, Grandoni, Pyatkin und Stepanov [FGPS08] gelang mit Satz 8.10 und Korollar 8.11 eine weitere Verbesserung für 3-DNP und ein allgemeineres Resultat, da sie auch die Exponentialzeit-Schranke für die Berechnung der domatischen Zahl verbesserten. Ihre Methode wird in der Übersichtsarbeit von Fomin, Grandoni und Kratsch [FGK05] als „Measure and Conquer“ bezeichnet.

Measure and Conquer

Satz 8.12 wurde von Yamamoto [Yam05] bewiesen. Die in Satz 8.13 angegebene Zeitschranke von $\tilde{\mathcal{O}}(2.6129^n)$ ist etwas besser als die in der Originalarbeit von Riege, Rothe, Spakowski und Yamamoto [RRSY07] bewiesene Schranke von $\tilde{\mathcal{O}}(2.695^n)$. Das liegt aber nur daran, dass in dieser Arbeit ein älteres Resultat von Fomin et al. [FGPS05] benutzt wurde, das diese Autoren später verbesserten [FGPS08].

Björklund und Husfeldt [BH06] konnten sogar einen $\tilde{\mathcal{O}}(2^n)$-Algorithmus für das Berechnen der domatischen Zahl finden. Allerdings erfordert dieser Algorithmus exponentiellen Raum. Außerdem geben sie einen $\tilde{\mathcal{O}}(2.8805^n)$-Algorithmus für dasselbe Problem an, der in polynomialem Raum arbeitet.

Riege et al. [RRSY07] gaben auch randomisierte Algorithmen für 3-DNP an, deren Laufzeiten vom maximalen Knotengrad des Eingabegraphen abhängen und die auf dem randomisierten CSP-Algorithmus von Schöning [Sch99] beruhen, siehe auch [Sch02] sowie die Übersichtsarbeiten von Schöning [Sch05] und Woeginger [Woe03].