Institut für Theoretische Informatik
Leibniz Universität Hannover

Bachelorarbeit

Komplexität von „Common Approximate Substring“

Luisa Simmet
Matrikelnummer: 2942260

6. August 2015

Erstprüfer: Prof. Dr. Heribert Vollmer
Zweitprüfer: Dr. Arne Meier
Betreuerin: Dipl.-Math. Irena Schindler

Erklärung

Hiermit versichere ich, dass ich die vorliegende Bachelorarbeit selbstständig verfasst und keine anderen als die angegebenen Hilfsmittel und Quellen genutzt habe.

Hannover, den 6. August 2015

_____

Luisa Simmet

Inhaltsverzeichnis

1 Einleitung

Das Finden einer Zeichenkette (String) in einem Text oder längeren String, genannt String-Matching, ist ein vielseitiges Problem, das in verschiedenen Bereichen auftaucht. Eine große Bedeutung hat es nicht nur für die Suche in Texten, sondern unter anderem auch in der Biologie, wo sich viele Probleme mit DNA- und Proteinsequenzen befassen, die als Zeichenketten über einem speziellen Alphabet aufgefasst werden können.

Sind nun gewisse Fehler erlaubt, sodass der gesuchte String nicht exakt auftreten muss, sondern nur ein ähnlicher String gesucht wird, wird das Problem als approximatives String-Matching bezeichnet und kann beispielsweise zur Korrektur falsch geschriebener Wörter eingesetzt werden, spielt aber auch in der Genetik eine große Rolle, um zum Beispiel DNA-Sequenzen zu erkennen, die durch Mutationen verändert wurden. Im Vergleich zur exakten Suche ist im approximativen String-Matching die Komplexität zugehöriger Probleme höher, da diese durch die Beachtung möglicher Fehler schwieriger und aufwendiger zu lösen sind.

Zu den Teilproblemen des approximativen String-Matching gehören Substring-Probleme, beispielsweise das Finden eines Substrings, der in zwei Strings mit einer gewissen erlaubten Abweichung vorkommt. Betrachtet man ein Substring-Problem für eine beliebige Menge von Strings, ergibt sich das Problem, einen String zu finden, der in allen Strings der Menge approximativ als Substring vorkommt. Dieses Problem wird mit CAS (Common Approximate Substring) bezeichnet.

In der Biologie ist es beispielsweise interessant, Gemeinsamkeiten in allen Formen eines Proteins zu finden, das in unterschiedlichen Organismen vorkommt, um die für das Protein charakteristischen Strukturen oder dessen Evolutionsgeschichte zu bestimmen.

In dieser Arbeit wird die Komplexität von CAS als Problem des approximativen String-Matching betrachtet. Wir werden zeigen, dass CAS NP-vollständig ist und verschiedene Algorithmen betrachten.

Da CAS ein schwieriges Problem ist, soll es im Rahmen der Theorie parametrisierter Komplexität untersucht werden. Diese liefert Methoden, um herauszufinden, welche Eigenschaften von Instanzen genau die algorithmische Schwierigkeit verursachen. Hierzu wollen wir einige so genannte Parameter betrachten, die sich aus der Struktur des Problems ergeben, darunter die Anzahl der Strings und die Größe des zugrunde liegenden Alphabets.

Außerdem wird ein Ausblick auf weitere verwandte Probleme gegeben, auf die einige der Resultate von CAS übertragen werden können. Beispielsweise ergibt sich bei der Verallgemeinerung von Substrings auf nicht notwendigerweise zusammenhängende Subsequenzen das verwandte Problem CASEQ (Common Approximate Subsequence).

2 Grundlagen

2.1 Approximatives String-Matching

Approximatives String-Matching bezeichnet das Problem, eine Zeichenkette p in einer längeren Zeichenkette t zu finden, wobei gewisse Abweichungen erlaubt sind. Im Gegensatz zum exakten String-Matching sollen also Teilzeichenketten gefunden werden, die p ähnlich sind. Dazu ist es notwendig, sich über den Begriff des Abstands zwischen Zeichenketten Gedanken zu machen. Da es mehrere Möglichkeiten gibt, diesen festzulegen, soll der Begriff des Abstands als Distanzfunktion in 2.1.1 formalisiert werden.

Das beschriebene Suchproblem ist wie folgt definiert:

Definition 2.1 (Suchproblem)**.** Gegeben ist ein Alphabet Σ, eine Zeichenkette $p \in \Sigma^*$, (das *Pattern*), eine Zeichenkette $t \in \Sigma^*$, (der Text). Weiterhin sei eine Distanzfunktion $d : \Sigma^* \times \Sigma^* \to \mathbb{N}$ und eine Fehlerschranke $k \in \mathbb{N}$ gegeben. Gesucht sind dann alle Positionen im Text t, an denen sich Teilzeichenketten t' befinden mit $d(t', p) \leq k$.

Weiterhin können Substring- bzw. Subsequenz-Probleme als Teilprobleme des approximativen String-Matching betrachtet werden, also beispielsweise das Problem, eine Subsequenz zu finden, die in zwei Strings enthalten ist. Also kann ein approximatives Vorkommen auch als ein Vorkommen als Subsequenz bzw. approximativer Substring aufgefasst werden.

2.1.1 Distanzfunktion

Allgemein definiert eine Distanzfunktion $d(u, v)$ nur die minimalen Kosten für die Transformation des Strings u in den String v über definierte Schritte. Eine Distanzfunktion ist hierbei eine Metrik über einer Menge von Wörtern [Par13].

Definition 2.2. $d : \Sigma^* \times \Sigma^* \to \mathbb{N}$ ist eine Distanzfunktion, wenn d Metrik auf Σ^* ist, d. h. für alle $u, v \in \Sigma^*$ gilt:
1. $d(u, v) \geq 0$,
2. $d(u, v) = 0$ genau dann, wenn $u = v$,
3. $d(u, v) = d(v, u)$, und
4. $d(u, v) \leq d(u, z) + d(z, v)$ für jedes $z \in \Sigma^*$.

Nun muss die Distanzfunktion weiter präzisiert werden, um praktisch an das Problem herangehen zu können. Je nach Anwendung gibt es verschiedene Möglichkeiten, diese zu definieren. Die Definition der Distanzfunktion kann zum Beispiel über Ausrichtungen, Korrespondenzen oder Edit-Operationen erfolgen.

Eine Ausrichtung erhält man, indem man die Zeichen der beiden Zeichenketten mit entsprechenden Lücken an einigen Positionen gegeneinander ausrichtet und Kosten für jeweils gegenüberliegende Zeichen festlegt.

Für eine Korrespondenz hingegen definiert man sich korrespondierende Zeichen in beiden Zeichenketten und legt Kosten für korrespondierende und nicht korrespondierende Zeichen fest.

Meist wird die Distanzfunktion jedoch im Sinne eines Edit-Distanz-Modells betrachtet; wir definieren diese also nun über Edit-Operationen, die eine Zeichenkette in die andere überführen. Im Edit-Distanz-Modell wird die Transformation im Allgemeinen als eine Reihe nacheinander angewandter Operatoren aufgefasst, wobei jeder Operator die Kosten für die Transformation eines Substrings in einen anderen angibt. Die Kosten der Abstandsfunktion ergeben sich dann als die Summe der atomaren Kosten der Operatoren. Die konkrete Wahl der Operatoren ist abhängig von der Anwendung, mögliche Operatoren sind Einfügen, Löschen, Ersetzen und Transponieren.

Je nach Wahl der Operatoren ergeben sich verschiedene Distanzmaße, die abhängig von der jeweiligen Anwendung genutzt werden können. Die wichtigsten Distanzen sind die Levenshtein-Distanz, Edit-Distanz und Hamming-Distanz, die nun definiert werden und Distanzfunktionen im obigen Sinne sind [Ric04; Par13; Kop+14]. Neben diesen sind aber auch noch andere Abstandsfunktionen möglich.

Definition 2.3 (Levenshtein-Distanz)**.** Die Levenshtein-Distanz $d_L(x, y)$ zwischen zwei Strings x und y ist die minimale Zahl von Einfügungen, Löschungen und Substitutionen einzelner Zeichen, die benötigt wird, um x in y zu überführen.

Definition 2.4 (Edit-Distanz)**.** Die Edit-Distanz $d_E(x, y)$ zwischen zwei Strings x und y ist die minimale Zahl von Einfügungen und Löschungen, die x in y überführen.

Definition 2.5 (Hamming-Distanz)**.** Für $|x| = |y|$ ist die Hamming-Distanz $d_H(x, y)$ zwischen zwei Strings x und y definiert als die Anzahl der Positionen in x und y, an denen verschiedene Zeichen stehen.
Ist $|x| \neq |y|$, so ist $d_H(x, y) = \infty$.
Erlaubte Operatoren sind also nur Ersetzungen. Da Einfüge- und Löschoperatoren fehlen, bietet sich die Hamming-Distanz zum Vergleich von Strings gleicher Länge an.

2.1.2 Substrings und Subsequenzen

In diesem Abschnitt sollen nun die Begriffe *Substring* und *Subsequenz*, welche für die späteren Probleme erforderlich sind, voneinander abgegrenzt werden [Kop+14; Par13].

Definition 2.6. Ein *Substring* z der Länge l von $s = (s_0, \ldots, s_{|s|-1}) \in \Sigma^*$ ist ein String der Form $(s_i, s_{i+1}, \ldots, s_{i+l-1})$ für $0 \leq i \leq |s| - l$.

z heißt *gemeinsamer Substring (common substring)* von x und y, falls z Substring von x und y ist.

Definition 2.7. Eine *Subsequenz* z der Länge l von $s = (s_0, \ldots, s_{|s|-1}) \in \Sigma^*$ ist ein String der Form $(s_{i_0}, \ldots, s_{i_{l-1}})$ mit $0 \leq i_0 < i_1 < \ldots < i_{l-1} < |s|$.

z heißt *gemeinsame Subsequenz (common subsequence)* von x und y, falls z Subsequenz von x und y ist.

Eine Subsequenz ist im Gegensatz zum Substring also nicht notwendigerweise zusammenhängend. Demnach sind alle Substrings Subsequenzen, aber nicht alle Subsequenzen sind auch Substrings.

Beispiel 2.1. Gegeben sei ein String $s = abcabdacb$. Dann sind *cabd*, *abc* und *dac* sowohl Substrings als auch Subsequenzen von s, *bcb* und *cda* sind jedoch nur Subsequenzen.

Definition 2.8. z heißt *Superstring* von x, falls x Substring von z ist.
z heißt *gemeinsamer Superstring* von x und y, falls z Superstring von x und y ist.

Definition 2.9. z heißt *Supersequenz* von x, falls x Subsequenz von z ist.
z heißt *gemeinsame Supersequenz* von x und y, falls z Supersequenz von x und y ist.

Fragestellungen, die sich mit Substrings bzw. Subsequenzen beschäftigen, können als Teil des approximativen String-Matching betrachtet werden. Zwei dieser Probleme werden nun definiert.

Definition 2.10 (Längster gemeinsamer Substring)**.** Die Länge des *längsten gemeinsamen Substrings* $lcf(s,t)$ („f“ für factor) von $s, t \in \Sigma^*$ ist die Länge eines längsten Strings, der ein gemeinsamer Substring von s und t ist (*longest common substring*).

Definition 2.11 (Längste gemeinsame Subsequenz)**.** Die Länge der *längsten gemeinsamen Subsequenz* $lcs(s,t)$ von $s, t \in \Sigma^*$ ist die Länge eines längsten Strings, der eine gemeinsame Subsequenz von s und t ist (*longest common subsequence*).

Beispiel 2.2. Seien zwei Strings *abcabba* und *cbabac* gegeben. Es gilt $lcs(abcabba, cbabac) \geq 4$, denn *baba* oder *cbba* sind gemeinsame Subsequenzen.

Diese Probleme für zwei Strings können nun mithilfe einer Distanzfunktion einfach berechnet werden [Par13].

2.1.3 Ansätze für Algorithmen

Zur Berechnung einer allgemeinen Distanzfunktion kann der Algorithmus von Wagner und Fischer genutzt werden, der auf dynamischer Programmierung beruht und mit einer Distanzmatrix arbeitet. Dieses Verfahren ist am universellsten einzusetzen. Es gibt aber verschiedene speziellere Verfahren, die effizienter sind und sich meist auf eine bestimmte Distanzfunktion beschränken, zum Beispiel wird im Algorithmus von Hirschberg oder im Verfahren von Ukkonen die Levenshtein-Distanz genutzt (siehe [Par13]).

Zur Berechnung von Substring- bzw. Subsequenz-Problemen zweier Zeichenketten, zum Beispiel *lcs*, kann ebenfalls der Wagner-Fischer-Algorithmus genutzt werden.

Wenn wir uns nun dem Suchproblem zuwenden, also dem Problem, Teilzeichenketten zu finden, die einer vorgegeben Zeichenkette ähnlich sind, d. h. einen gewissen Abstand erlauben, gibt es eine Vielzahl von Algorithmen. Einige wichtige sind der DotPlot-Algorithmus, der Algorithmus von Sellers, das Verfahren von Chang und Lampe und das Verfahren von Wu und Manber.

Die Klassifikation der Algorithmen kann zum Beispiel über Online- und Offline-Algorithmen erfolgen: Online-Algorithmen können die Suche sofort durchführen, ohne Vorberechnungen leisten zu müssen, Offline-Algorithmen hingegen benötigen Vorberechnungen auf dem Text. Eine andere Möglichkeit ist die Klassifikation nach dem Grundprinzip des Algorithmus: Dieser kann DP-basiert sein, d. h. auf einer Matrix der dynamischen Programmierung beruhend, oder auf Automaten oder Filtern basieren.

2.1.4 Anwendungen

Ein Gebiet, in dem approximatives String-Matching rege Anwendung findet, ist der Bereich der *Computational Biology*. In der Genetik und Molekularbiologie befassen sich viele der wichtigen Probleme mit DNA- und Proteinsequenzen. Diese können als Zeichenketten über einem speziellen Alphabet aufgefasst werden. Da beispielsweise die Erbinformation der Zellen in Form von langen Strängen aus vier verschiedenen Nukleinsäuren vorliegt, kann diese als eine Zeichenkette über einem vierelementigen Alphabet aufgefasst werden. Das Alphabet besteht aus den Symbolen A für Adenin, G für Guanin, C für Cytosin und T für Thymin.

Die Suche von spezifischen Sequenzen in diesen Zeichenketten ist eine wichtige Operation bei der Lösung von Problemen wie zum Beispiel der Suche nach bestimmten Eigenschaften in DNA-Strängen, dem Erkennen von DNA-Sequenzen, die durch Mutationen oder andere Manipulationen verändert wurden, der Bestimmung der Unterschiedlichkeit zweier genetischer Sequenzen oder dem Zusammensetzen von DNA-Fragmenten zu einem DNA-Strang.

Aufgrund leicht ungenauer experimenteller Daten, real aufgetretener Mutationen oder aufgrund der Problemstellung (Ähnlichkeitsbestimmung) ist hier eine Suche notwendig, die Fehler erlaubt.

Ein weiteres Anwendungsgebiet ist der Bereich des *Text Retrieval*. Beispielsweise kann approximatives String-Matching zur Korrektur falsch geschriebener Wörter eingesetzt werden, was im Information Retrieval eine große Bedeutung hat. Außerdem lässt sich zum Beispiel ein fehlerhaftes Wort in einer Datenbank wiederfinden. Weitere Anwendungen, die sich mit der Verarbeitung von Texten beschäftigen, sind Rechtschreibprüfungen, Schnittstellen zur Spracheingabe und computergestütztes Lernen.

Es folgen nun einige Angaben, um einen Begriff von den Größenordnungen der Problemparameter in der Praxis zu bekommen:

- Die *Größe des Alphabets* reicht von 4 Zeichen bei DNA-Sequenzen bis hin zu mehreren Hundert oder Tausend Zeichen in der natürlichen Sprache.

- Die *Länge des Textes* liegt zwischen wenigen Tausend Zeichen in der Computational Biology bis hin zu mehreren Millionen im Text Retrieval.
- Die *Länge des Pattern* schwankt zwischen einigen wenigen Zeichen bei der Textsuche bis hin zu mehreren Hundert Zeichen in der Computational Biology.
- Die *Fehlerschranke*, meist die Anzahl falscher Zeichen, überschreitet in der Regel nicht die halbe Patternlänge.

2.2 Parametrisierte Komplexität

Betrachtet man schwere Probleme, können folgende Fragen auftauchen: Welche Instanzen von etwa NP-vollständigen Problemen sind effizient zu lösen? Von welchen Parametern hängt die Laufzeit im Wesentlichen ab?

Die parametrisierte Komplexitätstheorie liefert Mechanismen zum Analysieren der Effekte von bestimmten Parametern auf die Komplexität des Problems, es wird also die Komplexität bezüglich eines von der Eingabe abhängigen Parameters k betrachtet.

Es folgen nun einige Definitionen:

Definition 2.12. Sei Σ ein Alphabet. Eine *Parametrisierung* von Σ^* ist eine Abbildung $\kappa : \Sigma^* \to \mathbb{N}$, die in Polynomialzeit berechenbar ist.

Definition 2.13. Ein *parametrisiertes Problem (PP)* über dem Alphabet Σ ist ein Paar (Q, κ) mit Sprache $Q \subseteq \Sigma^*$ und Parametrisierung κ von Σ^*.

Beispiel 2.3 (Clique als PP)**.**

CLIQUE

Instanz: Ein Graph $G = (V, E)$, eine positive ganze Zahl k.

Parameter: k

Frage: Gibt es eine Menge $V' \subseteq V$ mit k Knoten, die eine Clique von G ist?

Definition 2.14. Sei (Q, κ) ein PP und $l \in N$. Dann nennen wir $(Q, \kappa)_l$ eine *Scheibe* (engl. *slice*) von (Q, κ) mit $(Q, \kappa)_l = \{x \in \Sigma^* \mid x \in Q, \kappa(x) = l\}$.

2.2.1 Komplexitätsklassen

Die parametrisierten Komplexitätsklassen sollen nun definiert werden.

Definition 2.15. (Q, κ) ist *fixed-parameter tractable*, falls es eine deterministische Turingmaschine (DTM) gibt, die $x \in Q$ entscheidet und Laufzeit $f(\kappa(x)) \cdot p(|x|)$ hat, wobei f eine berechenbare Funktion und p ein Polynom ist.

Definition 2.16. **FPT** ist die Klasse der PP, die fixed-parameter tractable sind.

Die Klasse **FPT** kann als Analogon zu **P** verstanden werden, der Klasse der effizient lösbaren Probleme. Der Zusammenhang besteht darin, dass jede Scheibe eines PP (Q, κ) in **P** liegt, falls (Q, κ) in **FPT** liegt.

Da jedoch die Umkehrung nicht gilt, definieren wir eine weitere Klasse **XP**.

Definition 2.17. **XP** ist die Klasse der PP (Q, κ), für die $(Q, \kappa)_l \in \mathbf{P}$ für alle l gilt.

Erlaubt man nun statt einer DTM in der Definition von **FPT** eine NTM, erhält man die Klasse para-**NP**, das Analogon zur Klasse **NP**.

Definition 2.18. Die Klasse para-**NP** enthält die PP (Q, κ), $Q \subseteq \Sigma^*$, für die es eine nichtdeterministische Turingmaschine (NTM) M gibt, die $x \in Q$ für alle $x \in \Sigma^*$ mit Laufzeit $f(\kappa(x)) \cdot p(|x|)$ entscheidet, wobei f eine berechenbare Funktion und p ein Polynom ist.

Satz 2.19 ([FG02])**.** Ein PP Π ist schwer für para-**NP**, falls eine Vereinigung endlich vieler Scheiben bereits **NP**-schwer ist.

Nun definieren wir die Klasse **W**[**P**], die sich im Schnitt von **XP** und para-**NP** befindet.

Definition 2.20. Sei Σ ein Alphabet und κ eine Parametrisierung von Σ^*. Eine NTM M über Σ heißt *κ-beschränkt*, falls es berechenbare Funktionen h, f und ein Polynom p gibt, sodass für alle $x \in \Sigma^*$ gilt:

(1) Alle Rechenwege von $M(x)$ haben Laufzeit $\leq f(\kappa(x)) \cdot p(|x|)$.

(2) Auf allen Rechenwegen von $M(x)$ gibt es $\leq h(\kappa(x)) \cdot log|x|$ nichtdeterministische Verzweigungen.

W[**P**] ist die Klasse aller PP (Q, κ), die von κ-beschränkten NTMn entschieden werden können.

Schwere-Resultate ermöglichen es, untere Schranken für die Komplexität von Problemen abzuschätzen. Dazu führen wir den Begriff der Reduktion für die Klassen der parametrisierten Komplexität ein.

Definition 2.21. Seien (Q, κ), (Q', κ') PP, $Q \subseteq \Sigma^*$, $Q' \subseteq \Delta^*$. Eine *fpt-many-one-Reduktion* von (Q, κ) auf (Q', κ') ist eine Abbildung $h : \Sigma^* \to \Delta^*$ mit folgenden Eigenschaften:

(1) $x \in Q$ genau dann, wenn $h(x) \in Q'$ für alle $x \in \Sigma^*$.

(2) Es gibt eine DTM M, eine berechenbare Funktion f und ein Polynom p, sodass für alle $x \in \Sigma^*$ gilt: M berechnet $h(x)$ in Zeit $f(\kappa(x)) \cdot p(|x|)$.

(3) Es gibt eine berechenbare Funktion $g : \mathbb{N} \to \mathbb{N}$, sodass für alle $x \in \Sigma^*$ gilt: $\kappa'(h(x)) \leq g(\kappa(x))$.

Die ersten beiden Bedingungen sind analog zu den zwei Bedingungen, die für eine Polynomialzeit-many-one-Reduktion gelten müssen. Die dritte Bedingung fordert, dass der Parameter der Reduktionsinstanz beschränkt ist in Abhängigkeit des Parameters der ursprünglichen Instanz x .

Nun wollen wir die W-Hierarchie betrachten, wofür *Boolesche Schaltkreise* benötigt werden.

Definition 2.22 (Boolescher Schaltkreis)**.** Ein *Boolescher Schaltkreis* C mit n Eingängen ist ein gerichteter, azyklischer Graph mit Eingabeknoten $x_1, x_2, \ldots, x_n$, inneren Knoten (Gatter) sowie einem Ausgangsknoten mit einer Markierung aus $\{\wedge, \vee, \neg\}$. Die *Tiefe* ist die maximale Distanz von einem Eingangsknoten zu einem Ausgangsknoten. Der Schaltkreis C berechnet eine Funktion $f_c : \{0,1\}^n \to \{0,1\}$ entsprechend der Markierungen der Gatter. C heißt erfüllbar, falls es ein $x \in \{0,1\}^n$ gibt, sodass $f_c(x) = 1$. Die Anzahl der Einsen in x nennen wir das *Gewicht* von x.

Definition 2.23 (Weft)**.** Ein Gatter heißt *großes Gatter*, wenn die Anzahl der Eingänge eine festgelegte Grenze übersteigt. Die *Weft* eines Schaltkreises ist die maximale Anzahl von Alternierungen zwischen großen Und-Gattern und großen Oder-Gattern auf einem Eingabe-Ausgabe-Pfad, wobei keine Negation vor einem großen Gatter stehen darf.

Das *gewichtete Erfüllbarkeitsproblem für Schaltkreise* ist das folgende parametrisierte Problem:

WEIGHTED-CIRCUIT-SAT (WCS)

Instanz: Ein Boolescher Schaltkreis C, eine positive ganze Zahl k.

Parameter: k

Frage: Existiert eine erfüllende Belegung der Eingabegatter von C vom Gewicht k?

Mit $\text{WCS}_{t,h}$ bezeichnen wir das Problem WCS eingeschränkt auf Schaltkreise mit Tiefe h und Weft t. Nun können wir mithilfe von $\text{WCS}_{t,h}$ die Klasse $\mathbf{W}[\mathbf{t}]$ definieren:

Definition 2.24. Ein PP (Q, κ) gehört zur Komplexitätsklasse $\mathbf{W}[\mathbf{t}]$, $t \geq 1$, genau dann, wenn es ein h gibt, sodass (Q, κ) mittels einer parametrisierten Reduktion auf $\text{WCS}_{t,h}$ reduziert werden kann.

Nun folgt eine alternative Definition für die Klasse $\mathbf{W}[\mathbf{P}]$:

Satz 2.25. Ein PP (Q, κ) gehört zur Komplexitätsklasse $\mathbf{W}[\mathbf{P}]$ genau dann, wenn (Q, κ) mittels einer parametrisierten Reduktion auf WCS reduziert werden kann.

Die W-Hierarchie sieht nun wie folgt aus:

$\mathbf{W}[\mathbf{1}] \subseteq \mathbf{W}[\mathbf{2}] \subseteq \ldots \subseteq \mathbf{W}[\mathbf{P}]$

Insgesamt gelten für die Komplexitätsklassen folgende Inklusionen:

$\mathbf{FPT} \subseteq \mathbf{W}[\mathbf{1}] \subseteq \mathbf{W}[\mathbf{2}] \subseteq \ldots \subseteq \mathbf{W}[\mathbf{P}] \subseteq \mathbf{XP}$

3 Approximatives String-Matching für Mengen

In diesem Kapitel betrachten wir die Ähnlichkeit von mehr als zwei Strings sowie Probleme des approximativen String-Matching, die sich daraus ergeben. Das Ziel des approximativen String-Matching mit mehreren Strings ist es, einen String zu finden, der ähnlich zu jedem String in einer Menge ist, also zu jedem String der Menge höchstens die Distanz d hat.

Auch hier können Substring-Probleme für Mengen von Strings als Teilgebiet des approximativen String-Matching betrachtet werden, zum Beispiel kann die Ähnlichkeit zu allen Strings als approximatives Vorkommen als Substring in allen Strings interpretiert werden.

3.1 Bedeutung in der Praxis

Die Bedeutung von Problemen, die sich mit der Ähnlichkeit von mehr als zwei Strings beschäftigen, wird als Anwendung ebenfalls in der Biologie deutlich.

Dazu kann man zum Beispiel das Hämoglobin betrachten, ein Protein, das es in diversen Formen in den unterschiedlichsten Organismen gibt. Vergleicht man die Varianten von zwei Säugetieren, so wird man eine große Übereinstimmung finden, vergleicht man dagegen das Hämoglobin eines Säugetiers mit dem eines Insekts, wird man keine große Übereinstimmung finden. Aus biologischer Sicht ist es daher interessant, Gemeinsamkeiten in allen Formen des Hämoglobins zu finden, also mehr als zwei Varianten zu vergleichen, um die Strukturen zu finden, die charakteristisch für das Protein sind, oder um die Evolutionsgeschichte des Proteins zu bestimmen. Für diese und ähnliche Fragen benötigt man Methoden zum Vergleichen und Bewerten der Ähnlichkeit von mehr als nur zwei Zeichenketten.

3.2 Ähnlichkeit und Abstand

Nun soll betrachtet werden, wie man sich die Ähnlichkeit von mehr als zwei Strings vorstellen kann. Dazu wird die Distanzfunktion erweitert, die nun über multiple Ausrichtungen definiert wird und deren Berechnung im Gegensatz zur Distanz von zwei Strings schwierig ist [Par13].

Das Prinzip der Ausrichtung für zwei Strings wurde bereits in Kapitel 2 erwähnt. Ausrichtungen für mehr als zwei Strings heißen multiple Ausrichtungen und vergleichen einzelne Positionen von Sequenzen mit dem Ziel, sie so auszurichten, dass sie in möglichst vielen Positionen identisch oder ähnlich sind. Dazu ordnet man die Elemente eines untersuchten

Strings denen der anderen Strings so zu, dass die Reihenfolge erhalten bleibt und jedes Element einem anderen Element oder einer Leerstelle in jedem String zugeordnet ist.

Beispiel 3.1. Eine mögliche Ausrichtung der Zeichenketten *abccc*, *dbddbc* und *ddcddd* wäre:
$a\ b\ c\ -\ c\ -\ c$
$d\ b\ -\ d\ d\ b\ c$
$d\ d\ c\ d\ d\ -\ d$

Das Ziel ist es nun, eine optimale Ausrichtung unter einer Kostenfunktion zu finden. Dazu gibt es mehrere Möglichkeiten, die Distanz für eine Ausrichtung festzulegen bzw. diese zu bewerten.

Beispielsweise besteht eine Möglichkeit darin, die Kosten jeweils paarweise für zwei Zeichenketten zu bestimmen und diese Werte zu addieren.

3.3 String-Probleme

Im approximativen String-Matching tauchen einige schwere Probleme mit Substrings und Subsequenzen auf, zum Beispiel einen Substring zu finden, der approximativ in jedem String einer Menge vorkommt. Wegen der exponentiellen Anzahl möglicher Substrings sind solche Probleme schwierig.

3.3.1 Common Approximate Substring (CAS)

Nun wollen wir uns dem Problem Common Approximate Substring (CAS) zuwenden, das erstmalig von Evans, Smith und Wareham untersucht wurde [ESW03]. Dazu folgen zunächst einige Begriffe.

Ein *approximatives Auftreten* eines Strings x in einem String s bedeutet hier, dass es einen Substring von s mit einer bestimmten Distanz zu x gibt; x ist dann ein *approximativer Substring* von s. Ist x ein approximativer Substring von jedem String einer Menge, heißt er *common approximate substring (gemeinsamer ähnlicher Substring)*.

Nun definieren wir das Problem CAS:

COMMON APPROXIMATE SUBSTRING (CAS)

Instanz: Eine Menge $\mathcal{F} = \{S_1, \ldots, S_m\}$ von Strings über einem Alphabet Σ, $l \in \mathbb{N}$, $d \in \mathbb{N}$, $n \in \mathbb{N}$, sodass $|S_i| \leq n$, $1 \leq i \leq m$, $1 \leq l \leq n$, $1 \leq d \leq l$.

Frage: Gibt es einen String $C \in \Sigma^l$ (einen *Center String*), sodass es für alle $S \in \mathcal{F}$ einen Substring $s \in \Sigma^l$ mit $d_H(s, C) \leq d$ gibt?

Die Funktion d_H bezeichnet die Hamming-Distanz, die hier für die Definition des Abstands genutzt wird. Diese ist geeignet, da nur Substrings gleicher Länge l betrachtet werden.

Abhängig von l, n und d ergeben sich mehrere Spezialfälle von CAS:

- Für $l = n$ entspricht CAS dem Problem Closest String, ein Optimierungsproblem, in dem ein String gesucht wird, dessen Abstand zu jedem String der Menge möglichst klein ist.
- Für $l \leq n$ ergibt sich das Problem Closest Substring, ein Optimierungsproblem, in dem ein String gesucht wird, sodass jeder String der Menge einen Substring besitzt, zu dem der Abstand zum gesuchten String möglichst klein ist.
- Eine Variante mit $l < n$ und $d = 0$ ist das Problem Longest Common Substring, in dem der längste Substring gesucht wird, der in allen Strings der Menge vorkommt.

Beispiel 3.2. Sei $\mathcal{F} = \{abcdef, abdghi, abegij\}$ mit $n = 6$, $m = 3$. Sei $d = 1$ und $l = 3$. Dann ist *abc* ein möglicher *common approximate substring*, denn für jeden String aus $\mathcal{F}$ gibt es einen Substring der Länge drei, zu dem die Hamming-Distanz höchstens eins ist.

3.3.2 Common Approximate Subsequence (CASEQ)

Wenn wir nun Subsequenzen anstatt Substrings betrachten, führt uns dies zur nächsten Problemdefinition. Ein ähnliches Problem, das als eine Erweiterung von CAS gesehen werden kann, ist das Problem Common Approximate Subsequence (CASEQ).

COMMON APPROXIMATE SUBSEQUENCE (CASEQ)

Instanz: Eine Menge $\mathcal{F} = \{S_1, \ldots, S_m\}$ von Strings über einem Alphabet Σ, $l \in \mathbb{N}$, $d \in \mathbb{N}$, $n \in \mathbb{N}$, $g \in \mathbb{N} \cup \{\infty\}$, sodass $|Si| \leq n$, $1 \leq i \leq m$, $1 \leq l \leq n$, $0 \leq d \leq l$.

Frage: Gibt es einen String $C \in \Sigma^l$, sodass es für alle $S \in \mathcal{F}$ eine Subsequenz $s \in \Sigma^l$ mit $d_H(s, C) \leq d$ und Symbolabstand g gibt?

Hier können je nach d und g Spezialfälle von CASEQ unterschieden werden, wobei g die maximale Entfernung von zwei Zeichen aus der gesuchten Subsequenz in einem String der Menge ist, zum Beispiel gilt $g = 3$ für $s = axxxbyyc$ und $C = abc$.

- CASEQ ($d = 0$, $g = 0$) $\rightarrow$ Longest Common Substring
- CASEQ ($d \geq 0$, $g = 0$) $\rightarrow$ Common Approximate Substring
- CASEQ ($d = 0$, $g \geq 0$) $\rightarrow$ Longest Common Subsequence (LCS)
- CASEQ ($d \geq 0$, $g \geq 0$) $\rightarrow$ Common Approximate Subsequence

Beispiel 3.3. Sei $\mathcal{F} = \{abcdef, abdghi, abegij\}$ mit $n = 6$, $m = 3$. Sei $d = 1$ und $l = 3$. Dann ist *abg* eine mögliche *common approximate subsequence*, da es für jeden String aus $\mathcal{F}$ eine Subsequenz der Länge drei gibt, zu der die Hamming-Distanz höchstens eins ist.

Wir werden später sehen, dass sich die beiden Probleme in ihrer Komplexität ähnlich sind und einige Resultate der parametrisierten Komplexität von CAS auf CASEQ übertragen werden können.

4 Komplexität von CAS

Das Problem CAS ist NP-vollständig. Betrachten wir die verschiedenen Varianten, können wir nochmals differenzieren:

- $l = n$: Closest String
- $l \leq n$: Closest Substring
- $l < n$, $d = 0$: Longest Common Substring

Die Probleme Closest String und Closest Substring sind NP-schwer, Longest Common Substring liegt jedoch in P, da hier keine Abweichungen erlaubt sind, also exakte statt approximative Auftreten gesucht werden.

Diese Kontraste machen nochmals den Unterschied zwischen der Komplexität von approximativem und exaktem String-Matching deutlich. Das Erlauben gewisser Fehler macht CAS also zu einem schwierigen Problem.

4.1 NP-Vollständigkeit

Um die NP-Vollständigkeit von CAS zu zeigen, müssen sowohl die NP-Schwere als auch die Zugehörigkeit zur Klasse NP nachgewiesen werden.

Um zu zeigen, dass CAS in der Klasse NP liegt, geben wir einen Algorithmus an, der ein Zertifikat für CAS in Polynomialzeit überprüft [Smi04].

Der Algorithmus VALID prüft für einen String C, ob dieser ein gültiger *common approximate substring* ist. Dazu bildet er alle möglichen Substrings der Länge l der gegebenen Menge $\mathcal{F}$, vergleicht diese mit C, zählt die Anzahl der übereinstimmenden Positionen und gibt zurück, ob es für jeden String aus $\mathcal{F}$ einen Substring gibt, zu dem der Abstand zu C höchstens d ist, oder nicht.

Algorithmus 1: VALID-Algorithmus

Eingabe: Eine Menge $\mathcal{F} = \{S_1, ..., S_m\}$ von Strings, ein String $C \in \Sigma^l$ und eine ganze Zahl d mit $1 \leq d \leq l$.

Ausgabe: Ein Boolescher Wert, der angibt, ob es für jeden String aus $\mathcal{F}$ einen Substring der Länge l gibt, zu dem die Distanz zu C höchstens d ist.

VALID $(C, \mathcal{F}, d)$

```
1: l ← |C|
2: x ← true
3: for i ← 1 to m do
4:     x_i ← false
5:     for all Substrings s von S_i der Länge l do
6:         if  d_H(C,s) ≤ d then
7:             x_i ← true
8:         end if
9:     end for
10:    if x_i = false then
11:        x ← false
12:    end if
13: end for
14: return x
```

VALID hat eine Laufzeit von O(mnl), da für jeden String aus $\mathcal{F}$, also m-mal, die Distanz geprüft wird, was mit der Multiplikation der Längen, $n \cdot l$, abgeschätzt werden kann.

Also ist VALID ein deterministischer Polynomialzeit-Algorithmus und außerdem können wir $l \leq n$ annehmen, damit ist CAS polynomiell-überprüfbar und liegt somit in der Klasse NP.

Nun betrachten wir einen zweiten Algorithmus SCORE, der nützlich für Optimierungen ist und zur Anwendung kommt, wenn zusätzlich der *Radius* des Strings C benötigt wird.

Der Radius von C bezüglich der Menge $\mathcal{F}$ ist die kleinstmögliche notwendige Distanz, für die C in allen Strings aus $\mathcal{F}$ einen approximativen Substring besitzt. Dieser Begriff soll nun formal definiert werden.

Definition 4.1 (Radius)**.** Sei $\mathcal{F}$ eine Menge von Strings, l eine positive ganze Zahl. Für ein $S_i \in \mathcal{F}$ sei $s_{i,j}$ der j-te Substring der Länge l von S_i.

Für ein $C \in \Sigma^l$ ist der Radius von C bezüglich $\mathcal{F}$ definiert als:

$$radius_{\mathcal{F}}(C) = \max_{(S_i \in \mathcal{F})} \min_{(1 \leq j \leq m-l+1)} d_H(s_{i,j}, C).$$

Der Radius von $\mathcal{F}$ ist der kleinste Radius, den $\mathcal{F}$ bezüglich eines Centers C haben kann:

$$radius(\mathcal{F}) = \min_{(C \in \Sigma^l)} \max_{(S_i \in \mathcal{F})} \min_{(1 \leq j \leq m-l+1)} d_H(s_{i,j}, C).$$

Der Algorithmus SCORE bestimmt die minimale Distanz von C zu jedem Substring der Länge l aus $\mathcal{F}$ und gibt das Maximum dieser, also den Radius, zurück.

Algorithmus 2: SCORE-Algorithmus

Eingabe: Eine Menge von Strings $\mathcal{F} = \{S_1, \ldots, S_m\}$ und ein String $C \in \Sigma^l$.

Ausgabe: Der Radius von C bezüglich $\mathcal{F}$.

SCORE $(C, \mathcal{F})$

```
1:  l ← |C|
2:  d ← 0
3:  for i ← 1 to m do
4:      d_i ← ∞
5:      for all Substrings s von S_i der Länge l do
6:          if  d_H(C, s) ≤ d_i then
7:              d_i ← d_H(C, s)
8:          end if
9:      end for
10:     if d_i > d then
11:         d ← d_i
12:     end if
13: end for
14: return d
```

Aufgrund der ähnlichen Struktur der beiden Algorithmen stimmen die Laufzeiten von SCORE und VALID asymptotisch überein.

Für die NP-Vollständigkeit bliebe nun noch zu zeigen, dass CAS NP-schwer ist.

Dies wurde durch eine Reduktion von dem NP-vollständigen Problem 3-SAT auf das Problem Closest String nachgewiesen. Da Closest String ein Spezialfall von CAS mit $l = n$ ist, folgt daraus auch die NP-Schwere von CAS.

5 Parametrisierte Komplexität von CAS

Da CAS ein NP-vollständiges Problem ist und daher vermutlich nicht für beliebige Eingaben effizient gelöst werden kann, wollen wir Methoden der parametrisierten Komplexität nutzen, um zu betrachten, welche Effekte verschiedene Parameter auf die Komplexität haben. Die Struktur des Problems bietet dazu einige Parameter, die für die Analyse genutzt werden können.

Dazu führen wir die Schreibweise ein, für die jeweiligen parametrisierten Probleme die Parameter, die kleine Werte annehmen sollen, in Klammern anzugeben, zum Beispiel CAS$(m, |\Sigma|)$.

5.1 CAS als parametrisiertes Problem

Die parametrisierte Komplexität des Problems CAS wurde erstmalig von Evans, Smith und Wareham analysiert [ESW03].

COMMON APPROXIMATE SUBSTRING (CAS)

Instanz: Eine Menge $\mathcal{F} = \{S_1, \ldots, S_m\}$ von Strings über einem Alphabet Σ, $l \in \mathbb{N}$, $d \in \mathbb{N}$, $n \in \mathbb{N}$, sodass $|S_i| \leq n$, $1 \leq i \leq m$, $1 \leq l \leq n$, $1 \leq d \leq l$.

Parameter: $|\Sigma|$, m, n, l und d.

Frage: Gibt es einen String $C \in \Sigma^l$ (einen *Center String*), sodass es für alle $S \in \mathcal{F}$ einen Substring $s \in \Sigma^l$ mit $d_H(s, C) \leq d$ gibt?

Folgende Parameter werden wir also betrachten:

- Größe des Alphabets ($|\Sigma|$)
- Anzahl der Strings in $\mathcal{F}$ (m)
- Länge der Strings in $\mathcal{F}$ (n)
- Länge der gesuchten Substrings (l)
- Hamming-Distanz (d)

5.2 FPT-Resultate

Wir geben nun zwei Algorithmen für CAS an, die die Zugehörigkeit zur Klasse FPT für gewisse Parameter nachweisen.

Der erste Algorithmus generiert alle möglichen Strings der Länge l über Σ und prüft für jeden, ob er ein *Center String* für die Menge $\mathcal{F}$ ist. Die Anzahl dieser Strings ist $|\Sigma|^l$ und jeder kann in der Zeit $O(mnl)$ geprüft werden, indem der VALID-Algorithmus genutzt wird. Damit ist die Laufzeit insgesamt $O(|\Sigma|^l \cdot mnl)$.

Algorithmus 3: Einfache Enumeration

ENUMERATION($\mathcal{F}$)

```
1: for all Substrings s ∈ Σ^l do
2:     if VALID(s) then
3:         return s
4:     end if
5: end for
6: return „Es existiert kein Center.“
```

Satz 5.1. CAS $(|\Sigma|, l) \in$ FPT.

Beweis. Folgt aus Algorithmus 3, der CAS in Zeit $O(l|\Sigma|^l nm) = O(f(|\Sigma|, l) \cdot nm)$ löst. □

Im nächsten Algorithmus wird ein Center nach und nach durch Substitutionen aus einem String aus $\mathcal{F}$ entwickelt. Dafür führen wir ein neues Zeichen $x \notin \Sigma$ ein, den *blocking character*. Dieser induziert immer einen Fehler, wenn er mit einem Zeichen aus $\mathcal{F}$ verglichen wird, da diese alle aus Σ stammen. Sei C ein String der Länge l, in dem x höchstens d-mal auftritt, und s ein Substring der Länge l eines Strings $S \in \mathcal{F}$. Eine *Substitution* von C unter s ist eine Menge von Modifikationen von C, die einige der Vorkommen von x in C mit den Zeichen ersetzt, die in korrespondierenden Positionen in s auftauchen. Eine *minimal matching substitution* ist die kleinstmögliche Substitution, die in $d_H(C, s) \leq d$ resultiert.

Zu Beginn des Algorithmus wird ein beliebiger String $S \in \mathcal{F}$ aus $\mathcal{F}$ entfernt. Für jeden Substring s der Länge l von S wird $C = s$ gesetzt und dann für alle möglichen Mengen mit d Positionen in C jedes Zeichen an dieser Position zu x geändert. In der zweite Stufe wird rekursiv ein Center entwickelt, indem die Positionen, die mit x belegt sind, wieder durch Zeichen aus Σ substituiert werden. Für jeden rekursiven Aufruf wird dabei ein String S' aus $\mathcal{F}$ entfernt. Hat der entwickelte Center C höchstens die Distanz d zu einem Substring von S', findet sofort ein weiterer rekursiver Aufruf für den nächsten String statt. Wenn jedoch alle Substrings von S' in mehr als d Positionen von C abweichen, müssen alternative Center produziert werden, indem C modifiziert wird. Dazu wird für jeden Substring $s \in S$ mit $d_H(s, C) \leq 2d$ eine Menge $M = mm(C, s)$ von alternativen Centern genutzt, die die Strings einer *minimal matching substitution* von C unter s enthält. Für alle C' in M wird ein rekursiver Aufruf gestartet, wobei C' als Center für die weitere

Entwicklung übergeben wird. Wird die Menge $\mathcal{F}$ leer, ist der entwickelte Center gültig für die ursprüngliche Menge $\mathcal{F}$ und wird zurückgegeben.

DEVELOP-CENTER bekommt also als Eingabe $\mathcal{F}$ und C, der beim initialen Aufruf der leere String λ ist, und gibt einen Center zurück, falls dieser existiert.

Algorithmus 4: DEVELOP-CENTER

DEVELOP-CENTER($\mathcal{F}, C$)

1: **if** $\mathcal{F} = \emptyset$ **then**
2: **return** C
3: **end if**
4: **if** $C = \lambda$ **then**
5: Sei S ein beliebiger String aus $\mathcal{F}$.
6: **for all** Substrings s von S der Länge l **do**
7: $C \leftarrow s$
8: **for all** $B \subseteq \{1, ..., l\}$, sodass $|B| = d$ **do**
9: $\forall$ Positionen $p \in B$, substituiere $C[p] \leftarrow x$
10: DEVELOP-CENTER $(\mathcal{F} \setminus \{S\}, C)$
11: **end for**
12: **end for**
13: **end if**
14: **if** $C \neq \lambda$ **then**
15: Sei S ein beliebiger String aus $\mathcal{F}$.
16: branch $\leftarrow$ true
17: **for all** Substrings s von S der Länge l **do**
18: **if** $d_H(s, C) \leq d$ **then**
19: branch $\leftarrow$ false
20: **end if**
21: **end for**
22: **if** branch = false **then**
23: DEVELOP-CENTER $(\mathcal{F} \setminus \{S\}, C)$
24: **end if**
25: **if** branch = true **then**
26: **for all** Substrings s von S der Länge l **do**
27: **if** $d_H(s, C) \leq 2d$ **then**
28: Sei M die Menge $mm(C, s)$.
29: **for all** $C' \in M$ **do**
30: DEVELOP-CENTER $(\mathcal{F} \setminus \{S\}, C')$
31: **end for**
32: **end if**
33: **end for**
34: **end if**
35: **end if**

Die Laufzeit kann mithilfe des Rekursionsbaums bestimmt werden. Der Faktor $n\binom{l}{d}$ repräsentiert dabei den Ausgangsgrad der Wurzel. Die Rekursionstiefe ist maximal d und der Ausgangsgrad an jeder Verzweigung ist höchstens $n\binom{d}{d/2}$. Also ist die maximale Anzahl der Blätter $n\binom{l}{d}(n\binom{d}{d/2})^d$ und für jedes Blatt ist Zeit $O(mn)$ nötig. Also folgt gesamt: $O(n^2 m\binom{l}{d}(\binom{d}{d/2}n)^d)$, also $O(l^d d^{d^2} n^{d+2} m)$.

Satz 5.2. CAS$(n) \in$ FPT.

Beweis. Folgt aus $d \leq l \leq n$ und Algorithmus 4, der CAS in Zeit $O(l^d d^{d^2} n^{d+2} m) = O(f(n) \cdot m)$ löst. □

Aus diesen beiden Resultaten folgen alle FPT-Resultate in folgender Tabelle, die nach und nach weiter mit den Resultaten der parametrisierten Komplexität gefüllt wird:

Tabelle 5.1: FPT-Resultate von CAS

Parameter	-	$\lvert\Sigma\rvert$	m	m, $\lvert\Sigma\rvert$
-	NP-vollständig			
d				
l		FPT		FPT
n	FPT	FPT	FPT	FPT

Insbesondere folgen die FPT-Resultate in der vierten Zeile aus Satz 5.2, wobei weitere Parameter hinzugenommen werden können: Da CAS$(n) \in$ FPT, so gilt auch CAS$(n, |\Sigma|)$ $\in$ FPT, CAS$(n, m) \in$ FPT und CAS $(n, m, |\Sigma|) \in$ FPT.

Das verbleibende Resultat in Zeile drei folgt aus Satz 5.1: Da CAS$(|\Sigma|, l) \in$ FPT, so gilt auch CAS$(m, |\Sigma|, l) \in$ FPT.

5.3 Schwere-Resultate

Die Schwere einiger Varianten von CAS wird nun durch Reduktionen von Problemen mit bekannter parametrisierter Komplexität bewiesen. Folgende Probleme werden dazu genutzt:

CLIQUE

Instanz: Ein Graph $G = (V, E)$, eine positive ganze Zahl k.

Parameter: k

Frage: Gibt es eine Menge $V' \subseteq V$ mit k Knoten, die eine Clique von G ist (d. h. ein vollständiger Teilgraph von G)?

DOMINATING CLIQUE

Instanz: Ein Graph $G = (V, E)$, eine positive ganze Zahl k.

Parameter: k

Frage: Gibt es eine Menge $V' \subseteq V$ mit k Knoten, die eine Clique von G und ein *Dominating Set* von G ist (d. h. jeder Knoten ist entweder in V' oder adjazent zu einem Knoten in V')?

SET COVER

Instanz: Eine Menge $\mathcal{B}$, eine Familie $\mathcal{L}$ von Teilmengen $L_i \subseteq \mathcal{B}$ $(1 \leq i \leq |\mathcal{L}|)$ und eine positive ganze Zahl k.

Parameter: k

Frage: Gibt es eine Teilfamilie $R \subseteq \mathcal{L}$ der Größe k mit $\bigcup_{R_j \in R} R_j = \mathcal{B}$?

Der erste betrachtete Fall enthält die Parameter m, l und d, es wird also in wenigen Strings ein kurzer Center mit kleinem Fehler gesucht.

Satz 5.3. CAS(m, l, d) ist W[1]-schwer.

Beweis. Wir geben eine parametrisierte Reduktion von dem W[1]-vollständigen Problem CLIQUE an.

Dazu konstruieren wir eine Menge $\mathcal{F}$ aus m Strings über Σ, die genau dann einen Center der Länge l hat, wenn der gegebene Graph $G = (V, E)$ eine k-Clique enthält. Wir nehmen an, die Knotenmenge von G ist $V = \{1, ..., |V|\}$, die Knoten sind also nummeriert.

Damit die Konstruktion eine gültige FPT-Reduktion ist, muss sie in Zeit $f(k) \cdot |G|^{O(1)}$ für eine berechenbare Funktion f durchgeführt werden können, und die Parameter von CAS, m, l und d, müssen durch eine berechenbare Funktion von k beschränkt werden können.

Für m, l und d werden nun diese Funktionen festgelegt. Die Anzahl der Strings in $\mathcal{F}$ ist $m = f_1(k) = \binom{k}{2}$, die Länge des Centers C ist $l = f_2(k) = k + 2$ und die maximale Distanz ist $d = f_3(k) = k - 2$. Die maximale Länge eines Strings in $\mathcal{F}$, welche in dieser Reduktion kein Parameter ist, ist $n = f_4(G, k) = (2k + 4)(|E|)$.

Das Alphabet der Strings wird in drei Gruppen mit unterschiedlichen Funktionen für die Konstruktion aufgeteilt: $\Sigma = \Sigma_1 \cup \Sigma_2 \cup \Sigma_3$ mit

$$\begin{aligned}
&\Sigma_1 = \{1, \ldots, |V|\} && (\textit{vertex characters}),\\
&\Sigma_2 = \{\text{Zeichen, die je nur einmal in } \mathcal{F} \text{ auftauchen}\} && (\textit{unique characters}),\\
&\Sigma_3 = \{A, B\} && (\textit{alignment characters}).
\end{aligned}$$

Dabei stellen die *vertex characters* die Knoten des Graphen dar und sind im Center enthalten, sodass dieser die Clique repräsentiert. Die *alignment characters* begrenzen eine Komponente eines Strings. Die *unique characters*, die alle nur einmal vorkommen, werden durchgängig mit dem Symbol u bezeichnet. Ihre Anzahl muss genau wie die der *vertex characters* nicht beschränkt werden, da $|\Sigma|$ hier kein Parameter ist.

Für die Konstruktion der Strings in $\mathcal{F}$ beschreiben wir nun zunächst zwei Abkürzungen für Substring-Komponenten, aus denen die Strings im nächsten Schritt zusammengesetzt werden sollen:

$$\begin{aligned} &\text{Edge Selectors: } \langle edge(i,j)(p,q)\rangle = Au^{i-1}pu^{j-i-1}qu^{k-j}B \\ &\text{Separators: } \quad \langle separator\rangle \quad = u^{k+2} \end{aligned}$$

Die Edge Selectors stellen eine mögliche Cliquenkante dar und werden durch die *alignment characters* begrenzt. Dabei beschreibt das Indexpaar (i, j) die Kante zwischen dem i-ten und j-ten Knoten einer potenziellen Clique ($1 \leq i, j \leq k$) und (p, q) die Kante zwischen den Knoten p und q im Graphen ($1 \leq p, q \leq |V|$). Ist der Edge Selector ein Vorkommen des Centers, so bilden p und q den i-ten und j-ten Cliquenknoten. Im String sind die Zeichen p und q *vertex characters* aus Σ_1. Die Separators werden eingesetzt, um die Edge Selectors in einem String voneinander zu trennen: Da sie aus $k + 2$ *unique characters* bestehen, können sie nicht als Vorkommen für den Center gewählt werden, wie wir später sehen werden.

In der Konstruktion der Menge $\mathcal{F}$ enthält diese $\binom{k}{2}$ Strings $S_{i,j}$, die mit den $\binom{k}{2}$ Kanten (i, j) korrespondieren, die in einer k-Clique existieren. Dadurch ergibt sich die Menge $\mathcal{F}$ wie folgt:

$$\mathcal{F} = \{S_{i,j} : 1 \leq i < j \leq k\}.$$

Die einzelnen Strings $S_{i,j}$ werden nun wie folgt aus den Substring-Komponenten, also Edge Selectors und Separators, zusammengesetzt, wobei die iterierte Konkatenation mit Π abgekürzt wird:

$$S_{i,j} = \prod_{\substack{(p,q)\in E \\ p<q, i\leq p \\ j-i\leq q-p \\ q\leq |V|-k+j}} \langle edge(i,j)(p,q)\rangle\langle separator\rangle.$$

Die Positionen i und j im String $S_{i,j}$ sind also stets die Positionen, die von *vertex characters* belegt sind. In der Menge der $S_{i,j}$ kommen alle Kombinationen mit k Knoten vor, die eine Clique bilden können. Ein String $S_{i,j}$ ist also die Konkatenation aller Möglichkeiten, eine Kante im Graphen zu wählen, die den i-ten und j-ten Cliquenknoten verbindet.

Betrachten wir nun ein Beispiel. Gegeben sei ein Graph mit einer Clique der Größe vier mit den Knoten 1, 2, 4 und 5. Dann enthält die Clique sechs Kanten, es gibt also auch sechs Strings, die nach obiger Formel konstruiert werden und die Menge $\mathcal{F}$ bilden:

$\mathcal{F} = \{S_{12}, S_{13}, S_{14}, S_{23}, S_{24}, S_{34}\}$. Die einzelnen Strings werden nun nach obiger Formel konstruiert; beispielsweise werden für den String S_{12} für die jeweiligen Kanten zwischen p und q die Edge Selectors und Separators gebildet und konkateniert, was alle Möglichkeiten beschreibt, einen Kante im Graphen zu wählen, die den ersten und zweiten Cliquenknoten verbindet. Ein Center für $\mathcal{F}$ ist $A1245B$, der die *vertex characters* der Knoten der Clique enthält und in allen Strings aus $\mathcal{F}$ mit der Distanz $d = k - 2 = 2$ vorkommt, wobei die Fehler durch die *unique characters* entstehen und die Vorkommen in $\mathcal{F}$ eine Kante der Clique symbolisieren.

Nun zeigen wir, dass ein Center für $\mathcal{F}$ die Eigenschaften hat, dass er mit dem *alignment character* A beginnt und mit B endet und alle anderen Positionen mit *vertex characters* belegt werden, also keine Zeichen aus Σ_2 vorkommen.

Dazu sind zuerst einige Definitionen notwendig. Ein Vorkommen, das mit den *alignment characters* beginnt und endet, heißt *in-phase*, ansonsten *out-of-phase*. *Vertex positions* sind die Positionen, die mit *vertex characters* belegt sind. Für einen String S_{ij} sind diese also die Positionen i und j rechts von A. Für eine *vertex position* i ist die *vertex group* von i definiert als $V_i = \{S_{ix} : i < x \leq k\} \cup \{S_{xi} : 1 \leq x < i\}$, also die Menge aller Strings aus $\mathcal{F}$, die ebenfalls einen *vertex character* an Position i haben.

Wir zeigen zuerst, dass C mit A beginnt und mit B endet.

Nehmen wir an, C beginnt nicht mit A oder endet nicht mit B. Dann sorgt der Abstand von A und B in den Strings in $\mathcal{F}$ dafür, dass ein Vorkommen nicht mit A und B übereinstimmen kann. Alle Vorkommen müssen an vier Positionen mit C übereinstimmen, da der Center die Länge $k + 2$ hat und der maximale Fehler $k - 2$ ist. Deshalb muss also jedes Vorkommen in mindestens einer Position mit C übereinstimmen, die mit einem Zeichen aus Σ_2 belegt ist, da nur zwei Positionen Zeichen aus Σ_1 enthalten und eine A oder B. Da es in C nur $k + 2$ mögliche Positionen gibt, aber $\binom{k}{2}$ Vorkommen, führt dies ab $k > 4$ nach dem Schubfachprinzip zu einem Widerspruch dazu, dass die Zeichen in Σ_2 alle verschieden sind, da $\binom{k}{2} > k + 2$. Für $k \leq 4$ ist Clique in Polynomialzeit lösbar und wir wenden eine triviale Reduktion an.

Nun zeigen wir, dass keine Position in C mit einem Zeichen aus Σ_2 belegt ist.

Nehmen wir an, C enthält ein Zeichen aus Σ_2 an Position z. Dann stimmt höchstens ein Vorkommen mit C an dieser Position überein, da alle *unique characters* verschieden sind. Betrachten wir nun die *vertex group* V_z, also die Strings, die an der Stelle z einen *vertex character* haben. Wir unterscheiden nun zwei Fälle. Ein *Out-of-phase*-Vorkommen in einem String aus V_z muss einen *unique character* in C determinieren, da es aufgrund des Abstands von A und B nicht sowohl bezüglich des As als auch des Bs mit C übereinstimmen kann, aber höchstens zwei *vertex characters* enthält und vier Übereinstimmungen notwendig sind. Ein *In-phase*-Vorkommen in einem String aus V_z hat an der Position z einen *vertex character*, stimmt also nicht mit C an dieser Position überein, da C dort einen *unique character* enthält. Also muss das Vorkommen auch noch mindestens einen *unique character* determinieren, um eventuell zusammen mit A, B und dem zweiten *vertex character* an vier Positionen mit C übereinzustimmen. Alle

$k-1$ Vorkommen in Strings aus V_z determinieren also jeweils mindestens einen *unique character*. Da diese alle verschieden sind, passiert dies an $k-1$ verschiedenen Positionen in C. Also werden mindestens $k-1$ der k Positionen zwischen A und B im Center mit *unique characters* determiniert.

Alle *Out-of-phase*-Vorkommen in Strings aus V_z stimmen weder am Anfang noch am Ende mit C überein. Gibt es ein solches, determiniert es also mindestens drei *unique characters* statt nur einem in C, also werden mehr als k *unique characters* in C determiniert, was nicht möglich ist. Also sind alle Vorkommen *in-phase*.

Wir wissen nun: Kein Vorkommen stimmt an Stelle z mit C überein, denn C hat dort ein u, die Vorkommen aus V_z nicht. Die eventuell nicht determinierte Stelle y kann in C ein u enthalten. Dann hat C die Form Au^kB, sodass jedes Vorkommen mit zwei *unique characters* übereinstimmen muss, was aber wegen $2(k-1) > k$ ab $k > 2$ wieder nicht möglich ist. Enthält C an Stelle y einen *vertex character* (bzw. ein A oder B), so stimmt an dieser Stelle höchstens eines (bzw. keines) der Vorkommen überein, die anderen benötigen dafür noch zwei *unique characters*, was wiederum zu mindestens $2(k-2) > k$ determinierten *unique characters* und damit ab $k > 4$ zu einem Widerspruch führt.

Insgesamt gilt für alle Vorkommen, dass sie mit A beginnen und mit B enden, also *in-phase* sind. Nehmen wir an, ein Vorkommen ist *out-of-phase*. Dann kann dieses Vorkommen nicht mit A und B übereinstimmen, muss also neben zwei *vertex characters* noch mit einer Position übereinstimmen, die ein Zeichen aus Σ_2 enthält, was ein Widerspruch zur eben gezeigten Aussage ist.

Die Konstruktion läuft in FPT-Zeit bezüglich m, l und d und damit auch k und die Laufzeit bezüglich der Eingabe G ist polynomiell. Es bleibt also noch die Korrektheit zu zeigen.

Wenn es eine eine k-Clique in G gibt, wird ein Center gebildet, indem die Zeichen aus Σ_1, die mit den Knoten der Clique korrespondierenden, zwischen den *alignment characters* A und B platziert werden. Dieser String ist ein Center für $\mathcal{F}$, da es in jedem String in $\mathcal{F}$ ein Vorkommen in einem Edge Selector gibt, der nach Konstruktion mit einer Kante der k-Clique korrespondiert. Also stimmen die jeweiligen *vertex characters* im Edge Selector sowie die begrenzenden Zeichen A und B mit dem Center überein.

Wenn es einen Center C für eine Menge $\mathcal{F}$ gibt, so stimmen alle Vorkommen an zwei Positionen, die mit Zeichen aus Σ_1 belegt sind, mit C überein, da keine *unique characters* im Center vorkommen. Also korrespondieren die $\binom{k}{2}$ Strings in $\mathcal{F}$ nach der Konstruktion mit $\binom{k}{2}$ Kanten in G, die k Knoten untereinander verbinden, und damit gibt es eine k-Clique in G. □

Durch Weglassen von m als Parameter, d. h. es wird ein Center vieler Strings gesucht, wird das Problem potenziell schwerer. Im Folgenden zeigen wir sogar eine W[2]-Schwere.

Satz 5.4. CAS(l, d) ist W[2]-schwer.

Beweis. Wir ändern den Beweis aus Theorem 5.3 dahingehend ab, dass wir nun vom W[2]-vollständigen Problem Dominating Clique aus reduzieren.

Dazu konstruieren wir wieder eine Menge $\mathcal{F}$ mit m Strings über Σ, die aber nun genau dann einen Center der Länge l mit Distanz d besitzt, wenn der gegebene Graph $G = (V, E)$ eine *Dominating Clique* der Größe k enthält.

Das Alphabet der Strings und die Substring-Komponenten bleiben wie in obiger Reduktion definiert.

Die Parameter von CAS, l und d, müssen wieder durch Funktionen von k beschränkt werden können. Diese stimmen mit den Funktionen f_2 bis f_4 in obiger Reduktion überein. Die Anzahl der Strings in $\mathcal{F}$ ist $m = f_1(k, G) = \binom{k}{2} + |V|$, also hier nicht mehr unabhängig von $|G|$.

Die Strings in $\mathcal{F}$ werden in zwei Gruppen $\mathcal{F} = F_{Ge} \cup F_{Gv}$ aufgeteilt, die verschiedene Funktionen in der Konstruktion haben und die Bedingungen der Clique und des *Dominating Set* sicherstellen. Dabei besteht F_{Ge} aus $\binom{k}{2}$ Strings, die mit denen aus obiger Reduktion übereinstimmen und einen Center erzwingen sollen, der mit einer k-Clique in G korrespondiert. Die zweite Gruppe F_{Gv} ist dafür verantwortlich, zu prüfen, dass ein Center, der von F_{Ge} bestimmt wird, neben einer k-Clique auch mit einem *Dominating Set* korrespondiert:

$$F_{Gv} = \{S_{Vp} : 1 \leq p \leq |V|\}.$$

Die Menge F_{Gv} enthält also für jeden Knoten p des Graphen einen String. Dabei wird der String S_{Vp} aus allen Substring-Komponenten zusammengesetzt, die das Zeichen $q \in \Sigma_1$ enthalten, sodass q ein Nachbar von p ist. Die Komponenten werden dafür wie folgt verbunden, wobei $N[x]$ die Menge der Nachbarn von x ist:

$$S_{Vp} = \prod_{\substack{q \in N[p] \\ q' \in N[q] \\ 1 \leq i < j \leq k}} \begin{cases} \langle edge(i,j)(q',q) \rangle \langle separator \rangle & \text{falls } q' < q, \\ \langle edge(i,j)(q,q') \rangle \langle separator \rangle & \text{falls } q < q'. \end{cases}$$

Der String S_{Vp} gibt also für jeden Knoten p die Kanten an, über die p dominiert werden kann. Da solch ein String für jeden Knoten benötigt wird, kann die Anzahl m der Strings kein Parameter mehr sein.

Betrachten wir nun ein Beispiel. Gegeben sei ein Graph mit fünf Knoten und einer *Dominating Clique* der Größe drei mit den Knoten 2, 3 und 4. Für die Bedingung der Clique werden drei Strings S_{12}, S_{13}, S_{23}, die mit den Kanten der Clique korrespondieren, wie vorher konstruiert. Für die Bedingung des *Dominating Set* werden für alle Knoten p

die jeweiligen Strings $S_{V_1}, S_{V_2}, S_{V_3}, S_{V_4}, S_{V_5}$ gebildet, die mit den Kanten korrespondieren, über die der jeweilige Knoten dominiert wird. Die Menge $\mathcal{F}$ ergibt sich aus $\mathcal{F} = F_{Ge} \cup F_{Gv} = \{S_{12}, S_{13}, S_{23}\} \cup \{S_{V_1}, S_{V_2}, S_{V_3}, S_{V_4}, S_{V_5}\}$. Ein Center ist dann $A234B$, der genau die *vertex characters* der *Dominating Clique* enthält und in jedem String mit der Distanz $d = k - 2 = 1$ vorkommt, da jeder einen Bestandteil hat, der eine Cliquenkante oder eine dominierende Kante darstellt.

Die Konstruktion läuft in FPT-Zeit bezüglich l und d und damit auch k und die Laufzeit bezüglich der Eingabe G ist polynomiell. Es bleibt also noch die Korrektheit zu zeigen.

Wir wissen bereits, dass ein Center für F_{Ge} aus einer k-Clique in G konstruiert werden kann. Nehmen wir nun an, dass es eine Menge $V' \subset V$ gibt, die eine k-Clique und ein *Dominating Set* für G ist. Demnach existiert für alle Knoten $p \in V$ ein Knoten $q \in V'$, sodass q ein Nachbar von p ist. Deshalb fungiert für jeden Knoten p im Graphen der Substring von S_{Vp}, der eine mit p verbundene Kante der Clique darstellt, als ein Vorkommen für den Center. Wenn es also eine *Dominating k-Clique* in G gibt, existiert auch ein Center für $\mathcal{F}$.

Nehmen wir an, es gibt einen Center C für $\mathcal{F}$. Durch die obige Reduktion wissen wir, dass aus C eine k-Clique in G abgeleitet werden kann. Da C ein Center ist, kommt dieser in allen Strings aus F_{Gv} vor, wobei diese Vorkommen mit C an zwei Positionen übereinstimmen, die mit *vertex characters* belegt sind, also eine dominierende Kante darstellen. Deshalb sind die korrespondierenden Knoten in V mit einem Knoten in der Clique verbunden, die durch C spezifiziert wird. Also ist auch die Bedingung des *Dominating Set* erfüllt und damit existiert eine *Dominating Clique* der Größe k in G. □

Im Folgenden betrachten wir auch die Alphabetgröße $|\Sigma|$ als Parameter. Dies verbietet den Ansatz der vorherigen Reduktionen, mithilfe vieler *unique characters* eine bestimmte Form des Centers zu erzwingen. Stattdessen benötigen wir einen großen Center, der auch eine große Distanz zu den Strings in $\mathcal{F}$ haben kann.

Satz 5.5. CAS$(m, |\Sigma|)$ ist W[2]-schwer.

Beweis. Statt von einem Problem aus der Graphentheorie reduzieren wir nun vom kombinatorischen Problem Set Cover aus, das ebenfalls W[2]-vollständig ist.

Wir geben nun eine Menge $\mathcal{F}$ von m Strings über Σ an, die genau dann einen Center besitzt, wenn die Instanz $I = (\mathcal{B}, \mathcal{L})$ eine Überdeckung der Größe k hat. Wir können annehmen, dass $\mathcal{B} = \{1, \ldots, |\mathcal{B}|\}$ gilt.

Für die Parameter m und $|\Sigma|$ gilt $m = f_1(k) = 2k$ und $|\Sigma| = f_2(k) = 3k + 1$. Dagegen ist die maximale Distanz $d = f_3(\mathcal{L}, k) = (k-1)|\mathcal{B}|$ und damit nicht in k beschränkt, genauso die maximale Länge der Strings in $\mathcal{F}$, $n = f_4(\mathcal{L}, k) = 2(k|\mathcal{B}| + 2) \cdot (|\mathcal{L}| - 1)$, und die Länge des Centers $l = f_5(\mathcal{L}, k) = k|\mathcal{B}| + 2$.

Das Alphabet der Strings wird in drei Gruppen mit unterschiedlichen Funktionen für die

Konstruktion aufgeteilt: $\Sigma = \Sigma_1 \cup \Sigma_2 \cup \{A\}$ mit

$$\begin{aligned}
\Sigma_1 &= \{s_1, \ldots, s_k\} && (\textit{solution characters}),\\
\Sigma_2 &= \{u_{11}, u_{12}, u_{21}, u_{22}, \ldots, u_{k1}, u_{k2}\} && (\textit{unique characters}),\\
&\{A\} && (\textit{alignment character}).
\end{aligned}$$

Dabei wird angenommen, dass das Zeichen s_i aus Σ_1 für $1 \leq i \leq k$ die Zahl i ist. Die *solution characters* werden in der Konstruktion ein $L_i \in \mathcal{L}$ darstellen. Die Zeichen der Form $u_{ij} \in \Sigma_2$ sind identisch innerhalb eines String S_{ij}, aber es ist stets $u_{ij} \neq u_{i'j'}$, falls $i \neq i'$ oder $j \neq j'$. Da $j \in \{1, 2\}$, ist $|\Sigma_2|$ und damit $|\Sigma|$ durch k beschränkt. Der *alignment character* ist dazu da, die Komponenten eines Strings voneinander abzugrenzen, und bildet den Anfang und das Ende des Centers sowie aller Vorkommen, wie wir später sehen werden.

Für die Konstruktion der Strings in $\mathcal{F}$ beschreiben wir nun drei Abkürzungen für Substring-Komponenten, aus denen die Strings zusammengesetzt werden sollen.

$$\begin{aligned}
&\text{Fillers :} && \langle Filler(i)\rangle && = s_i^{(k-1)|\mathcal{B}|}\\
&\text{Separators :} && \langle Separator(i,p)\rangle && = u_{ip}^{(k|\mathcal{B}|+2)}\\
&\text{Subset Indicators :} && \langle Subset(i,j,p)\rangle && = \prod_{b\in\mathcal{B}} g(i,j,p,b)
\end{aligned}$$

Die Fillers sind Strings der Länge $(k-1)|\mathcal{B}|$, die nur aus einem wiederholten Vorkommen der Zahl s_i bestehen, wobei $\langle Filler(i)\rangle$ mit der Teilmenge $L_i \in \mathcal{L}$ korrespondiert. Die Separators sind Strings der Länge $k|\mathcal{B}| + 2$, wobei jeder aus einem Zeichen aus Σ_2 aufgebaut wird und die Variable p den Wert 1 oder 2 annimmt; diese kommen nicht im Center vor, wie wir später sehen werden. Die Subset Indicators werden genutzt, um die Teilmengen L_i anzugeben, die eine Überdeckung von $\mathcal{B}$ bilden. Es ist dabei g abhängig von $\mathcal{B}$ und $\mathcal{L}$ als eine Abkürzung für folgendes definiert:

$$g(i,j,p,b) = \begin{cases} s_i \text{ falls } b \in L_j, \\ u_{ip} \text{ sonst.} \end{cases}$$

Intuitiv sagt dies aus, dass L_j als i-te von k überdeckenden Teilmengen ausgewählt wird. Falls sie das Argument b enthält, so führt diese Auswahl zu einem Überdecken von $b \in \mathcal{B}$. In dem Fall ist s_i im String enthalten, ansonsten ein *unique character*.

Für die Konstruktion der Menge $\mathcal{F}$ wird jede der k Mengen einer Lösung R durch ein Paar von Strings in $\mathcal{F}$ repräsentiert. Dabei korrespondieren die Vorkommen in den Strings S_{i1} und S_{i2} aus $\mathcal{F}$ mit der i-ten Menge in R. Dafür werden S_{i1} und S_{i2} wie folgt definiert:

$$\begin{aligned}
S_{i1} &= \prod_{1\leq j\leq|\mathcal{L}|} A\langle Subset(i,j,1)\rangle\langle Filler(i)\rangle A\langle Separator(i,1)\rangle,\\
S_{i2} &= \prod_{1\leq j\leq|\mathcal{L}|} A\langle Subset(i,j,2)\rangle\langle Filler(i)\rangle A\langle Separator(i,2)\rangle.
\end{aligned}$$

Die Menge der Strings $\mathcal{F}$, die aus den Strings S_{i1} und S_{i2} gebildet wird, ist dann $\mathcal{F} = \{S_{11}, S_{12}, S_{21}, S_{22}, \ldots, S_{k1}, S_{k2}\}$.

Betrachten wir ein Beispiel. Sei $\mathcal{B} = \{1, 2, 3, 4, 5, 6, 7\}$, $L_1 = \{1, 4, 6\}$, $L_2 = \{1, 2, 4\}$, $L_3 = \{3, 5\}$, $L_4 = \{1, 2, 3, 7\}$ und $k = 3$. Eine Überdeckung ist durch die drei Teilmengen L_1, L_3 und L_4 gegeben. Dann ist $\mathcal{F} = \{S_{11}, S_{22}, S_{21}, S_{22}, S_{31}, S_{32}\}$, wobei die Instanzen nach obiger Formel konstruiert werden. Ein Center ist dann $A3331213111112222222333A$ mit $d = (k-1)|\mathcal{B}| = 14$, der durch die *alignment characters* begrenzt wird. Die ersten $|\mathcal{B}|$ Zeichen stehen dabei für die Indizes der $k = 3$ Teilmengen L_i aus R, die das entsprechende Zeichen aus $\mathcal{B}$ überdecken. Die restlichen Zeichen korrespondieren mit den Fillers, die die Indizes dieser Teilmengen aneinanderreihen.

Wir zeigen nun, dass ein Center für $\mathcal{F}$ sowie alle Vorkommen mit dem *alignment character* A beginnen und enden müssen. Dafür bemerken wir als erstes, dass jedes Zeichen aus $\Sigma \setminus \{A\}$ in höchstens zwei Strings vorkommt. Die ursprüngliche Rechnung aus [ESW03] nimmt inkorrekterweise an, dass es eine Spalte mit ausschließlich A und $l-1$ Spalten ohne A gibt; dies verallgemeinern wir dahingehend, dass die As an beliebigen Positionen stehen können.

Wir zeigen zunächst, dass C mit dem Zeichen A beginnt.

Sei $C \in \Sigma^l$ ein Center, bei dem kein A am Anfang steht. Sei X die Menge der entsprechenden Vorkommen S' bezüglich der Strings aus $\mathcal{F}$ mit Distanz jeweils höchstens d. Sei j eine Position, $1 \leq j \leq l$. Ist $C[j]$ nicht A, so stimmen höchstens zwei Zeilen an Stelle j mit C überein, weil alle Zeichen außer A höchstens zweimal pro Spalte vorkommen. In der Spalte j sind also $m-2$ Fehler. Sei $\#A$ die Anzahl der Spalten j mit $C[j] = A$. Dann hat X in den anderen $l - \#A$ Spalten also insgesamt bereits $(l - \#A)(m-2)$ Fehler, also hat nach dem Schubfachprinzip ein S' mindestens den Fehler $(l - \#A)(m-2)/m$. Für die $\#A$ Spalten j mit $C[j] = A$ gilt hingegen: Alle S' haben in diesen Spalten höchstens eine Übereinstimmung und mindestens $\#A - 1$ Fehler, da S', wenn A nicht am Anfang steht, höchstens mit einem A aus C übereinstimmen kann. Also gibt es ein S', das insgesamt mindestens den Fehler $d' = (\#A - 1) + (l - \#A)(m-2)/m$ hat.

Es ist $m = 2k$, $l = k|\mathcal{B}| + 2$ und $d = |\mathcal{B}|(k-1)$, wie oben definiert wurde, was wir nun in die Gleichung einsetzen können.

Demnach gibt es ein S' mit mindestens Fehler

$$\begin{aligned} d' &= \#A - 1 + (k|\mathcal{B}| + 2 - \#A)(2k-2)/2k \\ &= \#A - 1 + (k|\mathcal{B}| + 2 - \#A)(k-1)/k \\ &= \#A - 1 + k|\mathcal{B}|(k-1)/k + (2 - \#A)(k-1)/k \\ &= \#A - 1 + |\mathcal{B}|(k-1) + (2 - \#A)(k-1)/k. \end{aligned}$$

Aber es gilt

$$\begin{aligned} & \#A - 1 > (\#A - 2)(k-1)/k \\ \Longrightarrow \quad & \#A - 1 + (2 - \#A)(k-1)/k > 0 \\ \Longrightarrow \quad & \#A - 1 + |\mathcal{B}|(k-1) + (2 - \#A)(k-1)/k > |\mathcal{B}|(k-1) \\ \Longrightarrow \quad & d' > d \\ \Longrightarrow \quad & C \text{ ist kein Center, Widerspruch.} \end{aligned}$$

Analog lässt sich zeigen, dass ein Center mit dem Zeichen A enden muss.

Nun zeigen wir, dass auch jedes Vorkommen S' in X für einen Center C mit A anfängt und endet.

Da jedes S' genau dann mit A anfängt, wenn es mit A endet, unterscheiden wir wieder zwei disjunkte Mengen von Vorkommen: *in-phase*, $\mathcal{S}'_i$, und *out-of-phase*, $\mathcal{S}'_o$. Schreibe kurz $s = |\mathcal{S}'_o|$, dann ist unser Ziel, $s = 0$ zu zeigen.

Für die äußeren beiden Spalten gilt: Alle S' aus $\mathcal{S}'_i$ stimmen mit C überein, alle S' aus $\mathcal{S}'_o$ verursachen hier stets zwei Fehler, denn wir wissen bereits, dass ein Center mit A beginnen und enden muss. Also ergeben sich $2s$ Fehler für die äußeren Spalten. Für die inneren $l-2$ Spalten gilt: Alle Symbole kommen höchstens zweimal pro Spalte vor, oder sind A. Aber pro Zeile aus $\mathcal{S}'_o$ gibt es nur genau ein A, also enthalten diese $l-2$ inneren Spalten zusammen genau s As. Aus der Tatsache, dass kein Symbol außer A mehr als zweimal auftreten kann, bekommen wir unter der Annahme, dass A keine Übereinstimmung ist, wieder $2k-2$ Fehler pro Spalte, wobei wir wegen der potenziellen Übereinstimmungen durch A wieder bis zu $\#A = |\mathcal{S}'_o| = s$ Fehler abziehen müssen, denn A kann durchaus öfter als zweimal pro Spalte vorkommen, aber nur s-mal in den mittleren $l-2$ Spalten.

Die Gesamtzahl der Fehler ist also $2s + (l-2)(2k-2) - s = k|\mathcal{B}|(2k-2) + s$. Damit muss eine der $2k$ Zeilen mindestens den Fehler $(k|\mathcal{B}|(2k-2)+s)/2k = k|\mathcal{B}|(1-2/2k) + s/2k = |\mathcal{B}|(k-1) + s/2k$ haben, was für $s > 0$ ein Fehler $> d$ ist. Aber dann wäre C kein Center, also ist $s = 0$ und es gibt keine Instanzen, die *out-of-phase* sind.

Die Konstruktion läuft in FPT-Zeit bezüglich m und $|\Sigma|$ und damit auch k und die Laufzeit bezüglich der Eingabe G ist polynomiell. Es bleibt also die Korrektheit zu zeigen.

Nehmen wir an, es gibt eine Überdeckung R für $\mathcal{B}$ der Größe k. Daraus konstruieren wir wie folgt einen Center C für $\mathcal{F}$:

1. Die erste und letzte Position von C werden mit dem *alignment character* A belegt.

2. Die nächsten $|\mathcal{B}|$ Positionen repräsentieren die Elemente der Grundmenge und bekommen ein Zeichen zugewiesen, das eine der Teilengen in R angibt, die das korrespondierende Element überdeckt. Dazu wird für jedes $b \in \mathcal{B}$ ein $L_i \in R$ gewählt, das b enthält. Da R eine Überdeckung für $\mathcal{B}$ ist, kann also für jedes $b \in \mathcal{B}$ ein L_i gewählt werden. Wenn L_i gewählt wird, um b zu überdecken, bekommt die Position $b+1$ im Center das Zeichen s_i zugewiesen.

3. Die verbleibenden $(k-1)|\mathcal{B}|$ Positionen von C korrespondieren mit den Fillers. Wenn x_i Positionen ($0 \leq x_i \leq |\mathcal{B}|$) in C Zeichen zugewiesen wurden, die mit Elementen in L_i

korrespondieren, dann bekommen $|\mathcal{B}| - x_i$ Positionen im Filler-Teil von C nacheinander ebenfalls das Zeichen s_i zugewiesen.

Wenn $L_j \in \mathcal{L}$ die i-te Menge in R ist, stimmt der Substring von S_{i1} (und von S_{i2}), der mit dem *alignment character* beginnt und endet und den j-ten Subset Indicator enthält, also die Teilmenge L_j der Überdeckung repräsentiert, in genau $2 + |\mathcal{B}|$ Positionen mit C überein: Mit den As am Anfang und Ende und mit den $|\mathcal{B}|$ gleichen Zeichen, die den Index der Teilmenge darstellen. Zieht man von der Gesamtlänge $l = k|\mathcal{B}| + 2$ die Anzahl der Übereinstimmungen, $2 + |\mathcal{B}|$, ab, ergibt sich eine Distanz von $(k-1)|\mathcal{B}|$, also ist C ein Center für $\mathcal{F}$ mit einer Distanz von genau $d = (k-1)|\mathcal{B}|$ zu jedem Vorkommen.

Nehmen wir an, es gibt einen Center C für $\mathcal{F}$. Für jedes verschiedene i und i' stimmen die Vorkommen in S_{i1} und $S_{i'1}$ mit C an disjunkten inneren $(1 < j < l)$ Positionen überein, d. h. jede innere Position in C stimmt mit höchstens einem Vorkommen in den Strings $S_{11}, \ldots, S_{k1}$ überein. Dasselbe gilt für die Vorkommen in den Strings $S_{12}, \ldots, S_{k2}$. Jede innere Position j in C passt also zu genau einem S_{i1} und genau einem S_{i2}.

Wir zeigen nun, dass alle außer der ersten und letzten Position von C, da diese A enthalten, durch *solution characters* belegt sind, also nicht durch *unique characters* oder durch As. Es muss gelten, dass eine innere Position j in C genau dann zu S_{i1} passt, wenn sie zu S_{i2} passt. Andernfalls würde C an Stelle j mit einem S_{i1} und einem $S_{i'2}$ mit $i \neq i'$ übereinstimmen, aber diese haben an Stelle j stets verschiedene Zeichen. Die einzigen Zeichen, in denen S_{i1} und S_{i2} selbst übereinstimmen, sind aber *solution characters* und A, aber A kann ausgeschlossen werden, da alle Vorkommen *in-phase* sind.

Da jedes Vorkommen zwischen den As noch $(k|\mathcal{B}|+2)-((k-1)|\mathcal{B}|)-2 = |\mathcal{B}|$ Übereinstimmungen haben muss, müssen andersherum auch alle $k|\mathcal{B}|$ Symbole an den je $|\mathcal{B}|$ inneren Positionen der k Vorkommen in den S_{i1} mindestens einmal mit C übereinstimmen. Damit sind alle $k|\mathcal{B}|$ inneren Zeichen von C durch je genau ein Vorkommen determiniert. Also müssen die ersten $|\mathcal{B}|$ inneren Zeichen in C, die mit den Elementen aus $\mathcal{B}$ korrespondieren, ebenfalls aus einem Vorkommen stammen und damit aus einer der k ausgewählten Teilmengen L_i, deren überdeckte Elemente von den *solution characters* in den ersten inneren $|\mathcal{B}|$ Zeichen in S_{i1} angeben werden. □

Ein weiteres Schwere-Resultat für CAS($|\Sigma|$) lautet wie folgt:

Satz 5.6. CAS($|\Sigma|$) ist para-NP-vollständig.

Beweis. Folgt aus der NP-Schwere von CAS mit $|\Sigma| = 2$ [FL97], Theorem 2.19 und der Mitgliedschaft von CAS in NP [Smi04]. □

Wir sehen also, dass die Schwierigkeit des Problems nicht abnimmt, wenn wir uns auf binäre Strings beschränken.

Wir können nun die weiteren Ergebnisse für die Schwere in die Tabelle eintragen:

Tabelle 5.2: FPT- und Schwere-Resultate von CAS

Parameter	-	$\|\Sigma\|$	m	m, $\|\Sigma\|$
-	NP-vollständig	para-NP-vollständig	W[2]-schwer	W[2]-schwer
d	W[2]-schwer		W[1]-schwer	
l	W[2]-schwer	FPT	W[1]-schwer	FPT
n	FPT	FPT	FPT	FPT

Dabei folgen die Resultate in der ersten Zeile aus Satz 5.5: Da CAS$(m, |\Sigma|)$ W[2]-schwer ist, so auch CAS(m). Die Resultate in der ersten Spalte folgen aus Satz 5.4: CAS(l, d) ist W[2]-schwer, also auch CAS(d) und CAS(l). Die übrigen Resultate folgen aus Satz 5.3: CAS(m, l, d) ist W[1]-schwer, also auch CAS(m, d) und CAS(m, l).

5.4 Mitgliedschaft in der W-Hierarchie

Um Mitgliedschaften von parametrisierten Varianten von CAS in Klassen der W-Hierarchie zu zeigen, müssen Boolesche Schaltkreise mit begrenzter Weft konstruiert werden.

Die folgenden drei Schaltkreise müssen also genau dann erfüllbar sein, wenn CAS eine positive Lösung besitzt, und nutzen verschiedene Strategien, um Lösungen für Parametrisierungen von CAS zu testen. Sie werden mit *Center testing circuit*, *Instance testing circuit* und *Single instance + modification testing circuit* bezeichnet.

Center testing circuit:

Die Strategie dieses Schaltkreises ist es, von einem Center ausgehend zu prüfen, ob CAS eine positive Lösung besitzt. Jede positiv belegte Eingangsvariable entspricht dabei einem der l Symbole im Center, wobei der Schaltkreis noch prüfen muss, ob der gewählte Center eine ausreichend kleine Distanz besitzt.

Für eine Menge $\mathcal{F} = \{S_1, \ldots, S_m\}$ mit einem Center C bezeichnen wir den j-ten Substring der Länge l in S_i mit $S_i[j]$. Wir konstruieren eine Menge X, die genutzt wird, um alle möglichen Subsequenzen der Länge $(l - d)$ von einem String der Länge l zu indizieren, die also alle möglichen Übereinstimmungen mit dem Center bilden. Jedes X_p ist dabei eine Subsequenz der Länge $(l - d)$ von (1, . . . , l), kurz $X_p \prec (1, \ldots, l)$.

$$X = \left\{ X_p : X_p \prec (1, \ldots, l), |X_p| = l - d, 1 \leq p \leq \binom{l}{d} \right\}.$$

Nun definieren wir zwei Mengen von Booleschen Variablen. Die Menge der Variablen

$$A = \{a[i, j, p, q] : 1 \leq i \leq m, 1 \leq j \leq n - l + 1, 1 \leq p \leq |X|, 1 \leq q \leq l - d\}$$

bezeichnet die Position q in $X_p \cap S_i[j]$. Die Variablen der Menge

$$B = \{b[u, v] : 1 \leq u \leq l, 1 \leq v \leq |\Sigma|\}$$

geben an, ob das Zeichen v die Position u in C belegt. Die Variable $a[i, j, p, q]$ nimmt dabei genau dann den Wert von $b[u, v]$ an, wenn die Position q von X_p u ist und diese Position von dem Zeichen v in $S_i[j]$ belegt wird, sonst wird $a[i, j, p, q]$ auf 0 gesetzt.

Der Boolesche Ausdruck $E = E_1 E_2$ über der Menge der Variablen B wird wie folgt zusammengesetzt, wobei die Summe einer Oder-Verknüpfung und das Produkt einer Und-Verknüpfung entspricht:

$$E_1 = \prod_{u=1}^{l} \prod_{1 \leq v < v' \leq |\Sigma|} (\neg b[u, v] + \neg b[u, v']),$$

$$E_2 = \prod_{i=1}^{m} \sum_{j=1}^{n-l+1} \sum_{p=1}^{|X|} \prod_{q=1}^{l-d} a[i, j, p, q].$$

Der Zweck von E_1 ist, eine Korrespondenz zwischen erfüllenden Belegungen und Strings über Σ^l zu erzwingen. Dabei wird E_1 von einer Belegung mit Gewicht l falsifiziert, wenn $b[u, v]$ und $b[u, v']$ mit wahr belegt werden, also mehrere Zeichen einer Position zugewiesen werden. E_2 sorgt dafür, dass für jedes i, also jeden String aus $\mathcal{F}$, ein Substring $S_i[j]$ an $(l - d)$ Positionen mit dem Center übereinstimmt.

Betrachten wir nun ein Beispiel dafür, wie die Variablen belegt werden. Die Menge der Strings bestehe aus $S_1 = tggtca$, $S_2 = accgac$ und $S_3 = cggtag$ über dem Alphabet $\Sigma = \{a, c, g, t\}$. Wir nehmen die Reihenfolge $a = 1$, $c = 2$, $g = 3$ und $t = 4$ in Σ an. Für $X_p = (1, 2, 3)$ ist $a[1, 1, p, 1] = b[1, 4]$, weil beide mit dem Zeichen t an Position 1 in einem String der Länge l korrespondieren. $a[3, 1, p, 1] = b[1, 2]$ korrespondieren mit dem Zeichen c an Position 1. Für $X_p = (1, 3, 5)$ korrespondieren $a[2, 2, p, 2] = b[4, 3]$ mit dem Zeichen g an Position 4.

Wir zeigen nun, dass der Ausdruck E genau dann eine erfüllende Belegung besitzt, dessen Gewicht l ist, wenn es einen Center C für $\mathcal{F}$ gibt.

Wenn C existiert, ist es einfach zu zeigen, dass eine mit C korrespondierende wahre Belegung E erfüllt: $C[p] = S_i[j][p] = q$ gdw. $a[i, j, p, q] = true$. Die Belegung hat Gewicht l, da $a[i, j, p, q]$ genau l-mal auf wahr gesetzt wird, für jedes der l Zeichen des Centers.

Sei andersherum T eine erfüllende Belegung mit Gewicht l für E. E_1 versichert, dass T einen eindeutigen String $s \in \Sigma^l$ darstellt. E_2 versichert, dass für jedes i ein Substring $S_i[j]$ an $(l - d)$ Positionen mit s übereinstimmt. Dies impliziert, dass es in jedem S_i einen Substring der Länge l gibt, der höchstens die Distanz d von s hat, s ist also ein Center für $\mathcal{F}$.

Satz 5.7. $\mathrm{CAS}(l) \in \mathrm{W}[3]$ und $\mathrm{CAS}(m,l) \in \mathrm{W}[3]$.

Beweis. Der *Center testing circuit* hat Weft 3 und konstante Tiefe h. Aus der obigen Argumentation folgt die parametrisierte Reduktion von $\mathrm{CAS}(m,l)$ auf $\mathrm{WCS}_{3,h}$: Das Gewicht der gesuchten Belegung ist l und damit durch den alten Parameter l bzw. (m,l) beschränkt. Der Schaltkreis hat Größe $O(l|\Sigma|^2 + mnl^{d+1})$ und ist in FPT-Zeit bezüglich l bzw. (m,l) berechenbar. □

Instance testing circuit:

Wir konstruieren nun den *Instance testing circuit*, der eine andere Strategie als der *Center testing circuit* nutzt. Das Ziel ist hier, die Mitgliedschaft für Varianten von CAS zu zeigen, wenn die Länge l kein Parameter ist, weshalb von den Vorkommen in $\mathcal{F}$ ausgehend geprüft wird, ob es einen Center gibt. Die Idee dabei ist, m Vorkommen sowie für jedes Vorkommen d Positionen auszuwählen, die nicht mit dem Center übereinstimmen müssen, was also $(m + md)$ positiv belegten Eingangsvariablen entspricht.

Zunächst definieren wir wieder einige Mengen von Booleschen Variablen. Für die Menge der Variablen

$$B = \{b[i,j] : 1 \leq i \leq m, 1 \leq j \leq n - l + 1\}$$

wird $b[i,j]$ auf wahr gesetzt, wenn $S_i[j]$ ein Vorkommen von C ist.

Die Bedeutung der Menge der Variablen

$$W = \{w[i,r,p] : 1 \leq i \leq m, l \leq r \leq d, 1 \leq p \leq l\}$$

ist, dass ein Vorkommen von C in S_i an Position p nicht mit C übereinstimmen muss. Der Index r wird genutzt, um die Anzahl dieser Ausnahmen auf höchstens d zu begrenzen, da nur d Fehlerpositionen erlaubt sind.

Die Menge der Variablen

$$A = \{a[i,j,p,q] : 1 \leq i \leq m, 1 \leq j \leq n - l + 1, 1 \leq p \leq l, 1 \leq q \leq |\Sigma|\}$$

ist ein Alias für die Variablen aus B. Wie im ersten Schaltkreis nimmt die Variable $a[i,j,p,q]$ den Wert von $b[i,j]$ genau dann an, wenn $S_i[j]$ an Position p mit dem Zeichen q belegt wird. Sonst wird $a[i,j,p,q]$ auf falsch gesetzt.

Sei nun $E = E_1 E_2 E_3$ der Boolesche Ausdruck über der Menge von Variablen $B \cup W$ mit:

$$E_1 = \prod_{i=1}^{m} \prod_{1 \leq j < j' \leq n-l+1} (\neg b[i,j] + \neg b[i,j']),$$

$$E_2 = \prod_{i=1}^{m} \prod_{r=1}^{d} \prod_{1 \leq p < p' \leq l} (\neg w[i,r,p] + \neg w[i,r,p']),$$

$$E_3 = \prod_{p=1}^{l} \prod_{q=1}^{|\Sigma|} \prod_{i=1}^{m} \left(\sum_{r=1}^{d} w[i,r,p] + \sum_{j=1}^{n-l+1} a[i,j,p,q] \right).$$

Die Bedeutung von E_1 ist, dass die Anzahl der Vorkommen des Centers auf m begrenzt wird, indem für jeden S_i nur ein Vorkommen gewählt werden darf. E_2 sorgt dafür, dass es höchstens d Fehlerpositionen für ein Vorkommen geben kann und E_3 bedeutet, dass die Vorkommen überall außer an den erlaubten d Fehlerpositionen mit dem Center übereinstimmen.

Wir zeigen, dass E genau dann eine erfüllende Belegung besitzt, dessen Gewicht $(m+md)$ ist, wenn es einen Center C für $\mathcal{F}$ gibt.

Gibt es einen Center C, so kann eine erfüllende Belegung für E gefunden werden, indem $b[i,j]$ für jedes Vorkommen $S_i[j]$ von C auf wahr gesetzt wird. Ebenso wird $w[i,r,p]$ auf wahr gesetzt, wenn der r-te Fehler in dem Vorkommen in S_i an Position p auftritt. Die Anzahl der auf wahr gesetzten Variablen ist dann $(m+md)$, also m für die Vorkommen und md für die d Fehlerpositionen in jedem Vorkommen.

Sei andersherum T eine erfüllende Belegung für E mit Gewicht $(m+md)$. E_1 versichert, dass T mit höchstens m Vorkommen korrespondiert, eines in jedem S_i. E_2 versichert, dass höchstens d Fehlerpositionen für das Vorkommen in einem S_i ausgewählt werden. E_1 und E_2 sorgen dafür, dass T direkt mit einer Menge von Vorkommen korrespondiert und für jedes Vorkommen mit einer Menge von Positionen, die vom Center abweichen können. Wenn E_3 durch T erfüllt wird, impliziert dies, dass alle Vorkommen in allen Positionen mit Ausnahme der Fehlerpositionen übereinstimmen. Also besitzt $\mathcal{F}$ einen Center.

Satz 5.8. $\mathrm{CAS}(m,d) \in \mathrm{W}[2]$ und $\mathrm{CAS}(m,|\Sigma|,d) \in \mathrm{W}[2]$.

Beweis. Der *Instance testing circuit* hat Weft 2 und konstante Tiefe h. Wir reduzieren von $\mathrm{CAS}(m,|\Sigma|,d)$ auf $\mathrm{WCS}_{2,h}$. Das Gewicht der gesuchten Belegung ist $m+md$ und der Schaltkreis hat Größe $O(mn^2+mdl^2+l|\Sigma|n(d+n))$ und ist in Polynomialzeit berechenbar. □

Single instance + modification testing circuit:

Die Idee für den dritten Schaltkreis besteht darin, dass ein Center gefunden werden kann, indem ein String aus $\mathcal{F}$ isoliert wird (hier S_1) und modifiziert wird, bis ein Center entsteht, indem Substitutionen für Zeichen an bis zu d Positionen in jedem Substring $S_1[j]$ von S_1 angewendet werden. Durch die Auswahl von d Fehlerpositionen und einem Substring ergeben sich $(d+1)$ positiv belegte Eingangsvariablen.

Wir nutzen eine *guess-and-test*-Strategie: Als erstes wird ein Center geraten, indem ein $S_1[j]$ zusammen mit den Positionen und Zeichen ausgewählt wird, an denen der Center von $S_1[j]$ abweicht. Dann wird der Center getestet, indem nach einem Vorkommen in jedem $S_i, 2 \leq i \leq m$, also den verbleibenden Substrings, gesucht wird.

Das Ziel ist, eine erfüllende Belegung mit Gewicht $(d+1)$ zu haben, die die Auswahl von einem j $(1 \leq j \leq n-l+1)$ und d Substitutionen an Positionen von $S_1[j]$ repräsentiert, die $S_1[j]$ in einen Center transformieren.

Um die Eingabe des Schaltkreises zu beschreiben, nutzen wir die folgenden Mengen von Variablen:

$$X_1 = \{x_1[i,j,p,r] : 1 \leq i \leq m, 1 \leq j \leq n-l+1, 1 \leq p \leq l, 1 \leq r \leq |\Sigma|\},$$
$$X_2 = \{x_2[j] : 1 \leq j \leq n-l+1\},$$
$$X_3 = \{x_3[p,r] : 1 \leq p \leq l, 1 \leq r \leq |\Sigma|\}.$$

Der Wert von $x_1[i,j,p,r]$ korrespondiert mit dem Wahrheitswert davon, dass $S_i[j]$ mit dem Zeichen r an Position p belegt wird. Diese Werte sind für jedes Vorkommen fix und nicht Teil einer Belegung. $x_2[j]$ repräsentiert einen Substring von S_1 und X_3 die Substitutionen an Position p mit dem Zeichen r.

Die erfüllende Belegung mit Gewicht $(d+1)$ kommt daher, dass exakt ein Mitglied aus X_2 und d Mitglieder aus X_3 ausgewählt werden.

Wenn ein Center geraten wurde, muss er auf potenzielle Vorkommen in den verbleibenden Strings in $\mathcal{F}$ getestet werden. Anders als beim obigen Schaltkreis ist l kein Parameter, also können wir nicht die gleiche Strategie nutzen, um den Center zu testen, und führen weitere Variablen ein.

Die Menge der Variablen $\{g[p,r] : 1 \leq p \leq l, 1 \leq r \leq |\Sigma|\}$ beschreibt den geratenen Center, wobei

$$g[p,r] = \left(\sum_{j=1}^{n-l+1} (x_2[j] \cdot x_1[1,j,p,r]) \right) \cdot \left(\prod_{\substack{1 \leq r' \leq |\Sigma| \\ r' \neq r}} \neg x_3[p,r'] \right) + x_3[p,r].$$

Die unteren Schichten des Schaltkreises werden durch die Variablen

$$B = \{b[i,j,p] : 2 \leq i \leq m, 1 \leq j \leq n-l+1, 1 \leq p \leq l\}$$

beschrieben, mit der Interpretation, dass $b[i,j,p] = \mathit{true}$ genau dann, wenn $S_i[j]$ mit dem geratenen Center an Position p übereinstimmt oder einer von höchstens d Fehlern ist.

Mitglieder aus B treten in verschiedenen Tiefen auf. Wir stapeln die Variablen von B so, dass $b[i,j,p]$ von Variablen abhängt, die genutzt wurden, um $b[i,j,p-1]$ zu generieren. Der Zweck davon ist, zu vermeiden, die Anzahl der Fehler in einem einzelnen Level zählen zu müssen. Dies würde eine exponentielle Anzahl von Gattern verursachen. Die Strategie, die wir nutzen, ist, einen Zähler für die Fehler zu bestimmen, der jedes Mal dekrementiert wird, wenn ein Fehler auftritt. Die Menge der Variablen A stellt den Zähler für jedes $S_i[j]$ dar:

$$A = \{a[i,j,p,q] : 2 \leq i \leq m, 1 \leq j \leq n-l, 0 \leq p \leq l, 1 \leq q \leq d+1\},$$

sodass

$$a[i,j,p,q] = \left(a[i,j,p-1,q] \cdot \sum_{r=1}^{|\Sigma|} (g[p,r] \cdot x_1[i,j,p,r]) \right) + a[i,j,p-1,q+1].$$

Für alle i, j und p wird der Wert von $a[i, j, p, d+1]$ auf *false* gesetzt und für alle i, j und q wird der Wert von $a[i, j, 0, q]$ auf *true* gesetzt.

Wir definieren nun die Variablen aus B:

$$b[i, j, p] = a[i, j, p, 1] + \sum_{r=1}^{|\Sigma|}(g[p, r] \cdot x_1[i, j, p, r]).$$

Der Schaltkreis C wird durch den Ausdruck $E = E_1E_2E_3$ beschrieben mit:

$$E_1 = \prod_{1 \le j < j' \le n-l+1} (\neg x_2[j] + \neg x_2[j']),$$
$$E_2 = \prod_{p=1}^{l} \prod_{1 \le r < r' \le |\Sigma|} (\neg x_3[p, r] + \neg x_3[p, r']),$$
$$E_3 = \sum_{i=2}^{m} \sum_{j=1}^{n-l+1} \sum_{p=1}^{l} b[i, j, p].$$

E_1 beschreibt dabei den gewählten Substring von S_1, E_2 die Substitutionen und E_3 die Übereinstimmungen mit dem geratenen Center.

Der Schaltkreis ist also genau dann erfüllt, wenn ein geratener Center an mindestens $(l - d)$ Positionen in mindestens einem Substring für jedes Mitglied in $\mathcal{F}$ übereinstimmt.

Satz 5.9. CAS$(d) \in$ W[P].

Beweis. Wir reduzieren von CAS(d) auf WCS. Das Gewicht der gesuchten Belegung ist $d + 1$ und der *Instance testing circuit* hat Größe $O(nml(|\Sigma| + d))$. □

5.5 Zusammenfassung

Die Resultate für die Klassen der W-Hierarchie und das aus Satz 5.9 abgeleitete Resultat CAS$(d, |\Sigma|) \in$ W[P] vervollständigen die Tabelle, in der nun alle Resultate zusammengefasst sind:

Tabelle 5.3: Parametrisierte Komplexität von CAS

Parameter	-	$\|\Sigma\|$	m	m, $\|\Sigma\|$
-	NP-vollständig	para-NP-vollständig	W[2]-schwer	W[2]-schwer
d	W[2]-schwer, in W[P]	in W[P]	W[1]-schwer, in W[2]	in W[2]
l	W[2]-schwer, in W[3]	FPT	W[1]-schwer, in W[3]	FPT
n	FPT	FPT	FPT	FPT

Zusammenfassend erlaubt die Analyse der parametrisierten Komplexität, folgende Aussagen über die Bedeutung der Parameter für die Komplexität von CAS zu treffen:

FPT-Algorithmen wurden nur für die Fälle gefunden, in denen $|\Sigma|$ und l beide Parameter sind, für den Parameter $n \geq l$ folgt dies automatisch auch, für den Parameter $d \leq l$ jedoch nicht.

Außerdem gibt es noch einige Fälle, die offen sind oder verbessert werden können und weitere interessante Fragen nach sich ziehen:

Für zwei Varianten, die offenen Probleme CAS$(m, |\Sigma|, d)$ und CAS$(|\Sigma|, d)$, wurden weder Schwere-Resultate noch FPT-Resultate gefunden, weshalb sie mögliche Kandidaten für weitere Forschung sind.

Die Zellen in der Tabelle mit W[t]- bzw. W[P]-Resultaten können noch verbessert werden, indem die Unterschiede zwischen der Schwere für Klassen und Mitgliedschaft in Klassen angenähert werden, indem stärkere untere Schranken oder effizientere Schaltkreise gefunden werden.

Weiterhin wurden identische untere Schranken für d und l als Parameter gezeigt. Es könnte jedoch sein, dass der kleinere Parameter d zu einem schwierigeren Problem führt. Gibt es also einen Unterschied zwischen l und d, d. h., wie verhält sich das Problem, wenn ein langer Center mit wenigen Fehlern gesucht wird, wie zum Beispiel in der Analyse von DNA-Sequenzen?

Für die Fälle, in denen d unbeschränkt ist, also nur m beschränkt ist, wurden keine oberen Grenzen gefunden. Kann es sein, dass hier für die Mitgliedschaft in der W-Hierarchie Beweise mit Schaltkreisen nicht möglich sind? Können vielleicht gewisse Probleme nicht durch Schaltkreise ausgedrückt werden?

6 Weitere verwandte Probleme

6.1 CASEQ

Passende Parametrisierungen von Problemen können auch genutzt werden, um Resultate für verwandte Probleme abzuleiten und Faktoren zu beleuchten, die die Komplexität dieser Probleme beeinflussen.

Beispielsweise kann eine Erweiterung von CAS, das NP-vollständige Problem CASEQ (Common Approximate Subsequence), betrachtet werden, in dem Subsequenzen anstatt Substrings gesucht werden. Die NP-Schwere ergibt sich automatisch über CAS und die NP-Mitgliedschaft kann leicht über einen Algorithmus gezeigt werden, der die Subsequenzen und den Center rät.

Aus den Resultaten der Analyse der parametrisierten Komplexität von CAS können Implikationen für CASEQ abgeleitet werden.

COMMON APPROXIMATE SUBSEQUENCE (CASEQ)

Instanz: Eine Menge $\mathcal{F} = \{S_1, \ldots, S_m\}$ von Strings über einem Alphabet Σ, $l \in \mathbb{N}$, $d \in \mathbb{N}$, $n \in \mathbb{N}$, $g \in \mathbb{N} \cup \{\infty\}$, sodass $|S_i| \leq n$, $1 \leq i \leq m$, $1 \leq l \leq n$, $0 \leq d \leq l$.

Parameter: $|\Sigma|$, m, n, l und d.

Frage: Gibt es einen String $C \in \Sigma^l$, sodass es für alle $S \in \mathcal{F}$ eine Subsequenz $s \in \Sigma^l$ mit $d_H(s, C) \leq d$ und Symbolabstand g gibt?

Wir erinnern uns, dass die Variante CASEQ ($d = 0$, $g \geq 0$) das Problem Longest Common Subsequence (LCS) ist. Für LCS wurden ebenfalls Resultate der parametrisierten Komplexität gefunden.

Die Schwere-Resultate von CAS können für die korrespondierenden parametrisierten Versionen auf das Problem CASEQ übertragen werden. Darüber hinaus sind manche der Varianten schwer für höheren Stufen der W-Hierarchie, was sich aus bekannten Resultaten für LCS ableiten lässt, zum Beispiel ist CASEQ $(m, |\Sigma|)$ W[t]-schwer für $t \geq 1$.

Ob auch die FPT-Resultate von CAS oder LCS auf CASEQ übertragbar sind, ist ein offenes Problem.

CASEQ kann als Erweiterung von CAS gesehen werden, wobei sich die Definition von „approximativ“ so geändert hat, dass begrenzte Einfügungen in den Center erlaubt werden. Die Definition kann noch weiter geändert werden, indem man ebenfalls begrenzte Löschungen aus dem Center erlaubt.

6.2 Superstrings und Supersequenzen

Wir betrachten nun zwei verwandte Probleme von CAS und CASEQ, bei denen im Gegensatz zu Substrings und Subsequenzen nun Superstrings und Supersequenzen gesucht werden. Für diese NP-vollständigen Probleme liegen ebenfalls einige Resultate der parametrisierten Komplexität vor.

Betrachten wir nun das erste Problem, Shortest Common Superstring, in dem anstatt eines gemeinsamen Substrings ein gemeinsamer Superstring einer Menge von Strings gesucht wird [Bli+15]:

SHORTEST COMMON SUPERSTRING

Instanz: Eine Menge von n Strings $S = \{s_1, \ldots, s_n\}$ über einem Alphabet Σ und eine nichtnegative ganze Zahl l.

Frage: Gibt es einen String $s \in \Sigma^*$ der Länge höchstens l, der alle Strings aus S als Substrings enthält?

Eine Verallgemeinerung dieses Problems ist Partial Common Superstring:

PARTIAL COMMON SUPERSTRING

Instanz: Eine Multimenge S von Strings über einem Alphabet Σ und nichtnegative ganze Zahlen l, k.

Frage: Gibt es einen String $s \in \Sigma^*$ der Länge höchstens l, sodass s ein Superstring der Multimenge mit mindestens k Strings $S' \subseteq S$ ist?

Für $k = |S|$ entspricht diese Verallgemeinerung Shortest Common Superstring.

Ein interessantes Resultat ist, dass Partial Common Superstring parametrisiert mit k oder l in FPT liegt, CAS(l) hingegen ist W[2]-schwer.

In dem zweiten Problem, Shortest Common Supersequence, wird anstatt einer Subsequenz eine Supersequenz gesucht; hier sind ähnliche Resultate der parametrisierten Komplexität wie für Shortest Common Superstring bekannt [Pie03]:

SHORTEST COMMON SUPERSEQUENCE

Instanz: Eine Menge von n Strings $S = \{s_1, \ldots, s_n\}$ über einem Alphabet Σ und eine nichtnegative ganze Zahl l.

Frage: Gibt es einen String $s \in \Sigma^*$ der Länge höchstens l, der eine Supersequenz von jedem String in S ist?

7 Fazit und Ausblick

In dieser Arbeit haben wir die Komplexität von CAS (Common Approximate Substring) als Problem des approximativen String-Matching betrachtet.

Dazu wurde auf approximatives String-Matching für Mengen von Strings und dessen besondere Bedeutung für die Biologie, beispielsweise für die Forschung mit DNA-Sequenzen, sowie die zugehörigen Probleme CAS und CASEQ (Common Approximate Subsequence) eingegangen.

Für das Problem CAS haben wir die Komplexität analysiert. Dazu wurde die NP-Vollständigkeit nachgewiesen und außerdem eine Analyse der parametrisierten Komplexität durchgeführt.

Die betrachteten Parameter von CAS waren die Alphabetgröße $|\Sigma|$, die Anzahl der Strings m, die Länge der Strings n, die Länge des gesuchten Substrings l und die maximale Distanz d, für die einige Resultate der parametrisierten Komplexität gefunden wurden.

Für zwei parametrisierte Varianten des Problems, wenn $|\Sigma|$ und l oder n Parameter sind, liegt CAS in der Klasse FPT. Damit kann dieser Ansatz aber nicht für die beiden wichtigen Aufgaben der Textsuche (zu großes Alphabet) und Computational Biology (zu langer Center) angewendet werden. Weitere FPT-Resultate wurden nicht gefunden, für andere Varianten mit verschiedenen Parametern gibt es für CAS aber untere und obere Schranken in der W-Hierarchie.

Für zwei Varianten, wenn m, $|\Sigma|$ und d oder nur $|\Sigma|$ und d Parameter sind, wurden weder FPT-Resultate noch Schwere-Resultate gefunden, was also Möglichkeiten für weitere Forschung bietet. Ebenso können einige Resultate noch verbessert oder eingeschränkt werden, unter anderem für Varianten, für die Schwere und obere Schranken, aber keine Vollständigkeit nachgewiesen wurde.

Außerdem haben wir die Resultate von CAS auf ähnliche Probleme übertragen. Betrachtet man zum Beispiel Subsequenzen anstatt Substrings, können für das Problem CASEQ einige Resultate von CAS übernommen werden. Es wurde auch ein Ausblick auf verwandte Superstring- bzw. Supersequenz-Probleme gegeben, für die noch weitere Forschungsmöglichkeiten bestehen.

Literatur

[Bli+15] Ivan Bliznets u. a. *Parameterized Complexity of Superstring Problems*. CoRR **abs/1502.01461** (2015).

[ESW03] Patricia A. Evans, Andrew D. Smith, H.Todd Wareham. *On the complexity of finding common approximate substrings*. Theoretical Computer Science **306** (2003), no. 1–3, S. 407–430. ISSN: 0304-3975.

[FG02] Jörg Flum, Martin Grohe. *Describing Parameterized Complexity Classes*. STACS 2002. Hrsg. von Helmut Alt, Afonso Ferreira. **2285**. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2002, S. 359–371. ISBN: 978-3-540-43283-8.

[FL97] M. Frances, A. Litman. *On covering problems of codes*. English. Theory of Computing Systems **30** (1997), no. 2, S. 113–119. ISSN: 1432-4350.

[Kop+14] Dominik Kopczynski u. a. *Algorithmen auf Sequenzen*. 2014.

[Lan+03] J. Kevin Lanctot u. a. *Distinguishing string selection problems*. Information and Computation **185** (2003), no. 1, S. 41–55. ISSN: 0890-5401.

[Par13] Rainer Parchmann. *Approximative Zeichenkettensuche*. 2013.

[Pie03] Krzysztof Pietrzak. *On the parameterized complexity of the fixed alphabet shortest common supersequence and longest common subsequence problems*. Journal of Computer and System Sciences **67** (2003), no. 4. Parameterized Computation and Complexity 2003, S. 757–771. ISSN: 0022-0000.

[Ric04] Christoph Jan Richter. *Über die Vermeidung redundanter Betrachtungen beim Approximate String Matching*. Diss. Universität Dortmund, 2004.

[Smi04] Andrew David Smith. *Common Approximate Substrings*. AAINR06882. Diss. Fredericton, N.B., Canada, Canada: University of New Brunswick, 2004. ISBN: 0-494-06882-5.

[VC14] Heribert Vollmer, Maurice Chandoo. *Theorie der parametrisierten Komplexität*. 2014.