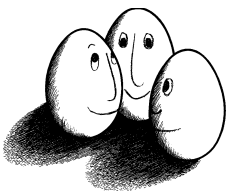
Diplomarbeit

Datenaggregation von Betriebssystemdaten durch Hierarchical Heavy Hitters

Peter Fricke



Diplomarbeit
Fakultät Informatik
Technische Universität Dortmund

Dortmund, 30. März 2010

Betreuer:

Prof. Dr. Katharina Morik
Dipl.-Inform. Marco Stolpe

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

1 Einleitung

Das vollständige Speichern von Datenströmen ist wegen der großen anfallenden Datenmengen oft nicht möglich, daher ist man an Methoden zur Zusammenfassung von Datenströmen interessiert. Klassische Methoden verwenden flache Zusammenfassungen wie die häufigsten Stromelemente als kondensierte Darstellung des Datenstroms. Die Elemente des Stroms weisen allerdings oft eine hierarchische Struktur auf. Die darin enthaltene Information geht bei einer solchen flachen Zusammenfassung des Stroms verloren. Neuere Verfahren berücksichtigen bei der Berechnung der kondensierten Darstellung des Datenstroms diese hierarchische Struktur. Algorithmen zur Berechnung der Hierarchical Heavy Hitters sind ein solches Verfahren.

Gegenstand dieser Diplomarbeit ist es, diesen Ansatz auf Datenströme von Betriebssystemaufrufen zu übertragen und kondensierte Darstellungen von Strömen von Systemaufrufen zu erzeugen, welche die hierarchische Struktur der Stromelemente erfassen.

1.1 Hintergrund

Betriebssystemaufrufe bilden die Schnittstelle zwischen den Benutzerprogrammen und dem Betriebssystem. Über sie stellt das System den Benutzerprogrammen seine Dienste zur Verfügung. Beispiele für Systemaufrufe sind `open` zum Öffnen einer Datei, `read` zum Lesen aus der geöffneten Datei und `write` zum Schreiben in die Datei. Über *Parameter* werden die Aufrufe genauer beschrieben, so wird zum Beispiel beim Systemaufruf `open` über einen Parameter festgelegt, ob die Datei zum Lesen, zum Schreiben oder für beides geöffnet werden soll.

Das Protokollieren von Systemaufrufen erlaubt einen detaillierten Einblick in die Funktionsweise des Systems. Anhand der gesammelten Daten können Potenziale für die Leistungssteigerung erkannt oder sicherheitsrelevante Vorfälle identifiziert werden. Allerdings treffen die Daten bei einem vollständigen Protokollieren der Systemaufrufe mit so hoher Geschwindigkeit und in so großer Menge ein, dass eine Speicherung längerer Zeiträume kaum möglich ist und für eine sofortige Verarbeitung nur sehr wenig Zeit je Element zur Verfügung steht. Man ist daher an einer kondensierten Darstellung der Daten interessiert, die so schnell berechnet werden kann, dass eine vollständige Zwischenspeicherung der Daten nicht erforderlich ist. Diese Darstellung soll platzsparend sein, aber die wesentlichen Informationen der ursprünglichen Daten enthalten.

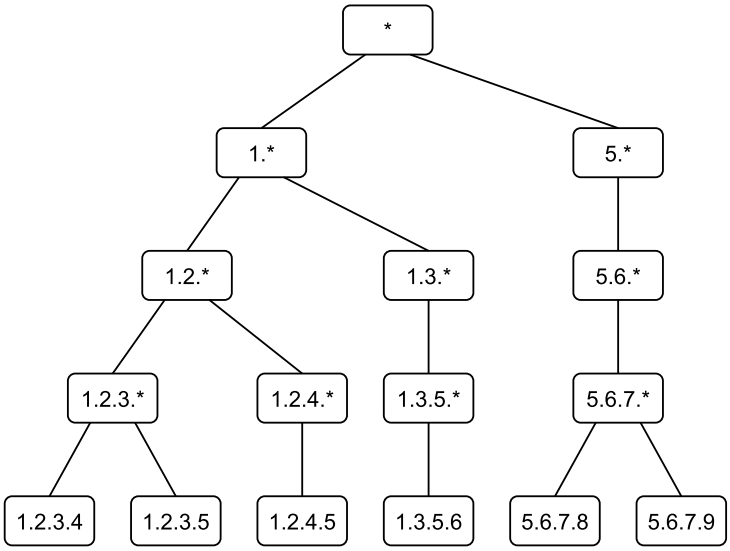


Abbildung 1.1: IP-Adressen (Blätter) und Subnetze (innere Knoten)

Dieser Problemtyp ist nicht auf das Protokollieren von Systemaufrufen beschränkt und gewinnt wegen der allerorts stetig steigenden Datenflut zunehmend an Bedeutung, die anfallenden Daten werden dabei als *Datenstrom* bezeichnet:

„In the data stream scenario, input arrives very rapidly and there is limited memory to store the input. Algorithms have to work with one or few passes over the data, space less than linear in the input size or time significantly less than the input size.“ ([Muthukrishnan, 2005], S. 5).

Ein gut untersuchtes Beispiel für Datenströme sind die Daten, die in IP-Netzwerken verarbeitet werden. An den Routern fallen große Mengen an Informationen über Absender, Ziel und Typ der vermittelten Pakete an, die einerseits wichtige Informationen über den Netzverkehr enthalten, andererseits aber zu umfangreich für eine vollständige Speicherung sind. Interessiert man sich beispielsweise nur für die Absender der Pakete, so kann man die IP-Adressen der Absender als Datenstrom auffassen. Auch Wettersatelliten und Netze von Überwachungskameras erzeugen Datenströme. Flugzeuge, LKW, komplexe medizinische Geräte und Industrieroboter, die Informationen über ihren Betriebszustand und Fehlermeldungen erfassen, erzeugen ebenfalls Datenströme.

Die Herausforderung ist in allen Fällen gleich: Der verfügbare Speicherplatz ist nicht ausreichend für eine vollständige Speicherung der Daten und die Zeit für die Verarbeitung jedes einzelnen Elementes des Datenstroms ist gering.

Wie ein solcher Datenstrom in eine kondensierte Darstellung überführt werden kann, ist von den Daten und dem Anwendungsbereich abhängig.

Man kann den Datenstrom zusammenfassen, indem man alle Elemente speichert, deren Häufigkeit einen bestimmten Wert übersteigt. Ist die Anzahl der Elemente im Strom N, so

werden Elemente, deren Häufigkeit nicht kleiner als ϕN ist, als *Heavy Hitters* bezeichnet ([Cormode et al., 2003]). Die Heavy Hitters können nützliche Informationen über den Datenstrom enthalten: In einem Datenstrom aus IP-Adressen sind es die Adressen der Rechner, die für einen bestimmten Anteil des gesamten Verkehrs verantwortlich sind.

In vielen Fällen besitzen die Elemente des Datenstroms eine hierarchische Struktur. IP-Adressen kann man beispielsweise in Netze, Subnetze und Subsubnetze einordnen, die einzelnen Subnetze werden durch Präfixe von IP-Adressen bezeichnet (siehe Abbildung 1.1). Es kann dann sinnvoll sein, die häufigen Elemente nicht nur auf der Ebene der Stromelemente zu bestimmen, sondern für alle Ebenen der Hierarchie. In Abbildung 1.1 können dann nicht nur Blätter der Hierarchie (Stromelemente, hier: IP-Adressen) häufige Elemente sein, sondern auch innere Knoten (Präfixe von IP-Adressen). Innere Knoten e repräsentieren die Aggregation der Stromelemente, die Nachfahren von e sind: Die Häufigkeit eines inneren Knotens e ist die Summe der Häufigkeiten der Stromelemente, die Nachfahren von e in der Hierarchie sind.

Hierarchical Heavy Hitters erweitern diesen Gedanken um einen weiteren Aspekt:

Hierarchical Heavy Hitters (HHH) sind solche Knoten der Hierarchie, deren Häufigkeit *nach Abzug der Häufigkeit von Nachfahren, die HHH sind*, größer als ϕN ist (vgl. [Hershberger et al., 2005]).

Die Häufigkeit von HHH-Nachfahren abzuziehen ist sinnvoll, weil sonst alle Vorfahren von häufigen Knoten ebenfalls häufig wären und die Mengen der Hierarchical Heavy Hitters wesentlich größer ausfallen würden, ohne zusätzliche Informationen zu enthalten. Für die Berechnung der Hierarchical Heavy Hitters gibt es verschiedene Approximationsalgorithmen. In [Cormode et al., 2008] wird der Einsatz dieser Algorithmen auf Datenströmen aus IP-Adressen beschrieben. Diese Algorithmen sind auch für den allgemeineren Fall geeignet, in dem die Elemente des Stroms Tupel aus mehreren Variablen sind, die jeweils einer Hierarchie entstammen. Ein Beispiel dafür ist ein Datenstrom, der nicht nur die IP-Adressen der Absender der Pakete enthält, sondern auch die Adressen der Empfänger. Absender- und Empfängeradressen haben eine hierarchische Struktur, daher werden die HHH in diesem Fall als *mehrdimensionale Hierarchical Heavy Hitters* bezeichnet.

1.2 Fragestellung

Auch Betriebssystemaufrufe weisen eine hierarchische Struktur auf: Viele Systemaufrufe arbeiten mit Dateisystempfaden, die eine natürliche Hierarchie bilden (*Pfadhierarchie*, siehe Abbildung 1.2). Auch die Systemaufrufe selbst lassen sich in eine Hierarchie einordnen: Systemaufrufe lassen sich anhand ihrer Funktion zu Gruppen zusammenfassen, und gleiche Systemaufrufe lassen sich anhand ihrer Parameter weiter hierarchisch einteilen (*Aufrufhierarchie*). Schließlich bilden auch Sequenzen von Systemaufrufen eine Hierarchie (*Sequenzhierarchie*). Wurden unmittelbar vor einem Systemaufruf s_1 die Aufrufe `open`, `write`, `read` in dieser Reihenfolge ausgeführt und vor s_2 die Aufrufe `write`, `write`, `read`, so kann man die Sequenzen `/read/write/open` und `/read/write/write` als Pfade

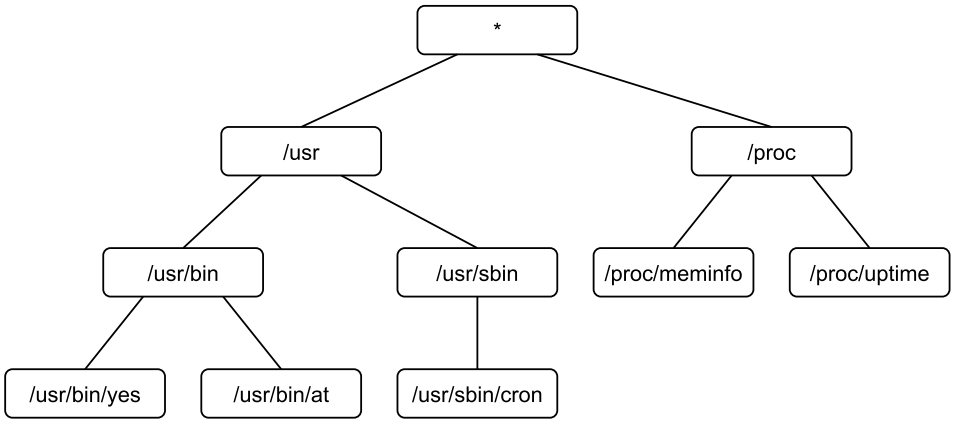


Abbildung 1.2: Pfadhierarchie: Dateisystempfade bilden eine natürliche Hierarchie.

in einer hierarchischen Darstellung der Sequenzen auffassen (siehe Abbildung 1.3). Der unmittelbare Vorgängeraufruf `read` erhält die höchste Position, entferntere Vorgänger erhalten tiefere Positionen. Die chronologische Reihenfolge wird also umgekehrt. Sowohl s_1 als auch s_2 haben einen `read`-Aufruf als Vorgänger und einen `write`-Aufruf als Vorvorgänger. Diese Gemeinsamkeit kann durch den inneren Knoten `/read/write` dargestellt werden.

In dieser Arbeit wird die Frage behandelt, wie man aus Datenströmen von Systemaufrufen eine kondensierte Darstellung als Menge von Hierarchical Heavy Hitters (HHH-Menge) berechnen kann. Es wird untersucht, welche Hierarchien sich für einen Strom

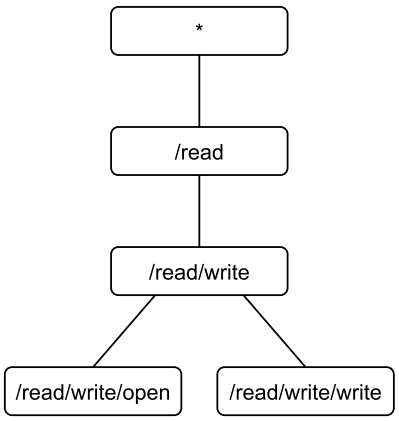


Abbildung 1.3: Sequenzhierarchie der Aufrufe `open`, `write`, `read` und `write`, `write`, `read`

von Systemaufrufen angeben lassen und wie die hierarchischen Informationen aus dem Strom der Aufrufe berechnet werden können.

Darüber hinaus wird der Speicherbedarf und die benötigte Rechenleistung der Algorithmen zur Berechnung der Hierarchical Heavy Hitters gemessen. Es wird untersucht, welche der Hierarchien, in die sich ein Systemaufruf einordnen lässt (Pfadhierarchie, Aufrufhierarchie, Sequenzhierarchie), wichtig sind und auf welche zugunsten einer schnelleren Verarbeitung verzichtet werden kann. Auch der Einfluss des Parameters, der die Approximationsgüte der Algorithmen steuert, auf den Ressourcenverbrauch wird gemessen und mit den theoretisch zu erwartenden Ergebnissen verglichen.

Ströme von Systemaufrufen durch HHH-Mengen darzustellen, ist ein neuer Ansatz und wirft einige Fragen auf. So ist die Größe der kondensierten Darstellung sehr gering. Es ist unklar, ob nach der Zusammenfassung eines Datenstroms, dessen Größe im Bereich einiger Terabyte liegt, zu einer Darstellung als HHH-Menge, deren Größe bei einigen hundert oder tausend Byte liegt, genug Information übrig bleibt.

Die Antwort auf diese Frage hängt davon ab, wofür die kondensierte Darstellung verwendet werden soll. Im Rahmen dieser Arbeit werden beispielhaft zwei Fälle betrachtet: Zum einen wird untersucht, ob man Anwendungen (Programme) an der HHH-Menge ihrer Systemaufrufe erkennen kann. Diese Fragestellung ist im Bereich der Intrusion Detection von Interesse, wenn manipulierte Varianten einer Anwendung vom Original unterschieden werden sollen ([Forrest et al., 2008] [Warrender et al., 1999]). Zum anderen wird untersucht, ob man Benutzer eines Rechnersystems an der HHH-Menge ihrer Systemaufrufe erkennen kann. Ein möglicher Anwendungsbereich dafür ist die Masquerade Detection, bei der erkannt werden soll, ob sich jemand über ein fremdes Benutzerkonto Zugang zu einem System verschafft hat ([Schonlau et al., 2001], [Garg et al., 2006]).

Das Erfassen aller Systemaufrufe kann die Leistung eines Rechners beeinträchtigen.[1] Daher wird am Beispiel der Anwendungserkennung untersucht, ob einzelne Aufrufe wichtiger sind als andere und ob bereits eine Teilmenge der Aufrufe ausreicht, um Anwendungen zuverlässig zu erkennen.

Die Fragestellung für diese Arbeit lässt sich also so zusammenfassen:

- *Konzept*: Wie kann man aus Datenströmen von Systemaufrufen eine kondensierte Darstellung als Menge von Hierarchical Heavy Hitters berechnen?
- *Ressourcenverbrauch*: Wie ist der Ressourcenverbrauch der HHH-Algorithmen bei der Verarbeitung von Betriebssystemdaten und wie ändert sich dieser bei Variation der Parameter, der Implementierungsdetails und des Detaillierungsgrades der Hierarchie?
- *Anwendungserkennung*: Lassen sich verschiedene Anwendungen anhand der Informationen, die durch die Zusammenfassung ihrer Systemaufrufe zu HHH-Mengen entstehen, überhaupt noch unterscheiden?

[1] Siehe Abschnitt 5.3 für ein Beispiel.

- *Benutzererkennung*: Lassen sich anhand dieser Informationen auch Benutzer voneinander unterscheiden oder Benutzergruppen zuordnen?
- *Reduzierter Datenstrom*: Gibt es wichtige und weniger wichtige Systemaufrufe? Ist es möglich, die Menge der erfassten Systemaufrufe zu beschränken, ohne dabei wesentliche Informationen über den Datenstrom zu verlieren?

1.3 Vorgehensweise

Zur Untersuchung dieser Fragen mussten zunächst Datenströme von Systemaufrufen erfasst werden. Ursprünglich sollten die Datensätze von Lehrstuhl Informatik 12 der TU Dortmund zur Verfügung gestellt werden. Da das Erzeugen dieser Daten aufwändiger als erwartet war, standen die Daten des Lehrstuhls 12 erst relativ spät zur Verfügung. Es war daher erforderlich, ersatzweise eigene Datensätze zu erzeugen. Als Betriebssystem wurde Linux gewählt. Es gibt verschiedene Werkzeuge, mit denen man die Aufrufe protokollieren kann, die in einem System erfolgen. Diese Werkzeuge unterscheiden sich in der Detailliertheit der protokollierten Daten. Für das Erfassen der Daten wurde das Werkzeug strace ausgewählt, weil es zu jedem Systemaufruf ausführliche Informationen über die verwendeten Parameter angibt. Mit strace wurden mehrfach alle Systemaufrufe von elf verschiedenen Anwendungen protokolliert. Diese Daten wurden später für die Anwendungserkennung verwendet (*Anwendungsdatensatz*).

Vom Lehrstuhl 12 wurde ein zweiter Datensatz zur Verfügung gestellt, der Aufrufe verschiedener Benutzer aus verschiedenen Benutzergruppen enthält. Wegen der oben erwähnten Leistungsbeeinträchtigung des Systems beim Protokollieren wurden für diesen Datensatz nur zwei von über 300 Systemaufrufen protokolliert. Dieser Datensatz (*Benutzerdatensatz*) wurde für die Benutzererkennung verwendet.

Für die Verarbeitung der Datensätze wurde ein System aus drei Komponenten entwickelt. Die erste Komponente extrahiert die hierarchischen Informationen aus den Datensätzen. Das ist erforderlich, weil die hierarchische Information nicht unmittelbar abgelesen werden kann, die Rohdaten sind keine geeignete Eingabe für die HHH-Algorithmen. Vor allem die Berechnung der verwendeten Pfade für die Einordnung in die Pfadhierarchie erfolgt in dieser Komponente.

Auch die Einordnung der einzelnen Systemaufrufe in die Aufrufhierarchie erfolgt in der ersten Komponente. Dazu wurde eine Taxonomie der Systemaufrufe und ihrer Parameter erstellt. Für die Einordnung der einzelnen Aufrufe in die Sequenzhierarchie speichert die erste Komponente die Vorgängeraufrufe.

Die zweite Komponente berechnet die Hierarchical Heavy Hitters und verwendet dabei die extrahierten hierarchischen Informationen der ersten Komponente. Für diese Berechnung wurden zwei Approximationsalgorithmen implementiert, die auch sehr große Datenmengen verarbeiten können, ohne viel Speicher zu benötigen ([Cormode et al., 2008]). Ein exakter Algorithmus zur HHH-Berechnung ([Cormode et al., 2004]) wur-

de ebenfalls implementiert, um die Qualität der Approximationslösungen bewerten zu können.

Aufgabe der dritten Komponente ist die Anwendungserkennung (bzw. Benutzererkennung). Wenn für einen neuen Datenstrom von Systemaufrufen entschieden werden soll, von welcher Anwendung er erzeugt wurde, wird mit der ersten und zweiten Komponente die HHH-Menge des Stroms berechnet. Diese wird mit den HHH-Mengen bekannter Anwendungen verglichen. Die k HHH-Mengen bekannter Anwendungen, die der HHH-Menge des neuen Stroms am ähnlichsten sind, werden als k nächste Nachbarn bezeichnet. Der neue Strom wird der Anwendung zugeordnet, die unter den k nächsten Nachbarn am häufigsten ist. Diese Vorgehensweise wird als Verfahren der nächsten Nachbarn bezeichnet (k-Nearest Neighbors, k-NN. Siehe [Aha et al., 1991], [Hastie et al., 2001], S. 14 und [Mitchell, 1997], S. 231).

Um die nächsten Nachbarn bestimmen zu können, sind Ähnlichkeitsmaße für HHH-Mengen erforderlich. Die üblichen Ähnlichkeitsmaße für Mengen wie Jaccard- und Dice-Index basieren auf der Größe der Schnittmenge. Solche Ähnlichkeitsmaße sind für HHH-Mengen nur schlecht geeignet, weil sie die hierarchische Struktur der Elemente der Mengen nicht erfassen können. Daher wurden hierarchiebasierte Ähnlichkeitsmaße aus der Literatur ([Ganesan et al., 2003]) implementiert und weiter entwickelt.

Die drei Komponenten wurden in das Data Mining Werkzeug RapidMiner integriert. RapidMiner stellt eine breite Palette von Algorithmen und Verfahren zur Verfügung, die bei der Durchführung der Experimente verwendet werden konnten.

Für das Erstellen der Komponenten waren umfangreiche Implementierungsarbeiten erforderlich. Auch das ursprünglich nicht eingeplante Erzeugen eigener Daten erforderte Zeit, so dass die Experimentierphase gestrafft wurde. Der Schwerpunkt der Arbeit liegt daher auf dem Lösungsansatz und der Implementierung.

1.4 Gliederung der Arbeit

In Kapitel 2 wird zunächst beschrieben, was Betriebssystemaufrufe sind und welche Funktion sie erfüllen. An einer Auswahl von Aufrufen wird dann die Funktionsweise genauer erläutert und gezeigt, welche Systemaufrufe das Kopieren einer Datei auslöst. In Kapitel 3 werden Hierarchical Heavy Hitters definiert und zwei Approximationsalgorithmen zu ihrer Berechnung beschrieben, bevor Kapitel 4 einen Überblick über Forschungsarbeiten zur Verarbeitung von Systemaufrufen gibt.

Der in dieser Arbeit gewählte Lösungsansatz für die Datenaggregation von Betriebssystemdaten mit Hierarchical Heavy Hitters wird in Kapitel 5 beschrieben. Zunächst wird dargestellt, welche hierarchischen Variablen verwendet werden und wie der Anwendungsdatensatz, der mit strace erzeugt wurde, verarbeitet wird, um daraus Tupel hierarchischer Variablen zu extrahieren. Ein Strom solcher Tupel bildet die Eingabe für die Algorithmen zur Berechnung der Hierarchical Heavy Hitters, Ausgabe sind Mengen von Hierarchical Heavy Hitters. Im zweiten Abschnitt werden Ähnlichkeitsmaße für sol-

che HHH-Mengen beschrieben, welche die hierarchische Struktur berücksichtigen. Diese Ähnlichkeitsmaße werden vom k-NN-Algorithmus zur Klassifikation verwendet. Im dritten Abschnitt wird das Protokollieren der Systemaufrufe einzelner Anwendungen und die Erstellung des Anwendungsdatensatzes beschrieben.

Nach dieser Beschreibung der grundlegenden Vorgehensweise und der Erstellung des Datensatzes werden in Kapitel 6 die Details der Implementierung erläutert. Zunächst wird das Einlesen der Daten und die Simulation des Betriebssystems beschrieben, die zur Extraktion der hierarchischen Variablen erforderlich ist. Anschließend wird die Implementierung der Algorithmen zur Berechnung der HHH-Mengen skizziert und ein Mechanismus zum Puffern der Ergebnisse vorgestellt, der die erforderlichen Rechenzeiten für die Experimente in Kapitel 8 erheblich reduziert hat. Das Kapitel schließt mit einer Beschreibung der Integration des Systems in das Data Mining Werkzeug RapidMiner.

Kapitel 7 beschreibt Experimente zum Ressourcenbedarf der Algorithmen. Kapitel 8 untersucht, ob man Anwendungen anhand der HHH-Mengen ihrer Systemaufrufe zuverlässig erkennen kann und ob es für zufriedenstellende Ergebnisse bereits ausreicht, eine Teilmenge der Systemaufrufe zu erfassen und die HHH-Mengen dieses reduzierten Datenstroms zu verwenden. Kapitel 9 untersucht, ob man Benutzer anhand ihrer HHH-Mengen erkennen kann.

In Kapitel 10 wird die Arbeit zusammengefasst und die Ergebnisse bewertet.

Grundbegriffe des maschinellen Lernens und des Data Mining werden vorausgesetzt, dabei wird auf die Bücher von [Mitchell, 1997], [Han und Kamber, 2006] und [Hastie et al., 2001] verwiesen.

2 Betriebssystemaufrufe

Im Folgenden wird zunächst beschrieben, was Betriebssystemaufrufe sind und welche Funktion sie erfüllen. An einer Auswahl von Aufrufen wird dann die Funktionsweise genauer erläutert. Im darauf folgenden Abschnitt wird an einem Beispiel gezeigt, welche Systemaufrufe das Kopieren einer Datei auslöst.

2.1 Definition und Überblick

Betriebssystemaufrufe bilden die Schnittstelle zwischen den Anwendungen und dem Betriebssystem. Über sie stellt das System den Prozessen seine Dienste zur Verfügung ([Tanenbaum, 2003], S.59).

Durch die Verwendung von Systemaufrufen ist es Anwendungen möglich, die Bearbeitung von Aufgaben wie das Lesen einer Datei zu veranlassen, für die sie selbst keine ausreichenden Rechte besitzen: Die Anwendungen laufen im Benutzermodus, der keinen Zugriff auf die Hardware erlaubt, die Systemaufrufe werden dagegen im Kernmodus ausgeführt, der vollen Zugriff auf die Hardware erlaubt.

Der Mechanismus zur Ausführung eines Systemaufrufs lässt sich wie folgt beschreiben: Die Anwendung stellt die Parameter des Aufrufs zur Verfügung, z. B. auf dem Stack. Außerdem wird die Nummer des gewünschten Systemaufrufs bereitgestellt, dies kann in einem Register erfolgen. Die Anwendung löst eine Unterbrechung (trap) aus oder führt eine spezielle Systemaufruf-Anweisung durch, durch welche die Kontrolle an das Betriebssystem abgegeben wird. Dieses kann anhand der Nummer des Systemaufrufs und anhand der Parameter bestimmen, welche Aufgabe für die aufrufende Anwendung gelöst werden soll. Die Aufgabe wird vom Betriebssystem im Kernmodus bearbeitet, der vollen Zugriff auf die Hardware besitzt. Anschließend wird die Kontrolle an die aufrufende Anwendung zurück gegeben, die ihre Arbeit mit der auf den Aufruf folgenden Anweisung fortsetzt.

Allerdings führen nur wenige Anwendungen die genannten Schritte tatsächlich selbst aus, meist wird stattdessen eine Bibliotheksfunktion verwendet. Diese Bibliotheksfunktionen sind Hüllen (Wrapper) für die Systemaufrufe und bieten einen komfortableren Zugriff auf die Funktionalität des Systemaufrufs, indem sie die technischen Details wie die Ablage der Aufrufnummer in einem Register und die Übergabe der Kontrolle an das Betriebssystem übernehmen. Aus Sicht der Anwendung erscheint der Systemaufruf dann wie der Aufruf einer normalen Bibliotheksfunktion ([Silberschatz et al., 2010], S. 55 ff., [Tanenbaum, 2003], S.60).

Systemaufruf	Beschreibung
`s = stat64( name, buf )`	Status einer Datei ermitteln
`fd = open( name, flags, mode )`	Datei öffnen, ggf. erzeugen
`n = read( fd, buf, count )`	Daten aus Datei in Puffer lesen
`n = write( fd, buf, count )`	Daten aus Puffer in Datei schreiben
`s = fstat64( fd, buf )`	Status einer Datei ermitteln
`s = close( fd )`	Offene Datei schließen
`s = execve( name, argv, env )`	Speicherabbild eines Prozesses ersetzen
`exit_group( status )`	Prozess beenden, Status zurückgeben
`s = fcntl64( fd, cmd, ... )`	Manipulieren von Dateideskriptoren
`tid = clone( ..., flags, ... )`	Erzeugen eines neuen Prozesses/Threads
`s = access( name, mode )`	Ermitteln der Zugriffsrechte für Datei
`fd = socket( domain, type, prot )`	Erzeugen eines Sockets

Tabelle 2.1: Liste ausgewählter Systemaufrufe, ihrer Parameter und Rückgabewerte

In Tabelle 2.1 (vgl. [Tanenbaum, 2003], S.62) ist eine kleine Auswahl von Systemaufrufen aufgeführt, die nachfolgend erläutert wird. Die ersten acht Aufrufe sind für das Verständnis des Beispiels in Abschnitt 2.2 erforderlich, die übrigen vier werden in den Beispielen in Kapitel 5 verwendet. Angegeben ist jeweils der Name des Aufrufs, die Parameter (in Klammern) und der Rückgabewert. Ausführlichere Informationen über jeden Systemaufruf erhält man unter unixoiden Systemen in Abschnitt zwei des Handbuchs (Manpages), also beispielsweise für `open` mit dem Befehl `man 2 open`.

Der Aufruf `stat64` berechnet Statusinformationen wie Größe und Zugriffsrechte für die Datei `name`. Diese werden in den Puffer geschrieben, der an der Speicheradresse `buf` beginnt. Der Rückgabewert `s` ist -1, wenn ein Fehler aufgetreten ist, ansonsten 0.

Der Aufruf `open` öffnet eine Datei mit dem Namen `name`, dabei wird über die `flags` unter anderem festgelegt, ob die Datei nur zum Lesen geöffnet werden soll (`O_RDONLY`), nur zum Schreiben (`O_WRONLY`) oder zum Lesen und Schreiben (`O_RDWR`). Existiert die Datei nicht und die `flags` enthalten den Wert `O_CREAT`, so wird die Datei erzeugt, die Zugriffsrechte werden dabei als Zahl codiert über den Parameter `mode` festgelegt. Die Verknüpfung der einzelnen Werte des Parameters `flags` erfolgt durch logisches Oder (z. B. `O_WRONLY|O_CREAT`). Der Rückgabewert `fd` ist ein sogenannter Dateideskriptor, ein kleiner ganzzahliger Wert, der ein Index in der Tabelle der offenen Dateien ist, die für jeden Prozess verwaltet wird. Die geöffnete Datei kann künftig innerhalb des Prozesses über diesen Dateideskriptor angesprochen werden, bis der Deskriptor (und die Datei) über den Aufruf `close` wieder geschlossen wird. Die Dateideskriptoren 0, 1 und 2 werden üblicherweise für die Standardeingabe (`stdin`), die Standardausgabe (`stdout`) und die Standardfehlerausgabe (`stderr`) verwendet und sind normalerweise mit dem Terminal vorbelegt ([Tanenbaum, 2003], S. 734).

Der Systemaufruf `read` verwendet einen solchen Dateideskriptor `fd` als Parameter, um Daten aus einer bereits geöffneten Datei zu lesen. Es wird versucht, `count` Byte aus dieser Datei zu lesen und in den Puffer zu schreiben, der an Speicheradresse `buf` beginnt. Die Anzahl der tatsächlich gelesenen Byte wird als Rückgabewert `n` übergeben. `n` kann kleiner als `count` sein, wenn das Dateiende erreicht wurde. Wenn ein Fehler aufgetreten ist, wird -1 zurückgegeben.

Der Aufruf `write` arbeitet analog zu `read` und versucht, in die durch den Dateideskriptor `fd` bezeichnete Datei `count` Byte aus dem Puffer `buf` zu schreiben.

Der Systemaufruf `fstat64` entspricht dem Aufruf `stat64`, bezeichnet die Datei aber nicht durch ihren Namen `name`, sondern durch den Dateideskriptor `fd`. Der Rückgabewert `s` ist -1, wenn ein Fehler aufgetreten ist, ansonsten 0.

Durch den Aufruf `execve` wird die Anwendung aus der Datei `name` in den Speicher geladen, der aktuelle Prozess wird überschrieben. Über den Wert `argv` werden die Parameter festgelegt, die dieser Anwendung übergeben werden sollen. In ähnlicher Weise werden über den Parameter `env` die Umgebungsvariablen festgelegt. In der Regel werden offene Dateideskriptoren durch den Aufruf von `execve` geschlossen, dieses Verhalten kann allerdings durch das Setzen eines Flags des Dateideskriptors verändert werden. Dieses `FD_CLOEXEC`-Flag kann beim Erzeugen des Deskriptors während des Öffnens der Datei gesetzt werden (über den Parameter `flag`) oder auch später von anderen Systemaufrufen verändert werden.

Durch den Systemaufruf `exit_group` wird ein Prozess beendet, dabei kann der Status des Prozesses (normale Beendigung oder Fehler) übergeben werden. Andere Prozesse können diesen Status abfragen. Ein Rückgabewert ist für diesen Aufruf nicht sinnvoll.

Der Aufruf `fcntl` dient dem Manipulieren von Dateideskriptoren. Er wird für verschiedene Aufgaben wie das Kopieren eines Deskriptors oder das Lesen und Setzen von Flags des Deskriptors verwendet. Ein Beispiel für ein solches Flag ist `FD_CLOEXEC`, das festlegt, ob ein Dateideskriptor während eines Aufrufs von `execve` geöffnet bleibt oder geschlossen wird. Die genaue Funktion des Aufrufs ist vom Wert des Parameters `cmd` abhängig. `F_SETFD` setzt das `FD_CLOEXEC`-Flag, `F_GETFD` liest es und gibt es zurück. `F_DUPFD` kopiert den Dateideskriptor, künftig kann die Datei über beide Deskriptoren angesprochen werden. `F_DUPFD_CLOEXEC` kopiert den Deskriptor ebenfalls, setzt aber zusätzlich das `FD_CLOEXEC`-Flag der Kopie. In den Handbuchseiten sind weitere mögliche Werte für `cmd` angegeben.

Neue Prozesse und neue Threads werden mit dem Systemaufruf `clone` erzeugt. Prozesse sind Ausführungsstränge eines Programms mit *eigenen* Betriebsmitteln, Threads sind Ausführungsstränge innerhalb eines Prozesses und nutzen Betriebsmittel, vor allem Speicher, *gemeinsam*. Über die Parameter des `clone`-Aufrufs lässt sich genau festlegen, ob und gegebenenfalls welche Ressourcen Elter und Kind gemeinsam nutzen sollen. Sowohl Prozesse als auch Threads besitzen eine im System eindeutige *Thread-ID* (TID). Der neue Prozess oder Thread ist eine Kopie des Elters, seine Thread-ID wird als Rückgabewert des Aufrufs ausgegeben. Neben der gemeinsamen Nutzung des Speichers lässt sich auch die gemeinsame Nutzung der Dateideskriptoren durch Kind und Elter einstellen:

Normalerweise erhält das Kind eine Kopie der Dateideskriptoren des Elters. Das Kind kann so vom Elter geöffnete Dateien verwenden, die Deskriptoren sind jedoch unabhängig voneinander und können von Elter und Kind ohne Auswirkungen auf den jeweils anderen geschlossen werden. Enthält der Parameter `flags` dagegen den Wert `CLONE_FILES`, arbeiten Elter und Kind mit denselben Deskriptoren. Änderungen wie das Schließen von Deskriptoren, die ein Thread vornimmt, wirken sich dann unmittelbar auf den jeweils anderen Thread aus.[1]

Das Programm strace, mit dem der Anwendungsdatensatz für diese Diplomarbeit erzeugt wurde, unterscheidet nicht zwischen Prozessen und Threads. Für strace sind Threads Prozesse, die Betriebsmittel gemeinsam nutzen. Beide werden über ihre Thread-ID identifiziert. In den Logdateien von strace sind Prozesse und Threads nicht voneinander zu unterscheiden, daher folge ich dieser Bezeichnungsweise und verwende den Begriff Prozess für Prozesse *und* Threads.

Um zu ermitteln, ob ein Prozess auf eine Datei zugreifen darf, wird der Aufruf `access` verwendet. `access` erhält neben dem Namen `name` der Datei einen Parameter `mode`, der festlegt, welche Zugriffsrechte untersucht werden sollen. Als `mode` wird entweder der Wert `F_OK` angegeben, in diesem Fall wird geprüft, ob die Datei existiert. Alternativ kann geprüft werden, ob der aufrufende Prozess Lese-, Schreib- oder Ausführungsrechte für die Datei besitzt, dazu werden anstelle von `F_OK` die Werte `R_OK`, `W_OK` und `X_OK` verwendet, die durch bitweises Oder verknüpft werden können. Der Aufruf gibt 0 zurück, wenn alle angefragten Rechte bestehen, ansonsten -1.

Mit dem Systemaufruf `socket` wird ein Kommunikationsendpunkt erzeugt, der von Prozessen auf verschiedenen Rechnern oder auf demselben Rechner verwendet werden kann. Über den Parameter `domain` wird die Protokollfamilie angegeben, mögliche Werte sind beispielsweise `LOCAL` für die Kommunikation auf demselben Rechner oder `INET` für das Internet-Protokoll. Sofern erforderlich, kann das verwendete Protokoll über den Parameter `prot` noch genauer festgelegt werden, meist ist dies allerdings nicht erforderlich und es wird 0 angegeben. Über den Parameter `type` wird die Art der Verbindung festgelegt (zuverlässig oder unzuverlässig, verbindungs- oder paketorientiert, siehe dazu [Silberschatz et al., 2010], S. 688 f.). Ein möglicher Wert ist etwa `SOCK_DGRAM` für Datagramme fester maximaler Länge ohne garantierte Zustellung, dagegen steht `SOCK_STREAM` für eine zuverlässige, verbindungsorientierte Kommunikation. Rückgabewert des Aufrufs ist ein Dateideskriptor, Lese- und Schreibvorgänge können daher mit den bereits beschriebenen Systemaufrufen `read` und `write` vorgenommen werden.

2.2 Ein Beispiel: Kopieren einer Datei

In Tabelle 2.2 ist ein Beispiel angegeben. Eine Datei `x` mit dem Inhalt „Premature optimization is the root of all evil.“ wurde kopiert, der Name der Kopie ist `y`. Der Kopiervorgang wurde unter Linux mit dem Befehl `cp x y` durchgeführt, die Betriebssystemaufrufe wur-

[1] Die Darstellung ist gekürzt, für Details siehe `man 2 clone`.

Zeile	TID	Systemaufruf
1	3298	`execve("/bin/cp", ["cp", "x", "y"], [/* 41 vars */]) = 0`
...		
154	3298	`stat64("y", 0xbfaf6b4c) = -1 ENOENT (No such file or directory)`
155	3298	`stat64("x", {st_mode=S_IFREG|0644, st_size=48, ...}) = 0`
156	3298	`stat64("y", 0xbfaf69bc) = -1 ENOENT (No such file or directory)`
157	3298	`open("x", O_RDONLY|O_LARGEFILE) = 3`
158	3298	`fstat64(3, {st_mode=S_IFREG|0644, st_size=48, ...}) = 0`
159	3298	`open("y", O_WRONLY|O_CREAT|O_EXCL|O_LARGEFILE, 0644) = 4`
160	3298	`fstat64(4, {st_mode=S_IFREG|0644, st_size=0, ...}) = 0`
161	3298	`read(3, "Premature optimization is the roo"..., 4096) = 48`
162	3298	`write(4, "Premature optimization is the roo"..., 48) = 48`
163	3298	`close(4) = 0`
164	3298	`close(3) = 0`
165	3298	`close(0) = 0`
166	3298	`close(1) = 0`
167	3298	`close(2) = 0`
168	3298	`exit_group(0) = ?`

Tabelle 2.2: Betriebssystemaufrufe beim Kopieren einer Datei, protokolliert mit strace

den mit dem Werkzeug strace (siehe Abschnitt 5.3.1) protokolliert.[2] Insgesamt wurden 168 Systemaufrufe durchgeführt, die 16 wichtigsten Aufrufe sind in der Tabelle angegeben. Alle Aufgaben wurden innerhalb eines Prozesses ausgeführt.

In Zeile 1 wird der Prozess durch den Systemaufruf `execve` mit dem Inhalt der ausführbaren Datei `/bin/cp` überschrieben, die den Befehl `cp` enthält. Zweiter Parameter des Aufrufs sind die Parameter `["cp", "x", "y"]` des Befehls `cp` (einschließlich des Befehlsnamens). Dritter Parameter sind die Umgebungsvariablen, die strace in abgekürzter Form darstellt (`[/* 41 vars */]`). Durch Verwendung eckiger Klammern kennzeichnet strace den zweiten und dritten Parameter des Aufrufs als Feld (Array).

In den Zeilen 2 bis 153 werden Aufrufe ausgeführt, die wenig zur unmittelbaren Bearbeitung der Aufgabe beitragen, sondern der Einbindung von Bibliotheken dienen.

In Zeile 154 werden mit dem Aufruf `stat64` Informationen über die Zieldatei `y` angefordert, die an die Speicheradresse `0xbfaf6b4c` geschrieben werden sollen. Da die Datei (noch) nicht existiert, ist der Aufruf nicht erfolgreich (Rückgabewert -1). In Zeile 155 werden mit dem Systemaufruf `stat64` erfolgreich (Rückgabewert 0) Informationen über die Quelldatei `x` angefordert. Diesmal schreibt strace als zweiten Wert in die Klammer keine Speicheradresse, sondern das Ergebnis des Systemaufrufs, welches das Betriebssystem an diese Speicheradresse geschrieben hat. Dieses Ergebnis ist als Struktur (struct) organisiert, was strace durch die Verwendung geschweifter Klammern kennzeichnet. Da die Struktur viele Komponenten enthält, wird sie von strace in abgekürzter Form (`...`) dargestellt, nur die wichtigsten Komponenten (`st_mode` und `st_size`) werden angege-

[2] `strace -f -olog cp x y` schreibt die Systemaufrufe des Befehls `cp x y` in die Datei `log`.

ben: Die Datei `x` umfasst 48 Byte (`st_size=48`) und ist kein Verzeichnis, sondern eine normale Datei (`S_IFREG`). Alle haben Leserechte und der Besitzer hat Schreibrechte (`0644`). Die Verknüpfung der Flags in (`st_mode=S_IFREG|0644`) erfolgt durch logisches Oder.

Zeile 156 wiederholt Zeile 154. In Zeile 157 wird die 48 Byte große Datei `x` durch den Aufruf `open` zum Lesen (`O_RDONLY`) geöffnet, sicherheitshalber in einer Weise, die auch für Dateien größer als 2 GiB geeignet ist (`O_LARGEFILE`). Zurückgegeben wird der Dateideskriptor 3, über den die Datei künftig ansprechbar ist. Dieser wird in Zeile 158 verwendet, um wie in Zeile 155 Informationen über Datei `x` einzuholen. In Zeile 155 wurde dazu der Aufruf `stat64` und der Dateiname verwendet, in Zeile 158 stattdessen der Aufruf `fstat64` und der Dateideskriptor. In Zeile 159 wird die Datei `y` zum Schreiben (`O_WRONLY`) geöffnet. Das Flag `O_CREAT` legt fest, dass die Datei erzeugt werden soll, sofern sie nicht bereits existiert, der dritte Parameter (`0644`) vergibt Leserechte für alle und Schreibrechte für den Besitzer. Die neue Datei erhält den Dateideskriptor 4, der in Zeile 160 verwendet wird, um zu erfahren, dass die Datei noch immer eine leere (`st_size=0`) reguläre Datei (`S_IFREG`) mit Leserechten für alle und Schreibrechten für den Besitzer (`0644`) ist.

Die eigentliche Arbeit geschieht in den Zeilen 161 und 162: Zunächst wird in Zeile 161 das Lesen von 4096 Byte aus der Datei mit dem Dateideskriptor 3 veranlasst, tatsächlich gelesen werden nur 48 Byte (Rückgabewert 48). Strace gibt die gelesenen Daten in gekürzter Form innerhalb der Klammer an (`"Premature optimization is the roo"...`). An dieser Stelle steht im eigentlichen Systemaufruf die Speicheradresse des Puffers, in den die Daten gelesen werden sollen. In Zeile 162 wird schließlich das Schreiben dieser 48 Byte in die Datei mit Dateideskriptor 4 veranlasst, tatsächlich können diese vollständig geschrieben werden (Rückgabewert 48).

In den Zeilen 163 bis 168 werden die Dateien `y` und `x` sowie `stdin`, `stdout` und `stderr` mit dem Aufruf `close` unter Verwendung ihrer Dateideskriptoren geschlossen und der Prozess beendet.

3 Datenaggregation durch Hierarchical Heavy Hitters

In diesem Kapitel werden Hierarchical Heavy Hitters und das Approximationsproblem zu ihrer Berechnung auf Datenströmen definiert. Ein Algorithmus zur Berechnung von Hierarchical Heavy Hitters im mehrdimensionalen Fall wird dargestellt und Schranken für Speicherbedarf und Laufzeit werden angegeben. Grundlage des gesamten Kapitels ist [Cormode et al., 2008]. Für die Grundbegriffe der Ordnungs- und Verbandstheorie siehe [Berghammer, 2008].

3.1 Notation und Definitionen

3.1.1 Eine Dimension

Elemente eines Datenstroms haben oft eine hierarchische Struktur. Ein einfaches Beispiel ist ein Strom von IP-Adressen. IP-Adressen entstammen einer Hierarchie (siehe Abbildung 3.1). Die Tiefe der Hierarchie wird als h bezeichnet, die Wurzel (dargestellt als $*$) liegt auf Ebene null, die Blätter auf Ebene h. Die Elemente auf einer Ebene l bilden die Menge $Level(l)$ und sind im Fall der IP-Hierarchie Präfixe der Stromelemente auf Ebene h. Der Elter $par(e)$ eines Elementes e ist das Element, das man erhält, wenn man das Element e in der Hierarchie um eine Ebene nach oben schiebt. Ein Element p ist *generalisierbar* zu q, kurz $p \prec q$ wenn es eine Folge von Elementen $r_1, \ldots, r_n$ mit $n > 1$ gibt, so dass $p = r_1$, $q = r_n$ und für alle i mit $1 \leq i < n$ gilt: $r_{i+1} = par(r_i)$. Es ist dann q eine Generalisierung von p. Eine Generalisierung eines Elementes ist im Fall der IP-Hierarchie ein Präfix des Elementes. Das durch die Generalisierung entfernte Suffix wird als $*$ („beliebig“) dargestellt. Es ist $p \preceq q \Leftrightarrow (p \prec q) \vee (p = q)$. Ein Element ist voll spezifiziert, wenn es keine Generalisierung eines anderen Elementes ist.

Für eine Menge P wird definiert $q \preceq P \Leftrightarrow (\exists p \in P : q \preceq p)$.

Beispiel: In Abbildung 3.1 ist die Hierarchietiefe $h = 4$, $Level(1) = \{(1.*), (5.*)\}$, $par((1.2.3.*)) = (1.2.*)$, $(1.2.3.4) \preceq (1.2.*)$ und $(1.2.3.4) \preceq Level(1)$.

Eine Adresse (1.2.3.4) kann also auf verschiedenen Ebenen betrachtet werden: Als voll spezifiziertes Element (1.2.3.4) oder generalisiert auf beispielsweise 24 Bit (1.2.3.$*$). Methoden, die bei der Zusammenfassung des Stroms nur eine Ebene betrachten, werden als flache Methoden bezeichnet. In der Regel wird dabei die Ebene der voll spezifizierten Elemente betrachtet.

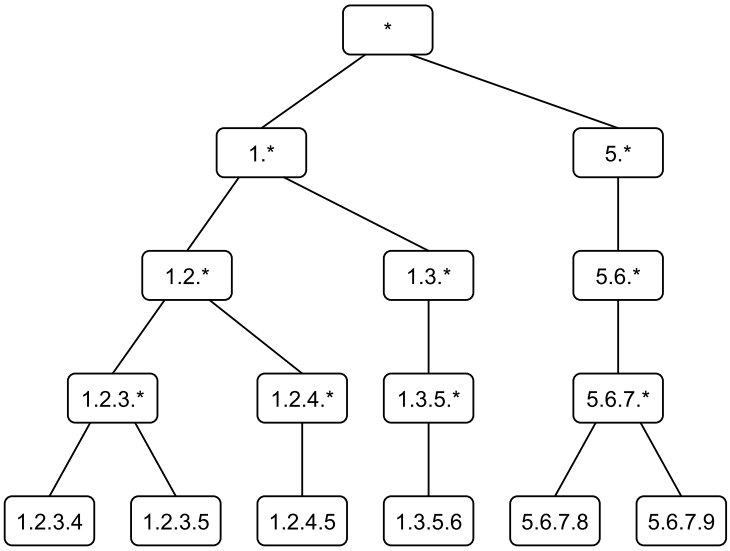


Abbildung 3.1: IP-Adressen (Blätter) und Subnetze

Ein Beispiel für solche flachen Methoden sind die Verfahren zur Berechnung der Heavy Hitters.[1]

Definition 1 (**Heavy Hitter**)**.** Gegeben sei eine Multimenge S der Größe N und ein Schwellwert ϕ mit $0 < \phi < 1$. Ein *Heavy Hitter* ist ein Element e, dessen Häufigkeit $f(e)$ in S nicht kleiner als ϕN ist. Die Menge der Heavy Hitters ist definiert als $HH = \{e \mid f(e) \geq \phi N\}$.

Das *Heavy Hitters Problem* besteht darin, alle Heavy Hitters und ihre Häufigkeiten in einem Datensatz zu finden. Einen Überblick über Lösungen für dieses Problem geben [Cormode et al., 2003]. Ein früher zählerbasierter Algorithmus stammt von [Misra und Gries, 1982]. Mit *Lossy Counting* schlagen [Manku und Motwani, 2002] einen deterministischen Approximationsalgorithmus für Datenströme vor, der die Grundlage der in dieser Arbeit verwendeten Algorithmen zur Berechnung der Hierarchical Heavy Hitters bildet.

Heavy Hitters berücksichtigen die hierarchische Struktur der Stromelemente nicht. Ein Element (1.2.3.$*$) auf der Hierarchieebene 3 kann häufig sein, obwohl alle voll spezifizierten Nachfahren auf Ebene 4 selten sind. In diesem Fall erzeugt ein Subnetz insgesamt viel Datenverkehr, obwohl keiner der einzelnen Rechner viel Verkehr erzeugt. Bei der Zusammenfassung des Stroms durch Heavy Hitters auf der Ebene der einzelnen Rechner geht diese möglicherweise nützliche Information verloren. Dieses Problem lässt sich naiv lösen, indem die Heavy Hitters auf allen Ebenen der Hierarchie berechnet werden. Da alle Generalisierungen von häufigen Elementen ebenfalls häufig sind, wird die Ausgabe allerdings groß und redundant.

[1] Heavy Hitter: Einflussreiche Persönlichkeit (Quelle: `http://dict.leo.org`).

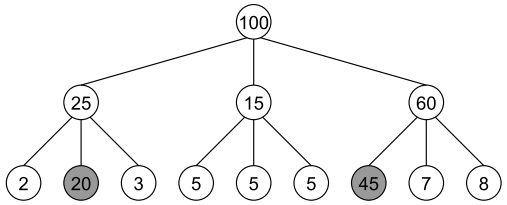


Abbildung 3.2: Heavy Hitters auf der Ebene voll spezifizierter Elemente

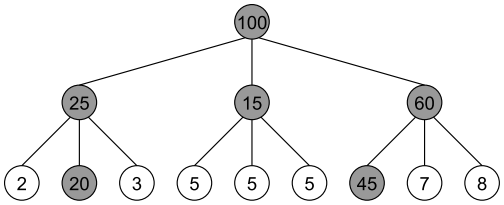


Abbildung 3.3: Naive Lösung, Heavy Hitters auf allen Ebenen

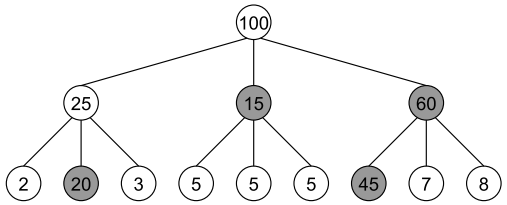


Abbildung 3.4: Menge der Hierarchical Heavy Hitters

Die Menge der Hierarchical Heavy Hitters enthält die hierarchische Information, ist jedoch kleiner als die naive Lösung, weil die Entscheidung, ob ein Element p ausgegeben wird, nicht von seiner (absoluten) Häufigkeit, sondern von einer angepassten Häufigkeit F'_p abhängt. Die angepasste Häufigkeit ist die Häufigkeit nach Abzug der Häufigkeit von Stromelementen, die Nachfahren von Hierarchical Heavy Hitters sind. Abbildung 3.2 zeigt für $N = 100$ und $\phi = 0.1$ die Heavy Hitters auf der Ebene voll spezifizierter Elemente, Abbildung 3.3 zeigt die naive Lösung und Abbildung 3.4 die Hierarchical Heavy Hitters. Die ausgegebenen Elemente sind grau markiert; alle Elemente sind mit ihrer (absoluten) Häufigkeit beschriftet. Die Häufigkeit eines inneren Elementes e ist die Summe der Häufigkeiten der Stromelemente, die Nachfahren von e sind. Die Abbildungen sind entnommen aus [Cormode et al., 2008].

Definition 2 (**Hierarchical Heavy Hitters**)**.** Gegeben sei eine Multimenge S der Größe N mit Elementen e aus einer hierarchischen Domäne D der Tiefe h sowie $0 < \phi < 1$. Die Menge der Hierarchical Heavy Hitters ist induktiv definiert:

- HHH_h, die Hierarchical Heavy Hitters aus $Level(h)$, sind die Heavy Hitters von S.
- Für ein Präfix p aus $Level(l), 0 \leq l < h$ in der Hierarchie wird F'_p definiert als

  $$F'_p = \sum f(e) : (e \in S) \wedge (e \preceq p) \wedge (e \not\preceq HHH_{l+1}).$$

  Die Menge HHH_l ist definiert als

  $$HHH_l = HHH_{l+1} \cup \{p : (p \in Level(l)) \wedge (F'_p \geq \phi N)\}.$$

- Die Menge der Hierarchical Heavy Hitters HHH ist die Menge HHH_0.

Das *HHH-Problem* besteht in der Berechnung der Hierarchical Heavy Hitters eines Datenstroms. Eine exakte Lösung erfordert im allgemeinen Fall linearen Platz zur Größe des Stroms, daher wird das Problem als Approximationsproblem definiert.

Definition 3 (**HHH-Problem**)**.** Gegeben sei eine Multimenge S der Größe N mit Elementen e aus einer hierarchischen Domäne D der Tiefe h. Die Häufigkeit der Elemente e ist $f(e)$, gegeben sind ferner ein Schwellwert $0 < \phi < 1$ und ein Fehlerparameter $0 < \varepsilon < \phi$. Gesucht ist eine Menge von Präfixen $P \subseteq D$ und Häufigkeitsschranken $f_{min}(p)$ und $f_{max}(p)$ für alle $p \in P$, so dass gilt:

- Genauigkeit: Die wahre Häufigkeit $f^*(p) = \sum_{e \preceq p} f(e)$ wird angenähert durch

  $$f_{min}(p) \leq f^*(p) \leq f_{max}(p)$$

  und für die Schranken gilt $f_{max}(p) - f_{min}(p) \leq \varepsilon N$.
- Abdeckung: Für alle Präfixe $q \notin P$ gilt $\phi N > \sum f(e) : (e \preceq q) \wedge (e \not\preceq P)$.

[Cormode et al., 2008] beschreiben einen Algorithmus, der das HHH-Problem löst. Die nachfolgend beschriebenen Algorithmen für mehrere Dimensionen lösen auch das eindimensionale HHH-Problem.

3.1.2 Mehrere Dimensionen

Bisher stammten die Elemente des Datenstroms aus einer Hierarchie. In vielen Datenströmen bestehen die Elemente des Datenstroms aus d-Tupeln, deren Komponenten aus Hierarchien stammen. Ein Beispiel ist ein Datenstrom, dessen Elemente Paare aus den IP-Adressen von Absender und Empfänger von Datenpaketen sind. Für diesen Datenstrom könnte man das HHH-Problem einmal für die Absender- und einmal für die Empfängeradressen lösen. Ausgabe wären eine Menge der Netze, die viele Pakete versenden und eine Menge der Netze, die viele Pakete erhalten. Mehrdimensionale Hierarchical Heavy Hitters behandeln die Komponenten der Tupel gemeinsam. Ausgabe ist eine Menge von Netzpaaren, die häufig kommunizieren.

Im d-dimensionalen Fall hat ein Element bis zu d Eltern, daher muss die Notation erweitert werden.

Die Hierarchietiefe in der i-ten Dimension wird mit h_i bezeichnet. Der Elter $par(e, i)$ eines Elementes e ist das Element, das man erhält, wenn man die i-te Komponente von e in deren Hierarchie um eine Ebene nach oben schiebt. Ein Element p ist *generalisierbar* zu q, kurz $p \prec q$ wenn es eine Folge von Elementen $r_1, \ldots, r_n$ mit $n > 1$ gibt, so dass $p = r_1$, $q = r_n$ und es für alle i mit $1 \leq i < n$ ein j zwischen 1 und d gibt, so dass gilt: $r_{i+1} = par(r_i, j)$. Es ist dann q eine (strikte) Generalisierung von p.

Wie zuvor ist $p \preceq q \Leftrightarrow (p \prec q) \vee (p = q)$ und für eine Menge P wird wieder definiert $q \preceq P \Leftrightarrow (\exists p \in P : q \preceq p)$.

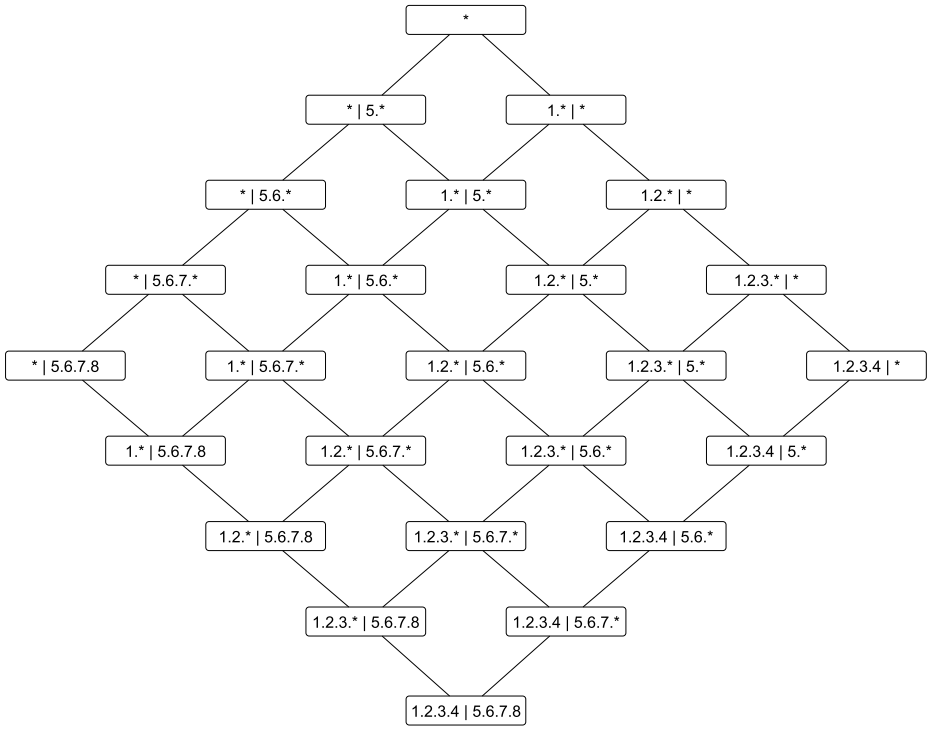


Abbildung 3.5: Verband, der durch Element (1.2.3.4, 5.6.7.8) induziert wird

Die Menge aller Stromelemente und ihrer Generalisierungen bildet mit der Relation $\preceq$ eine Halbordnung. Jedes Paar von Elementen besitzt ein Supremum.[2]

Alle Paare von Generalisierungen *desselben* Stromelementes besitzen darüber hinaus ein Infimum, daher bilden die Generalisierungen *eines* Stromelementes mit der Relation $\preceq$ einen Verband. Abbildung 3.5 (entnommen aus [Cormode et al., 2008]) zeigt den Verband, der durch das Element $(1.2.3.4, 5.6.7.8)$ induziert wird. Die Verbände, die durch verschiedene Stromelemente induziert werden, sind isomorph ([Hershberger et al., 2005]), daher wird auch von *dem* Verband gesprochen.

Die Größe des Verbands H ergibt sich aus den Hierarchietiefen h_i der einzelnen Dimensionen als $H = \prod_{i=1}^{d}(h_i + 1)$. Um einzelne Punkte im Verband beschreiben zu können, wird ein d-dimensionaler Vektor verwendet. Die i-te Komponente gibt an, wie oft ein Element noch in der i-ten Dimension generalisiert werden kann. Der Vektor wird als Label bezeichnet. Das Element $(1.2.3.*, 5.6.7.8)$ hat also das Label $[3, 4]$. *Level*(j), die j-te Ebene im Verband, enthält die Label, deren Komponenten die Summe j haben. *Level*(j) wird auch verwendet, um die Menge der Elemente mit einem Label in *Level(j)* zu beschreiben. Die oberste Ebene bildet *Level*(0), die unterste *Level*(L) mit $L = \sum_{i=1}^{d} h_i$, also gibt es $L + 1$ Ebenen. Die Funktion `GeneralizeTo`(e,*label*) generalisiert Element e auf Label *label*.

[2] In jeder Dimension wird das längste gemeinsame Präfix gewählt. Falls alle längsten gemeinsamen Präfixe die Länge 0 haben, ist das Supremum das Element $*$.

Beispiel: In Abbildung 3.5 ist $d = 2$, $h_1 = 4$, $h_2 = 4$, $H = 25$ und $L = 9$. Es ist $par((1.*, 5.*), 1) = (*, 5.*)$ und $par((1.*, 5.*), 2) = (1.*, *)$ Das Element * ist das einzige Element mit Label $[0, 0]$, Element (*, 5.6.*) hat Label $[0, 2]$ und (1.*, 5.*) hat Label $[1, 1]$. $Level(2) = \{[2, 0], [1, 1], [0, 2]\}$. `GeneralizeTo`$((1.2.3.4, 5.6.7.8),[1, 3]) = (1.*, 5.6.7.*)$.

Es ist im mehrdimensionalen Fall nicht selbstverständlich, wie die Häufigkeit eines Elementes aus der Häufigkeit der Kinder berechnet werden sollte. Im eindimensionalen Fall wird die volle Häufigkeit eines Kindes zum (einzigen) Elter addiert. Im d-dimensionalen Fall gibt es bis zu d Eltern. Wird die volle Häufigkeit des Kindes zu allen Eltern addiert, so spricht man von der Overlap-Regel. Wird die Häufigkeit auf die Eltern verteilt, spricht man von der Split-Regel. Im Folgenden wird nur die Overlap-Regel behandelt, die einfachere Split-Regel wird in [Cormode et al., 2004] beschrieben.

Definition 4 (Hierarchical Heavy Hitters mit Overlap-Regel). Gegeben sei eine Multimenge S der Größe N mit Elementen e mit den Häufigkeiten $f(e)$. Die Elemente sind d-Tupel, deren Komponenten aus Hierarchien der Tiefe h_i, $1 \leq i \leq d$ stammen. Es sei $L = \sum_{i=1}^{d} h_i$ und $0 < \phi < 1$. Die Hierarchical Heavy Hitters sind induktiv definiert:

- HHH_L, die Hierarchical Heavy Hitters aus $Level(L)$, sind die Heavy Hitters von S: $HHH_L = \{e : (e \in S) \wedge (f(e) \geq \phi N)\}$.
- Für ein Element p aus $Level(l)$, $l < L$ wird die angepasste Häufigkeit $f_l(p)$ definiert als
$$f_l(p) = \sum f(e) : (e \in S) \wedge (e \preceq p) \wedge (e \not\preceq HHH_{l+1}).$$
Die Menge HHH_l ist definiert als
$$HHH_l = HHH_{l+1} \cup \{p : (p \in Level(l)) \wedge (f_l(p) \geq \phi N)\}.$$
- Die Menge HHH der Hierarchical Heavy Hitters mit Overlap-Regel ist die Menge HHH_0.

[Cormode et al., 2004] geben einen Algorithmus zur exakten Berechnung der Hierarchical Heavy Hitters mit Overlap-Regel an. Wie schon im eindimensionalen Fall erfordert eine exakte Lösung im allgemeinen Fall linearen Platz zur Größe des Stroms, daher wird das *HHH-Problem mit Overlap-Regel* für Datenströme als Approximationsproblem definiert.

Definition 5 (HHH-Problem mit Overlap-Regel). Gegeben sei eine Multimenge S der Größe N mit Elementen e mit den Häufigkeiten $f(e)$ sowie ein Schwellwert $0 < \phi < 1$ und ein Fehlerparameter $0 < \varepsilon < \phi$. Die Elemente sind d-Tupel, deren Komponenten aus Hierarchien der Tiefe h_i, $1 \leq i \leq d$ stammen. Gesucht ist eine Menge P von Elementen der Halbordnung und Häufigkeitsschranken $f_{min}(p)$ und $f_{max}(p)$ für alle $p \in P$, so dass gilt:

- Genauigkeit: Die wahre Häufigkeit $f^*(p) = \sum_{e \preceq p} f(e)$ wird angenähert durch
$$f_{min}(p) \leq f^*(p) \leq f_{max}(p)$$
und für die Schranken gilt $f_{max}(p) - f_{min}(p) \leq \varepsilon N$.
- Abdeckung: Für alle Präfixe $q \notin P$ gilt $\phi N > \sum f(e) : (e \preceq q) \wedge (e \not\preceq P)$.

3.2 Algorithmen

3.2.1 Überlick

Die Algorithmen verwalten eine Datenstruktur T, die aus Tupeln besteht. Jedes Tupel t_e enthält ein Element e der Halbordnung und einige Hilfsvariablen. Mit den Hilfsvariablen kann die Häufigkeit von e geschätzt und der Fehler der Schätzung beschränkt werden.

Der Strom ist in Fenster der Breite $w = \lceil \frac{1}{\varepsilon} \rceil$ eingeteilt. Während seiner Verarbeitung werden die Stromelemente e in die Datenstruktur eingefügt. Ist Tupel t_e noch nicht in der Datenstruktur vorhanden, wird ein neues Tupel erzeugt. Andernfalls wird der Häufigkeitszähler des Tupels erhöht. Um Speicher zu sparen, werden nach der Verarbeitung von w Stromelementen (am Ende eines Fensters) die Zähler der Tupel der Datenstruktur geprüft. Tupel mit kleinem Zähler werden gelöscht. Die Zähler gelöschter Tupel werden in geeigneter Weise an einige der Vorfahren propagiert. Sind die Tupel der Vorfahren nicht Bestandteil der Datenstruktur, so werden sie eingefügt.

Auf diese Weise wird der Datenstrom elementweise verarbeitet. Das Einfügen erfolgt in der `Insert`-Methode und das Löschen in der `Compress`-Methode.

Nach der Verarbeitung des Stroms werden die Hierarchical Heavy Hitters in der `Output`-Methode ermittelt. Eine *diskontierte Häufigkeit* der Elemente wird geschätzt. Diese soll die Häufigkeit von Elementen angeben nach Abzug der Häufigkeit von Stromelementen, die Nachfahren von Elementen sind, die als HHH ausgeben wurden. Die diskontierte Häufigkeit eines Elementes wird über die geschätzte absolute Häufigkeit von Elementen und über die Fehlerschranken geschätzt. Übersteigt die Schätzung einen bestimmten Wert, wird das Element als Hierarchical Heavy Hitter ausgegeben.

Hinweis: Die diskontierte Häufigkeit entspricht nicht der zuvor verwendeten angepassten Häufigkeit. Bei der Berechnung der angepassten Häufigkeit wird von der absoluten Häufigkeit die Häufigkeit von Stromelementen abgezogen, die Nachfahren von Hierarchical Heavy Hitters sind. Bei der diskontierten Häufigkeit wird die Häufigkeit von Stromelementen abgezogen, die Nachfahren von Elementen sind, *die vom Approximationsalgorithmus als Hierarchical Heavy Hitters ausgegeben wurden.*

3.2.2 Hilfsvariable

Der Datenstrom ist in Fenster der Breite $w = \lceil \frac{1}{\varepsilon} \rceil$ eingeteilt. Für die aktuelle Fensternummer gilt $b_{current} = \lceil \varepsilon N \rceil$.

Die Datenstruktur T enthält Tupel t_e, die neben dem Element e die Hilfsvariablen g_e, Δ_e und m_e enthalten. Es gilt $\Delta_e \geq 0$ und $m_e \geq 0$.

Die Zähler g_e werden so verwaltet, dass

$$f_p = \sum_{e \preceq p} g_e$$

eine gute Schätzung der absoluten Häufigkeit von p ist. Der Wert Δ_p soll den Fehler dieser Schätzung beschränken:

$$f_{min}(p) = f_p - \Delta_p$$

und

$$f_{max}(p) = f_p + \Delta_p,$$

sofern p in der Datenstruktur enthalten ist. Der Wert m_e wird im nächsten Abschnitt beschrieben.

3.2.3 Fehlerschranken und die Hilfsvariable m_e

Ein Tupel t_e wird von der `Compress`-Methode nur dann aus der Datenstruktur gelöscht, wenn $|g_e| + \Delta_e \leq b_{current}$. Wird das Element e später neu eingefügt, ist die Information über den alten Zähler g_e verloren gegangen. Außerdem ist $b_{current}$ um mindestens 1 gestiegen, denn die `Compress`-Methode wird am Ende des Fensters ausgeführt. Der mögliche Fehler durch verlorene Zähler soll durch die Fehlerschranke Δ_e begrenzt werden. Der Fehler ist durch $b_{current} - 1$ begrenzt, weil Tupel nur gelöscht werden können, wenn $|g_e| + \Delta_e \leq b_{current}$ und $b_{current}$ seit der Löschung um mindestens 1 gestiegen ist.

Beim Einfügen eines Elementes kann als Fehlerschranke Δ_e daher $b_{current} - 1$ verwendet werden. Genauere Werte lassen sich mit der Hilfsvariablen m_e bestimmen. Die Verwaltung von m_e stellt sicher, dass $m_e = \max(|g_{d(e)}| + \Delta_{d(e)})$ über alle Nachfahren $d(e)$ von e, deren Zähler an e propagiert wurden. Wird ein solcher Nachfahr $d(e)$ neu eingefügt, kann seine Fehlerschranke $\Delta_{d(e)}$ und sein Wert $m_{d(e)}$ auf das Minimum der m_e-Werte der Vorfahren gesetzt werden. Gibt es keine Vorfahren, wird $b_{current} - 1$ gewählt.

Elemente werden nur gelöscht, wenn $|g_e| + \Delta_e \leq b_{current}$, also ist $m_e \leq b_{current} - 1$ und für die Fehlerschranke Δ_e gilt $\Delta_e \leq b_{current} - 1 \leq \varepsilon N$.

Der m_p-Wert kann auch für Elemente p berechnet werden, die nicht in der Datenstruktur enthalten sind. Dazu wird er wie zuvor auf das Minimum der m_e-Werte der Vorfahren gesetzt. Damit können $f_{min}(p)$ und $f_{max}(p)$ auch dann berechnet werden, wenn p nicht in der Datenstruktur enthalten ist:

$$f_{min}(p) = f_p - m_p$$

und

$$f_{max}(p) = f_p + m_p.$$

3.2.4 Full Ancestry und Partial Ancestry

Es gibt zwei Arten der Verwaltung der Datenstruktur. Beim Full Ancestry Algorithmus werden beim Einfügen eines Elementes e alle Vorfahren des Elementes mit Zähler null in die Datenstruktur eingefügt, sofern sie nicht schon enthalten sind. Elemente können nur dann gelöscht werden, wenn sie keine Nachfahren haben. Beides führt dazu, dass zu

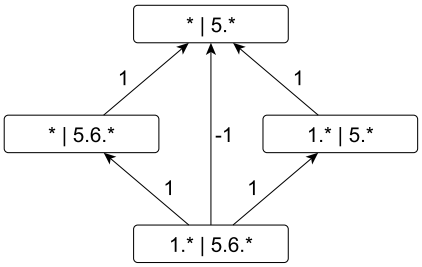


Abbildung 3.6: Inklusions–Exklusionsprinzip

jedem Zeitpunkt für jedes in der Datenstruktur vorhandene Element auch alle Vorfahren vorhanden sind.

Um Speicherplatz zu sparen, wählt der Partial Ancestry Algorithmus eine andere Strategie. Er fügt lediglich das einzelne Element e ein, nicht jedoch die Vorfahren. Elemente mit Nachfahren können gelöscht werden.

3.2.5 Inklusions–Exklusionsprinzip

Nach der Overlap-Regel muss im zweidimensionalen Fall der Zähler eines gelöschten Elementes e an beide Eltern übergeben werden. Ohne Korrektur würde dies zu Zählfehlern beim gemeinsamen Großelter führen. Dieser würde den Zähler von e doppelt erhalten, einmal über jeden Elter von e. Daher wird der Zähler eines gelöschten Elementes e nicht nur zu den Eltern addiert, sondern zusätzlich vom gemeinsamen Großelter abgezogen. Diese Vorgehensweise wird als Inklusions–Exklusionsprinzip bezeichnet.

Abbildung 3.6 stellt das Inklusions–Exklusionsprinzip dar. Der Zähler des Elementes $e = (1.*, 5.6.*)$ wird zu beiden Eltern addiert. Daher wird er indirekt doppelt dem gemeinsamen Großelter zugerechnet. Ohne Korrektur würde dessen Häufigkeit überschätzt. Um dies zu vermeiden, wird der Zähler von e vom Zähler des gemeinsamen Großelters abgezogen. Der Zähler des Großelters kann auf diese Weise negativ werden.

Für die Verallgemeinerung auf den d-dimensionalen Fall sei a der Vorfahr des zu löschenden Elementes e, der durch einmaliges Generalisieren in jeder Dimension erreicht wird. Das Inklusions–Exklusionsprinzip betrifft alle Elemente p mit $e \preceq p \preceq a$. Dieser Teilverband enthält 2^d Elemente und bildet einen d-dimensionalen Hyperwürfel.[3] Die Elemente p können dargestellt werden durch einen Bitstring der Länge d, der an Position i genau dann eine Eins enthält, wenn das Element e in der i-ten Dimension generalisiert wurde. Der Zähler von e wird zu den Eltern addiert, von den Großeltern abgezogen und zu den Urgroßeltern addiert. Der Vorgang wird mit alternierendem Vorzeichen fortgesetzt, bis a erreicht ist.

[3] Hier wird vereinfachend angenommen, dass e noch in allen Dimensionen generalisierbar ist.

Algorithmus 1 : Mehrdimensionaler Algorithmus, Partial Ancestry

```
Insert(element e, count c):
1  if t_e exists in T then
2      g_e = g_e + c;
3  else
4      create t_e with (g_e = c, Δ_e = m_e = b_current − 1);
5      for p in ancestors of e in T do
6          Δ_e = m_e = min(m_e, m_p);

Compress():
1  for l = L downto 0 do
2      foreach node t_e at level l do
3          if |g_e| + Δ_e ≤ b_current then
4              for j = 1 to 2^d − 1 do
5                  p = e; parcount = 0;
6                  for i = 1 to d do
7                      if bit(i, j) = 1 then
8                          p = par(p, i);
9                          parcount = parcount + 1;
10                 if p in domain then
11                     factor = 2 · bit(1, parcount) − 1;
12                     Insert(p, g_e · factor);
13                     if parcount == 1 then
14                         m_p = max(m_p, m_e, |g_e| + Δ_e);
15             delete(t_e);

Output(threshold φ):
1  F_e = f_e = 0 for all e;
2  for l = L downto 0 do
3      forall label ∈ Level(l) do
4          forall e ∈ D, Level(e) ≥ l do
5              p = GeneralizeTo(e,label);
6              f_p = f_p + g_e;
7              if ∄h ∈ P : (e ⪯ h) ∧ (h ⪯ p) then
8                  F_p = F_p + g_e;
9          forall h ∈ P, Level(h) ≥ l do
10             p = GeneralizeTo(h,label);
11             if ∄q ∈ P : (h ≺ q) ∧ (q ⪯ p) then
12                 F_p = F_p + Δ_h;
13         forall h, h' ∈ P, Level(h) ≥ l, Level(h') ≥ l do
14             p = GeneralizeTo(glb(h, h'), label);
15             if (h ⪯ p) ∧ (h' ⪯ p) ∧ (∄q ∈ P : ((h ≺ q) ∨ (h' ≺ q)) ∧ (q ⪯ p)) then
16                 F_p = F_p + Δ_glb(h,h');
17     forall p ∈ Level(l) with f_p > 0 do
18         if F_p + Δ_p ≥ φN then
19             P = P ∪ {p};
20             print(p, f_p − Δ_p, f_p + Δ_p);
```

3.2.6 Verarbeitung des Datenstroms

Algorithmus 1 zeigt den mehrdimensionalen Partial Ancestry Algorithmus. Der Full Ancestry Algorithmus unterscheidet sich von diesem nur in zwei Punkten: In Zeile 4 der `Insert`-Methode fügt Full Ancestry alle Eltern des Elementes mit Zähler 0 ein. In der `Compress`-Methode werden in Zeile 3 nur solche Elemente zur Löschung freigegeben, die keine Nachfahren haben. Die übrigen Verarbeitungsschritte sind für beide Algorithmen gleich.

Die `Insert`-Methode wird für jedes Element des Datenstroms aufgerufen. Ist das Element bereits in der Datenstruktur enthalten, wird sein Zähler aktualisiert (Zeile 2). Andernfalls wird es neu eingefügt (Zeilen 4–6). Die Werte der Hilfsvariablen Δ_e und m_e werden wie in Abschnitt 3.2.3 beschrieben gesetzt.

In der `Compress`-Methode werden die Tupel der Datenstruktur ebenenweise aufsteigend untersucht (Zeile 1). Die Reihenfolge der Bearbeitung der Elemente innerhalb einer Ebene ist ohne Bedeutung (Zeile 2). Wenn die Prüfung eines Elementes e in Zeile 3 ergibt, dass das Element aus der Datenstruktur gelöscht werden soll, wird sein Zähler in den Zeilen 4–14 an die Elemente des Hyperwürfels propagiert. Dazu wird in Zeile 4 eine Variable j eingeführt. Die unteren d Bits von j bilden den Bitstring, der beschreibt, wie die Elemente p des Hyperwürfels durch Generalisierung in den mit eins besetzten Dimensionen aus e erzeugt werden können. Die Elemente p werden erzeugt, indem zunächst ein Element p als Kopie von Element e angefertigt wird (Zeile 5). Anschließend werden in den Zeilen 6–7 die Bits von j gelesen und in Zeile 8 die entsprechenden Generalisierungen vorgenommen. Die Funktion $\text{bit}(i, j)$ berechnet das i-te Bit von j. Die Variable *parcount* zählt die Generalisierungen und wird in Zeile 11 verwendet, um das Vorzeichen zu bestimmen, mit dem in Zeile 12 der Zähler von e an p propagiert wird. Die Variable *parcount* wird außerdem verwendet, um in Zeile 13 festzustellen, ob p Elter von e ist. In diesem Fall wird die Hilfsvariable m_p aktualisiert, so dass weiter gilt: $m_p = \max(|g_{d(p)}| + \Delta_{d(p)})$ über alle Nachfahren $d(p)$ von p, deren Zähler an p propagiert wurden.

Nach Verarbeitung des gesamten Hyperwürfels wird das Element e in Zeile 15 gelöscht.

3.2.7 Häufigkeitsschranken

Die Häufigkeitsschranken sind für die Genauigkeit in Definition 5 von Bedeutung. Darüber hinaus wird die geschätzte (absolute) Häufigkeit zur Schätzung der diskontierten Häufigkeit verwendet.

Lemma 1. f_{min} und f_{max} sind gültige untere und obere Schranken der Häufigkeit aller Elemente.

Beweis. Siehe [Cormode et al., 2008].

3.2.8 Ausgabe der Hierarchical Heavy Hitters

Nach der Verarbeitung des Stroms mit der `Insert`- und der `Compress`-Methode enthält die Datenstruktur T die Informationen, mit denen die Hierarchical Heavy Hitters berechnet werden können. Die Berechnung erfolgt in der `Output`-Methode und kann beliebig oft mit verschiedenen Schwellwerten ϕ wiederholt werden. Da die Berechnung nicht während der Verarbeitung des Stroms erfolgt, ist sie nicht zeitkritisch.

Wegen der Abdeckbedingung aus Definition 5 müssen alle Elemente p ausgegeben werden, deren diskontierte Häufigkeit mindestens ϕN ist. Die diskontierte Häufigkeit ergibt sich aus der absoluten Häufigkeit durch Subtraktion der Häufigkeit von Elementen, deren Vorfahren bereits als HHH ausgegeben wurden. Da die absoluten Häufigkeiten nicht exakt bekannt sind, wird auch die diskontierte Häufigkeit nur geschätzt. Falschnegative Treffer sind nach Definition unzulässig. Die Schätzung muss daher konservativ sein und darf die wahre diskontierte Häufigkeit nicht unterschätzen. Man ist an einer oberen Schranke der diskontierten Häufigkeit interessiert und gibt nur solche Elemente *nicht* aus, deren obere Schranke unter ϕN liegt.

In der `Output`-Methode wird diese obere Schranke berechnet. Ausgangspunkt ist die obere Schranke der absoluten Häufigkeit $f_{max}(p)$. Diese überschätzt die diskontierte Häufigkeit, weil sie die Häufigkeit der Nachfahren von Elementen enthält, die als HHH ausgegeben wurden. Dieser (erste) Fehler soll korrigiert werden. Sei P die Menge der bereits ausgegebenen HHH, dann ist für beliebiges p die Menge H_p definiert als

$$H_p = \{h \,|\, h \in P \wedge (h \prec p) \wedge (\nexists h' \in P : h \prec h' \prec p)\}.$$

Als obere Schranke der diskontierten Häufigkeit könnte man den Wert

$$f_{max}(p) - \sum_{h \in H_p} f_{min}(h)$$

berechnen. Es wird jeweils nur die untere Schranke $f_{min}(h)$ der absoluten Häufigkeit abgezogen, weil man an einer oberen Schranke der diskontierten Häufigkeit interessiert ist. Der Wert kann die diskontierte Häufigkeit jedoch unterschätzen, weil die Häufigkeit von Elementen, die Nachfahren mehrerer Elemente aus H_p sind, mehrfach abgezogen werden.

Abbildung 3.7 zeigt ein Element c, das Nachfahr der Elemente a und b ist, die als Hierarchical Heavy Hitters ausgegeben wurden und Elemente von H_p sind. Werden für die Berechnung der diskontierten Häufigkeit von p die Häufigkeiten von a und b subtrahiert, wird die Häufigkeit von c doppelt abgezogen. Die Häufigkeit von Elementen, die Nachfahr von zwei Elementen aus H_p sind, wird also doppelt abgezogen und muss daher wieder zur diskontierten Häufigkeit addiert werden.

Für die Korrektur dieses zweiten Fehlers sind zwei weitere Begriffe erforderlich: Das Infimum oder die größte untere Schranke $q = glb(h, h')$ ist das eindeutige Element, für das gilt:

$$\forall p : (q \preceq p) \wedge (p \preceq h) \wedge (p \preceq h') \Rightarrow p = q.$$

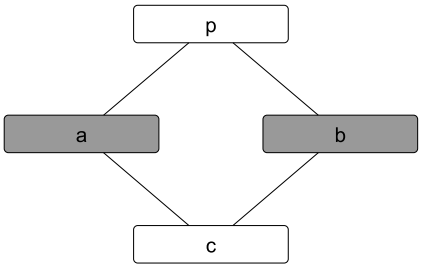


Abbildung 3.7: Die Elemente a und b wurden als Hierarchical Heavy Hitters ausgegeben und liegen in H_p. Beide enthalten die Häufigkeit von c. Wenn bei der Berechnung der diskontierten Häufigkeit von p ihre Häufigkeit abgezogen wird, wird die Häufigkeit von c doppelt berücksichtigt. Daher muss die Häufigkeit von c einmal zur diskontierten Häufigkeit von p addiert werden.

Die dominierende Menge von Element q bezüglich H_p ist $Dom(q, H_p) = \{h \in H_p | q \prec h\}$. In der Formel $f_{max}(p) - \sum_{h \in H_p} f_{min}(h)$ wird die Häufigkeit von q einmal für jedes Element seiner dominierenden Menge abgezogen, obwohl sie insgesamt nur einmal abgezogen werden sollte. Dieser Fehler lässt sich korrigieren, indem die Häufigkeit für solche Elemente q in angemessenem Umfang wieder addiert wird. Da eine obere Schranke der diskontierten Häufigkeit gefunden werden soll, wird die obere Schranke der Häufigkeit von q addiert:

$$f_{max}(p) - \sum_{h \in H_p} f_{min}(h) + \sum_{q = glb(h \in H_p, h' \in H_p)} f_{max}(q)(|Dom(q, H_p)| - 1).$$

Durch Umformung lässt sich der Wert effizient abschätzen. Dabei wird ausgenutzt, dass die geschätzte Häufigkeit von Element p berechnet wird als $f_p = \sum_{e \preceq p} g_e$. Subtrahiert man von f_p die untere Schranke $f_{min}(q)$ der Häufigkeit eines Nachfahren q, so ist das Ergebnis

$$f_p - f_{min}(q) = \Delta_q + \sum_{(e \preceq p) \wedge (e \not\preceq q)} g_e.$$

Es wird also nicht addiert und dann subtrahiert, sondern nur der relevante Teil der Zähler g_e addiert.

Eine konservative Schätzung F_p der diskontierten Häufigkeit von p kann daher durch Addition von drei Termen berechnet werden. Der erste Term sind die Zähler der relevanten Nachfahren von p. Der zweite Term sind die Δ-Werte der Elemente h aus H_p. Diese sind das Ergebnis der Subtraktion der unteren Schranken $f_{min}(h)$ der Häufigkeiten zur Korrektur des ersten Fehlers. Der dritte Term sind die Δ-Werte der *glb*-Elemente von Elementpaaren aus H_p. Sie sind das Ergebnis der Addition der oberen Schranken der absoluten Häufigkeit von *glb*-Elementen zur Korrektur des zweiten Fehlers.

Die geschätzte diskontierte Häufigkeit wird also als Summe von drei Termen berechnet:[4]

$$F_p = \sum_{(e \preceq p) \wedge (\forall h \in H_p : e \not\preceq h)} g_e \; + \sum_{h \in H_p} \Delta_h \; + \sum_{(h,h' \in H_p) \wedge (q = glb(h,h'))} \Delta_q .$$

Die obere Schranke der diskontierten Häufigkeit von p ist $F_p + \Delta_p$.

Nach diesen Vorbemerkungen lässt sich die Berechnung der Hierarchical Heavy Hitters wie folgt beschreiben:

In der ersten Zeile werden die Schätzungen der absoluten und der diskontierten Häufigkeit f_e und F_e für alle Elemente auf null gesetzt. Anschließend werden die Label ebenenweise aufsteigend betrachtet. Für jedes Element e wird mit der Funktion `GeneralizeTo`(e, *label*) der Vorfahr p mit dem aktuellen Label *label* ermittelt und dessen geschätzte Häufigkeit f_p um g_e erhöht (Zeile 6). Dies stellt sicher, dass nach Bearbeitung des Labels gilt: $f_p = \sum_{e \preceq p} g_e$. Die Schätzung der diskontierten Häufigkeit ist aufwändiger: In den Zeilen 7–8 wird der Beitrag von e zum ersten Term von F_p berechnet. In den Zeilen 9–12 wird für die Elemente $h \in H_p$ der Vorfahr p mit dem aktuellen Label bestimmt (Zeilen 10–11) und der Beitrag von h zum zweiten Term von F_p verbucht (Zeile 12).

In den Zeilen 13–16 wird der dritte Term behandelt. In den Zeilen 13–14 wird der Vorfahr mit aktuellem Label von *glb*-Elementen von Paaren aus P bestimmt. In Zeile 15 wird geprüft, ob das Paar in H_p liegt. In Zeile 16 wird bei Bedarf der Beitrag zum dritten Term verbucht.

Nach Bearbeitung aller Label einer Ebene werden die Elemente dieser Ebene untersucht. Elemente mit $F_p + \Delta_p \geq \phi N$ werden in die Menge P eingefügt. Da $F_p + \Delta_p$ eine obere Schranke der diskontierten Häufigkeit ist, werden alle erforderlichen Elemente gefunden und die Abdeckung in Definition 5 erfüllt. In Zeile 20 werden neu eingefügte Elemente mit Informationen über ihre Häufigkeitsschranken ausgegeben.

Theorem 1. Algorithmus 1 löst das HHH-Problem mit Overlap-Regel. Er berechnet die Hierarchical Heavy Hitters in mehreren Dimensionen mit Genauigkeit εN. Der Speicherbedarf für den Algorithmus in der Variante Full Ancestry ist $O(\frac{H}{\varepsilon} \log \varepsilon N)$. Das Einfügen eines Elementes benötigt amortisierte Zeit $O(H \log \varepsilon N)$.

Beweis. Siehe [Cormode et al., 2008].[5]

Für Speicherbedarf und Laufzeit des Partial Ancestry Algorithmus lassen sich keine theoretischen Schranken angeben. In Experimenten hat Partial Ancestry einen geringeren Speicherbedarf als Full Ancestry.

[4] Die Formel in [Cormode et al., 2008], S. 29 enthält drei Tippfehler: Die Menge P muss jeweils durch H_p ersetzt werden.

[5] Im Beweis wird nicht erwähnt, dass wegen der Existenz negativer Zähler im mehrdimensionalen Fall nicht gilt: $f_{max}(p) - f_{min}(p) \leq \varepsilon N$, sondern nur: $f_{max}(p) - f_{min}(p) \leq 2 \cdot \varepsilon N$. Wird der Algorithmus mit $\varepsilon' = 0.5 \cdot \varepsilon$ gestartet, gilt die gewünschte Genauigkeit.

4 Intrusion Detection und alternative Repräsentationen

In diesem Kapitel werden einige Arbeiten aus dem Bereich der Intrusion Detection (Angriffs- oder Einbruchserkennung) auf der Grundlage von Systemaufrufen vorgestellt. Es wird beschrieben, dass bei der anomaliebasierten Intrusion Detection ein Modell normalen Verhaltens verwendet wird, anhand dessen abweichendes Verhalten erkannt und als möglicher Angriff eingeordnet wird.

Dieses Modell wird aus Datenströmen von Systemaufrufen erstellt und kann daher als Zusammenfassung derartiger Datenströme aufgefasst werden. Die Vorstellung dieser alternativen Ansätze zur Zusammenfassung erlaubt es, die in Kapitel 5 beschriebene Vorgehensweise zur Zusammenfassung von Strömen von Systemaufrufen einzuordnen.

4.1 Missbrauchs- und anomaliebasierte Intrusion Detection

Die Intrusion Detection Subgroup des National Security Telecommunications Advisory Committee gibt folgende Definition: „Eine Intrusion ist der unberechtigte Zugriff auf und/oder die unberechtigte Aktivität in einem Informationssystem.“ [Intrusion Detection Subgroup, 1997]

Aufgabe eines Intrusion Detection Systems (IDS) ist das automatische Erkennen solcher Einbrüche. [Zanero, 2006] vergleicht Systeme zur Intrusion Detection mit Alarmanlagen.

Systeme zur Intrusion Detection lassen sich in zwei Gruppen einteilen (vgl. [Kosoresow und Hofmeyer, 1997], [Varghese und Jacob, 2007b]): Bei der missbrauchsbasierten Intrusion Detection, die hier nicht näher betrachtet werden soll, wird eine Wissensbasis (eine Menge von so genannten Signaturen) verwaltet, die Angriffe beschreibt. Das System erkennt Angriffe anhand dieser Wissensbasis. Bei der anomaliebasierten Einbruchserkennung wird dagegen ein Modell normalen Verhaltens verwendet, mit dem abweichendes Verhalten erkannt und als möglicher Angriff eingeordnet werden kann.

Sowohl das Verhalten von Benutzern als auch das Verhalten einzelner Anwendungen können modelliert werden. Abweichendes Verhalten eines Benutzers kann darauf hinweisen, dass sich jemand unter einem fremden Benutzernamen Zugriff auf das System verschafft hat. Abweichendes Verhalten einer Anwendung kann ein Hinweis auf eine Manipulation der Anwendung sein. Ein Beispiel für eine solche Manipulation ist das Ausnutzen von *Pufferüberläufen*, das darin besteht, eine Anwendung mit Sicherheitslücken zu ver-

anlassen, große Datenmengen in einen dafür zu kleinen Speicherbereich zu schreiben, nachfolgende Speicherbereiche zu überschreiben und den dort eingefügten Code auszuführen.

Anomaliebasierte Intrusion Detection kann das Verhalten von Benutzern und Anwendungen auf verschiedenen Ebenen betrachten. [Imsand und Hamilton, 2007] erfassen als Verhalten von Benutzern die Art, in der sie die grafische Benutzerschnittstelle (Menüs, Symbole, Tastenkombinationen) verwenden. [Garg et al., 2006] betrachten ebenfalls das Verhalten von Benutzern an der grafischen Benutzerschnittstelle, erfassen jedoch Mausbewegungen und Mausklicks. [Schonlau et al., 2001] untersuchen dagegen Befehle, die von Benutzern über die Kommandozeile eingegeben werden.

In den folgenden Abschnitten wird der Fall betrachtet, in dem das Verhalten einer Anwendung in Form eines Datenstroms von Systemaufrufen erfasst wird. Die grundsätzliche Vorgehensweise ist jeweils gleich: Es liegt eine Menge von Datenströmen von Systemaufrufen vor, die von der Anwendung unter normalen Bedingungen erzeugt wurden. Diese Datenströme werden zu einem Modell normalen Verhaltens zusammengefasst. Wird die Anwendung erneut ausgeführt, soll anhand des Datenstroms ihrer Systemaufrufe und des Modells normalen Verhaltens entschieden werden, ob ein Angriff vorliegt.

4.2 Sequenzenbasierte Repräsentationen

In den meisten Arbeiten zur Intrusion Detection anhand von Systemaufrufen werden kurze Sequenzen von Systemaufrufen verwendet, um normales Verhalten zu beschreiben ([Kang et al., 2005], [Liao und Vemuri, 2002]). Diesem Ansatz liegt die Annahme zugrunde, dass die Ausführungsreihenfolge durch die Anwendung vorgegeben ist und Abweichungen auf einen Angriff hinweisen.

Sequenzen fester Länge

[Warrender et al., 1999] beschreiben ein Verfahren zur Anomalieerkennung für privilegierte Unix-Prozesse. Sie wählen einen sehr einfachen Ansatz für die Modellierung normalen Verhaltens. Es werden nur die Namen der Aufrufe verwendet (z. B. `open` oder `read`); die Informationen über Parameter und Rückgabewerte werden ignoriert. Jeder Datenstrom wird in kurze Sequenzen konstanter Länge k (z. B. $k = 6$) eingeteilt. Dazu wird ein Fenster der Länge k über den Strom geschoben und alle Sequenzen, die mindestens einmal auftreten, werden in der Datenbasis normalen Verhaltens gespeichert. Die Häufigkeit der Sequenzen wird nicht erfasst. Der Vorgang wird für jeden Datenstrom aus der Menge der normalen Ausführungen wiederholt.

Neue Prozesse werden ebenfalls in Sequenzen der Länge k eingeteilt. Jede Sequenz, die nicht in der Datenbasis normalen Verhaltens enthalten ist, wird als Fehler gewertet. Die Anzahl der Fehler innerhalb der letzten 20 Sequenzen wird als Locality Frame Count (LFC) bezeichnet. Übersteigt der LFC einen Schwellwert, wird der Prozess als Angriff gewertet. Die Verwendung des LFC beruht auf der Beobachtung, dass Anomalien

während eines Angriffs nicht kontinuierlich, sondern zeitlich gehäuft auftreten ([Forrest et al., 2008], [Warrender et al., 1999]).

Dieser Ansatz wurde in verschiedenen Varianten untersucht. [Hofmeyr et al., 1998] werten nicht in der Datenbasis enthaltene Sequenzen nur dann als Fehler, wenn die Hammingdistanz zwischen der Sequenz und der ähnlichsten Sequenz in der Datenbasis einen Schwellwert übertrifft.

Lookahead pairs

[Forrest et al., 1996] wählen einen noch einfacheren Ansatz: Die so genannten *lookahead pairs*. Wieder wird ein Fenster der Länge k über die Ströme geschoben. Alle Paare von Aufrufen x und y, die mindestens einmal im Abstand l, $1 \leq l < k$ in einem Fenster auftreten, werden in der Datenbasis normalen Verhaltens als Eintrag $\langle x, y \rangle_l$ gespeichert.

Für neue Prozesse wird gezählt, wie oft Paare von Aufrufen x und y im Abstand l auftreten, für die es keinen Eintrag $\langle x, y \rangle_l$ in der Datenbasis gibt. Der Anteil dieser Paare an der maximal möglichen Anzahl solcher Paare wird zur Unterscheidung normaler und anomaler Prozesse verwendet.

4.3 Häufigkeitsbasierte Repräsentationen

Bei der Verwendung häufigkeitsbasierter Repräsentationen wird das Verhalten einer Anwendung über die Häufigkeit der von ihr verwendeten Systemaufrufe charakterisiert. Diese kann global für die gesamte Anwendung erfasst werden, auf der Ebene einzelner Prozesse oder auf der Ebene von Teilsequenzen eines Prozesses.

Wortvektoren

Im Bereich des Information Retrieval und in der Textklassifikation ist es üblich, Texte d_j aus einer Menge D von Dokumenten durch Vektoren von Wortgewichten zu repräsentieren ([Joachims, 2001]). Dabei wird vereinfachend angenommen, die Reihenfolge der Wörter eines Textes sei bedeutungslos. Die Texte werden als Mengen von Wörtern w_i (*Bag of Words*) aufgefasst und die Relevanz der einzelnen Wörter wird durch ihr Gewicht dargestellt.

Es sei M die Anzahl der verschiedenen Wörter w_i („Terme“) in D, N die Anzahl der Dokumente und n_i die Anzahl der Dokumente in D, in denen w_i vorkommt. Eine $M \times N$–Matrix $\mathbf{A} = (a_{ij})$ enthält die Wortgewichte; dabei stellt a_{ij} das Gewicht von Wort w_i in Dokument d_j dar. Jede Spalte der Matrix ist ein Vektor von Wortgewichten, der ein Dokument repräsentiert.

Für die Wahl von a_{ij} gibt es verschiedene Möglichkeiten. Im einfachsten Fall ist a_{ij} genau dann 1, wenn Wort w_i in Dokument d_j vorkommt. Wird als Gewicht a_{ij} die Häufigkeit f_{ij}

von Wort w_i in Dokument d_j verwendet, so spricht man von der *Term Frequency* (TF). Oft wird als Gewicht die *Term Frequency – Inverse Document Frequency* (TF–IDF) gewählt, bei welcher der Vektor der Term Frequency auf Länge 1 normiert wird und das Gewicht die globale Häufigkeit des Worts in der Dokumentensammlung D berücksichtigt:

$$a_{ij} = \frac{f_{ij}}{\sqrt{\sum_{l=1}^{M} f_{lj}^2}} \cdot \log \frac{N}{n_i}.$$

Obige Darstellung wurde [Liao und Vemuri, 2002] entnommen. [Salton und Buckley, 1988] stellen weitere Möglichkeiten für die Wahl der Gewichte dar.

Die Ähnlichkeit von Dokumenten wird über die Ähnlichkeit ihrer Gewichtsvektoren definiert. Die Ähnlichkeit der Vektoren kann über den euklidischen Abstand oder die Kosinus-Ähnlichkeit bestimmt werden. Im Fall der Kosinus-Ähnlichkeit gilt für die Dokumente x und d_j

$$sim(x, d_j) = \frac{x \cdot d_j}{||x|| \cdot ||d_j||}.$$

[Liao und Vemuri, 2002] übertragen diese Art der Repräsentation aus dem Bereich der Textdokumente auf den Bereich der Systemaufrufe. Sie repräsentieren Prozesse durch Vektoren von Wortgewichten. Die einzelnen Systemaufrufe werden als Wörter aufgefasst und die Folge der Aufrufe eines Prozesses als Dokument. Die Gewichtung erfolgt mit TF–IDF. Die Datenbasis normalen Verhaltens ist eine Menge von Vektoren von Wortgewichten, die jeweils einen normalen Prozess repräsentieren. Neue Prozesse werden mit den Prozessen der Datenbasis anhand der Kosinus-Ähnlichkeit ihrer Vektordarstellung verglichen. Liegt die durchschnittliche Ähnlichkeit des neuen Prozesses zu seinen k ähnlichsten Prozessen der Datenbasis unterhalb eines Schwellwerts, wird er als Angriff gewertet.

Mengen ähnlicher Prozess–Sequenzen

[Varghese und Jacob, 2007a] bilden ein Modell normalen Verhaltens, indem Informationen über die Häufigkeit (und in einigen Fällen über die Streuung der Häufigkeit) der einzelnen Systemaufrufe gesammelt werden. Diese Informationen werden nicht global erfasst, sondern für Mengen ähnlicher Sequenzen einer Anwendung.

Die Folge der Systemaufrufe eines Prozesses einer Anwendung wird als (Prozess–)Sequenz bezeichnet. Jeder Prozess erzeugt eine Prozess–Sequenz; eine Ausführung einer Anwendung kann eine oder mehrere Prozess–Sequenzen erzeugen. Die Menge aller Prozess–Sequenzen aller Ausführungen einer Anwendung wird in Mengen ähnlicher Prozess–Sequenzen zerlegt. Diese werden als *Sequenzmengen* der Anwendung bezeichnet. Die genaue Vorgehensweise bei der Zerlegung wird nicht beschrieben. Der Einsatz von Clusteringverfahren ist denkbar. Für jede Sequenzmenge werden Informationen über die Häufigkeit der einzelnen Systemaufrufe erfasst. Drei Fälle werden unterschieden: Ist die Streuung der Häufigkeit des Aufrufs in der Sequenzmenge niedrig, werden Häufigkeit und Streuung gespeichert (*Matching Profile*). Ist die Streuung der Häufigkeit des Aufrufs in

der Sequenzmenge hoch, wird die Position des Aufrufs in der Häufigkeitsrangfolge der Aufrufe erfasst (*Frequency Pattern*). Kommt der Aufruf in der Sequenzmenge nicht vor, wird dies gespeichert (*Presence of System Calls*).

Die Anomalie neuer Prozess–Sequenzen wird dadurch ermittelt, dass der Häufigkeitsvektor der Prozess–Sequenz mit den gespeicherten Informationen der Sequenzmengen verglichen wird. Für Aufrufe mit normalerweise niedriger Streuung ist der Anomaliewert davon abhängig, ob die Häufigkeit in der neuen Prozess–Sequenz im Rahmen der üblichen Streuung liegt. Für Aufrufe mit normalerweise hoher Streuung wird der Rang in der Häufigkeitsrangfolge der neuen Prozess–Sequenz mit dem üblichen Rang verglichen. Für Aufrufe, die normalerweise nicht in der Sequenzmenge vorkommen, wird ein hoher Anomaliewert gewählt.

4.4 Parameterbasierte Repräsentationen

Alle bisher beschriebenen Ansätze verwenden ausschließlich die Namen (den Typ) der Systemaufrufe. Auch die weit überwiegende Mehrheit der in der Literatur vorgeschlagenen Ansätze zur Intrusion Detection anhand von Systemaufrufen verwendet ausschließlich die Namen der Systemaufrufe und beachtet die Parameter- und Rückgabewerte der Aufrufe nicht (vgl. [Peisert et al., 2007], [Mutz et al., 2006], [Matuszewski, 2009]). Diese Vorgehensweise ist einerseits verständlich, da die Parameter der Aufrufe sich in Anzahl und Typ unterscheiden, teilweise komplexe Datenstrukturen enthalten und die Erfassung und Verarbeitung mit erheblichem Aufwand verbunden sein können. Andererseits können die Parameterwerte wichtige Informationen über Angriffe enthalten, die bei der ausschließlichen Berücksichtigung der Namen verloren gehen. Die nachfolgend beschriebenen Arbeiten verwenden daher die Parameterwerte der Aufrufe.

Modelle normaler Parameterwerte

[Kruegel et al., 2003] verwenden die Parameterwerte der Systemaufrufe zur Angriffserkennung und beachten die Information über die Reihenfolge der Ausführung nicht. Sie erstellen für jede Anwendung für jeden Systemaufruf ein *Profil*, das die normalerweise auftretenden Werte aller Parameter des Aufrufs in der Anwendung repräsentiert. Normale Werte von einzelnen Parametern werden durch so genannte *Modelle* beschrieben, die für einen bestimmten Wert des Parameters dessen Normalität in Form einer Kennzahl von 0 bis 1 ausgeben. Die Modelle werden mit der Datenbasis normalen Verhaltens erstellt.

Jedes Modell bezieht sich auf einen Parameter eines Systemaufrufs einer Anwendung (z. B. auf den Dateinamen in `open`-Aufrufen der Anwendung top). Ein Parameter desselben Aufrufs kann für verschiedene Anwendungen verschiedene Modelle haben, weil z. B. für den Dateinamen in `open`-Aufrufen der Anwendung top andere Werte normal sein können als für die Anwendung Firefox.

Es werden vier verschiedene Klassen von Modellen eingesetzt, von denen hier zwei skizziert werden sollen:

Das *String Length Model* wird für Parameter verwendet, die Zeichenketten sind; meist handelt es sich um Dateisystempfade. Über die Datenbasis normalen Verhaltens werden arithmetisches Mittel und empirische Varianz der Zeichenkettenlängen bestimmt. Diese von den Autoren als μ und σ^2 bezeichneten Werte stellen eine Schätzung von Erwartungswert $\dot{\mu}$ und Varianz $\dot{\sigma}^2$ der wahren Verteilung der Zeichenkettenlängen dar.

Für neue Zeichenketten wird die Kennzahl 1 („normal“) ausgegeben, wenn die Länge l der Zeichenkette höchstens μ ist, denn kurze Zeichenketten sind unverdächtig. Ansonsten wird als Kennzahl

$$p(l) = \frac{\sigma^2}{(l-\mu)^2}$$

verwendet.[1] Grundlage der Berechnung ist die Tschebyschow-Ungleichung, mit der eine obere Schranke für die Wahrscheinlichkeit berechnet werden kann, dass der Abstand des Werts einer Zufallsvariablen zu ihrem Erwartungswert eine bestimmte Schwelle übersteigt.

Das *Token Finder Model* wird für Parameter verwendet, die nur wenige verschiedene Werte annehmen können (z. B. Verknüpfungen von Flags). Über die Datenbasis normalen Verhaltens werden alle im Normalbetrieb vorkommenden Werte ermittelt. Wenn Anwendungen auf anomales Verhalten untersucht werden, erhalten bekannte Werte die Kennzahl 1, unbekannte die 0.

Für die Beschreibung von zwei weiteren Modellen, die *String Character Distribution* und die *Structural Inference*, sei auf [Kruegel et al., 2003] verwiesen.

Eine Anomaliekennzahl AS für den gesamten Aufruf wird aus den Kennzahlen p_m der M Modelle ermittelt. Der Einfluss von p_m-Werten nahe 0 wird durch 10^{-6} begrenzt:

$$AS = \sum_{m=1}^{M} -\log(\ \max(\ p_m, 10^{-6}\ )\ ).$$

Für jedes Profil wird ein Schwellwert für die Anomaliekennzahl bestimmt, indem die maximale Anomaliekennzahl AS_{max} des Aufrufs in der Anwendung unter normalen Bedingungen mit Hilfe der Datenbasis ermittelt und der Schwellwert auf einen etwas höheren Wert (z. B. $1.1 \cdot AS_{max}$) gesetzt wird. Im Produktionsbetrieb werden Aufrufe, deren Anomaliekennzahl diesen Schwellwert übertrifft, als Angriff gewertet.

Die Anzahl der erfassten Systemaufrufe liegt bei 22 ([Zanero, 2006], S. 101). Die Funktionalität des Systems zur Anomalieerkennung ist in Form der Bibliothek *libAnomaly* und des Systems *SyscallAnomaly* frei verfügbar.[2]

[1] Werte $p(l) > 1$ werden im Quellcode auf 1 gesetzt, im Artikel fehlt diese Angabe.
[2] `http://www.cs.ucsb.edu/~seclab/projects/libanomaly`

Cluster normaler Parameterwerte

[Zanero, 2006] erweitert diesen Ansatz, indem er für jeden Parameter jedes Systemaufrufs jeder Anwendung die Menge der Parameterwerte in Cluster ähnlicher Werte zerlegt und für jeden Cluster ein Modell normaler Werte erstellt. Dieser Ansatz beruht auf der Überlegung, dass es für manche Aufrufe keine einzelne normale Verwendung gibt, sondern möglicherweise eine Menge verschiedener normaler Verwendungen. [Zanero, 2006] nennt den Systemaufruf `open` als Beispiel, dessen Parameterwerte beim Öffnen einer Bibliothek sich sehr von den Parameterwerten beim Öffnen einer Benutzerdatei unterscheiden, obwohl beides normale Verwendungen des Aufrufs sind.

5 Lösungsansatz

In dieser Arbeit wird die Frage behandelt, wie man aus Datenströmen von Systemaufrufen eine kondensierte Darstellung als Menge von Hierarchical Heavy Hitters berechnen kann. Es wird untersucht, welche Hierarchien sich für einen Strom von Systemaufrufen angeben lassen und wie die hierarchischen Informationen aus dem Strom der Aufrufe berechnet werden können. Außerdem werden Ähnlichkeitsmaße für HHH-Mengen entwickelt, welche die hierarchische Struktur der Elemente erfassen. In diesem Kapitel wird die grundlegende Vorgehensweise beschrieben, die Details der Implementierung werden in Kapitel 6 erläutert.

Zunächst wird dargestellt, welche hierarchischen Variablen verwendet werden und wie der Anwendungsdatensatz, der mit strace erzeugt wurde, verarbeitet wird, um daraus Tupel hierarchischer Variablen zu extrahieren. Ein Strom solcher Tupel bildet die Eingabe für die Algorithmen zur Berechnung der Hierarchical Heavy Hitters, Ausgabe sind Mengen von Hierarchical Heavy Hitters. Im zweiten Abschnitt werden Ähnlichkeitsmaße für solche HHH-Mengen beschrieben, welche die hierarchische Struktur berücksichtigen. Im dritten Abschnitt wird das Protokollieren der Systemaufrufe mit dem Werkzeug strace und die Erstellung des Anwendungsdatensatzes beschrieben.

5.1 Extraktion der hierarchischen Variablen

Eingabe für die Algorithmen zur Berechnung der Hierarchical Heavy Hitters ist ein Strom von Elementen. Die einzelnen Elemente sind Tupel von Variablen, deren Werte jeweils einer Hierarchie entstammen. Es ist daher erforderlich, aus den Rohdaten, die strace erzeugt, derartige hierarchische Variablen zu extrahieren.

Ein Beispiel für eine solche Variable sind die absoluten Pfadnamen, mit denen Dateien und Verzeichnisse bezeichnet werden. Diese Pfade bilden eine natürliche Hierarchie und lassen sich als Baum darstellen, Generalisierungen um eine Stufe verkürzen den Pfad jeweils um eine Komponente und ersetzen das Kind durch den eindeutigen Elter. In dieser Hierarchie hat die Datei `/tmp/log` den Elter `/tmp`, der wiederum den Elter `*` (für „beliebig“) hat. In Abschnitt 5.1.1 wird die Konstruktion dieser hierarchischen Variablen ausführlicher beschrieben. Die Schrägstriche (`/`) zur Trennung der Hierarchieebenen werden nicht nur für die Pfade, sondern auch für die übrigen hierarchischen Variablen verwendet.

Auch aus den Betriebssystemaufrufen selbst und ihren Parametern lässt sich eine hierarchische Variable konstruieren. Da es in diesem Fall keine eindeutige natürliche Hierarchie

gibt, wurde eine Taxonomie der Betriebssystemaufrufe und ihrer Parameter erstellt, anhand derer die Generalisierungsvorgänge vorgenommen werden können. Eine genauere Beschreibung erfolgt in Abschnitt 5.1.2.

Die dritte hierarchische Variable ist die Sequenz der Betriebssystemaufrufe. In diesem Fall wird der einzelne Aufruf nicht hierarchisch behandelt wie im vorigen Abschnitt, sondern (flach) durch seinen Namen beschrieben. Parameterwerte werden nicht berücksichtigt. Die hierarchische Struktur entsteht durch die Sequenz der einzelnen Namen. Die Sequenzen bilden eine natürliche Hierarchie, die Generalisierung erfolgt durch Verkürzen der Sequenz: `/close/read` ist Elter von `/close/read/open`, der wiederum Elter von `/close/read/open/stat` ist. Die Extraktion dieser hierarchischen Variablen wird in Abschnitt 5.1.3 beschrieben.

Neben der Extraktion der hierarchischen Variablen werden bei der Verarbeitung der Rohdaten zwei weitere Arbeitsschritte durchgeführt: Die neben den Betriebssystemaufrufen in den Logdateien enthaltenen Signale zur Interprozesskommunikation (siehe [Silberschatz et al., 2010], S. 837) werden nicht benötigt und daher verworfen, und unvollständig ausgeführte Betriebssystemaufrufe werden wieder zusammengesetzt. Zur unvollständigen Ausführung von Aufrufen kommt es, wenn ein Prozess blockierende Eingabe/Ausgabe vornimmt (siehe [Silberschatz et al., 2010], S. 570). Während der Prozess auf das Gerät wartet, werden andere Prozesse ausgeführt. Strace schreibt in diesem Fall zwei Einträge in die Logdaten: Zum Zeitpunkt des blockierenden Aufrufs wird der Name des Betriebssystemaufrufs mit dem Vermerk `<unfinished...>` geschrieben, zum Zeitpunkt der Beendigung werden Parameter und Rückgabewerte mit dem Vermerk `<...resumed>` geschrieben. Im Rahmen der Verarbeitung der Rohdaten werden die unvollständigen blockierenden Aufrufe der einzelnen Prozesse gespeichert und nach Beendigung zu einem vollständigen Aufruf zusammengesetzt. Der Speicherbedarf für das Zusammensetzen unvollständiger Aufrufe ist durch die Anzahl der laufenden Prozesse begrenzt.

Die Extraktion der hierarchischen Variablen wird in den folgenden Abschnitten beschrieben. Sie erfolgt während des Einlesens der Daten, so dass das Gesamtsystem in der Lage ist, direkt auf dem von strace erzeugten Datenstrom zu arbeiten. Für die in Kapitel 8 beschriebenen Experimente wurden die Rohdaten allerdings nicht direkt aus dem Datenstrom gelesen, sondern aus Logdateien.

5.1.1 Pfade als eine hierarchische Variable

Da Dateisystempfade eine natürliche Hierarchie bilden, ist es naheliegend, sie für die Konstruktion der hierarchischen Variablen zu verwenden.

Drei Fälle lassen sich unterscheiden:

Bei einigen Systemaufrufen wird der Pfad, mit dem der Aufruf arbeitet, unmittelbar angegeben (`open`, `stat`, `execve`). In diesem Fall kann der Pfad unmittelbar als Wert der hierarchischen Variablen verwendet werden und es sind keine weiteren Verarbeitungsschritte erforderlich.

Eine andere Gruppe von Aufrufen verwendet gar keine Pfade (`gettimeofday`, `brk`, `clock_gettime`), in diesem Fall wird der Pfadvariablen der allgemeinstmögliche Wert `*` zugewiesen.

Eine dritte Gruppe von Aufrufen (`read`, `write`, `fstat`) verwendet Pfade, die durch Dateideskriptoren dargestellt werden. Dateideskriptoren sind Indizes in der Tabelle der offenen Dateien, die für jeden Prozess verwaltet wird. Im einfachsten Fall wird eine Datei mit dem Systemaufruf `open` unter direkter Angabe ihres Pfadnamens geöffnet, der dabei zurückgegebene Dateideskriptor wird vom Systemaufruf `read` genutzt, um die Datei zu lesen, die anschließend mit dem Aufruf `close` unter erneuter Verwendung des Dateideskriptors geschlossen wird (siehe [Silberschatz et al., 2010], S. 465, für ein Beispiel siehe Kapitel 2).

Um auch diese durch Dateideskriptoren bezeichneten Pfade verwenden zu können, müssen die Deskriptoren im Rahmen der Extraktion der hierarchischen Variablen wieder in Pfade umgewandelt werden. Damit für jeden einzelnen Prozess die dafür erforderlichen Informationen bekannt sind, muss während der Verarbeitung des Datenstroms der Teil des Betriebssystems, der die Tabelle der offenen Dateien verwaltet, simuliert werden. Dazu wird für jeden Prozess eine Tabelle der offenen Dateien angelegt, die jeweils aktualisiert wird, wenn ein Systemaufruf im Datenstrom Auswirkungen auf die Dateideskriptoren hat.

In der Regel ist die Situation dabei komplizierter als im oben angegeben Fall (`open-read-close`), weil Dateideskriptoren eines Prozesses an dessen Kindprozesse übergeben werden können. Dies kann als Kopie geschehen oder auch in Form der gemeinsamen Nutzung, bei der sich `open` und `close`-Aufrufe des einen Prozesses auf die Dateideskriptoren des jeweils anderen Prozesses auswirken. Insbesondere können Prozesse also Dateideskriptoren verwenden, die sie nie selbst erzeugt haben.

Darüber hinaus kann durch bestimmte Systemaufrufe ein Flag (`FD_CLOEXEC`) für einzelne Dateideskriptoren gesetzt werden, welches das Verhalten während eines `execve`-Aufrufs beeinflusst: Normalerweise wird während eines `execve`-Aufrufs die Anwendung in der als Parameter übergebenen Datei in den Speicher geladen, der aktuelle Prozess dabei überschrieben und die Dateideskriptoren geschlossen. Deskriptoren, deren `FD_CLOEXEC`-Flag gesetzt ist, bleiben dagegen geöffnet. Auch dies muss im Rahmen der Simulation berücksichtigt werden.

Die folgenden Systemaufrufe sind für die Auflösung der Dateideskriptoren von Bedeutung (vereinfacht):

- `open`: Öffnet Datei, neuer Eintrag in der Tabelle offener Dateien und neuer Dateideskriptor werden erzeugt. Je nach Parametern wird `FD_CLOEXEC`-Flag des Deskriptors gesetzt.
- `close`: Schließt Datei, Eintrag in Tabelle und Deskriptor werden frei.
- `fork`, `vfork`: Erzeugt Kindprozess, der Kopie des aktuellen Prozesses ist. Das Kind erhält eine Kopie der Tabelle offener Dateien.

TID	Systemaufruf	Pfadvariable
8364	`open("/proc/meminfo", O_RDONLY) = 3`	`/proc/meminfo`
8364	`fcntl64(3, F_DUPFD_CLOEXEC, 4) = 4`	`/proc/meminfo`
8364	`clone( ... flags=CLONE_FILES|CLONE_SIGHAND ...) = 8380`	`*`
8380	`access("/tmp/log", X_OK|R_OK) = 0`	`/tmp/log`
8380	`fstat64(4, {st_mode=S_IFREG|0444, st_size=0, ...}) = 0`	`/proc/meminfo`
8380	`socket(PF_FILE, SOCK_STREAM, 0) = 5`	`/Socket`

Tabelle 5.1: Extraktion der hierarchischen Pfadvariablen aus den Systemaufrufen

- `clone`: Erzeugt Kindprozess. Je nach Parametern erhält das Kind eine Kopie der Tabelle offener Dateien oder Elter und Kind nutzen die Tabelle gemeinsam, in diesem Fall wirken sich Änderungen an den Dateideskriptoren des einen Prozesses auf den jeweils anderen Prozess aus.
- `unshare`: Beendet die gemeinsame Nutzung der Tabelle offener Dateien aus `clone` und erzeugt echte Kopien.
- `execve`: Lädt die Anwendung aus der übergebenen Datei in den Speicher, der aktuelle Prozess wird überschrieben. Ob die Dateideskriptoren dabei geschlossen werden, hängt jeweils von ihrem `FD_CLOEXEC`-Flag ab.
- `dup, dup2, dup3`: Kopiert Dateideskriptoren, beeinflusst je nach Parametern das `FD_CLOEXEC`-Flag.
- `fcntl`, `fcntl64`: Ähnlich dup, kopiert Deskriptoren, beeinflusst je nach Parametern das `FD_CLOEXEC`-Flag.

Darüber hinaus arbeiten verschiedene weitere Betriebssystemaufrufe mit Dateideskriptoren (`socket`, `pipe`, `inotify_init` und deren Varianten). Die Deskriptoren stehen dabei allerdings nicht für normale Dateien im Dateisystem, sondern (gemäß dem Unix-Prinzip „Alles ist eine Datei“) für eine Anzahl sehr unterschiedlicher Dinge von Pipes zur Interprozesskommunikation über Sockets bis zu Informationsquellen zur Abfrage von Ereignissen. Um die darin enthaltenen Informationen ebenfalls nutzen zu können, werden diese Dateideskriptoren durch einfache Pseudopfade wie `/Socket` gekennzeichnet und die hierarchische Pfadvariable mit dieser Information angereichert.

Tabelle 5.1 zeigt die Extraktion der hierarchischen Pfadvariablen aus den Rohdaten an einigen Beispielen. In der ersten Zeile wird die Datei `/proc/meminfo` zum Lesen geöffnet und der Deskriptor 3 zurück gegeben. In der zweiten Zeile ist die Auflösung des Deskriptors 3 zu erkennen. Der Deskriptor wird durch den Aufruf `fcntl` kopiert, nun bezieht sich auch Deskriptor 4 auf die Datei `/proc/meminfo`. Die nächsten beiden Zeilen zeigen die Behandlung von Aufrufen, die keine Pfade enthalten. In der vierten Zeile entsteht durch `clone` ein neuer Prozess mit Prozess-ID 8380. Dieser kann in der vorletzten Zeile den Deskriptor 4 des Elters verwenden. In der letzten Zeile ist die Verwendung eines Pseudopfads für ein Socket zu sehen.

open	dup3	execve	socket	inotify_init1
close	clone	fcntl	pipe	accept
dup	fork	fcntl64	pipe2	accept4
dup2	vfork	unshare	inotify_init	

Tabelle 5.2: Systemaufrufe, die für die Auflösung der Dateideskriptoren erforderlich sind

Die Systemaufrufe, welche die Auflösung von Dateideskriptoren beeinflussen, sind in Tabelle 5.2 zusammengefasst. Die Kenntnis der Aufrufe in den ersten drei Spalten ist für die Berechnung der Pfadnamen zwingend erforderlich. Wenn aufgrund der in Abschnitt 5.3 beschriebenen Kosten des Protokollierens auf das Protokollieren einiger Aufrufe verzichtet werden soll (Merkmalsselektion, siehe Abschnitt 6.3.4), müssen diese Aufrufe unter den protokollierten Aufrufen sein, sofern Pfade als hierarchische Variable verwendet werden sollen. Andernfalls ist eine Auflösung der Deskriptoren nicht möglich. Die Aufrufe der letzten beiden Spalten sind für die Auflösung der Dateideskriptoren von Pseudopfaden erforderlich. Sofern diese Aufrufe nicht protokolliert werden, geht keine Information über die echten Pfadnamen verloren, allerdings werden dann Zugriffe auf Sockets, Pipes und andere spezielle Dateien als Zugriffe auf unbekannte Dateien gekennzeichnet. Insbesondere ist es in diesem Fall nicht möglich zu entscheiden, ob ein Dateideskriptor ein Socket, eine Pipe oder eine andere spezielle Datei bezeichnet.

Für die Auflösung der Deskriptoren muss für jeden Prozess eine Tabelle der offenen Dateien verwaltet werden; der Speicherbedarf für eine einzelne Datei ist konstant, weil die Länge von Pfaden (und der Speicherbedarf der Flags) beschränkt ist. Daher ist der Speicherbedarf für die Auflösung der Deskriptoren durch die Anzahl der gleichzeitig geöffneten Dateien und die Anzahl der Prozesse begrenzt. Das vollständige Speichern des Datenstroms ist nicht erforderlich.

5.1.2 Systemaufrufe als eine hierarchische Variable

Aus den Systemaufrufen und ihren Parametern lässt sich eine hierarchische Variable konstruieren. Zunächst lassen sich die Systemaufrufe gemäß ihrer Funktion in verschiedene Gruppen einteilen: Es gibt Systemaufrufe, die das Dateisystem betreffen (`open`), andere dienen der Prozessverwaltung (`clone`) oder der Kommunikation (`socket`). Diese Einteilung lässt sich anhand der jeweils verwendeten Parameter weiter verfeinern: `open` öffnet beispielsweise in Abhängigkeit des Parameters `flags` die Datei zum Lesen (`O_RDONLY`), zum Schreiben (`O_WRONLY`) oder für beides (`O_RDWR`).

[Silberschatz et al., 2010] teilen Systemaufrufe in sechs verschiedene Gruppen ein: Process Control, File Manipulation, Device Manipulation, Information Management, Communication und Protection (S.60 ff.), in älteren Ausgaben des Buchs fehlte die Gruppe Protection noch ([Silberschatz et al., 2000], S. 56). [Stallings, 2009] beschreibt die sechs Gruppen Dateisystem, Prozesse, Scheduling, Interprozesskommunika-

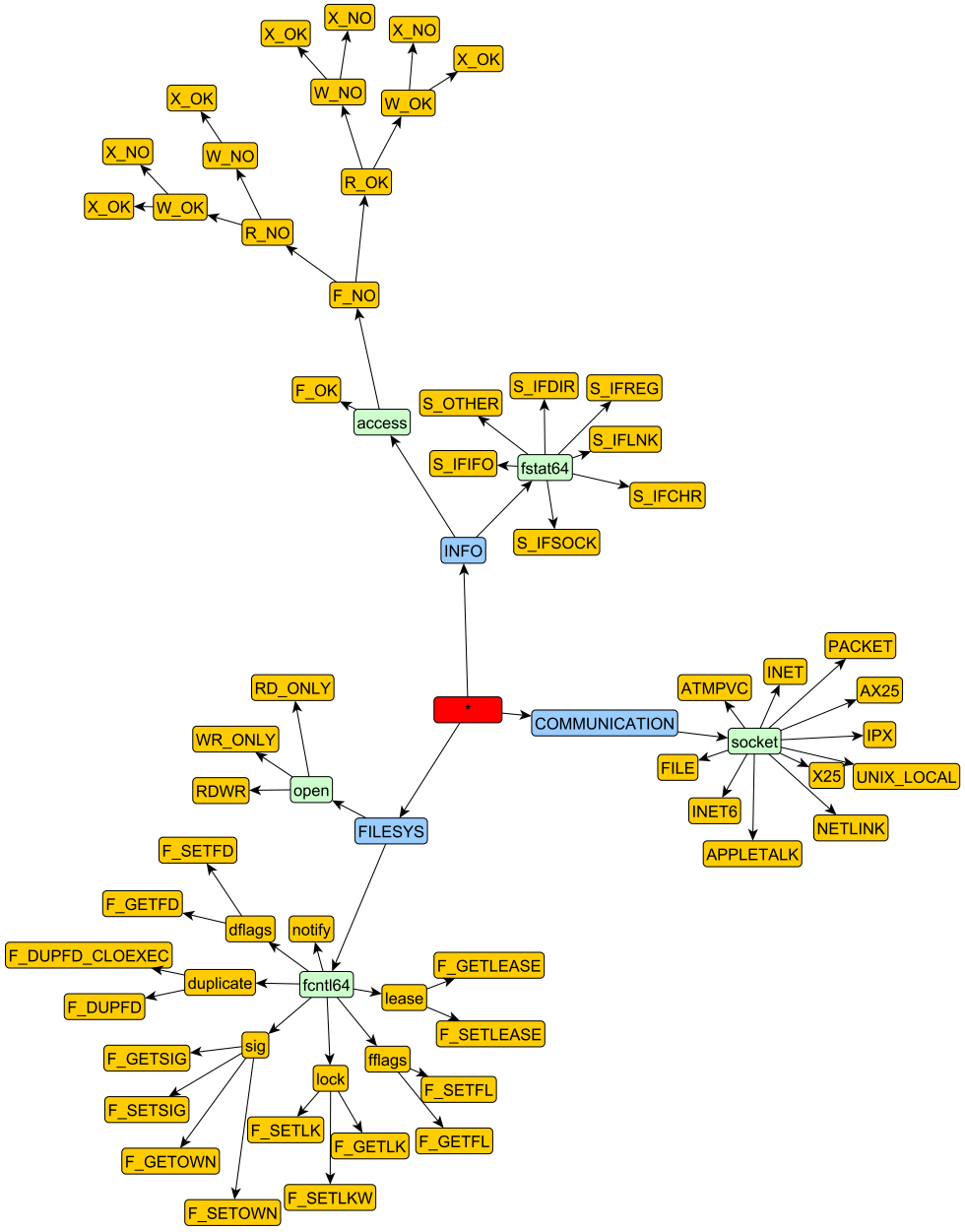


Abbildung 5.1: Ausschnitt aus der Taxonomie der Systemaufrufe und ihrer Parameter

tion, Socket/Netzwerk und Sonstige (S. 100). [Tanenbaum, 2003] nennt dagegen vier Gruppen: Prozessmanagement, Dateimanagement, Verzeichnis- und Dateimanagement und Verschiedenes (S. 62).

In Anlehnung an [Silberschatz et al., 2010] habe ich die Aufrufe in fünf Gruppen eingeordnet: Prozess- und Speicherverwaltung (`PROCMEM`), Dateisystem (`FILESYS`), Geräte (`DEVICE`) und Information (`INFO`). Die Kategorie Protection habe ich nicht übernommen, da in den Logdateien nur wenige Aufrufe aus diesem Bereich zu finden waren. Auch die Gruppe `DEVICE` ist dünn besetzt.

Diese Einteilung der Aufrufe in Gruppen bildet die oberste Hierarchieebene, die Aufrufe selbst bilden die zweite Hierarchieebene (siehe Abbildung 5.1). Um diese Einteilung zu verfeinern, wurden auch die Parameter der Aufrufe verwendet. Diese können einfache Zahlen (z. B. Dateideskriptoren), Zeichenketten (z. B. Dateinamen) oder Flags sein (z. B. `FD_CLOEXEC`), aber auch komplexere Gebilde wie Strukturen (struct), die Felder (Arrays) von Strukturen und Zeichenketten enthalten. Das Einlesen dieser Parameter aus den unstrukturierten Logdaten wird in Abschnitt 6.1 beschrieben.

Wie diese Parameter für die Verfeinerung der Hierarchie verwendet werden, ist vom jeweiligen Aufruf abhängig. Einige Beispiele sind in Abbildung 5.1 dargestellt und werden im Folgenden kurz erläutert. Die grundsätzliche Funktionsweise der dargestellten Aufrufe wurde bereits in Abschnitt 2.1 erklärt, eine ausführlichere Beschreibung findet sich in Abschnitt zwei des Handbuchs (Manpages).

Der Aufruf `open` wurde bereits beschrieben, die Behandlung der Parameter des Aufrufs `fcntl64( fd, cmd, arg )` ist etwas aufwändiger, weil hier ein Parameter noch weiter hierarchisch unterteilt wird. `fcntl64` manipuliert den Dateideskriptor `fd`, die durchzuführende Operation wird durch den Wert des Parameters `cmd` festgelegt, die Bedeutung des Parameters `arg` ist unterschiedlich und von dieser Operation abhängig. `cmd` kann 16 Werte annehmen, die ich in sieben Gruppen eingeteilt habe. So gibt es die Gruppe `duplicate` zum Kopieren eines Deskriptors mit den Werten `F_DUPFD` (Kopieren des Deskriptors) und `F_DUPFD_CLOEXEC` (Kopieren des Deskriptors und Setzen des `FD_CLOEXEC`-Flags der Kopie). Eine weitere Gruppe ist `dflags` zum Lesen und Ändern des Deskriptorflags mit den Werten `F_GETFD` zum Lesen und `F_SETFD` zum Schreiben des `FD_CLOEXEC`-Flags des Deskriptors. Weitere Gruppen dienen dem Manipulieren von Flags der Datei (`fflags`), dem Sperren von Regionen der Datei (`lock`) und der Verwaltung bestimmter Signaleinstellungen. Diese Gruppeneinteilung bildet die dritte Hierarchieebene, die einzelnen Werte bilden die vierte Ebene.

Der Aufruf `access( name, mode )` wird anhand des Parameters `mode` feiner eingeteilt: Auf der dritten Hierarchieebene wird die Einteilung anhand des Wertes `F_OK` vorgenommen. Hat `mode` den Wert `F_OK`, so wird der Aufruf verwendet, um zu bestimmen, ob die Datei `name` existiert. In diesem Fall ist eine weitere Verfeinerung auf tieferen Ebenen nicht erforderlich. Hat `mode` nicht den Wert `F_OK` (dies wird dargestellt als `F_NO`), so wird der Aufruf verwendet, um die Lese-, Schreib- und/oder Ausführungsrechte der Datei zu ermitteln. Welche Rechte ermittelt werden sollen, wird durch Oder-Verknüpfung der Flags `R_OK` (Lesen), `W_OK` (Schreiben) und `X_OK` (Ausführen) über den Parameter

TID	Systemaufruf
8364	`open("/proc/meminfo", O_RDONLY) = 3` `/FILESYS/open/RD_ONLY`
8364	`fcntl64(3, F_DUPFD_CLOEXEC, 4) = 4` `/FILESYS/fcntl64/duplicate/F_DUPFD_CLOEXEC`
8364	`clone(child_stack=0xb5a7f4a4, flags=CLONE_FILES, ...) = 8380` `/PROCMEM/clone/CLONE_FILES/stackNotNull/NO_CLONE_THREAD`
8380	`access("/tmp/log", X_OK|R_OK) = 0` `/INFO/access/F_NO/R_OK/W_NO/X_OK`
8380	`fstat64(4, {st_mode=S_IFREG|0444, st_size=0, ...}) = 0` `/INFO/fstat64/S_IFREG`
8380	`socket(PF_FILE, SOCK_STREAM, 0) = 5` `/COMMUNICATION/socket/FILE`

Tabelle 5.3: Extraktion der hierarchischen Aufrufvariablen. Für jeden Eintrag sind zunächst die Rohdaten angegeben, in der Zeile darunter ist die extrahierte Aufrufvariable angegeben.

`mode` festgelegt. In diesem Fall kann `mode` sieben Werte annehmen, je nachdem, welche Kombination von Lese-, Schreib- und/oder Ausführungsrechten ermittelt werden soll. Es wäre möglich gewesen, diese sieben Werte gleichberechtigt auf der vierten Hierarchieebene darzustellen. Stattdessen habe ich die Werte weiter hierarchisch aufgespalten, zunächst anhand der Frage, ob Leserechte ermittelt werden sollten, dann anhand der Schreib- und Ausführungsrechte. Durch diese weitere hierarchische Aufspaltung lässt sich beispielsweise die Menge der `access`-Aufrufe, die Lese- und Schreibrechte ermitteln (Ausführungsrechte beliebig), durch ein einzelnes Präfix beschreiben, in einer gleichberechtigten Darstellung auf der vierten Hierarchieebene wäre das nicht möglich gewesen. Da die HHH-Algorithmen die Häufigkeiten von Mengen über derartige Präfixe verwalten, erschien mir diese Aufspaltung sinnvoll. Da `access`-Aufrufe alle Kombinationen von Lese-, Schreib- und Ausführungsrechten ermitteln können, bilden die Parameter allerdings einen dreidimensionalen Raum, der sich nicht vollständig in eine monohierarchische Struktur pressen lässt: Beispielsweise ist die Menge der `access`-Aufrufe, die Ausführungs- und Schreibrechte ermitteln sollen (Leserechte beliebig), daher nicht durch ein einzelnes Präfix beschreibbar. Auch in anderen Fällen war es erforderlich, die Parameter eines Aufrufs, die wegen derartiger Kombinationsmöglichkeiten eigentlich mehrdimensional waren, auf ähnliche Weise in eine monohierarchische Struktur einzuordnen.

Für `fstat64( fd, buf )` ist die Situation wieder einfacher, hier gibt es nur eine dritte Hierarchieebene, die durch die Werte von `st_mode` gebildet wird. `st_mode` ist eine Komponente der Struktur, die vom Betriebssystem in den Puffer `buf` geschrieben wird und den Status der untersuchten Datei beschreibt. Anders als bei den übrigen beschriebenen Aufrufen erfolgt die Einteilung hier also nicht unmittelbar auf der Ebene der Parameter, sondern über den Rückgabewert. Mögliche Werte sind unter anderem `S_IFDIR` für Verzeichnisse, `S_IFREG` für reguläre Dateien und `S_IFSOCKET` für Sockets.

FILESYS	COMMUNICATION	PROCMEM	INFO	DEVICE
open	recvmsg	mmap2	access	ioctl
read	recv	munmap	getdents	
write	send	brk	getdents64	
lseek	sendmsg	clone	clock_gettime	
_llseek	sendfile	fork	gettimeofday	
writev	sendto	vfork	time	
fcntl	rt_sigaction	mprotect	uname	
fcntl64	pipe	unshare	poll	
dup	pipe2	execve	fstat	
dup2	socket	futex	fstat64	
dup3	accept	nanosleep	lstat	
close	accept4		lstat64	
			stat	
			stat64	
			inotify_init	
			inotify_init1	
			readlink	
			select	

Tabelle 5.4: Liste der verwendeten Systemaufrufe. Die übrigen Aufrufe stellen weniger als 1 % der Aufrufe in den Logdateien.

Die Situation für `socket( domain, type, prot )` ist ähnlich einfach, hier bildet der Parameter `domain` die dritte Hierarchieebene. Mögliche Werte sind unter anderem `FILE` für lokale Kommunikation, `INET` für das IP-Protokoll und `INET6` für Version 6 des IP-Protokolls.

In Tabelle 5.3 ist die Extraktion der hierarchischen Variablen aus den Systemaufrufen dargestellt.

Insgesamt wurden auf diese Weise 54 Aufrufe (von etwa 319 Aufrufen in aktuellen Linux-Systemen) in eine Taxonomie eingeordnet. Die übrigen Aufrufe treten nur sehr selten oder gar nicht in den Logdateien auf und stellen insgesamt weniger als 1 % aller Aufrufe. Da die Algorithmen zur Berechnung der HHH seltenen Stromelementen wenig Beachtung schenken, ist nicht davon auszugehen, dass diese Aufrufe einen wesentlichen Einfluss auf die Berechnung der HHH-Mengen haben. Die verwendeten Aufrufe sind in Tabelle 5.4 dargestellt.

5.1.3 Sequenzen als eine hierarchische Variable

Die Sequenzen der Namen von Betriebssystemaufrufen bilden eine natürliche Hierarchie und lassen sich als hierarchische Variable verwenden. Die Erfassung der Sequenzen erfolgt auf der Ebene der einzelnen Prozesse der Anwendung, nicht auf der Ebene der gesam-

TID	Systemaufruf	Sequenzvariable
8364	`open("/proc/meminfo", O_RDONLY) = 3`	*
8364	`fcntl64(3, F_DUPFD_CLOEXEC, 4) = 4`	/open
8364	`clone( ... flags=CLONE_FILES|CLONE_SIGHAND ...) = 8380`	/fcntl64/open
8380	`access("/tmp/log", X_OK|R_OK) = 0`	*
8380	`fstat64(4, {st_mode=S_IFREG|0444, st_size=0, ...}) = 0`	/access
8380	`socket(PF_FILE, SOCK_STREAM, 0) = 5`	/fstat64/access

Tabelle 5.5: Extraktion der Sequenzvariablen aus den Systemaufrufen

ten Anwendung. Tabelle 5.5 zeigt dies beispielhaft für zwei Prozesse einer Anwendung. Der jeweils vorausgegangene Systemaufruf wird auf der höchsten Hierarchieebene der Sequenzvariablen eingeordnet, weiter zurückliegende Aufrufe auf tieferen Ebenen. In der ersten und vierten Zeile gibt es für den jeweiligen Prozess noch keinen Vorgängeraufruf, daher wird der Sequenzvariablen der Wert * zugewiesen.

5.2 Ähnlichkeitsmaße

Nach der Extraktion der hierarchischen Variablen aus den Rohdaten werden die Algorithmen zur Berechnung der Hierarchical Heavy Hitters (Kapitel 3) ausgeführt. Auf diese Weise wird aus jedem Datenstrom eine HHH-Menge berechnet. Die Klassifikation neuer Ströme erfolgt durch den Vergleich ihrer HHH-Mengen mit den HHH-Mengen von Datenströmen, deren Klasse bekannt ist. Dazu ist es erforderlich, Ähnlichkeitsmaße für diese Mengen zu definieren. Auch Datenströme unterschiedlicher Länge lassen sich über die HHH-Mengen gut vergleichen: Der Einfluss unterschiedlicher Stromlängen N auf die absoluten Häufigkeiten der einzelnen Elemente wird relativiert, weil die einzelnen HHH bestimmt werden, indem die absolute Häufigkeit der Elemente mit ϕN verglichen wird.

In diesem Abschnitt werden zunächst einige „flache“ Ähnlichkeitsmaße beschrieben, welche die Ähnlichkeit zweier Mengen über die Größe der Schnittmenge definieren. Anhand eines Beispiels wird gezeigt, dass diese Ähnlichkeitsmaße für HHH-Mengen ungeeignet sind, weil sie die hierarchische Struktur der Variablen unberücksichtigt lassen. Anschließend werden einige Ähnlichkeitsmaße beschrieben, die diese hierarchische Struktur berücksichtigen: Das Optimistic Genealogy Measure ([Ganesan et al., 2003]) ist ein hierarchisches Ähnlichkeitsmaß für den eindimensionalen Fall, ich beschreibe eine eigene Verallgemeinerung für den mehrdimensionalen Fall und eine von [Cormode et al., 2008] vorgeschlagene Verallgemeinerung. Ich zeige an einem Beispiel die Schwächen dieser Verallgemeinerungen und schlage zwei Varianten vor. Zuletzt beschreibe ich, warum es sinnvoll sein könnte, nicht die berechneten HHH-Mengen, sondern die Datenstruktur des HHH-Algorithmus als kondensierte Repräsentation des Datenstroms aufzufassen und schlage ein Ähnlichkeitsmaß für Datenstrukturen vor. Alle beschriebenen Ähnlichkeitsmaße sind implementiert worden und können zum Vergleich der Datenströme verwendet werden.

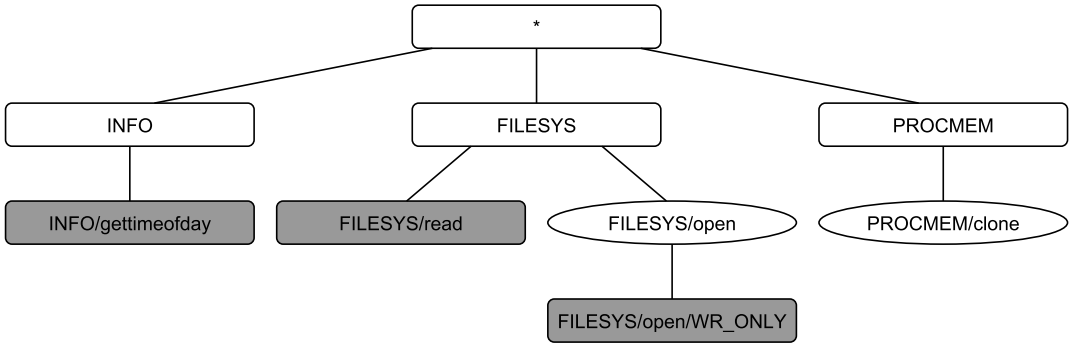


Abbildung 5.2: Beispiel für Ähnlichkeitsmaße auf Mengen. Die Ähnlichkeitswerte der Mengen C_1 (grau) und C_2 (elliptisch) sind in Tabelle 5.6 dargestellt.

5.2.1 Flache Ähnlichkeitsmaße

Jaccard-Koeffizient und Dice-Koeffizient sind flache Ähnlichkeitsmaße für Mengen, die eine mögliche hierarchische Struktur der Elemente nicht berücksichtigen. Beide definieren Ähnlichkeit über die Größe der Schnittmenge und unterscheiden sich lediglich in der Art der Normalisierung ([Ganesan et al., 2003]): Für Mengen C_1 und C_2 ist der Jaccard-Koeffizient definiert als

$$sim_{Jacc}(C_1, C_2) = \frac{|C_1 \cap C_2|}{|C_1 \cup C_2|},$$

für den Dice-Koeffizienten gilt

$$sim_{Dice}(C_1, C_2) = \frac{2 \cdot |C_1 \cap C_2|}{|C_1| + |C_2|}.$$

Ein einfaches Beispiel zeigt, warum diese flachen Ähnlichkeitsmaße für HHH-Mengen nicht gut geeignet sind: Die Mengen $M_1 = \{$`/FILESYS/read`$\}$, $M_2 = \{$`/FILESYS/write`$\}$ und $M_3 = \{$`/INFO/gettimeofday`$\}$ sind paarweise disjunkt, daher sind alle Jaccard- und Dice-Koeffizienten null, obwohl M_1 und M_2 einander intuitiv ähnlicher sind als M_1 und M_3 oder M_2 und M_3. Die im Folgenden beschriebenen hierarchischen Ähnlichkeitsmaße erkennen diese Ähnlichkeit und weisen M_1 und M_2 einen größeren Ähnlichkeitswert zu.

5.2.2 Optimistic Genealogy Measure (OGM)

Das Optimistic Genealogy Measure (OGM) wurde von ([Ganesan et al., 2003]) vorgeschlagen und ist ein Ähnlichkeitsmaß für Mengen von eindimensionalen Elementen mit hierarchischer Struktur. Die Mengen enthalten dabei lediglich die Blätter der Hierarchie. HHH-Mengen können dagegen mehrdimensionale Elemente enthalten, und die einzelnen hierarchischen Variablen der Elemente können auch innere Knoten ihrer Hierarchie sein.

Das hier dargestellte Ähnlichkeitsmaß OGM ist meine mehrdimensionale Verallgemeinerung des eindimensionalen Ähnlichkeitsmaßes von [Ganesan et al., 2003].[1]

Werden zwei Mengen C_1 und C_2 von Elementen verglichen, so wird zu jedem Element $l_1 \in C_1$ ein Ähnlichkeitsbeitrag bestimmt, indem ein zu l_1 ähnlichstes Element aus C_2 gesucht und die Ähnlichkeit der beiden bestimmt wird. Entsprechend werden die Ähnlichkeitsbeiträge für jedes Element $l_2 \in C_2$ bestimmt. Die Ähnlichkeit von Elementen wird definiert über die Ebene des Supremums der Elemente. Dieses Supremum existiert auch im mehrdimensionalen Fall für jedes Paar von Elementen, deren hierarchische Variablen aus denselben Hierarchien stammen (siehe Kapitel 3).

Definiert man für $l_1 \in C_1$

$$\mathit{suplevel}(l_1, C_2) = \max_{l_2 \in C_2} \mathit{level}(\mathit{sup}(l_1, l_2)),$$

dann lässt sich der Ähnlichkeitsbeitrag von l_1 bzgl. C_2 definieren als

$$\mathit{contrib}_{OGM}(l_1, C_2) = \begin{cases} 1, \text{ falls } l_1 = * \\ \frac{\mathit{suplevel}(l_1, C_2)}{\mathit{level}(l_1)} \text{ sonst.} \end{cases}$$

Die Ähnlichkeit der Mengen C_1 und C_2 wird definiert als durchschnittlicher Ähnlichkeitsbeitrag ihrer Elemente:

$$\mathit{sim}_{OGM}(C_1, C_2) = \frac{\sum_{l_1 \in C_1} \mathit{contrib}_{OGM}(l_1, C_2) + \sum_{l_2 \in C_2} \mathit{contrib}_{OGM}(l_2, C_1)}{|C_1| + |C_2|}.$$

Das Ähnlichkeitsmaß OGM lässt sich auch dann anwenden, wenn die hierarchischen Variablen der Elemente nicht nur Blätter, sondern auch innere Knoten der Hierarchien enthalten, sofern für den Wert $*$ und beliebige Mengen C der Ähnlichkeitsbeitrag $\mathit{contrib}_{OGM}(*, C) = 1$ gesetzt wird. Die Festlegung für den Ähnlichkeitsbeitrag $\mathit{contrib}_{OGM}(*, C)$ ist erforderlich, um eine Division durch null zu vermeiden, denn es gilt $\mathit{level}(*) = 0$. Welcher Wert dabei angesetzt werden sollte, ist nicht unmittelbar offensichtlich. Der Wert 1 vermeidet einen logischen Bruch, denn für alle Werte $l_1 \neq *$ ist $\mathit{contrib}_{OGM}(l_1, C) = 1$, wenn l_1 Vorfahr seines ähnlichsten Elementes in C ist.

Abbildung 5.2 zeigt einen Ausschnitt aus der Hierarchie, aus der die Elemente der Mengen $C_1 = \{$`/FILESYS/read`, `/FILESYS/open/WR_ONLY`, `/INFO/gettimeofday`$\}$ und $C_2 = \{$`/FILESYS/open`, `/PROCMEM/clone`$\}$ stammen. Es gilt beispielsweise für den Ähnlichkeitsbeitrag $\mathit{contrib}_{OGM}($`/FILESYS/read`$, C_2) = \frac{1}{2}$, denn $\mathit{level}($`/FILESYS/read`$) = 2$ und $\mathit{sup}($`/FILESYS/read`, `/FILESYS/open`$) =$ `/FILESYS` sowie $\mathit{level}($`/FILESYS`$) = 1$.

Tabelle 5.6 zeigt die suplevel-Werte und die Ähnlichkeitsbeiträge der Elemente der Mengen C_1 und C_2 für verschiedene Ähnlichkeitsmaße.

[1] Das von [Ganesan et al., 2003] verwendete $LCA_{T_1,T_2}(l_1)$ ist nur im eindimensionalen Fall eindeutig. Durch meine Definition von $\mathit{suplevel}(l_1, C_2)$ wird eine Mehrdeutigkeit im mehrdimensionalen Fall vermieden.

	Element	level	suplevel	match	OGM	COR	MCOR	MOGM
C_1	`/FILESYS/read`	2	1	2	$\frac{1}{2}$	1	1	$\frac{1}{2}$
	`/FILESYS/open/WR_ONLY`	3	2	2	$\frac{2}{3}$	1	$\frac{1}{2}$	$\frac{5}{6}$
	`/INFO/gettimeofday`	2	0	2	0	2	2	0
C_2	`/FILESYS/open`	2	2	3	1	0	$\frac{1}{2}$	$\frac{5}{6}$
	`/PROCMEM/clone`	2	0	2	0	2	2	0
Ähnlichkeit C_1 und C_2:					$\frac{13}{30}$	-5	-5	$\frac{13}{30}$

Tabelle 5.6: Übersicht der level-, suplevel- und matchlevel-Werte und der Ähnlichkeitsbeiträge der Elemente aus Abbildung 5.2 für verschiedene Ähnlichkeitsmaße.

5.2.3 Optimistic Genealogy Measure nach Cormode et al. (COR)

[Cormode et al., 2008] schlagen für den Fall, dass die zu vergleichenden Mengen mehrdimensionale Elemente und nicht nur Blätter, sondern auch innere Knoten enthalten, eine Variante (nachfolgend abgekürzt als COR) des Optimistic Genealogy Measure vor, die nicht Verhältnisse, sondern Differenzen verwendet. Der Ähnlichkeitsbeitrag (hier eigentlich ein Distanzbeitrag) wird definiert als

$$contrib_{COR}(l_1, C_2) = level(l_1) - suplevel(l_1, C_2).$$

Die Ähnlichkeit der Mengen C_1 und C_2 wird definiert als:[2]

$$sim_{COR}(C_1, C_2) = 1 - \sum_{l_1 \in C_1} contrib_{COR}(l_1, C_2) - \sum_{l_2 \in C_2} contrib_{COR}(l_2, C_1).$$

Eine Normierung auf den Bereich $[0, 1]$ nehmen [Cormode et al., 2008] nicht vor.

Für die Mengen C_1 und C_2 aus Abbildung 5.2 gilt $contrib_{COR}(\texttt{/FILESYS/read}, C_2) = 1$, denn wie zuvor ist $level(\texttt{/FILESYS/read}) = 2$ und $level(\texttt{/FILESYS}) = 1$.

5.2.4 Modified Genealogy Measure Cormode (MCOR)

Die dargestellten hierarchischen Ähnlichkeitsmaße OGM und COR haben den Nachteil, in bestimmten Fällen für *verschiedene* Mengen die Ähnlichkeit eins (Abstand null) zu berechnen. Abbildung 5.3 zeigt ein Beispiel. Wenn Menge C_3 Obermenge von Menge C_2 ist und alle Elemente der Differenzmenge Vorfahren ihres ähnlichsten Elementes in C_2 sind, gilt für alle Elemente $l_3 \in C_3$, dass $suplevel(l_3, C_2) = level(l_3)$.

Dieses Problem habe ich behoben, indem ich nicht nur das Level von l_3, sondern auch das Level der zu l_3 ähnlichsten Elemente aus C_2 berücksichtige.

[2] Die Formel in [Cormode et al., 2008], S. 39 enthält einen Vorzeichenfehler.

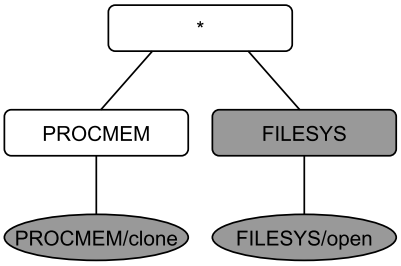


Abbildung 5.3: Verschiedene Mengen mit Ähnlichkeit 1: Die Menge der grauen Knoten C_3 ist Obermenge der Menge der elliptischen Knoten C_2. Die Mengen unterscheiden sich, haben aber nach OGM und COR Ähnlichkeit 1 (Abstand null), weil der Knoten aus $C_3 \backslash C_2$ Vorfahr seines besten Partners aus C_2 ist.

Zunächst soll gelten:

$$match(l_1, C_2) = \{l_2 \in C_2 | level(sup(l_1, l_2)) = suplevel(l_1, C_2)\}$$

und

$$matchlevel(l_1, C_2) = \min_{l_2 \in match(l_1, C_2)} level(l_2),$$

das *matchlevel* gibt also das Level eines ähnlichsten Elementes an.

Nun kann man ein Ähnlichkeitsmaß (Modified Genealogy Measure Cormode, MCOR) definieren, welches das Ähnlichkeitsmaß COR um das Level eines ähnlichsten Elementes ergänzt:

$$contrib_{MCOR}(l_1, C_2) = \frac{1}{2}(level(l_1) + matchlevel(l_1, C_2) - 2 \cdot suplevel(l_1, C_2)).$$

Die Ähnlichkeit wird wie zuvor über die Beiträge der Elemente berechnet:

$$sim_{MCOR}(C_1, C_2) = 1 - \sum_{l_1 \in C_1} contrib_{MCOR}(l_1, C_2) - \sum_{l_2 \in C_2} contrib_{MCOR}(l_2, C_1).$$

Vergleicht man die Mengen C_2 und C_3 aus Abbildung 5.3 mit dem Ähnlichkeitsmaß COR, erhält man Ähnlichkeit eins, denn $sup($`/FILESYS`, `/FILESYS/open`$) =$ `/FILESYS`, daher ist $suplevel($`/FILESYS`$, C_2) = level($`/FILESYS`$) = 1$. Daher gilt $contrib_{COR}($`/FILESYS`$, C_2) = 0$.

Dagegen berücksichtigt das Ähnlichkeitsmaß MCOR, dass sich das ähnlichste Element zu `/FILESYS` in C_2 von `/FILESYS` unterscheidet. Es gilt $matchlevel($`/FILESYS`$, C_2) = 2$ und $contrib_{MCOR}($`/FILESYS`$, C_2) = \frac{1}{2}(1 + 2 - 2) = \frac{1}{2}$. Tabelle 5.7 zeigt die suplevel-Werte und die Ähnlichkeitsbeiträge der Elemente der Mengen C_2 und C_3 für verschiedene Ähnlichkeitsmaße.

	Element	level	suplevel	match	OGM	COR	MCOR	MOGM
C_2	`/FILESYS/open`	2	2	2	1	0	0	1
	`/PROCMEM/clone`	2	2	2	1	0	0	1
C_3	`/FILESYS/open`	2	2	2	1	0	0	1
	`/FILESYS`	1	1	2	1	0	$\frac{1}{2}$	$\frac{3}{4}$
	`/PROCMEM/clone`	2	2	2	1	0	0	1
Ähnlichkeit C_2 und C_3:					1	1	$\frac{1}{2}$	$\frac{19}{20}$

Tabelle 5.7: Übersicht der level-, suplevel- und matchlevel-Werte und Ähnlichkeitsbeiträge der Elemente aus C_2 und C_3 für verschiedene Ähnlichkeitsmaße

5.2.5 Modified Optimistic Genealogy Measure (MOGM)

In ähnlicher Weise kann man ein Ähnlichkeitsmaß MOGM definieren, welches das Ähnlichkeitsmaß OGM um das Level eines ähnlichsten Elementes ergänzt:

$$contrib_{MOGM}(l_1, C_2) = \begin{cases} 1 \text{ für } (l_1 = *) \wedge (* \in C_2) \\ \frac{1}{2} \text{ für } (l_1 = *) \text{ XOR } (matchlevel(l_1, C_2) = 0) \\ \frac{1}{2}\left(\frac{suplevel(l_1,C_2)}{level(l_1)} + \frac{suplevel(l_1,C_2)}{matchlevel(l_1,C_2)}\right) \text{ sonst.} \end{cases}$$

Die Ähnlichkeit der Mengen C_1 und C_2 wird wie zuvor definiert als durchschnittlicher Ähnlichkeitsbeitrag ihrer Elemente:

$$sim_{MOGM}(C_1, C_2) = \frac{\sum_{l_1 \in C_1} contrib_{MOGM}(l_1, C_2) + \sum_{l_2 \in C_2} contrib_{MOGM}(l_2, C_1)}{|C_1| + |C_2|}.$$

Die Festlegung von $contrib_{MOGM}(l_1, C_2)$ für $(l_1 = *) \wedge (* \in C_2)$ und $(l_1 = *)$ XOR $(matchlevel(l_1, C_2) = 0)$ vermeidet Divisionen durch null. Für $(l_1 = *) \wedge (* \in C_2)$ ist der Wert 1 angemessen, weil das zu l_1 ähnlichste Element in C_2 identisch zu l_1 ist. Für $(l_1 = *)$ XOR $(matchlevel(l_1, C_2) = 0)$ ist der Wert $\frac{1}{2}$ angemessen, weil dann l_1 mit den ähnlichsten Elementen in C_2 vergleichbar ist. Die $contrib_{MOGM}$-Werte bei Vergleichbarkeit auf niedrigeren Ebenen legt den Wert $\frac{1}{2}$ für die oberste Ebene nahe.

Vergleicht man die Mengen C_2 und C_3 aus Abbildung 5.3 mit dem Ähnlichkeitsmaß OGM, erhält man Ähnlichkeit eins, denn $suplevel(\texttt{/FILESYS}, C_2) = level(\texttt{/FILESYS}) = 1$. Also gilt $contrib_{OGM}(\texttt{/FILESYS}, C_2) = 1$.

Dagegen berücksichtigt das Ähnlichkeitsmaß MOGM, dass sich das ähnlichste Element zu `/FILESYS` in C_2 von `/FILESYS` unterscheidet. Es gilt $matchlevel(\texttt{/FILESYS}, C_2) = 2$ und $contrib_{MOGM}(\texttt{/FILESYS}, C_2) = \frac{1}{2}(1 + \frac{1}{2}) = \frac{3}{4}$.

5.2.6 Ähnlichkeitsmaße für die Datenstrukturen (DSM)

Die Hierarchical Heavy Hitters sind eine anschauliche kondensierte Beschreibung des Datenstroms. Sie werden anhand eines Parameters ϕ aus der gespeicherten Datenstruktur des HHH-Algorithmus nach Verarbeitung des Datenstroms berechnet. Mit einer solchen Datenstruktur lassen sich die HHH-Mengen für beliebige Werte $\phi > \varepsilon$ berechnen. Man kann also sagen, dass die eigentliche kondensierte Beschreibung des Stroms nicht die HHH-Menge ist, sondern die Datenstruktur selbst. Ähnlichkeit von Datenströmen kann also auch als Ähnlichkeit der Datenstrukturen, die sie erzeugen, verstanden werden. Die Ähnlichkeit über die Datenstruktur und nicht über die HHH-Mengen zu definieren, hat zwei Vorteile:

- Die mehr oder weniger willkürliche Wahl des Parameters ϕ entfällt,
- und ein möglicher Informationsverlust bei der Berechnung der HHH-Mengen wird vermieden.

Mit der Datenstruktur können die absoluten Häufigkeiten *aller Präfixe p aller Stromelemente* geschätzt werden. Da diese Schätzungen in gewissen Schranken fehlerhaft sein dürfen, ist dazu nur eine relativ kleine Datenstruktur erforderlich. Insbesondere ist es für die meisten Präfixe im Rahmen der Fehlerschranken zulässig, die Häufigkeit auf null zu schätzen. Die Datenstruktur speichert die geschätzte Häufigkeit der Präfixe der Stromelemente allerdings nicht explizit. Gespeichert wird für eine kleine Anzahl von Präfixen p der Wert g_p (der sich interpretieren lässt als der Anteil an der Schätzung der Häufigkeit von p, der nicht bereits durch Nachfahren von p abgedeckt ist). In der Output-Methode des Algorithmus (siehe Algorithmus 1, Seite 24, Output-Methode, Zeile 6) werden die g_p-Werte zu den geschätzten Häufigkeiten f_p zusammengesetzt. Man könnte die Ähnlichkeit von zwei Datenstrukturen direkt über die g_p-Werte definieren. Ich verwende stattdessen die Output-Methode zum Zusammensetzen der geschätzten Häufigkeiten der Präfixe (nur für $f_p > 0$ muss tatsächlich gerechnet werden) und definiere die Ähnlichkeit von Datenstrukturen über die Ähnlichkeit der geschätzten Häufigkeiten der Präfixe.

Die Ähnlichkeit der Datenstrukturen D_1 und D_2 nach Verarbeitung der Datenströme S_1 und S_2 soll berechnet werden. Sei f_p^i die mittels D_i geschätzte Häufigkeit von Präfix p in Strom S_i und P_i die Menge der Präfixe p von Elementen des Stroms S_i, für die $f_p^i > 0$.

Dann definiere ich für $p \in P_1 \cap P_2$

$$contrib_{DSM}(p) = \frac{2 \cdot \min(f_p^1, f_p^2)}{\min(f_p^1, f_p^2) + \max(f_p^1, f_p^2)}$$

und für die Ähnlichkeit der Datenstrukturen

$$sim(D_1, D_2) = \frac{\sum\limits_{p \in P_1 \cap P_2} contrib_{DSM}(p)}{|P_1 \cup P_2|}.$$

Die Ähnlichkeit der Datenstrukturen kann über die Schnittmenge berechnet werden, weil die hierarchische Information bereits in den geschätzten Häufigkeiten der Präfixe enthalten ist.

5.3 Datengenerierung

Ursprünglich sollten die Betriebssystemdaten zur Durchführung der Experimente vom Lehrstuhl Informatik 12 der TU Dortmund zur Verfügung gestellt werden. Die Daten sollten die Betriebssystemaufrufe von verschiedenen Benutzern aus verschiedenen Benutzergruppen enthalten. Mit diesen Daten sollte untersucht werden, ob es möglich ist, Benutzer anhand ihrer Betriebssystemdaten zu unterscheiden und Benutzergruppen zuzuordnen.

Da das Erzeugen dieser Daten aufwändiger als erwartet war, standen die Daten des Lehrstuhls 12 erst relativ spät zur Verfügung. Ich habe daher ersatzweise eigene Datensätze erzeugt. Diese Entscheidung erwies sich als richtig, denn die letztlich vom Lehrstuhl 12 zur Verfügung gestellten Daten enthalten nur zwei von 319 Systemaufrufen und sehr wenige Ansatzpunkte für eine Extraktion der hierarchischen Variablen, mit denen die HHH-Algorithmen arbeiten. Der Schwerpunkt der Experimente liegt daher bei den Experimenten mit den selbst erzeugten Daten (Kapitel 8), die Experimente mit den Daten des Lehrstuhls 12 werden in Kapitel 9 dargestellt.

5.3.1 Datenerfassung mit strace

Die Werkzeuge für die Erfassung von Betriebssystemaufrufen lassen sich in zwei Gruppen aufteilen. Werkzeuge wie trace, truss, ktrace und strace dienen der Protokollierung einzelner Prozesse oder Anwendungen und werden zur Suche nach Fehlern in den Anwendungen verwendet. Werkzeuge wie auditctl können dagegen das Gesamtsystem überwachen und werden als Intrusion Detection System (IDS) zur Entdeckung von Einbrüchen und unautorisiertem Verhalten regulärer Benutzer eingesetzt.

Beiden Arten von Werkzeugen ist gemeinsam, dass sie die Ausführungsgeschwindigkeit der überwachten Anwendungen erheblich beeinträchtigen können. IDS-Systeme werden daher in der Regel so konfiguriert, dass sie nur eine vergleichsweise kleine Menge sicherheitsrelevanter Vorgänge protokollieren. Für die Anwendung der HHH-Algorithmen ist es dagegen vorteilhaft, alle Systemaufrufe einschließlich ihrer Parameter zu protokollieren, um sie (nach Erstellung einer geeigneten Taxonomie) als hierarchische Variablen verwenden zu können.

Der Plan, alle Betriebssystemaufrufe aller Benutzer am Lehrstuhl 8 mit auditctl zu protokollieren, erwies sich allerdings als nicht durchführbar. Die Zeit für den Start der Anwendung RapidMiner stieg beim Erfassen aller Systemaufrufe auf einem Rechner mit zwei AMD Dual-Core Opteron 2220 Prozessoren und 26.624 MB Hauptspeicher unter Linux von 27 Sekunden auf etwa neun Minuten, die Reaktionszeiten wurden so lang, dass eine Bedienung kaum noch möglich war. Die Daten wurden auf dem überwachten Rechner selbst erfasst, möglicherweise ist die Leistungsbeeinträchtigung geringer, wenn die Daten auf einem anderen Rechner erfasst werden.

Stattdessen habe ich einzelne Anwendungen mit dem Werkzeug strace protokolliert. Strace kann die Systemaufrufe einzelner Prozesse aufzeichnen und gibt dabei auch die

Anwendung	Version	Funktion	Abkürzung
Firefox	3.0.15	Webbrowser	ff
top	3.2.7	Anzeige laufender Prozesse	tp
Rhythmbox	0.12.0	Audiospieler	rb
Geyes	2.26.1	Wachsame Augen	gy
NEdit	5.5	Editor	ne
Vinagre	2.26.1	VNC-Software	vg
XEmacs	21.4.21	Editor	xe
Kate	3.2.2	Editor	ka
xterm	241	Terminal-Emulator	xt
Tomboy	0.14.0	Editor für Notizen	tb
Epiphany	2.26.1	Webbrowser	ep

Tabelle 5.8: Liste der Anwendungen, deren Systemaufrufe protokolliert wurden

jeweils übergebenen Parameter und Rückgabewerte aus. Strace protokolliert bei Angabe der Option `-f` auch die Betriebssystemaufrufe aller Kindprozesse eines Prozesses, so dass alle Prozesse einer Anwendung gleichzeitig überwacht werden können. Tabelle 2.2 zeigt eine Ausgabe von strace. Angegeben ist jeweils die Prozess-ID, der Name des Systemaufrufs, die Parameter und der Rückgabewert. In Kapitel 2 wird die Bedeutung der einzelnen Zeilen der Tabelle ausführlich erläutert.

Auch bei der Verwendung von strace reagieren die überwachten Anwendungen träge. Da jedoch jeweils nur eine Anwendung und nicht das Gesamtsystem überwacht wird, ist in den meisten Fällen eine Bedienung der Anwendung noch möglich. Bei umfangreichen Anwendungen wie RapidMiner, Eclipse und gedit traten allerdings zumindest vereinzelt Probleme auf, so dass diese Anwendungen nicht überwacht werden konnten.

In Tabelle 5.8 sind die Anwendungen aufgeführt, die protokolliert wurden. Jede Anwendung wurde zehnmal ausgeführt, fünfmal wurde fünf Minuten und fünfmal zehn Minuten lang protokolliert. Verwendet wurde strace in der Version 4.5.17 auf Ubuntu, Kernel 2.6.27, 32 Bit.[3] Ubuntu lief unter VirtualPC 2007 unter Windows XP. Strace wurde jeweils mit Option `-f` gestartet, so dass alle Prozesse der Anwendung überwacht wurden. Die einhundert erzeugten Dateien umfassen etwa 23 Millionen Zeilen. Die Zahl der Systemaufrufe liegt unwesentlich niedriger, da gelegentlich auch Signale für die Interprozesskommunikation protokolliert werden und weil blockierende Systemaufrufe zwei Zeilen beanspruchen (siehe Abschnitt 5.1.1). Die einzelnen Aufrufe werden unterschiedlich häufig verwendet. In Abbildung 5.4 sind die Häufigkeiten der Aufrufe im Anwendungsdatensatz dargestellt. Die Gesamtgröße aller Logdateien liegt bei 1.8 GiB. Die Daten sind stark redundant, die Größe der komprimierten Daten liegt bei 101 MiB (gzip) bzw. 60 MiB (bzip2).

[3] Auf die Herausforderungen von 64-Bit Betriebssystemen ist das schon recht betagte strace offenbar unzureichend vorbereitet: Es kommt zu Abstürzen, bei einigen Systemaufrufen wird inkonsistent zwischen dezimaler und hexadezimaler Darstellung gewechselt und anstelle des Wertes -1 wird $2^{32}-1$ ausgegeben.

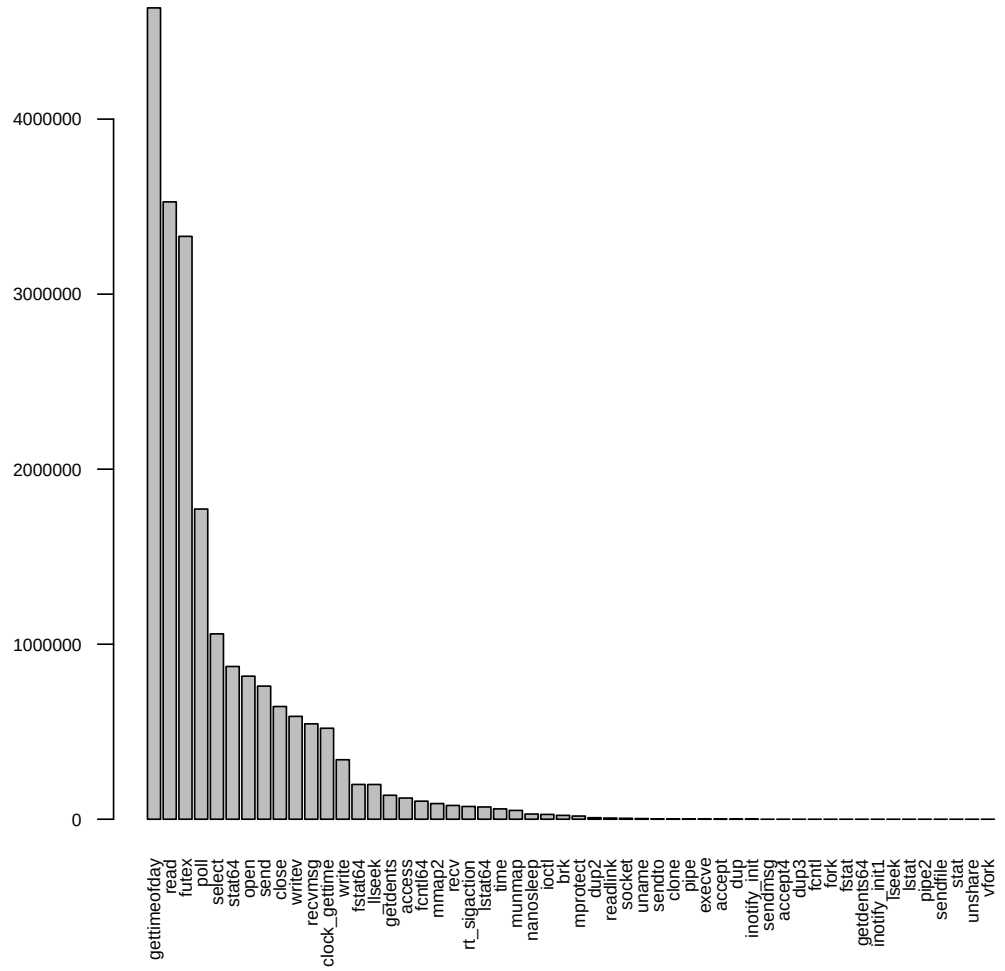


Abbildung 5.4: Häufigkeiten der Systemaufrufe im Anwendungsdatensatz

5.3.2 Firefox und Epiphany

Zunächst umfassten die protokollierten Daten nur zehn Anwendungen, darunter die Webbrowser Firefox und Epiphany; Tomboy fehlte noch. Die HHH-Mengen von Firefox und Epiphany zeigten bei den Experimenten deutliche Auffälligkeiten, sie wurden seltener korrekt klassifiziert als andere Anwendungen. Häufig wurden HHH-Mengen von Firefox-Logdateien Epiphany zugeordnet und umgekehrt. In Tabelle 5.9 ist der Grund für dieses Verhalten zu sehen: Die HHH-Mengen von Firefox und Epiphany sind einander ähnlicher, als dies für andere Anwendungspaare der Fall ist. Die Tabelle zeigt die durchschnittlichen Ähnlichkeiten aller Anwendungspaare unter Verwendung des Ähnlichkeitsmaßes MOGM, bei Verwendung anderer Ähnlichkeitsmaße ergibt sich ein ähnliches Bild. Für jeden Eintrag in der Tabelle wurden alle Paare von Logdateien der beiden Anwendungen verglichen und der Durchschnitt gebildet. Die Hauptdiagonale enthält daher Durchschnittswerte über 45 Paare (verschiedene Paare der zehn Logdateien einer Anwendung), die übrigen Einträge sind Durchschnittswerte über 100 Paare.[4]

Bei der Suche nach Gründen für diese Auffälligkeit stellte sich heraus, dass Firefox und Epiphany im Wesentlichen identisch sind: Kernaufgabe für Webbrowser ist die Darstellung von Webseiten. Firefox und Epiphany benutzen für diese Darstellung der Seiten die Rendering Engine Gecko in derselben Version 1.9, es wird also überwiegend derselbe Code ausgeführt. Die Anwendungen sind anhand ihrer HHH-Mengen entsprechend schlecht zu unterscheiden.

Für die Entscheidung über den weiteren Umgang mit Firefox und Epiphany war die folgende Überlegung ausschlaggebend: Die Daten für die Experimente sollten der tatsächlichen Verteilung entsprechen. Wenn es verschiedene Anwendungen gibt, die denselben Kern besitzen, sollte dieser Sonderfall auch in den Daten vertreten sein. Der Sonderfall sollte allerdings in den Daten mit derselben Häufigkeit auftreten wie in der zugrunde liegenden unbekannten Verteilung. Meiner Einschätzung nach ist es eher ungewöhnlich, dass verschiedene Anwendungen denselben Kern besitzen, der Sonderfall wäre dann in den Daten deutlich überrepräsentiert. Nach Entfernen des Sonderfalls aus den Daten wäre er leicht unterrepräsentiert, das entspräche der tatsächlichen Verteilung eher als eine deutliche Überrepräsentierung. Daher schien es mir vertretbar, Epiphany aus dem Datensatz zu entfernen und durch Tomboy zu ersetzen.

Es ist nicht unproblematisch, die Daten in dieser Weise zu bereinigen, daher wäre es wünschenswert gewesen, die Experimente jeweils zweifach auszuführen, einmal mit den bereinigten Daten und einmal mit dem Datensatz, der Firefox und Epiphany enthält. Wegen der erheblichen erforderlichen Rechenzeiten und des begrenzten Zeitrahmens der Diplomarbeit war dies nicht möglich.

Die auffällige Ähnlichkeit der HHH-Mengen von Firefox und Epiphany ist ein interessantes erstes Ergebnis: Sie zeigt, dass die Ähnlichkeiten von HHH-Mengen zumindest in manchen Fällen geeignet sein können, die Ähnlichkeit von Anwendungen zu messen. Die auffällige Ähnlichkeit der HHH-Mengen von Firefox und Epiphany deckte eine mir

[4] Als hierarchische Variablen wurden Pfade und Betriebssystemaufrufe verwendet, siehe dazu Abschnitt 5.1.

	gy	tp	ne	rb	xe	xt	ka	tb	vg	ff	ep
gy	1.00	0.29	0.46	0.47	0.39	0.30	0.37	0.32	0.33	0.35	0.37
tp	0.29	0.99	0.57	0.57	0.60	0.73	0.72	0.70	0.71	0.72	0.73
ne	0.46	0.57	0.90	0.73	0.77	0.67	0.74	0.73	0.75	0.73	0.74
rb	0.47	0.57	0.73	0.94	0.72	0.65	0.68	0.77	0.82	0.79	0.82
xe	0.39	0.60	0.77	0.72	0.91	0.72	0.77	0.77	0.77	0.75	0.77
xt	0.30	0.73	0.67	0.65	0.72	0.86	0.73	0.76	0.75	0.75	0.76
ka	0.37	0.72	0.74	0.68	0.77	0.73	0.93	0.81	0.77	0.76	0.76
tb	0.32	0.70	0.73	0.77	0.77	0.76	0.81	0.97	**0.91**	0.84	0.84
vg	0.33	0.71	0.75	0.82	0.77	0.75	0.77	**0.91**	0.96	0.87	0.88
ff	0.35	0.72	0.73	0.79	0.75	0.75	0.76	0.84	0.87	0.92	**0.86**
ep	0.37	0.73	0.74	0.82	0.77	0.76	0.76	0.84	0.88	**0.86**	0.93

Tabelle 5.9: Durchschnittliche Ähnlichkeiten der HHH-Mengen verschiedener Anwendungen

zuvor nicht bekannte Ähnlichkeit der Anwendungen (Verwendung der gleichen Rendering Engine) auf.

Neben Firefox und Epiphany fällt in der Tabelle ein weiteres Paar ähnlicher Anwendungen auf: Vinagre und Tomboy. Tomboy ist ein Editor zur Verwaltung von Notizen und Vinagre ein VNC-Client, mit dem auf die grafische Benutzerschnittstelle eines entfernten Rechners zugegriffen werden kann. Die beiden Anwendungen erfüllen also sehr unterschiedliche Aufgaben. Allerdings verwenden beide die *GNOME development platform*, ein Framework, das vom GNOME Project[5] zur vereinfachten Erstellung von Anwendungen für die Arbeitsumgebung GNOME bereitgestellt wird. Es kann vermutet werden, dass die auffällige Ähnlichkeit auch hier auf der Benutzung derselben Bibliotheken basiert. Die Verwendung eines solchen Frameworks ist nicht ungewöhnlich, daher wurden die Anwendungen nicht aus dem Datensatz entfernt.

[5] `http://www.gnome.org`

6 Implementierung

In Kapitel 5 wurden die grundlegende Vorgehensweise zur Extraktion der hierarchischen Variablen und die Ähnlichkeitsmaße zum Vergleich von Mengen von Hierarchical Heavy Hitters vorgestellt. Gemeinsam mit den Algorithmen zur Berechnung der HHH-Mengen (Kapitel 3) bilden sie die Grundbausteine des Systems zu Datenaggregation von Betriebssystemdaten durch Hierarchical Heavy Hitters. Abbildung 6.1 zeigt das System im Überblick.

In den folgenden Abschnitten wird die Implementierung dieser Bausteine dargestellt. Die Implementierung erfolgte in Java und wurde in das Data Mining Werkzeug RapidMiner integriert. Zunächst wird das Einlesen der Daten und die Simulation des Betriebssystems beschrieben, die zur Extraktion der hierarchischen Variablen erforderlich ist. Anschließend wird die Implementierung der Algorithmen zur Berechnung der HHH-Mengen skizziert, dabei wird auch ein Mechanismus zum Puffern der Ergebnisse vorgestellt, der die erforderlichen Rechenzeiten für die Experimente in Kapitel 8 erheblich reduziert hat. Zuletzt wird die Integration des Systems in RapidMiner beschrieben. Durch die Implementierung der Schnittstellen von RapidMiner konnte die breite Palette von Werkzeugen und Algorithmen, die RapidMiner bereitstellt, für die Durchführung der Experimente in Kapitel 8 und 9 verwendet werden.

6.1 Einlesen, Simulieren und Extrahieren

In Tabelle 2.2 ist ein Beispiel für die Daten abgebildet, die strace erzeugt und die vom System eingelesen und verarbeitet werden müssen. Die Daten liegen nicht in einer für Maschinen leicht lesbaren Form wie XML- oder JSON-Dateien vor, sondern in einer Form, die offenbar für menschliche Leser gedacht ist. Für das Einlesen konnten daher keine Standardbibliotheken verwendet werden.

Das Einlesen erfolgt zeilenweise für jeweils einen Aufruf. Neben dem Namen des Aufrufs werden auch die übergebenen Parameter und Rückgabewerte eingelesen. Anzahl und Art der Parameter sind vom Systemaufruf abhängig, die Parameter können einfache Werte oder auch komplexere zusammengesetzte Werte wie Strukturen von Feldern und Zeichenketten sein. Das Einlesen der Aufrufe erfolgt daher über ein zweischichtiges Verfahren: Auf der unteren Ebene wird über reguläre Ausdrücke die Funktionalität zur Ermittlung der Rückgabewerte und einer Liste der Parameter des Aufrufs bereitgestellt. Zusammengesetzte Parameter sind zu diesem Zeitpunkt noch nicht aufgelöst und bilden einen einzelnen Wert. Auf der oberen Ebene kann damit für die einzelnen Aufrufe der Zugriff auf ihre Parameter abstrakt beschrieben werden. Da auf der Ebene der einzel-

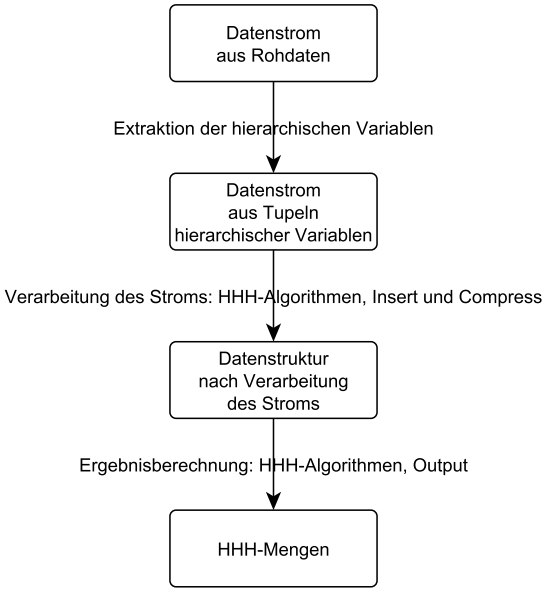


Abbildung 6.1: Überblick Gesamtsystem

nen Aufrufe auch bekannt ist, welche der Parameter zusammengesetzt sind, kann für diese Parameter wiederum eine Auflösung in eine Werteliste (durch die Funktionalität der unteren Ebene) veranlasst werden. Sofern auch in dieser Liste einzelne Werte zusammengesetzt sind, können diese wiederum durch die untere Ebene in eine Werteliste aufgelöst werden.

Nachdem die Parameter des einzelnen Aufrufs ermittelt wurden, kann die Simulation des Teils des Betriebssystems erfolgen, der für die Verwaltung der Dateideskriptoren zuständig ist. Dies ist erforderlich, um im Rahmen der Extraktion der hierarchischen Pfadvariablen die Dateideskriptoren wieder in Pfade zurück verwandeln zu können. Für die Simulation wird für jeden einzelnen Prozess eine „Prozesstabelle“ angelegt, in der die offenen Dateien und Deskriptoren verwaltet werden. Darüber hinaus wird in der Prozesstabelle das `FD_CLOEXEC`-Flag der Dateideskriptoren gespeichert. In einem zirkulären Puffer wird zudem jeweils der aktuelle Systemaufruf des Prozesses abgelegt. Wenn ein Systemaufruf gelesen wird, der Auswirkungen auf die Deskriptoren hat (siehe Tabelle 5.2), wird die Prozesstabelle aktualisiert.

Die Extraktion der hierarchischen Variablen ist nach diesen Vorarbeiten sehr einfach: Da Systemaufruf und Parameter ermittelt wurden, kann die hierarchische Aufrufvariable sofort konstruiert werden. Für die Pfadvariable wird, sofern erforderlich, auf die Datenstruktur zum Auflösen der Dateideskriptoren zugegriffen. Für die Sequenzvariable wird der zirkuläre Puffer des aktuellen Prozesses ausgelesen. Zeilen der Logdateien, die Signale enthalten, werden verworfen. Unvollständige Zeilen blockierender Aufrufe (sie-

he Abschnitt 5.1) werden in der Prozesstabelle zwischengespeichert, bis sie zu einem vollständigen Aufruf zusammengesetzt werden können.

Bereits während des Einlesens der Daten lassen sich drei Einstellungen vornehmen:

- Es kann festgelegt werden, dass nur eine Teilmenge der hierarchischen Variablen extrahiert werden soll. Dies wird in Kapitel 7 verwendet, um die Auswirkungen der Dimensionalität auf den Ressourcenverbrauch der Algorithmen zu untersuchen.
- Die Hierarchietiefe der hierarchischen Variablen kann begrenzt werden, so dass z. B. nur noch eine gewisse Anzahl von Komponenten eines Pfades berücksichtigt wird. In Kapitel 7 wird damit die Auswirkung der Hierarchietiefe auf die Laufzeit untersucht.
- Die Menge der verwendeten Aufrufe kann eingeschränkt werden. Nur Zeilen, die Aufrufe aus der Liste der verwendeten Aufrufe enthalten, werden zur Konstruktion der hierarchischen Variablen benutzt, die übrigen Zeilen werden verworfen. Die Möglichkeit zur Einschränkung der verwendeten Aufrufe wird eingesetzt, um im Rahmen der Merkmalsselektion in Kapitel 8 zu ermitteln, welche Aufrufe protokolliert werden sollten, um gute Ergebnisse zu erzielen.

6.2 Berechnung der Hierarchical Heavy Hitters

In Kapitel 3 wurden die Algorithmen zur Berechnung der Hierarchical Heavy Hitters beschrieben. Diese wurden im Rahmen dieser Arbeit in den Varianten Full Ancestry und Partial Ancestry implementiert. Zusätzlich wurde ein exakter Algorithmus (siehe [Cormode et al., 2004]) implementiert, um die Ergebnisqualität der Approximationsalgorithmen messen zu können.

6.2.1 Datenstruktur

Eingabe für die Algorithmen zur Berechnung der Hierarchical Heavy Hitters ist ein Strom von Elementen. Die einzelnen Elemente sind Tupel, deren Komponenten jeweils einer Hierarchie entstammen. Die Algorithmen verwalten Elemente dieses Datenstroms und Generalisierungen von solchen Stromelementen. Beide haben dieselbe interne Darstellung und werden im Folgenden als „Elemente“ bezeichnet. Für eine im Vergleich zur Stromgröße kleine Zahl von Elementen werden Informationen über die Häufigkeit und Hilfsvariablen gespeichert. Diese Informationen werden zu Tupeln zusammengefasst und über Hashtabellen verwaltet, Schlüssel ist jeweils das Element. Die Verwaltung der Elemente erfolgt ebenenweise, jede Ebene des Verbandes erhält eine eigene Hashtabelle. So ist es möglich, während der `Compress`- und `Output`-Methode die Elemente ebenenweise von unten nach oben zu verarbeiten, um sicherzustellen, dass Elemente erst nach ihren Nachfahren bearbeitet werden. Die Reihenfolge der Wahl der Elemente innerhalb einer Ebene ist nicht entscheidend und kann in der Reihenfolge der internen Speicherung der Schlüssel erfolgen. Die Gesamtheit der Elemente und ihrer Tupel bildet die Datenstruk-

tur. Da alle Iterationen über diese Struktur innerhalb einer Ebene oder (bei Zugriffen auf Vorfahren) in Richtung von spezielleren zu generelleren Elementen verlaufen, ist eine explizite Verzeigerung nicht erforderlich, das Tupel eines Vorfahren kann durch Generalisieren des Elementes und anschließenden Zugriff auf die Hashtabelle ermittelt werden. Für die Hashtabellen wurden die Klassen des Java Collections Frameworks und der Google Collections Library verwendet.

Die Elemente enthalten die hierarchischen Variablen, deren interne Darstellung von ihrer Art abhängig ist: Die Pfadvariable wird intern als Zeichenkette dargestellt, die Systemaufrufvariable und die Sequenzvariable werden kompakter als einzelner 32-Bit Wert dargestellt. Für die Sequenzvariable wird dieser Wert in vier Bereiche zu je 8 Bit aufgeteilt, die jeweils einen Aufruf codieren. Für die Systemaufrufvariable werden sechs Hierarchieebenen mit 4, 8, 6, 6, 4 und 4 Bit zugeteilt. Die Bitzahl je Ebene ist von der Anzahl möglicher Werte auf der Ebene abhängig, 4 Bit für die 5 Gruppen von Systemaufrufen enthalten also eine Reserve von einem Bit, 8 Bit für 54 Systemaufrufe enthalten eine Reserve von zwei Bit für mögliche Erweiterungen. Die Aufteilung der 32 Bit auf die einzelnen Hierarchieebenen ist allerdings so flexibel gestaltet, dass auch nachträgliche Änderungen der Aufteilung leicht möglich sind und nur eine einzelne Veränderung an einer Stelle des Quellcodes erfordern. Durch diese kompakte Darstellung der hierarchischen Variablen in einem einzelnen int-Wert sinkt der Speicherbedarf. Darüber hinaus lassen sich so Operationen wie Generalisieren und Berechnung von Infimum und Supremum über schnelle Bitoperationen realisieren.

Die Pfadvariable wird als einfache Zeichenkette dargestellt, Optimierungen wurden nicht implementiert. Durch eine entropiebasierte Codierung der einzelnen Pfadkomponenten[1] ließe sich Speicher sparen. Da der Speicherbedarf der Algorithmen nicht sehr hoch ist (siehe Kapitel 7), könnte es allerdings sinnvoller sein, die Pfadkomponenten in konstanter Länge (z. B. als Integer-Werte) zu codieren. So ließe sich schnell bestimmen, ob zwei Komponenten der Pfadvariablen verschiedener Elemente gleich sind. Folglich wäre der häufig durchgeführte Test, ob ein Element Vorfahr eines anderen Elementes ist, schneller durchführbar als bei einer Darstellung als Zeichenkette.

6.2.2 Optimierungen

Vermeiden unnötiger Einfügungen

Während der Kompression werden beim Partial Ancestry Algorithmus Elemente gelöscht, für die auf den nächsthöheren Ebenen teilweise neue Elemente erzeugt werden (Zeile 12 der `Compress`-Methode, siehe Algorithmus 1). Ein Teil dieser neuen Elemente wird bei der anschließenden Kompression dieser Ebenen sofort wieder gelöscht. Der Versuch, durch Zwischenspeichern während der Kompression nur solche Elemente neu in die Datenstruktur einzufügen, die nicht sofort wieder gelöscht werden und die Information über die Häufigkeiten der übrigen Elemente getrennt zu verwalten, führte nicht zu besseren Ergebnissen und wurde nicht weiter verfolgt.

[1] Pfadkomponenten: Der Pfad `/tmp/log` hat die Komponenten tmp und log.

Puffern der Ergebnisse

Eine bessere Einsparmöglichkeit stellt das Puffern der Ergebnisse der HHH-Algorithmen dar. Bei den Experimenten in Kapitel 8 wird die Klassifikation von Anwendungen bei Verwendung verschiedener Ähnlichkeitsmaße und verschiedener Werte für die Anzahl der Nachbarn k untersucht, die mehrfache Berechnung der jeweils gleichen HHH-Mengen wäre nicht sinnvoll. Es wäre möglich, die Ergebnisse während eines Experiments im Arbeitsspeicher zu halten, etwa durch einen geeigneten Versuchsaufbau in RapidMiner, allerdings wäre auch dann eine gemeinsame Nutzung bereits berechneter Ergebnisse über verschiedene Experimente hinweg nicht möglich. Auch der häufige Fall, dass nach Abschluss eines Experiments dieses in leicht veränderter Form wiederholt werden soll, würde bei einer einfachen Pufferung im Arbeitsspeicher eine vollständige Neuberechnung erfordern. Daher werden die HHH-Mengen in Dateien gespeichert, so dass sie über verschiedene Experimente hinweg nutzbar sind. Die Speicherung in Dateien ist schnell und flexibel: Verwaltet man die Dateien während der Experimente in einer RAM-Disk[2], sind die Zugriffszeiten sehr kurz. Nach Abschluss der Experimente kann dann entschieden werden, ob die Daten in einen nichtflüchtigen Speicher überführt werden sollen. Verschiedene Puffer können bei Bedarf sehr leicht vereinigt werden, indem die Dateien in ein gemeinsames Verzeichnis kopiert werden.

Grundsätzlich kann das Puffern der Ergebnisse der HHH-Algorithmen auf zwei Arten erfolgen: Durch Speichern der Datenstruktur (nach Verarbeitung des Datenstroms) oder durch Speichern der HHH-Mengen. Das Speichern der Datenstruktur hat den Vorteil, dass bei festem ε die Verwendung derselben Datei für verschiedene ϕ-Werte möglich ist: Die Berechnung der Datenstruktur ist abhängig von ε (so dass für verschiedene ε verschiedene Datenstrukturen gespeichert werden müssen), aber unabhängig von ϕ. Bei gegebener Datenstruktur können die HHH-Mengen für beliebige ϕ mit der relativ schnellen Ausgabemethode berechnet werden. Für Experimente, bei denen für einen ε-Wert verschiedene ϕ-Werte untersucht werden, ist das von Vorteil. Nachteil der Speicherung der Datenstruktur ist der im Vergleich zu den HHH-Mengen höhere Platzbedarf. Da mir am Lehrstuhl nur 3 GiB Festplattenplatz zur Verfügung standen und der hohe Speicherbedarf auch die Verwendung der RAM-Disk erschwert hätte, habe ich stattdessen die HHH-Mengen gespeichert. Für jede Logdatei wird für jeden ϕ- und ε-Wert also eine eigene Pufferdatei angelegt. Da nur in relativ wenigen Experimenten für ein gegebenes ε verschiedene ϕ-Werte untersucht wurden, war der Nachteil im Vergleich zur Speicherung der Datenstruktur gering. Allerdings bedeutet die Speicherung der HHH-Mengen, dass bei Verwendung des datenstrukturbasierten Ähnlichkeitsmaßes DSM immer die Ausführung der HHH-Algorithmen erforderlich ist.

6.3 Integration in RapidMiner

RapidMiner ([Mierswa et al., 2006]) ist ein erweiterbares Werkzeug zur Durchführung von Data Mining Experimenten. Über eine grafische Benutzerschnittstelle können Expe-

[2] Zum Beispiel `/dev/shm` unter Linux

rimente aus einer großen Zahl von Operatoren zusammengesetzt werden. Die Operatoren decken verschiedene Aufgabenbereiche wie die Ein- und Ausgabe, Vorverarbeitung, Modellbildung und Evaluation ab. RapidMiner ist Open-Source-Software und leicht erweiterbar: Über Schnittstellen lassen sich eigene Komponenten in RapidMiner integrieren, so dass die breite Palette von Werkzeugen und Algorithmen, die RapidMiner bereitstellt, für Experimente mit den selbst erstellten Komponenten voll nutzbar ist.

Der Aufbau der Experimente wird im XML-Format gespeichert, damit ist ein Standard für den Austausch von Data Mining Experimenten gegeben.

6.3.1 Repräsentation der Daten

Normalerweise werden Beispiele für Lernverfahren in RapidMiner als Merkmalsvektoren fester Länge verwaltet. Für jedes Merkmal wird die Merkmalsausprägung des Beispiels intern als double-Wert gespeichert. Für nominale Attribute, deren Wert sich nicht unmittelbar als double-Wert speichern lässt, werden die Attributwerte in einer Liste verwaltet. Der double-Wert dient als Index für den Zugriff auf die Attributwerte der Liste. Neben den Merkmalsvektoren wird gegebenenfalls auch die Klasse des Beispiels gespeichert. Eine Menge solcher Beispiele bildet ein „Exampleset“ und ist die wesentliche Eingabe für die Lernverfahren. Ein Vektor von Attributgewichten kann zusätzlich gespeichert werden, um die unterschiedliche Bedeutung der einzelnen Attribute für das Lernverfahren zu modellieren. Dieser Vektor wird auch für die Merkmalsselektion verwendet.

Diese Form der Verwaltung lässt sich auf die Logdaten, die strace erzeugt, nicht unmittelbar anwenden. Zwar lässt sich die Klasse leicht angeben: Sie ist für die Experimente aus Kapitel 8 der Name der Anwendung. Ein Merkmalsvektor konstanter Länge ist dagegen nicht erkennbar: Vor der Ausführung der HHH-Algorithmen entspricht ein Beispiel einer vollständigen Logdatei („rohes Beispiel“), nach der Zusammenfassung des Datenstroms entspricht ein Beispiel einer HHH-Menge oder einer Datenstruktur, die ebenfalls eine Menge von Präfixen von Stromelementen ist („extrahiertes Beispiel“). Weder Logdateien noch HHH-Mengen oder Datenstruktur haben eine konstante Größe und es ist nicht unmittelbar ersichtlich, wie sie sich als Merkmalsvektoren konstanter Länge darstellen lassen.

Die Lösung dieses Problems gestaltet sich für rohe und extrahierte Beispiele unterschiedlich.

Die rohen Beispiele vor der Ausführung der Algorithmen erhalten jeweils ein einzelnes Attribut, das den Dateinamen der Logdatei enthält. Es ist weder möglich (Speicherbedarf) noch erforderlich, die Logdateien in ausführlicherer Form in RapidMiner darzustellen, denn sie werden nur zur Berechnung der HHH-Mengen benötigt. Für deren Berechnung wurde ein Extraktionsoperator (siehe Abschnitt 6.3.2 für eine ausführlichere Beschreibung) implementiert, der für jedes Beispiel den Dateinamen ausliest und anhand der Datei die HHH-Mengen berechnet. Ausgabe des Operators ist wieder eine Menge von Beispielen, die diesmal nicht die Logdateien, sondern die unterschiedlich großen HHH-Mengen enthalten.

Auch für diese extrahierten Beispiele ergibt sich das Problem, dass eine sinnvolle Darstellung als Merkmalsvektor konstanter Länge nicht offensichtlich ist. Da auf diese Daten auch von anderen Operatoren zugegriffen werden soll, ist eine indirekte Speicherung über einen Dateinamen nicht möglich. Die Vorgehensweise bei der Speicherung der HHH-Mengen ähnelt daher dem Fall der nominalen Attribute: Die HHH-Mengen werden in einer Liste gespeichert. Für die Beispiele wird ein einzelnes neues Attribut erzeugt, das für die Speicherung der HHH-Mengen verwendet wird. Der double-Wert, der für das einzelne Beispiel den Wert dieses Attributs speichert, wird als Index verwendet, über den auf die in der Liste gespeicherte HHH-Menge zugegriffen wird. Über diesen Mechanismus („ObjectAttribute“) können beliebige Objekte mittels Mapping durch einen Attributwert dargestellt werden. Darum kann in derselben Weise die Datenstruktur über ein weiteres Attribut gespeichert werden. Die Implementierung dieses Mechanismus stammt von [Jungermann, 2009].

6.3.2 Operatoren

Über die selbst erstellten Operatoren werden die Methoden zur Extraktion der hierarchischen Variablen und zur Berechnung der Hierarchical Heavy Hitters in RapidMiner eingebunden, so dass die breite Palette von Werkzeugen und Algorithmen, die RapidMiner bereitstellt, für Experimente mit diesen Methoden voll nutzbar wird.

HHHExtractionPlain

Der wichtigste Operator für Experimente mit strace-Daten ist HHHExtractionPlain, der sowohl die Extraktion der hierarchischen Variablen als auch die Berechnung der HHH-Mengen realisiert. Eingabe des Operators ist eine Menge roher Beispiele, Ausgabe ist eine Menge extrahierter Beispiele. Die folgenden Parameter für die Extraktion der hierarchischen Variablen und die Berechnung der Hierarchical Heavy Hitters sind über die grafische Benutzerschnittstelle einstellbar:

- Der Algorithmus: Full Ancestry oder Partial Ancestry
- ε und ϕ. Für die wenig sinnvolle Einstellung $\phi < \varepsilon$ wird $\phi = \varepsilon$ gesetzt und eine Warnung ausgegeben.
- Die zu verwendenden hierarchischen Variablen (z. B. use_path für die hierarchische Pfadvariable) und die Hierarchietiefe (z. B. path_cap = 8 für die Begrenzung der Pfadvariable auf Hierarchietiefe 8).
- Ein- und Ausschalten der Pufferung (siehe Abschnitt 6.2) und ggf. Angabe des Verzeichnisses für die Pufferung. Durch die Wahl des Verzeichnisses wird auch festgelegt, ob der Puffer persistent oder in der RAM-Disk verwaltet wird.
- Result: Wenn dieser Parameter eingeschaltet ist, gibt der Operator zusätzlich zur Menge der extrahierten Beispiele ein sogenanntes ResultObject aus, das die HHH-Mengen enthält und ihre Darstellung in einer Tabelle ermöglicht. Durch die Mög-

lichkeit, die Tabelle auf verschiedene Arten zu sortieren, können die HHH-Mengen genauer untersucht werden.

- write_data_struct legt fest, ob neben den HHH-Mengen auch die Datenstrukturen in die extrahierten Beispiele geschrieben werden. Dies ist erforderlich, wenn das datenstrukturbasierte Ähnlichkeitsmaß DSM verwendet werden soll.

HistoExtraction

Der Operator HistoExtraction arbeitet ebenfalls auf einer Menge roher Beispiele, zählt aber in jeder Logdatei lediglich die Häufigkeit der verschiedenen Aufrufe. Ausgabe ist für jedes Beispiel ein Vektor, der für jeden Systemaufruf die relative Häufigkeit des Aufrufs darstellt. Diese Repräsentation von Logdateien ähnelt der Textrepräsentation durch Vektoren von Worthäufigkeiten (Term Frequency), die in der Dokumentenklassifikation und im Information Retrieval üblich ist und ist vergleichbar zu der kondensierten Darstellung von Systemaufrufen, die [Liao und Vemuri, 2002] beschreiben (siehe Abschnitt 4.3).

Über diese Vektoren und verschiedene Ähnlichkeitsmaße (im einfachsten Fall Skalarprodukt oder euklidischer Abstand) lassen sich Ähnlichkeiten von Logdateien berechnen, die genau wie die HHH-basierten Ähnlichkeiten von k-NN-Algorithmen zur Klassifikation verwendet werden können. Dieser Operator wurde für die Experimente in Kapitel 8 als einfache Vergleichsmethode verwendet, um den Klassifikationsfehler der HHH-basierten Algorithmen besser einordnen zu können.

Der Operator benötigt keine Parameter.

HHHExtractionUser

Die Betriebssystemdaten des Lehrstuhls Informatik 12 der TU Dortmund (siehe Kapitel 9 für eine ausführlichere Beschreibung) unterscheiden sich von den mit strace erzeugten Daten, so dass für die Berechnung der HHH-Mengen für diese Daten ein eigener Operator erforderlich ist. Bei den strace-Daten entspricht eine Logdatei einem Beispiel, die Datei enthält nur Aufrufe einer einzelnen Anwendung. Bei den Daten vom Lehrstuhl 12 habe ich nicht Anwendungen, sondern Benutzer und Benutzergruppen klassifiziert. Die Daten der Benutzer liegen nicht in einzelnen Dateien, sondern alle Aufrufe aller Benutzer sind Bestandteil eines gemeinsamen Stroms, der wegen seiner Größe auf 437 Dateien aufgeteilt wurde. Diese Aufteilung ist sequenziell, in allen Dateien können Aufrufe aller Benutzer auftreten. Da die Daten einen anderen Aufbau als die strace-Daten haben, müssen auch die rohen Beispiele anders dargestellt werden. Das Verzeichnis der 437 Dateien ist für alle Beispiele gleich und wird nur einmalig über die grafische Benutzerschnittstelle angegeben, die rohen Beispiele dienen lediglich dazu festzulegen, für welche Benutzer die HHH-Mengen berechnet werden sollen. Über die Angabe der Klasse (label) kann zudem festgelegt werden, ob einzelne Benutzer oder Benutzergruppen erkannt werden sollen.

Der Operator führt für jeden in den rohen Beispielen genannten Benutzer eine Instanz des HHH-Algorithmus aus und gibt das Ergebnis als Menge extrahierter Beispiele aus.

Der Aufbau dieser extrahierten Beispiele unterscheidet sich nicht mehr vom Aufbau der extrahierten strace-Beispiele.

Parameter des Operators sind die Werte ε und ϕ, der HHH-Algorithmus und das Verzeichnis der 437 Logdateien.

6.3.3 Ähnlichkeitsmaße

Neben den einfachen Ähnlichkeitsmaßen Precision, Recall, Jaccard und Dice wurden die hierarchischen Ähnlichkeitsmaße Optimistic Genealogy Measure (OGM), Cormode (COR), Modified Optimistic Genealogy Measure (MOGM), Modified Genealogy Measure Cormode (MCOR) und Datastructure Similarity Measure (DSM) implementiert (siehe Abschnitt 5.2). Die Ähnlichkeitsmaße lassen sich einerseits zur Bestimmung der k nächsten Nachbarn einer HHH-Menge für die Klassifikation verwenden. Zum anderen kann mit ihnen auch die Ähnlichkeit der HHH-Mengen der Approximationsalgorithmen mit den exakten HHH-Mengen (und damit die Approximationsgüte) gemessen werden (siehe Kapitel 7).

Erweiterungen von RapidMiner erfolgen normalerweise über neue Operatoren. Für die Erweiterung von RapidMiner um die neuen Ähnlichkeitsmaße war es dagegen erforderlich, die neuen Ähnlichkeitsmaße in bereits bestehende Operatoren zu integrieren. RapidMiner bietet dazu seit Version 4.4 eine Schnittstelle SimilarityMeasure, welche die neuen Ähnlichkeitsmaße implementieren. Nachdem die Ähnlichkeitsmaße beim Programmstart über einen speziellen Mechanismus global registriert worden sind, stehen sie in allen Operatoren, die Ähnlichkeitsmaße verwenden, zur Verfügung. Auf diese Weise kann auf den extrahierten Beispielen nicht nur der k-NN-Algorithmus, sondern z. B. auch ein Clusteringalgorithmus ausgeführt werden.

6.3.4 Merkmalsselektion

In Abschnitt 1.2 wurde beschrieben, dass es ein Ziel dieser Arbeit ist herauszufinden, ob angesichts der hohen Kosten des Protokollierens von Systemaufrufen bereits eine Teilmenge der Systemaufrufe ausreicht, um Anwendungen gut anhand der HHH-Mengen des Stroms der Systemaufrufe zu erkennen. Dies lässt sich als Problem der Merkmalsselektion auffassen.

In dem einfachen Fall, in dem Beispiele durch Merkmalsvektoren konstanter Länge dargestellt werden, ist das Problem der Merkmalsselektion die Auswahl einer relevanten Teilmenge der Merkmale ([Kohavi und John, 1997]). Das verwendete Lernverfahren, z. B. k-NN, benutzt dann nur diese Merkmale.

Ziel ist es ([Hall, 2000]),

- durch die Begrenzung der Dimensionalität der Daten einen effizienteren Lernvorgang zu ermöglichen,
- Modelle zu erzeugen, die auf ungesehenen Daten bessere Ergebnisse erzielen,
- und kompaktere, besser verständliche Modelle zu erhalten.

Es gibt zwei Ansätze zur Merkmalsselektion: Bei einem Wrapperverfahren wird das Lernverfahren selbst verwendet, um die Qualität einer Merkmalsmenge zu schätzen. Bei einem Filterverfahren wird dagegen der Nutzen von Merkmalen unabhängig von einem Lernverfahren anhand von Heuristiken beurteilt ([Hall, 1999], S. 3).

Die Information über die Qualität von Merkmalsmengen wird verwendet, um eine Suche nach guten Merkmalsmengen im Teilmengenverband der Merkmale zu steuern. Für die Organisation der Suche gibt es verschiedene Möglichkeiten. Gibt es nur sehr wenige Merkmale, kann eine vollständige Suche möglich sein. Ist dies nicht möglich, werden Suchheuristiken verwendet. Ein einfaches Verfahren beginnt mit einer leeren Merkmalsmenge und fügt schrittweise einzelne Merkmale hinzu (*Vorwärtsselektion*). Bei der *Rückwärtsselektion* beginnt man dagegen mit der Menge aller Merkmale und entfernt schrittweise einzelne Merkmale ([Kohavi und John, 1997], [Han und Kamber, 2006], S.76). Genetische Algorithmen zur Merkmalsselektion (siehe [Hall, 1999]) verwenden eine aufwändigere Suchstrategie, sind aber weniger anfällig für lokale Optima als die beiden zuvor genannten Verfahren. Einige Filterverfahren vereinfachen die Suche, indem nicht Mengen von Merkmalen, sondern einzelne Merkmale bewertet werden (siehe [Guyon und Elisseeff, 2003], [Hall, 1999]). Die Merkmalsselektion erfolgt in diesem Fall durch Auswahl der k besten Merkmale oder durch Auswahl aller Merkmale, deren Bewertung einen Schwellwert überschreitet ([Hall, 2000]).

Man kann die Merkmalsselektion auffassen als Bestimmung eines Bitvektors mit einem Bit pro Merkmal, der festlegt, welche Teile der Daten verwendet werden sollen und welche nicht.

Die Anwendungsdaten liegen nicht als Merkmalsvektoren fester Länge vor. Dennoch lassen sich Verfahren zur Merkmalsselektion anwenden, um eine Teilmenge von Systemaufrufen zu bestimmen, die ausreicht, um Anwendungen gut voneinander zu trennen. In diesem Fall ist das Ziel das Bestimmen eines Bitvektors, der festlegt, welche Systemaufrufe verwendet werden sollen. Wie im Wrapperverfahren kann die Qualität von Teilmengen der Menge der Systemaufrufe mit dem Lernverfahren (hier k-NN) geschätzt werden und zum Steuern der Suche im Teilmengenverband der Aufrufe verwendet werden. Dazu wird für die jeweilige Teilmenge S der Aufrufe für jede Logdatei eine Instanz des HHH-Algorithmus ausgeführt. Während der Verarbeitung der Datenströme werden alle Aufrufe verworfen, die nicht in S enthalten sind. Anschließend wird der Klassifikationsfehler von k-NN bei Verwendung der Merkmalsmenge S durch Kreuzvalidierung geschätzt. Insgesamt wird so die Situation simuliert, dass nur ein Teil der Systemaufrufe protokolliert wird.

Diese Vorgehensweise lässt sich nutzen, um die Kosten des Protokollierens von Systemaufrufen zu reduzieren: Zunächst erstellt man einen Datensatz, der alle Aufrufe enthält, und bestimmt auf diesem mittels Merkmalsselektion die relevanten Merkmale. Im späteren Produktionsbetrieb, etwa bei der Intrusion Detection, müssen dann nur noch die relevanten Aufrufe erfasst werden. Zu den drei oben genannten Zielen der Merkmalsselektion kann also ein viertes hinzugefügt werden: Wenn das Erfassen der Merkmale Kosten verursacht, kann man mit der Merkmalsselektion die Kosten für das Erfassen neuer Daten senken.

Attributgewichte

RapidMiner stellt eine Reihe von Methoden zur Merkmalsselektion zur Verfügung, unter anderem die Vorwärts- und Rückwärtsselektion und einen genetischen Algorithmus zur Merkmalsselektion. Diese Methoden sind konstruiert für den üblichen Fall, in dem Beispiele sich als Merkmalsvektor konstanter Länge darstellen lassen und die Merkmalsselektion eine Auswahl aus diesen Merkmalen treffen soll. Dazu wird ein Vektor mit binären Attributgewichten verwendet, mit dem Merkmale ein- und ausgeschaltet werden können. Jeder Vektor von Attributgewichten entspricht einer Teilmenge der Menge der Merkmale und repräsentiert einen Punkt im Suchraum. Der Operator zur Merkmalsselektion gibt über den Vektor der Attributgewichte eine Merkmalsmenge vor. Diese wird von einem sogenannten inneren Operator bewertet. Die Bewertung kann darin bestehen, die Ergebnisqualität eines Lernverfahrens zu ermitteln, das genau diese Merkmale der Beispiele verwendet (Wrapperverfahren). Anhand dieser Bewertung der Merkmalsmengen und der von der Merkmalsselektion realisierten Suchstrategie wird der nächste Punkt im Suchraum festgelegt. Dieser wird wiederum durch den inneren Operator bewertet.

Die Operatoren zur Merkmalsselektion benutzen also Attributgewichte als Schnittstelle zu Beispielmengen und Lernverfahren. Um diese Operatoren ebenfalls nutzen zu können, habe ich die rohen Beispiele, die lediglich ein einzelnes Attribut mit dem Namen der Logdatei besaßen, so erweitert, dass sie dieselbe Schnittstelle anbieten. Für jeden Systemaufruf habe ich ein Attribut eingeführt, dessen Wert für alle Beispiele 1 ist und das nie ausgelesen wird. Die Attribute dienen ausschließlich dazu, Systemaufrufe ein- und auszuschalten. Wenn während der Merkmalsselektion über die Attributgewichte einzelne Aufrufe ausgeschaltet werden, werden diese Aufrufe bei der Extraktion der hierarchischen Variablen verworfen. Die ermittelten HHH-Mengen sind also von den ausgewählten Aufrufen abhängig, die Ähnlichkeiten und die Klassifikationsgüte ebenfalls, daher erhält der Operator für die Merkmalsselektion über die Klassifikationsgüte eine Rückmeldung über die Eignung der gewählten Menge von Systemaufrufen und kann die Suche im Teilmengenverband in Regionen nützlicher Systemaufrufe lenken.

Der Dateiname ist ein „spezielles“ Attribut. Spezielle Attribute in RapidMiner werden von der Merkmalsselektion ignoriert, so dass dieses Merkmal nicht ausgeschaltet werden kann.

Über die Schnittstelle Attributgewichte lassen sich alle RapidMiner-Operatoren zur Merkmalsselektion anwenden. Da jedoch während der Merkmalsselektion für jeden neuen

Punkt des Suchraums für jedes Beispiel die HHH-Algorithmen ausgeführt werden müssen, kommen wegen der erforderlichen Rechenzeiten nur einfache Suchstrategien wie Vorwärtsselektion in Frage.

Erweiterte Merkmalsselektion: UnseenClassValidation

Merkmalsselektion wird normalerweise in folgendem Kontext durchgeführt: Man hat eine feste Menge von Klassen und möchte eine Teilmenge der Merkmale finden, mit der sich diese Klassen gut trennen lassen. Für die Anwendungsklassifikation würde das bedeuten, dass man für eine feste Menge von Anwendungen herausfinden möchte, mit welchen Systemaufrufen sich diese Anwendungen gut anhand der HHH-Mengen ihrer Aufrufe erkennen lassen.

Wegen der hohen Kosten der Merkmalsselektion ist es eine interessante Frage, ob man Merkmalsmengen, die für das Trennen bestimmter Klassen relevant sind, auch für das Trennen anderer Klassen verwenden kann. Im Allgemeinen kann man nicht davon ausgehen, dass dies immer möglich ist. Andererseits ist es durchaus denkbar, dass bestimmte Merkmale generell relevanter sind als andere Merkmale. Ein Beispiel für Merkmale, die für generell wenig relevant gehalten werden, sind Stoppwörter wie „die“, „der“ und „oder“ in der Textklassifikation ([Joachims, 2001], S.27).

Ich habe daher mit dem Wrapperverfahren für den Fall der Anwendungsklassifikation untersucht, ob das Ergebnis einer Merkmalsselektion, die für eine bestimmte Menge von Klassen (Anwendungen) erfolgte, auch für eine neue Menge von Klassen anwendbar ist: Bei der Evaluierung der klassischen Merkmalsselektion misst man die Qualität der Selektion (z. B. den Anteil korrekt klassifizierter Beispiele) auf ungesehenen Beispielen *bekannter* Klassen. Bei der Evaluierung der erweiterten Merkmalsselektion messe ich die Qualität der Selektion auf ungesehenen Beispielen *ungesehener* Klassen.

Für den Fall der Anwendungsklassifikation bedeutet dies, dass mit den Logdateien von fünf Anwendungen eine Merkmalsselektion durchgeführt wird und die Qualität der Merkmalsselektion auf Logdateien der anderen fünf Anwendungen gemessen wird. Diese Vorgehensweise lässt sich als Evaluierung eines Lernprozesses auf einer übergeordneten Ebene auffassen: Man „lernt“ ein Modell (eine Menge relevanter Merkmale) anhand von „Trainingsbeispielen“ (hier: Klassen) und überprüft die Qualität des Modells anhand ungesehener „Beispiele“ (hier: Klassen). Es ist daher naheliegend, die üblichen Verfahren zur Evaluation von Lernergebnissen in angepasster Form anzuwenden. Der Operator UnseenClassValidation setzt genau diesen Gedanken um:

Es sei E die Menge der vorhandenen Beispiele und C die Menge der Klassen dieser Beispiele. Die Beispiele haben Merkmale aus der Menge M der Merkmale. Es wird eine zufällige Menge $S \subset C$ ausgewählt mit $|S| = \frac{|C|}{2}$. $E_S \subset E$ sei die Menge der Beispiele aus E mit einer Klasse aus S, $E_{C \setminus S} = E \setminus E_S$. Auf einer Teilmenge E_1 von E_S wird eine Merkmalsselektion durchgeführt. Die Qualität der ausgewählten Merkmalsmenge M^* wird auf einer Menge $E_2 \subset E_{C \setminus S}$ gemessen, indem der kreuzvalidierte Klassifikati-

onsfehler bestimmt wird. Um das so gemessene Ergebnis beurteilen zu können, werden zwei Vergleichswerte berechnet:

- Der Klassifizierungsfehler bei Verwendung der Merkmale aus M^* auf ungesehenen Beispielen $E_3 \subset E_S \backslash E_1$ bekannter Klassen (wie bei der Evaluierung einer normalen Merkmalsselektion).
- Der Klassifikationsfehler auf E_2 bei Verwendung aller Merkmale aus M.

Da die Schwierigkeit eines Klassifikationsproblems von der Anzahl der Klassen und der Anzahl der Trainingsbeispiele abhängig ist, sind diese Vergleichswerte nur dann verwendbar, wenn $|E_2| = |E_3|$ und $|S| = |C \backslash S|$. Der Operator stellt dies automatisch sicher. Die Klassifikationsfehler sind daher vergleichbar und lassen Rückschlüsse darüber zu, wie sehr sich eine Merkmalsselektion auf den „falschen“ Klassen auf den Klassifikationsfehler auswirkt. Der gesamte Vorgang wird für verschiedene zufällige Mengen S wiederholt.

Der Operator UnseenClassValidation erhält als Eingabe also eine Menge E von Beispielen. Er besitzt zwei so genannte innere Operatoren: Der erste Operator ist ein Versuchsaufbau, der eine Merkmalsselektion durchführt und einen Vektor von Attributgewichten ausgibt, der die Menge M^* beschreibt. Der zweite innere Operator ist ein Versuchsaufbau, der die Qualität der Attributgewichte auf ungesehenen Daten überprüft, indem ein Lerner mit der vorgegebenen Merkmalsauswahl trainiert und der kreuzvalidierte Klassifikationsfehler bestimmt wird. Die Verarbeitungsschritte des Operators UnseenClassValidation sind in Algorithmus 2 dargestellt.

Algorithmus 2 : UnseenClassValidation

Eingabe: Anzahl der Iterationen n, Menge E der Beispiele mit Klassen aus der Menge C der Klassen. Beispiele haben Merkmale aus der Menge M der Merkmale.

for $i = 1$ **to** n **do**
wähle $S \subset C$ mit $|S| = \frac{|C|}{2}$ zufällig;
$E_S \subset E$ sei die Menge der Beispiele mit einer Klasse aus S, $E_{C \backslash S} = E \backslash E_S$;
wähle $E_1 \subset E_S$, $E_2 \subset E_{C \backslash S}$ und $E_3 \subset E_S \backslash E_1$ zufällig, so dass $|E_2| = |E_3|$;
berechne mit erstem Operator eine Merkmalsmenge $M^* \subset M$ durch Merkmalsselektion auf E_1;
bewerte M^* durch Anwenden des zweiten Operators auf Beispielen E_2 und gib Ergebnis aus;
bewerte M^* durch Anwenden des zweiten Operators auf Beispielen E_3 und gib Ergebnis aus;
bewerte M durch Anwenden des zweiten Operators auf Beispielen E_2 und gib Ergebnis aus;

Die Verwendung des Operators ist nicht auf Systemaufrufdaten beschränkt, er kann auch bei Verwendung anderer Daten zur Evaluierung der erweiterten Merkmalsselektion verwendet werden.

7 Ressourcenbedarf der Algorithmen

In diesem Kapitel werden Experimente zu Speicherbedarf, Laufzeit und Approximationsgüte des Full Ancestry und des Partial Ancestry Algorithmus dargestellt. Auch die Änderung des Ressourcenverbrauchs bei Begrenzung der Hierarchietiefe wird beschrieben. Alle Messungen wurden auf Dateien des Anwendungsdatensatzes ausgeführt, der etwa 23 Millionen Systemaufrufe enthält. Verwendet wurden Aufruf-, Pfad- und Sequenzhierarchie. Die Aufrufhierarchie hat Tiefe sechs, die Pfadhierarchie Tiefe zwölf und die Sequenzhierarchie Tiefe vier. Die Laufzeiten enthalten nur die Rechenzeiten der Algorithmen während der Verarbeitung des Stroms ohne den Zeitaufwand für das Einlesen und die Extraktion der hierarchischen Variablen. Auch die Rechenzeiten der Output-Methode, die erst nach der Verarbeitung des Stroms ausgeführt wird, sind nicht enthalten. Die Messungen wurden auf einem Rechner mit Intel Core 2 Duo E6300 Prozessor mit 2 GHz und 2 GB Hauptspeicher durchgeführt.

Die obere Schranke für die amortisierte Laufzeit des Full Ancestry Algorithmus ist $O(H \log \varepsilon N)$ für das Verarbeiten eines Stromelementes, die Schranke für den Speicherbedarf ist $O(\frac{H}{\varepsilon} \log \varepsilon N)$. Es handelt sich um asymptotische obere Schranken, die keine konkreten Prognosen für Veränderungen der Laufzeiten bei Variation eines Parameters erlauben: Die Halbierung einer oberen Schranke, etwa durch Halbierung von H, erlaubt keine Aussage über die tatsächliche Veränderung der Laufzeit. Tatsächlich sind die oberen Schranken nicht eng, wie man am Beispiel eines Stroms identischer Elemente sehen kann.

7.1 Speicherbedarf und Laufzeiten

Tabelle 7.1 zeigt den Speicherbedarf der Algorithmen für eine, zwei und drei Dimensionen. Es gibt elf Anwendungen. Zu jeder Anwendung gibt es zehn Logdateien. Für jede Anwendung wurde die erste ihrer zehn Logdateien ausgewählt. Auf diesen elf Logdateien von verschiedenen Anwendungen wurden die Algorithmen ausgeführt. Für jede dieser Logdateien wurde der Speicherbedarf berechnet als die maximale Anzahl der Tupel, die in der Datenstruktur während der Ausführung des Algorithmus gespeichert wurden. Angegeben sind Minimum, Maximum und Mittelwert über diese elf Logdateien. ε wurde auf $5 \cdot 10^{-4}$ gesetzt. In der ersten Spalte sind die Werte bei Verwendung der Aufrufhierarchie (C) angegeben, in der zweiten Spalte wird zusätzlich die Pfadhierarchie (P) verwendet und in der dritten Spalte auch die Sequenzhierarchie (S). Sind die Hierarchietiefen h_i für die d Dimensionen gegeben, so ist $H = \prod_{i=1}^{d} (h_i + 1)$. Für die erste Spalte ist daher $H = 7$, für die zweite Spalte ist $H = 91$ und für die dritte Spalte ist $H = 455$.

		Speicher			Laufzeit		
		C	CP	CPS	C	CP	CPS
Full Ancestry	Mittelwert	111	5988	48820	79	472	6569
	Minimum	19	25	736	16	31	78
	Maximum	151	9971	73403	219	922	14422
Partial Ancestry	Mittelwert	70	2837	10547	74	2328	74342
	Minimum	7	7	141	15	31	78
	Maximum	105	4671	18058	219	5109	150781

Tabelle 7.1: Speicherbedarf (in Tupeln) und Laufzeit (in ms) der Algorithmen. Der Wert ε wurde auf $5 \cdot 10^{-4}$ gesetzt. Angegeben sind Minimum, Maximum und Mittelwert über die erste Logdatei aller Anwendungen.

Der Speicherbedarf steigt mit zunehmender Anzahl der Dimensionen stark an. In allen Fällen ist der Speicherbedarf für Partial Ancestry niedriger als für Full Ancestry. Allerdings ist der Speicherbedarf in allen Fällen sehr niedrig und liegt auch für Full Ancestry nur bei einigen hundert Kilobyte. Verschiedene Logdateien benötigen unterschiedlich viel Speicher für die Verarbeitung. Geyes benötigt sehr wenig Speicher, weil in einer Endlosschleife von 14 Aufrufen immer dieselben Aufrufe ausgeführt werden. Alle Minimalwerte für den Speicherbedarf in der Tabelle stammen von Geyes. Rhythmbox und Firefox haben einen sehr hohen Speicherbedarf.

In Tabelle 7.1 sind auch die Laufzeiten angegeben (Durchschnittswerte über die erste Logdatei aller Anwendungen in ms), wieder ist ein starker Anstieg bei zunehmender Anzahl der Dimensionen zu erkennen. Partial Ancestry ist in einer Dimension etwas schneller als Full Ancestry und in höherer Dimension deutlich langsamer. Ein Anstieg der Laufzeiten für Partial Ancestry tritt auch bei [Cormode et al., 2008] auf, ist hier allerdings deutlich stärker ausgeprägt. Der Anstieg der Laufzeiten des Partial Ancestry Algorithmus in höheren Dimensionen kann vermutlich abgeschwächt werden durch eine verbesserte Implementierung einer bestimmten Methode, die beim Partial Ancestry Algorithmus sehr häufig ausgeführt wird (Erfragen des m-Werts der Vorfahren, siehe Algorithmus 1, Insert, Zeile 5/6). Wegen der besseren Approximationsgüte des Full Ancestry Algorithmus (siehe Abschnitt 7.3) und aus Zeitmangel wurde dieser Ansatz nicht weiter verfolgt.

Die Rechenzeiten für die verschiedenen Logdateien sind sehr unterschiedlich. Geyes und NEdit benötigen sehr wenig Zeit, Rhythmbox und Firefox dagegen sehr viel.

In Abbildung 7.1 und 7.2 werden Speicherbedarf und Rechenzeiten bei Variation von ε am Beispiel der größten Logdatei (Rhythmbox 4) dargestellt. Wegen der Größe der Datei liegen die Werte teilweise über den Werten in Tabelle 7.1. Wieder wird im eindimensionalen Fall die Aufrufhierarchie verwendet, im zweidimensionalen Fall zusätzlich die Pfade und im dreidimensionalen Fall zusätzlich die Sequenzen von Betriebssystemaufrufen.

Für den Speicherbedarf ist jeweils die maximale Anzahl gespeicherter Tupel angegeben. Der Speicherbedarf sinkt erwartungsgemäß bei steigendem ε, dagegen steigt die

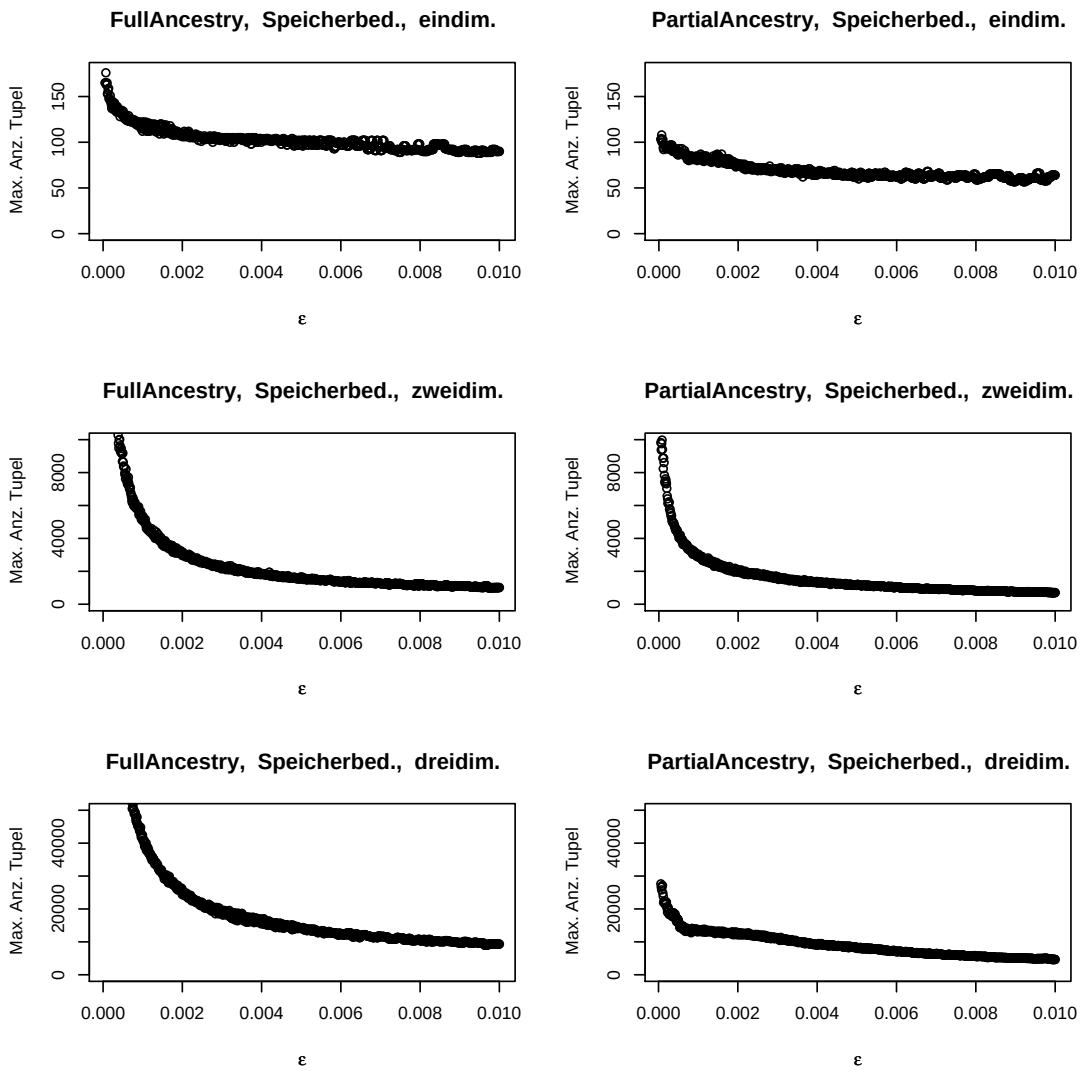


Abbildung 7.1: Speicherbedarf (in Tupeln) für verschiedene Dimensionen bei Variation von ε am Beispiel der größten Logdatei Rhythmbox 4

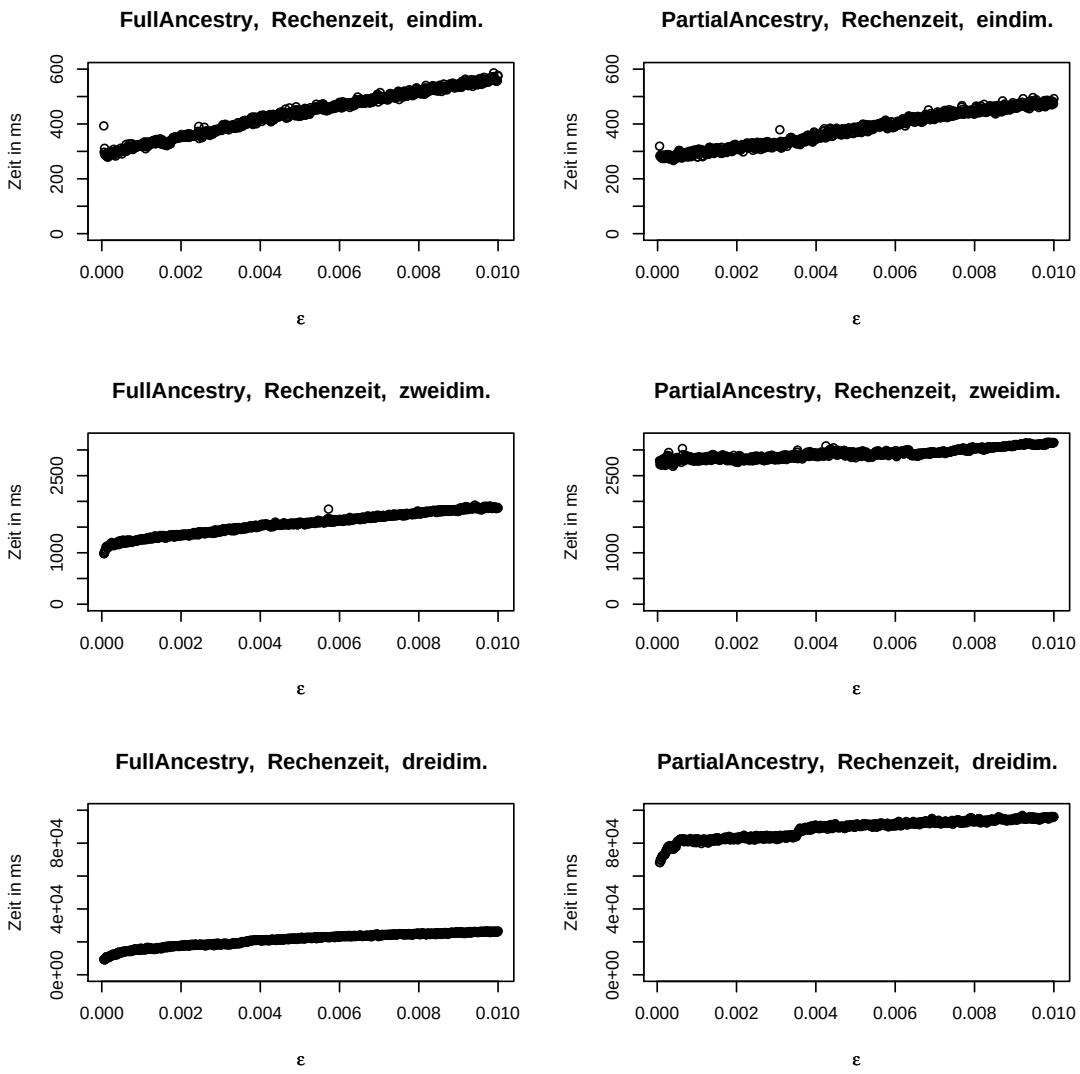


Abbildung 7.2: Rechenzeit (in ms) für verschiedene Dimensionen bei Variation von ε am Beispiel der größten Logdatei Rhythmbox 4

		Speicher			Laufzeit		
		C	CP	CPS	C	CP	CPS
Full Ancestry	Mittelwert	111	5988	48820	79	472	6569
	Minimum	19	25	736	16	31	78
	Maximum	151	9971	73403	219	922	14422
Full Ancestry, Cap	Mittelwert	71	4732	21430	75	369	2339
	Minimum	13	19	202	15	31	47
	Maximum	96	8365	36813	219	766	4765
Partial Ancestry	Mittelwert	70	2837	10547	74	2328	74342
	Minimum	7	7	141	15	31	78
	Maximum	105	4671	18058	219	5109	150781
Partial Ancestry, Cap	Mittelwert	52	2153	5134	77	1128	12291
	Minimum	6	6	50	16	31	47
	Maximum	73	3591	8887	219	2297	25532

Tabelle 7.2: Speicherbedarf (in Tupeln) und Laufzeit (in ms) bei begrenzter Hierarchietiefe. Der Wert ε wurde auf $5 \cdot 10^{-4}$ gesetzt. Angegeben sind Minimum, Maximum und Mittelwert über die erste Logdatei aller Anwendungen.

Laufzeit bei steigendem ε. Die dreidimensionalen Werte für die Laufzeit sind Durchschnittswerte über drei Ausführungen, die übrigen Werte sind Durchschnittswerte über zwölf Ausführungen. Die dennoch vereinzelt auftretenden Ausreißer sind Messfehler, die bei Wiederholung des Versuchs an anderen Punkten auftreten.

7.2 Begrenzung der Hierarchietiefe

Tabelle 7.2 zeigt den Speicherbedarf und die Laufzeit der Algorithmen bei begrenzter Hierarchietiefe (Durchschnittswerte über die erste Logdatei aller Anwendungen). Für die Zeilen mit dem Zusatz „Cap“ wurden die Hierarchietiefen für alle Dimensionen halbiert. Die Hierarchietiefe für Aufrufe ist dann drei, für Pfade sechs und für Sequenzen zwei. Angegeben sind die Durchschnittswerte für die jeweils erste Logdatei aller Anwendungen, ε wurde auf $5 \cdot 10^{-4}$ gesetzt.

Sowohl für die Laufzeit als auch für den Speicherbedarf könnte man eine Halbierung des Ressourcenverbrauchs in einer, eine Viertelung in zwei und eine Achtelung in drei Dimensionen erwarten. Die tatsächlichen Werte weichen davon ab, weil die theoretischen Schranken nicht eng sind und die Auswirkung der Begrenzung von der Verteilung der Stromelemente abhängt.

	C	CP	CPS
Full Ancestry	0.997 ± 0.006	0.994 ± 0.003	0.987 ± 0.008
Partial Ancestry	0.985 ± 0.010	0.957 ± 0.017	0.921 ± 0.026

Tabelle 7.3: Durchschnittliche Approximationsgüte, gemessen mit dem Ähnlichkeitsmaß MOGM. ε wurde auf $5 \cdot 10^{-4}$ und ϕ wurde auf $2 \cdot 10^{-3}$ gesetzt. Angegeben sind Mittelwert und Standardabweichung über die erste Logdatei aller Anwendungen.

7.3 Approximationsgüte

Tabelle 7.3 zeigt die Güte der Approximationslösungen für $\varepsilon = 5 \cdot 10^{-4}$ und $\phi = 2 \cdot 10^{-3}$ (Durchschnittswerte über die erste Logdatei aller Anwendungen). Dazu wurde zu jeder Eingabe die exakte Lösung berechnet und mit der Approximationslösung verglichen. Der Vergleich wurde mit dem Ähnlichkeitsmaß MOGM durchgeführt. Für andere Ähnlichkeitsmaße sind die Ergebnisse ähnlich: Die Lösungsqualität sinkt bei konstanten Werten für ε und ϕ mit zunehmender Dimension, und Full Ancestry erzielt bessere Ergebnisse als Partial Ancestry. In niedriger Dimension entspricht die Approximationslösung in vielen Fällen der exakten Lösung.

Tabelle 7.4 zeigt die durchschnittlichen Größen der HHH-Mengen (Durchschnittswerte über die erste Logdatei aller Anwendungen) und erklärt die unterschiedlichen Approximationsgüten der Algorithmen: Partial Ancestry erzielt vermutlich schlechtere Ergebnisse als Full Ancestry, weil die HHH-Mengen größer als bei Full Ancestry ausfallen. Auch [Cormode et al., 2008] (S. 38) beobachten, dass Partial Ancestry größere HHH-Mengen berechnet und stellen einen Bezug zur Ergebnisqualität her (S. 42). Leider führen sie keinen Vergleich der Ergebnisqualitäten von Full Ancestry und Partial Ancestry durch. Die

		C	CP	CPS
Exakt	Mittelwert	27	131	282
	Minimum	7	7	21
	Maximum	40	569	701
Full Ancestry	Mittelwert	27	134	302
	Minimum	7	7	21
	Maximum	40	571	709
Partial Ancestry	Mittelwert	29	191	560
	Minimum	7	7	23
	Maximum	43	799	1069

Tabelle 7.4: Größe der HHH-Mengen. ε wurde auf $5 \cdot 10^{-4}$ und ϕ wurde auf $2 \cdot 10^{-3}$ gesetzt. Angegeben sind Minimum, Maximum und Mittelwert über die erste Logdatei aller Anwendungen.

für Full Ancestry angegeben Werte zwischen null und eins werden als „Score“ bezeichnet, die Berechnung wird nicht erklärt.

Die HHH-Mengen sind bei gleichem ϕ für verschiedene Anwendungen unterschiedlich groß. Geyes hat sehr kleine HHH-Mengen (alle Minimalwerte in Tabelle 7.4 stammen von Geyes), Tomboy, top und xterm haben dagegen große HHH-Mengen.

In Abbildung 7.3 ist die Approximationsgüte bei Variation von ε für $\phi = 10^{-3}$ am Beispiel der Logdatei Rhythmbox 4 dargestellt. Die Approximationsgüte wurde erneut mit dem Ähnlichkeitsmaß MOGM durch Vergleich der Approximationslösung mit der exakten Lösung gemessen. Im eindimensionalen Fall wird die Aufrufhierarchie verwendet, im zweidimensionalen Fall zusätzlich die Pfade und im dreidimensionalen Fall zusätzlich die Sequenzen von Betriebssystemaufrufen. Erwartungsgemäß erzeugen kleinere ε bessere Ergebnisse. Auch hier ist eine schlechtere Approximation bei Verwendung von Partial Ancestry und bei steigender Anzahl der Dimensionen zu erkennen. Allerdings wird auch deutlich, dass selbst in den schwierigen Fällen durch Absenken von ε eine gute Approximation erreicht werden kann.

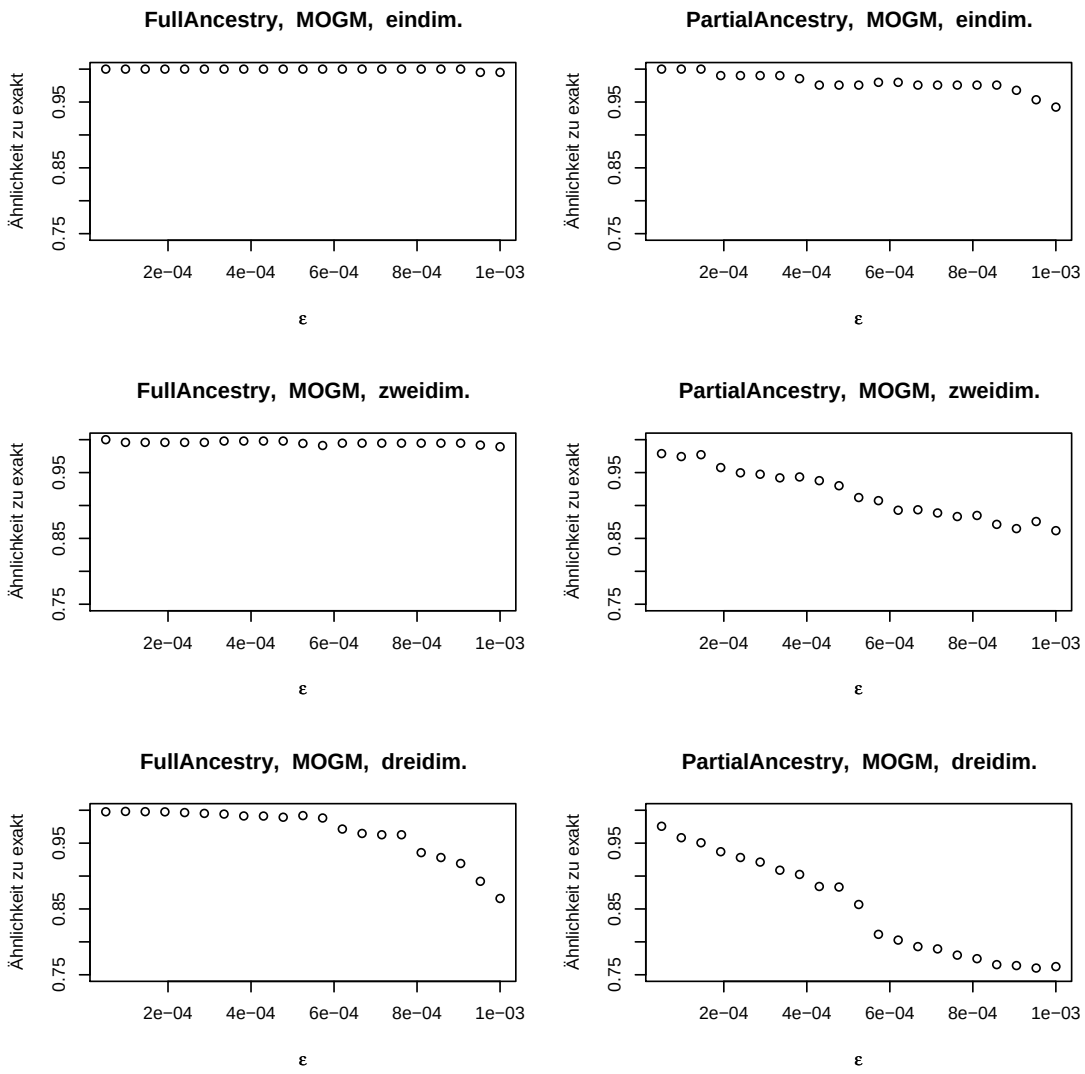


Abbildung 7.3: Approximationsgüte bei Variation von ε. Der Wert ϕ wurde auf 10^{-3} gesetzt, Logdatei ist Rhythmbox 4.

8 Erkennen von Anwendungen

In diesem Kapitel werden Experimente beschrieben, bei denen Anwendungen anhand der kondensierten Darstellung ihrer Systemaufrufe durch HHH-Mengen erkannt werden sollen.

In Abschnitt 8.1 wird gezeigt, dass es möglich ist, Anwendungen an den HHH-Mengen ihrer Systemaufrufe zu erkennen. Der Fehler ist dabei so gering, dass keine Aussage darüber gemacht werden kann, welche hierarchischen Variablen und Ähnlichkeitsmaße besonders gut geeignet sind. Die Experimente in den folgenden Abschnitten wurden daher mit aufgespaltenen Logdateien durchgeführt. Dies entspricht der Simulation kürzerer Logzeiträume. In Abschnitt 8.2 wird mit diesen aufgespaltenen Dateien die Eignung der hierarchischen Variablen und ihrer Kombinationen, der Einfluss der Parameter ε und ϕ und der Einfluss des Ähnlichkeitsmaßes auf den Fehler untersucht. In Abschnitt 8.3 wird der Einfluss der Anzahl der Nachbarn k gemessen und der Klassifikationsfehler der HHH-Verfahren mit dem Fehler eines einfachen, nur auf der Häufigkeit der Aufrufe basierenden Verfahrens verglichen. In Abschnitt 8.4 wird untersucht, ob es möglich ist, nicht nur Anwendungen, sondern auch Gruppen von Anwendung (Editoren und Nicht-Editoren) voneinander zu unterscheiden. Zuletzt werden in Abschnitt 8.5 Experimente zur Merkmalsselektion dargestellt und bewertet.

8.1 Fehler beim Erkennen von Anwendungen

In Tabelle 8.1 sind die Ergebnisse der Anwendungsklassifikation in Abhängigkeit der hierarchischen Variablen und der Ähnlichkeitsmaße dargestellt. Aus gegebenen Datenströmen in Logdateien wurden die HHH-Mengen berechnet. Ziel war es, durch den Vergleich einer HHH-Menge mit den HHH-Mengen bekannter Anwendungen die Anwendung zu schätzen, die den Datenstrom erzeugt hat. Diese Fragestellung kann im Bereich der Intrusion Detection von Interesse sein, wenn manipulierte Varianten einer Anwendung vom Original unterschieden werden sollen.

Verwendet wurden jeweils zehn Logdateien der zehn protokollierten Anwendungen (ohne Epiphany), insgesamt also 100 Beispiele. Für die Berechnung der HHH-Mengen wurde der Full Ancestry Algorithmus mit $\varepsilon = 5 \cdot 10^{-4}$ und $\phi = 2 \cdot 10^{-3}$ verwendet. Diese Werte erwiesen sich bei vorab durchgeführten Experimenten als sinnvoll. Die Anzahl der Nachbarn im k-NN-Algorithmus wurde auf $k = 1$ gesetzt. Die hierarchischen Variablen C (Calls, Aufrufhierarchie), P (Pfade) und S (Sequenzen) wurden einzeln und in Kombination verwendet. Der Klassifikationsfehler wurde mit Leave-One-Out Kreuzvalidierung

	C	P	S	CP	CS	PS	CPS
OGM	0.0	1.0	1.0	1.0	1.0	1.0	1.0
COR	1.0	1.0	1.0	1.0	1.0	1.0	1.0
MCOR	1.0	1.0	1.0	1.0	1.0	1.0	1.0
MOGM	0.0	1.0	1.0	1.0	1.0	1.0	1.0
DSM	1.0	1.0	1.0	1.0	1.0	1.0	1.0

Tabelle 8.1: Klassifikationsfehler bei der Anwendungsklassifikation mit 1-NN in Prozent. ε wurde auf $5 \cdot 10^{-4}$ und ϕ wurde auf $2 \cdot 10^{-3}$ gesetzt.

(siehe [Hastie et al., 2001], S. 214, [Mitchell, 1997], S. 111) geschätzt, daher sind keine Standardabweichungen angegeben.

Die Ergebnisse sind eindeutig: Es ist mit dem hier beschriebenen Verfahren möglich, Anwendungen anhand der HHH-Mengen ihrer Systemaufrufe zu erkennen. In den meisten Fällen liegt der Fehler bei 1 %, in einigen Fällen werden alle Anwendungen richtig erkannt.

Dieses Ergebnis stellt zugleich ein Problem dar, denn damit sind keine Aussagen darüber möglich, welche hierarchischen Variablen und welche Ähnlichkeitsmaße gut und welche weniger gut geeignet sind. Für die folgenden Experimente wurden daher die 100 Logdateien jeweils in 30 gleich große Dateien aufgespalten. Dies entspricht der Simulation eines kürzeren Logzeitraums.

8.2 Fehler nach Aufspalten der Logdateien

Der Klassifikationsfehler auf den aufgespaltenen Dateien ist in Tabelle 8.2 dargestellt. Wieder wurde der Full Ancestry Algorithmus mit $\varepsilon = 5 \cdot 10^{-4}$ und $\phi = 2 \cdot 10^{-3}$ sowie 1-NN eingesetzt. Bei diesem und allen folgenden Experimenten wurden, sofern nicht anders angegeben, 300 der 3000 aufgespaltenen Dateien verwendet.

	C	P	S	CP	CS	PS	CPS
OGM	10.3	43.3	9.0	7.3	8.3	8.3	7.3
COR	10.7	45.7	8.3	8.7	8.0	7.7	8.3
MCOR	11.0	38.3	9.3	10.7	9.3	8.0	8.7
MOGM	11.0	38.0	8.7	7.7	7.3	6.3	6.7
DSM	7.3	13.3	7.0	10.7	7.7	8.3	8.3

Tabelle 8.2: Klassifikationsfehler bei der Anwendungsklassifikation mit 1-NN auf aufgespaltenen Dateien in Prozent. ε wurde auf $5 \cdot 10^{-4}$ und ϕ wurde auf $2 \cdot 10^{-3}$ gesetzt.

Der Fehler liegt deutlich über den Werten in Tabelle 8.1. Es ist – wie erwartet – schwieriger, Anwendungen anhand kürzerer Logzeiträume zu erkennen. Besonders auffällig ist der starke Anstieg des Fehlers bei ausschließlicher Verwendung der Pfadhierarchie (P). Der Grund dafür ist, dass es durch die Verkürzung des Logzeitraums wesentlich schwieriger wird, die Dateideskriptoren aufzulösen. Bei kurzen Logzeiträumen tritt häufiger der Fall auf, dass ein Dateideskriptor verwendet wird, der bereits vor Beginn des Logzeitraums erzeugt wurde und daher nicht aufgelöst werden kann. In diesem Fall wird der Dateiname „Datei unbekannt“ verwendet, der das Erkennen der Anwendung nicht erleichtert. Da das Aufspalten der Logdateien kürzere Logzeiträume simulieren soll und dieses Problem bei kürzeren Logzeiträumen tatsächlich genau so auftritt, wurde die Information zum Auflösen der Dateideskriptoren nicht künstlich bereitgestellt.

Die Unterschiede der Klassifikationsfehler sind (mit Ausnahme der Fehler bei alleiniger Verwendung der Pfadhierarchie) gering, die Fehler bewegen sich meist zwischen 7 % und 11 %. Kombinationen von hierarchischen Variablen führen meist zu einer etwas niedrigeren Fehlerzahl, selbst das Hinzufügen der einzeln nicht sehr nützlichen Pfadvariable senkt die Fehlerzahl eher als sie zu erhöhen.

Die alleinige Verwendung der Sequenzvariable führt mit Fehlern zwischen 7 % und 9.3 % zu vergleichsweise guten Ergebnissen. Der Ressourcenverbrauch ist im eindimensionalen Fall erheblich geringer als im mehrdimensionalen Fall. Daher kann die alleinige Verwendung der Sequenzvariablen bei Ressourcenknappheit eine gute Wahl sein. Der zusätzliche Nutzen der Pfad- und Parameterinformationen ist in diesem Experiment gering. Das ist eine interessante Beobachtung, denn in vielen Arbeiten zur Intrusion Detection mit Systemaufrufen (z. B. [Warrender et al., 1999], [Hofmeyr et al., 1998]) werden Sequenzen von Systemaufrufen konstanter Länge verwendet, um das Verhalten einer Anwendung zu charakterisieren. Dabei werden wie bei der alleinigen Verwendung der Sequenzvariablen nur die Namen der Systemaufrufe und keine Parameter der Aufrufe und auch keine Dateisystempfade verwendet. Allerdings muss berücksichtigt werden, dass der Nutzen der Pfadvariablen bei diesem Experiment durch die kurzen Logzeiträume stark eingeschränkt war.

Die Ähnlichkeitsmaße OGM, COR, MOGM und MCOR (siehe Abschnitt 5.2) erzielen vergleichbare Ergebnisse, keines ist klar besser oder klar schlechter als die übrigen Ähnlichkeitsmaße. Die modifizierten Ähnlichkeitsmaße MOGM und MCOR erzielen keine besseren Ergebnisse als ihre unmodifizierten Varianten. Das Ähnlichkeitsmaß DSM ist im eindimensionalen Fall (besonders für die hierarchische Pfadvariable P) deutlich besser und insgesamt zumindest nicht schlechter als die übrigen Ähnlichkeitsmaße. Da DSM auf der Datenstruktur operiert, entfällt die Wahl des Parameters ϕ. Für die Experimente aus Tabelle 8.2 wurde mit $\phi = 2 \cdot 10^{-3}$ bereits ein guter Wert gewählt, trotzdem sind die Ergebnisse für DSM zumindest nicht schlechter als die Ergebnisse der übrigen Ähnlichkeitsmaße. Durch die Wahl des Ähnlichkeitsmaßes DSM kann also ohne Beeinträchtigung der Ergebnisqualität die Suche nach einem geeigneten Wert für ϕ entfallen, die Verwendung von DSM kann daher vorteilhaft sein.

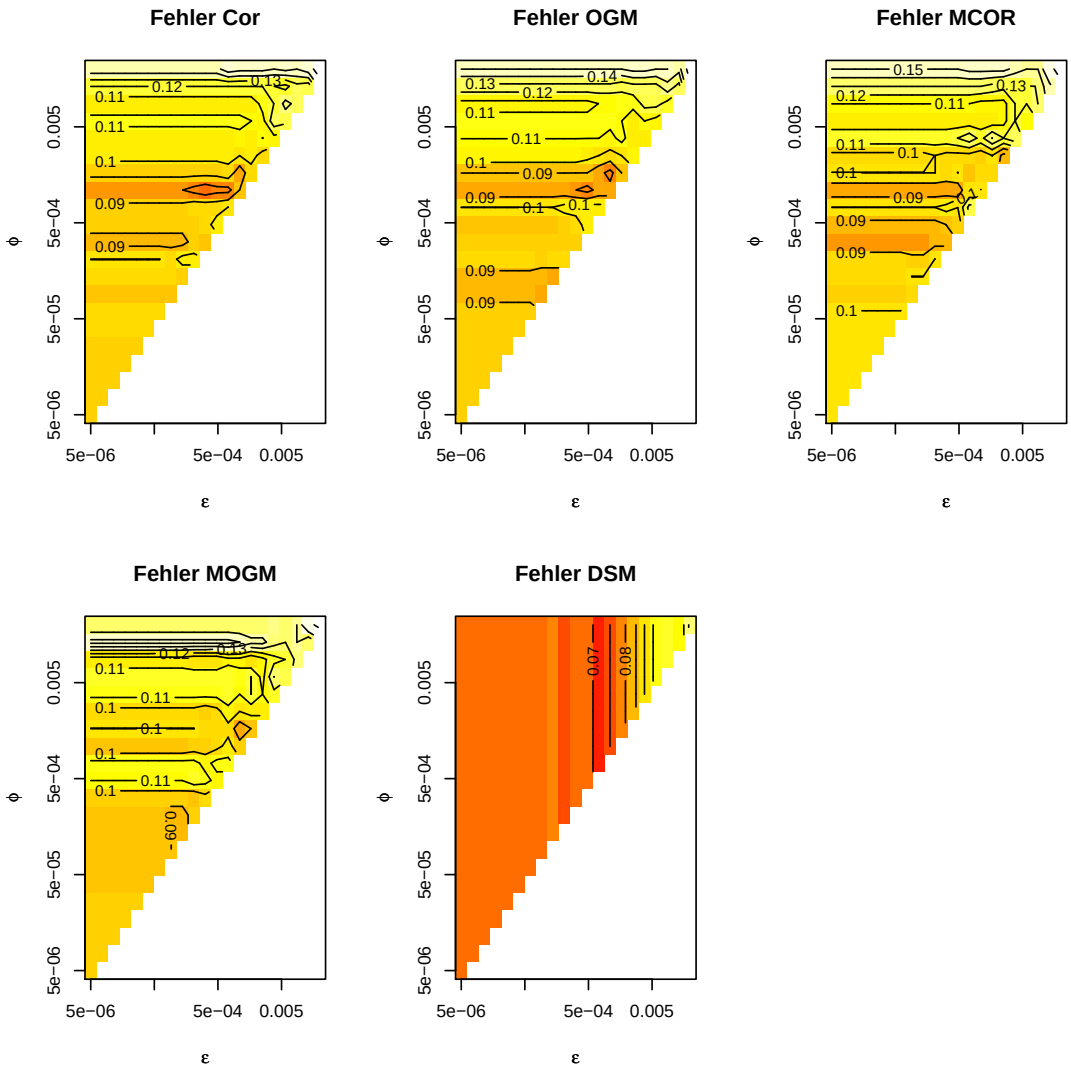


Abbildung 8.1: Verschiedene Ähnlichkeitsmaße: Fehler für die Aufrufhierarchie bei Variation von ε und ϕ

8.3 Einfluss der Ähnlichkeitsmaße und Parameter

Abbildung 8.1 zeigt den Klassifikationsfehler für $k = 1$ bei Verwendung der Aufrufhierarchie und Variation von ε und ϕ für verschiedene Ähnlichkeitsmaße. Für beide Achsen wurde eine logarithmische Darstellung gewählt. Werte für $\phi < \varepsilon$ sind nach Problemdefinition unzulässig und nicht angegeben.

Für die Ähnlichkeitsmaße COR, OGM, MCOR und MOGM ist zu erkennen, dass der Parameter ϕ, der die Größe der HHH-Mengen steuert, größeren Einfluss auf den Klassifikationsfehler als der Fehlerparameter ε hat. Für große ϕ-Werte steigt der Fehler deutlich an. Es ist nachvollziehbar, dass bei sehr kleinen HHH-Mengen die Zuverlässigkeit der Ähnlichkeitsmaße sinkt. Die Fehler für diese vier Ähnlichkeitsmaße sind unter den angegebenen Rahmenbedingungen bei guter Parameterwahl (z. B. $\varepsilon = 5 \cdot 10^{-4}$, $\phi = 2 \cdot 10^{-3}$) vergleichbar. COR und MCOR sind robuster gegenüber suboptimalen Parametereinstellungen.

DSM verursacht weniger Fehler als die übrigen Ähnlichkeitsmaße und ist robuster gegenüber den Parametereinstellungen: Der Parameter ϕ braucht gar nicht gewählt zu werden, der Einfluss des Parameters ε auf den Fehler ist gering.

Tabelle 8.3 zeigt den Klassifikationsfehler bei Verwendung der Aufrufhierarchie für $\varepsilon = 5 \cdot 10^{-4}$ und $\phi = 2 \cdot 10^{-3}$ bei Variation der Anzahl der Nachbarn k, zusätzlich ist der Fehler bei Verwendung eines einfachen Vergleichsverfahrens (TF) angegeben. Dazu wurden die relativen Häufigkeiten der Systemaufrufe für jede Logdatei in einem Vektor erfasst (siehe Operator HistoExtraction in Abschnitt 6.3.2) und der euklidische Abstand der Vektoren zur k-NN-Klassifikation verwendet.

Es ist deutlich zu erkennen, dass kleinere Werte für k zu geringeren kreuzvalidierten Fehlern führen. Der Fehler wird jeweils für $k = 1$ minimal. Der Fehler bei Ermittlung der Nachbarn über den euklidischen Abstand der TF-Vektoren ist jeweils um etwa $\frac{1}{3}$ größer als bei den anderen Verfahren. Bereits ein sehr einfaches Verfahren ohne jegliche Optimierungen erzielt also Ergebnisse, die nicht viel schlechter sind als die Ergebnisse der HHH-Methoden. Möglicherweise lassen sich durch Modifikationen der einfachen Methode die Ergebnisse sogar noch verbessern.

k	OGM	COR	MCOR	MOGM	DSM	TF
1	10.3	10.7	11.0	11.0	7.3	14.7
3	12.0	10.7	12.0	13.0	10.3	17.0
5	13.0	10.7	11.0	12.7	12.7	18.7
7	13.7	13.3	14.3	14.7	14.0	21.7
9	15.0	15.0	14.0	14.0	14.0	21.0

Tabelle 8.3: Klassifikationsfehler bei der Anwendungsklassifikation in Prozent. Variation von Ähnlichkeitsmaß und k bei Verwendung der Aufrufhierarchie. ε wurde auf $5 \cdot 10^{-4}$ und ϕ wurde auf $2 \cdot 10^{-3}$ gesetzt.

k	OGM	COR	MCOR	MOGM	DSM	TF
1	8.3	8.0	9.3	7.3	7.7	14.7
3	9.3	11.0	11.7	8.7	7.7	17.0
5	9.7	13.3	14.0	9.7	8.7	18.7
7	10.7	14.0	13.0	9.3	8.7	21.7
9	12.0	14.3	13.0	10.3	9.0	21.0

Tabelle 8.4: Klassifikationsfehler bei der Anwendungsklassifikation in Prozent. Variation von Ähnlichkeitsmaß und k bei Verwendung von Aufrufhierarchie und Sequenzhierarchie. ε wurde auf $5 \cdot 10^{-4}$ und ϕ wurde auf $2 \cdot 10^{-3}$ gesetzt.

Tabelle 8.4 zeigt die Ergebnisse desselben Experiments wie Tabelle 8.3, diesmal unter Verwendung der Aufruf- und Sequenzhierarchie. Die Beobachtungen zur Anzahl der Nachbarn k bestätigen sich. Der Fehler sinkt im Vergleich zur alleinigen Verwendung der Aufrufhierarchie leicht, der Abstand zum Fehler bei Verwendung von TF ist daher etwas größer.

Tabelle 8.5 zeigt die Ergebnisse desselben Experiments bei Verwendung der Aufrufhierarchie und bei Verwendung von 1500 statt 300 Logdateien. Erwartungsgemäß fällt der Fehler bei größerer Trainingsmenge geringer als in Tabelle 8.3 aus.

8.4 Erkennen von Anwendungsgruppen

In den bisherigen Experimenten wurde versucht, einzelne Anwendungen zu erkennen. Die Annahme war, dass verschiedene Ausführungen derselben Anwendung Ströme von Systemaufrufen erzeugen, deren HHH-Mengen einander ähnlich sind. Die bisherigen Experimente haben gezeigt, dass diese Annahme zutrifft: Auch wenn die Benutzer während verschiedener Ausführungen derselben Anwendung diese Anwendung in unterschiedlicher Weise benutzen, werden offenbar hinreichend oft dieselben Codeabschnitte der Anwendung ausgeführt, um ähnliche Ströme von Systemaufrufen zu erzeugen.

k	OGM	COR	MCOR	MOGM	DSM	TF
1	4.2	4.5	3.5	3.5	1.3	5.8
3	4.4	4.0	3.8	3.9	1.9	6.0
5	5.1	5.5	5.1	5.2	2.5	8.1
7	6.3	6.0	5.8	6.1	3.7	8.6
9	6.9	6.7	6.5	6.7	4.3	9.1

Tabelle 8.5: Klassifikationsfehler bei der Anwendungsklassifikation in Prozent. Variation von Ähnlichkeitsmaß und k bei Verwendung der Aufrufhierarchie mit 1500 Beispielen. ε wurde auf $5 \cdot 10^{-4}$ und ϕ wurde auf $2 \cdot 10^{-3}$ gesetzt.

k	OGM	COR	MCOR	MOGM	DSM	TF
1	6.0	7.3	7.0	5.7	5.3	7.3
3	5.7	6.7	7.0	7.0	6.7	9.3
5	7.0	5.7	8.0	7.7	8.3	10.7
7	9.0	8.7	10.3	10.0	8.0	12.3
9	7.7	9.0	7.7	8.3	8.3	14.7

Tabelle 8.6: Klassifikationsfehler bei Erkennen der Anwendungsgruppe unter Verwendung der Aufrufhierarchie in Prozent. ε wurde auf $5 \cdot 10^{-4}$ und ϕ wurde auf $2 \cdot 10^{-3}$ gesetzt.

Im Folgenden soll untersucht werden, ob es möglich ist, Gruppen von Anwendungen mit ähnlicher Funktion anhand der Ströme ihrer Systemaufrufe zu erkennen. Dazu wurden die Anwendungen des Anwendungsdatensatzes in zwei Klassen eingeteilt: Die vier Anwendungen XEmacs, NEdit, Kate und Tomboy wurden zur Klasse der Editoren zusammengefasst, die übrigen sechs Anwendungen zur Klasse der Nicht-Editoren. Ziel war die Klassifikation von Anwendungen als Editor oder Nicht-Editor durch den Vergleich der HHH-Mengen ihrer Datenströme mit denen bekannter Editoren und Nicht-Editoren.

Die Annahme war in diesem Fall, dass verschiedene Anwendungen, die ähnliche Aufgaben erfüllen, Ströme von Systemaufrufen erzeugen, deren HHH-Mengen einander ähneln.

Wieder wurden die HHH-Mengen der Datenströme mit dem Full Ancestry Algorithmus mit $\varepsilon = 5 \cdot 10^{-4}$ und $\phi = 2 \cdot 10^{-3}$ berechnet, die Klassifikation als Editor oder Nicht-Editor erfolgte mittels k-NN. Tabelle 8.6 zeigt die Ergebnisse bei Verwendung der Aufrufhierarchie, Tabelle 8.7 stellt die Ergebnisse bei Verwendung der Aufruf- und Sequenzhierarchie dar.

Es gibt keinen Hinweis darauf, dass bei diesen Versuchen etwas über Editoren gelernt wurde. Die Fehler in Tabelle 8.6 sind geringer als in Tabelle 8.3, allerdings ist das Problem einfacher, da nur zwei Klassen voneinander getrennt werden mussten. Am Beispiel von $k = 1$ wird dies deutlich. Aus Tabelle 8.3 ist bekannt, dass in vielen Fällen der nächste Nachbar einer Logdatei L_1 eine Logdatei L_2 ist, die von derselben Anwendung erzeugt

k	OGM	COR	MCOR	MOGM	DSM	TF
1	6.3	6.7	7.3	5.3	5.0	7.3
3	7.7	8.3	9.0	6.7	5.0	9.3
5	10.0	11.0	10.3	8.0	6.3	10.7
7	7.7	10.0	10.3	7.0	6.3	12.3
9	8.3	11.3	12.0	7.0	4.7	14.7

Tabelle 8.7: Klassifikationsfehler bei Erkennen der Anwendungsgruppe unter Verwendung der Aufruf- und Sequenzhierarchie in Prozent. ε wurde auf $5 \cdot 10^{-4}$ und ϕ wurde auf $2 \cdot 10^{-3}$ gesetzt.

wurde. In diesem Fall wird in Tabelle 8.6 die Logdatei L_1 korrekt als Editor/Nicht-Editor klassifiziert, weil der nächste Nachbar L_2 sogar von derselben Anwendung erzeugt wurde. Wenn das Verfahren darüber hinaus etwas über Editoren/Nicht-Editoren gelernt hätte, müssten in den Fällen, in denen der nächste Nachbar von einer *anderen* Anwendung erzeugt wurde (die Fehlerfälle in Tabelle 8.3), Editoren häufiger nächste Nachbarn von Editoren sein, als dies aufgrund ihrer Häufigkeit zu erwarten wäre. Nicht-Editoren müssten häufiger nächste Nachbarn von Nicht-Editoren sein, als dies aufgrund ihrer Häufigkeit zu erwarten wäre.

Dies ist nicht der Fall: In Tabelle 8.3 liegt der durchschnittliche Fehler für $k = 1$ bei 10.06, bei 300 Logdateien wird für durchschnittlich 30.18 Logdateien die Anwendung nicht erkannt. Nimmt man an, dass jede Anwendung gleich häufig falsch klassifiziert und bei einer Fehlklassifikation jede der verbliebenen neun Anwendungen gleich häufig gewählt wird, müssten $30.18 \cdot \frac{4}{10} \cdot \frac{3}{9} = 4.024$ Editoren zusätzlich zufällig korrekt klassifiziert und $30.18 \cdot \frac{6}{10} \cdot \frac{5}{9} = 10.06$ Nicht-Editoren zusätzlich korrekt klassifiziert werden. Bereits die Vereinfachung des Problems auf zwei Klassen sollte also eine Verbesserung von 14.084 zusätzlich korrekt klassifizierten Logdateien verursachen. Tatsächlich werden in Tabelle 8.6 durchschnittlich 18.78 Logdateien falsch klassifiziert, die Verbesserung von 11.4 Dateien im Vergleich zu den 30.18 Logdateien in Tabelle 8.3, die nun zusätzlich korrekt klassifiziert werden, ist also durch die Vereinfachung des Problems zu erklären. Für $k > 1$ sind die Überlegungen komplizierter, weil korrekt klassifizierte Anwendungen überstimmt werden können. Bei Verwendung der Aufruf- und Sequenzhierarchie bestätigt sich für $k = 1$ das Ergebnis, dass nichts über Editoren gelernt wurde.

8.5 Merkmalsselektion

Im Folgenden werden Experimente zur Merkmalsselektion dargestellt. Die Merkmale sind die Systemaufrufe. Die Experimente sollen zwei Fragen beantworten.

Erstens soll ermittelt werden, ob es für eine feste Menge von Klassen möglich ist, die Kosten des Protokollierens der Systemaufrufe durch Merkmalsselektion zu reduzieren. In diesem Fall könnte man zunächst einen Datensatz für diese Klassen erstellen, der alle Systemaufrufe enthält und auf diesem mittels Merkmalsselektion die relevanten Aufrufe bestimmen. Im späteren Produktionsbetrieb, etwa bei der Intrusion Detection, müssten dann nur noch die relevanten Aufrufe erfasst werden. Diese Art der Merkmalsselektion wird nachfolgend als *normale Merkmalsselektion* bezeichnet.

Zweitens soll untersucht werden, wie gut sich durch Merkmalsselektion ermittelte Aufrufmengen, die für das Trennen bestimmter Klassen relevant sind, für das Trennen *anderer* Klassen verwenden lassen. Die Motivation wird in Abschnitt 6.3.2 genauer erläutert. Diese Art der Merkmalsselektion wird nachfolgend als *erweiterte Merkmalsselektion* bezeichnet.

Um die Rechenzeiten zu reduzieren, wurden beide Arten von Experimenten zusammen ausgeführt, so dass die Kosten der Merkmalsselektion nur einmal anfielen und die ermittelten Merkmalsmengen für beide Experimente verwendet werden konnten. Dazu

	Alle Merkmale	Normale Selektion	Erweiterte Selektion
Korrekt klassifiziert	35.11 ± 9.53	72.84 ± 9.49	65.72 ± 11.40

Tabelle 8.8: Merkmalsselektion auf dem UCI Automobile Data Set. Vorhersage des Herstellers mit 1-NN, korrekt klassifizierte Beispiele in Prozent.

wurde ein eigens entwickelter RapidMiner-Operator UnseenClassValidation verwendet, der beide Arten der Merkmalsselektion bewertet. In Abschnitt 6.3.2 wird der Operator beschrieben.

Der Operator wird mit dem Wrapperansatz der Merkmalsselektion verwendet. Es wird also während der Suche im Teilmengenverband der Merkmale für jede untersuchte Merkmalsmenge die Qualität durch Anwendung des Lernverfahrens mit dieser Merkmalsmenge bestimmt. Derartige Wrapperansätze sind rechenintensiv. Im vorliegenden Fall sind die Rechenzeiten besonders lang, weil für jede untersuchte Merkmalsmenge für jedes einzelne Beispiel (Logdatei) der Algorithmus zur Berechnung der HHH-Menge bei ausschließlicher Verwendung der gewählten Merkmale (Systemaufrufe) ausgeführt werden muss. Die gesamte Merkmalsselektion und Evaluierung wird mehrfach ausgeführt, um die Stabilität der Ergebnisse einschätzen zu können. Auch dies trägt zu langen Rechenzeiten bei. Als Merkmalsselektion wurde nur eine einfache Vorwärtsselektion verwendet, damit die Rechenzeiten handhabbar blieben.

Um die grundsätzliche Funktionsfähigkeit des Ansatzes der erweiterten Merkmalsselektion zu zeigen, wurden zunächst Experimente mit einem einfacheren Problem durchgeführt. Dazu wurde das UCI Automobile Data Set („1985 Auto Imports Database“) aus dem UCI Machine Learning Repository verwendet [Asuncion und Newman, 2007]. Der Datensatz enthält 205 Beispiele mit 26 Attributen. Nach Entfernen eines Attributs mit 42 fehlenden Werten („normalized-losses“) und anschließendem Entfernen aller Beispiele, die dennoch fehlende Werte hatten, verblieben 193 Beispiele mit 25 Attributen. Das Attribut „make“ (Hersteller) mit 22 Werten wurde als Klassenlabel verwendet. Die erweiterte Merkmalsselektion wurde also auf Beispielen von elf Herstellern durchgeführt und auf Beispielen der anderen elf Hersteller bewertet. Als Lernverfahren wurde k-NN gewählt, die Ergebnisse wurden mit Leave-One-Out-Kreuzvalidierung geschätzt. Zusätzlich zu den regulären Attributen wurden 26 standardnormalverteilte Rauschattribute hinzugefügt.

Tabelle 8.8 zeigt die Ergebnisse für das UCI Automobile Data Set. Der Operator UnseenClassValidation wählte und bewertete 1000 Merkmalsmengen. Der erste innere Operator führte eine Vorwärtsselektion nach Wrapperansatz mit k-NN als Lerner durch, der zweite innere Operator bewertete die Merkmalsmengen mit k-NN. Die normale Merkmalsselektion (Spalte: Normal) führt zu deutlich besseren Ergebnissen als die Verwendung aller Attribute (Spalte: Alle). Die erweiterte Merkmalsselektion (Spalte: Erweitert) führt ebenfalls zu deutlich besseren Ergebnissen als die Verwendung aller Attribute. Der Fehler ist allerdings etwas größer als bei der normalen Merkmalsselektion.

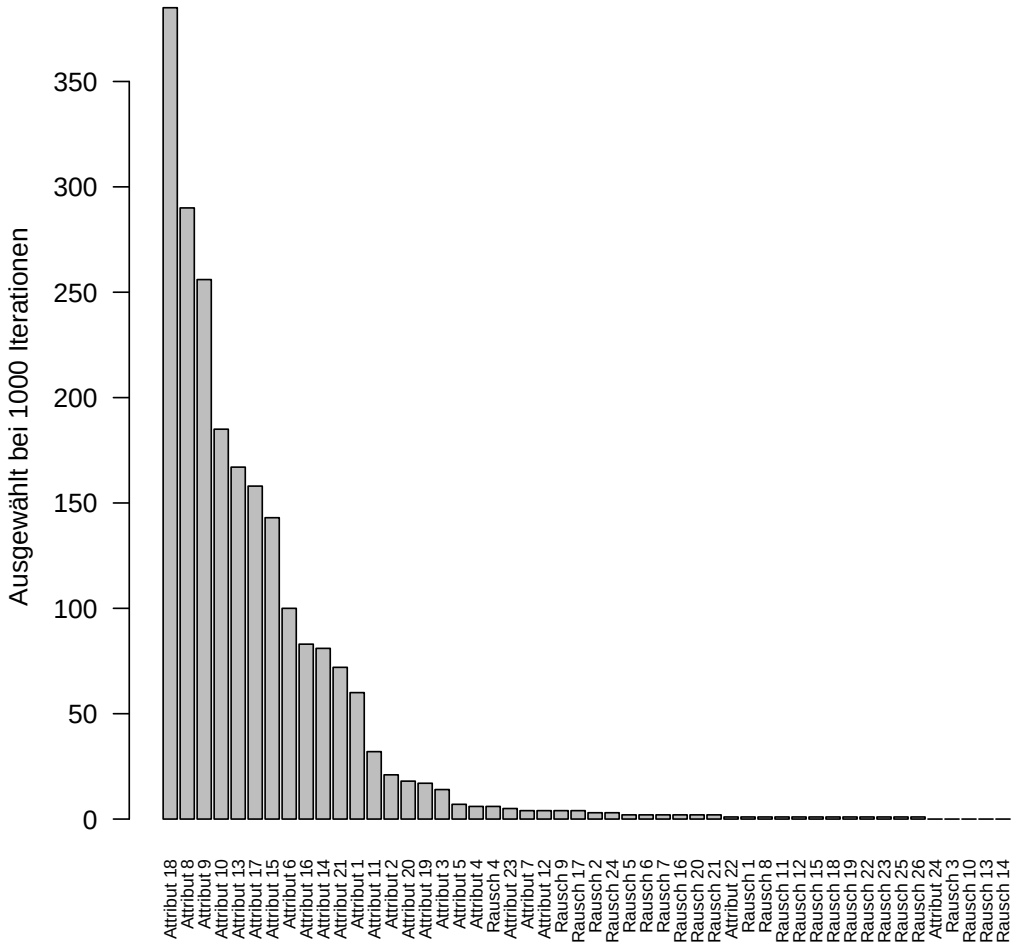


Abbildung 8.2: Merkmalsselektion UCI Automobile Data Set, ausgewählte Attribute

In diesem Fall funktioniert die erweiterte Merkmalsselektion also, führt aber zu einem größeren Fehler als die normale Merkmalsselektion.

Abbildung 8.2 zeigt, wie oft die einzelnen Attribute bei der Merkmalsselektion ausgewählt wurden. Die Rauschattribute wurden fast nie gewählt und unter den echten Attributen wurden meist dieselben Attribute gewählt.

Wegen der hohen Rechenzeiten waren für die Merkmalsselektion auf Systemaufrufdaten nur zwanzig Iterationen möglich. Verwendet wurde die Aufrufhierarchie, $\varepsilon = 5 \cdot 10^{-4}$, $\phi = 2 \cdot 10^{-3}$, Ähnlichkeitsmaß MCOR und 5-NN. In Abbildung 8.3 ist angegeben, wie oft die einzelnen Systemaufrufe ausgewählt wurden. 26 Aufrufe wurden nie ausgewählt,

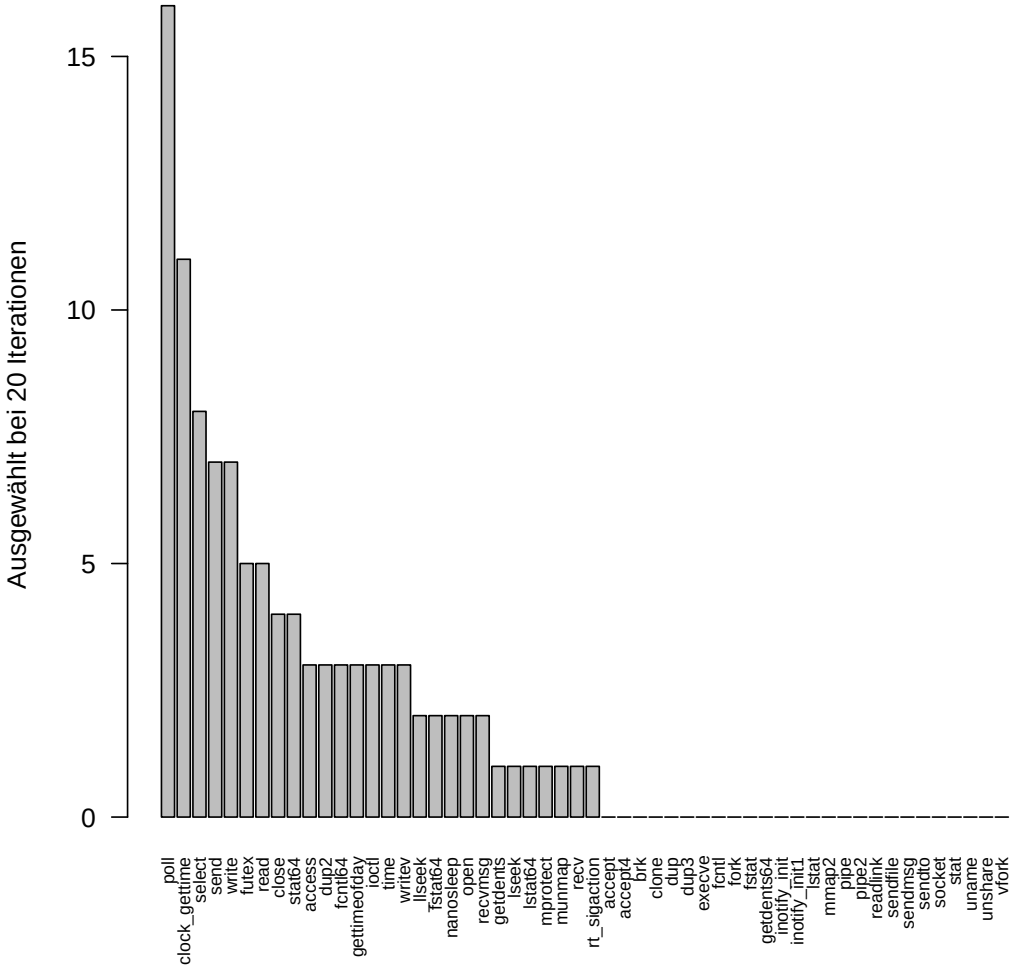


Abbildung 8.3: Merkmalsselektion für Systemaufrufe, ausgewählte Aufrufe

nur 21 von 54 Aufrufen wurden mehr als einmal und nur sieben Aufrufe wurden in mindestens 25 % der Fälle gewählt.

Tabelle 8.9 zeigt die Ergebnisse der Merkmalsselektion auf Systemaufrufdaten. Angegeben ist der Klassifikationsfehler bei Verwendung aller Aufrufe, bei normaler und bei erweiterter Merkmalsselektion. Drei Dinge sind zu erkennen:

Erstens ist die Streuung bereits bei Verwendung aller Aufrufe relativ groß, die Fehler liegen zwischen 1.3 % und 12.93 %. Dies wird vermutlich verursacht durch die zufällige Auswahl der fünf verwendeten Klassen durch den Operator (siehe Abschnitt 6.3.2). Da die Klassen unterschiedlich schwierig voneinander zu trennen sind, variieren die Fehler.

Iteration	Normal	Alle	Erweitert
1	9.4	4.7	55.0
2	0.7	13.0	15.0
3	21.6	2.3	3.9
4	9.8	3.6	41.3
5	5.9	3.3	11.7
6	3.3	6.0	12.0
7	3.0	10.0	26.2
8	15.9	1.3	8.4
9	7.9	7.3	5.9
10	16.8	1.3	14.2
11	4.9	11.1	34.1
12	7.3	3.6	11.9
13	13.2	3.0	9.2
14	3.3	7.2	6.9
15	4.4	12.2	28.1
16	2.0	12.6	37.9
17	7.4	8.0	23.4
18	2.7	7.7	13.3
19	3.0	5.4	25.8
20	6.0	2.3	3.7
Mittelwert	7.4	6.3	19.4
Standardabweichung	5.6	3.9	14.1

Tabelle 8.9: Merkmalsselektion für Systemaufrufe, Klassifikationsfehler in Prozent. ε wurde auf $5 \cdot 10^{-4}$ und ϕ wurde auf $2 \cdot 10^{-3}$ gesetzt.

Zweitens ist es in diesem Fall möglich, die Kosten des Protokollierens der Systemaufrufe durch normale Merkmalsselektion zu reduzieren. Allerdings muss ein leichter Anstieg des durchschnittlichen Fehlers (von 6.29 % auf 7.42 %) und ein Anstieg der Standardabweichung in Kauf genommen werden.

Drittens ist zu erkennen, dass in diesem Fall die erweiterte Merkmalsselektion nicht funktioniert. Mengen von Systemaufrufen, die durch Merkmalsselektion auf fünf Anwendungen ermittelt wurden, lassen sich nicht zuverlässig für neue Anwendungen verwenden.

Der Fehler[1] ist bei normaler Merkmalsselektion im Vergleich zur Verwendung aller Merkmale etwas größer. Bei der erweiterten Merkmalsselektion ist der Fehler dagegen deutlich größer als bei Verwendung aller Merkmale (19.41 % gegenüber 6.29 %) und die Streuung ist sehr hoch (Standardabweichung 14.12 %). In der ersten Zeile liegt der Fehler bei Verwendung aller Aufrufe bei 4.7 %, bei normaler Merkmalsselektion bei 9.4 % und bei erweiterter Merkmalsselektion bei 55.03 %. Unter diesen Umständen ist es nicht sinnvoll,

[1] Die Fehlerwerte in den drei Spalten sind untereinander, aber nicht mit den Fehlerwerten in Tabelle 8.3 vergleichbar, weil hier nur 5 Klassen voneinander getrennt werden müssen.

die Ergebnisse einer Merkmalsselektion für andere Klassen zu verwenden. Die Merkmalsselektion sollte nur für eine feste Menge von Klassen erfolgen.

Die 26 niemals ausgewählten Aufrufe sind für weniger als 0.7 % aller ausgeführten Systemaufrufe im Anwendungsdatensatz verantwortlich. Diese Aufrufe nicht zu protokollieren, führt nicht zu einer wesentlichen Ersparnis.

Der Anwendungsdatensatz enthält nur zehn verschiedene Anwendungen. Möglicherweise wären mit einer größeren Anzahl von Anwendungen stabilere Ergebnisse zu erreichen.

9 Erkennen von Benutzern

9.1 Beschreibung des Datensatzes

Der Benutzerdatensatz wurde vom Lehrstuhl Informatik 12 der TU Dortmund zur Verfügung gestellt. Protokolliert wurde eine Teilmenge der Betriebssystemaufrufe der Benutzer eines Serverrechners über einen Zeitraum von einer Woche.

Die Logdateien haben eine Gesamtgröße von 41 GiB, bei Kompression mit gzip ist die Gesamtgröße 2.33 GiB. Die Dateien enthalten etwa 340 Millionen Systemaufrufe. Es gibt 34 verschiedene Benutzer, die zwölf Benutzergruppen zugeordnet sind. Die Benutzer werden über ihre User ID (uid) identifiziert und sind Mitglied von Gruppen, die über eine Group ID (gid) identifiziert werden. Nur zwei Gruppen haben mehr als einen Benutzer: Die Gruppe der Mitarbeiter (gid: 1000) hat vier Benutzer, die Gruppe der Studenten (gid: 10000) zwanzig. Die übrigen zehn Benutzer sind Root und Dienste wie der Mailserver.

Protokolliert wurden lediglich zwei Betriebssystemaufrufe: `close` zum Schließen einer Datei und `execve` zum Ausführen eines Programms. Abbildung 9.1 zeigt einen Ausschnitt einer Logdatei. Es wird jeweils der Name des Betriebssystemaufrufs (syscall), ein Zeitstempel (time), die Prozess-ID (pid), die ID des Elternprozesses (ppid), der Name des ausgeführten Programms (execname), die ID des Benutzers (uid) und seine Gruppe (gid) angegeben. Für `close`-Aufrufe wird zusätzlich der vollständige Name (einschl. Pfad) der zu schließenden Datei (file) angegeben. Für `execve`-Aufrufe wird zusätzlich der vollständige Name der Datei genannt, in der sich das zu startende Programm befindet (param).

Die Verteilung ist schief: 260 Millionen der 340 Millionen Aufrufe werden von einem einzelnen Benutzer (uid: 10026) verursacht, elf der 34 Benutzer verursachen weniger als 1000 Aufrufe und die weniger aktive Hälfte der Benutzer verursacht zusammen weniger als 1.3 ‰ der Aufrufe. 0.6 ‰ der Aufrufe sind `execve`-Aufrufe, die übrigen 99.94 % sind `close`-Aufrufe.

```
syscall: close, time: 5728032676, pid: 1701, ppid: 1700, execname: maxlifetime, file: /etc/php5/, uid: 0, gid: 0
syscall: close, time: 5728033453, pid: 1703, ppid: 1702, execname: maxlifetime, file: /dev/null, uid: 0, gid: 0
syscall: compat_execve, time: 5728033497, pid: 1703, ppid: 1702, execname: maxlifetime, param: /bin/sed , uid: 0, gid: 0
syscall: close, time: 5729174549, pid: 1703, ppid: 1702, execname: sed, file: /etc/ld.so.cache, uid: 0, gid: 0
syscall: close, time: 5729174601, pid: 1703, ppid: 1702, execname: sed, file: /lib/i686/cmov/libc.so.6, uid: 0, gid: 0
```

Tabelle 9.1: Ausschnitt aus dem Benutzerdatensatz. Zwischen den `close`-Aufrufen gibt es auch vereinzelte `execve`-Aufrufe.

syscall: close, time: 7620201552, pid: 29137, ppid: 21481, execname: htop, file: /proc/25895/task, uid: 10026, gid: 10000
syscall: close, time: 7620201574, pid: 29137, ppid: 21481, execname: htop, file: /proc/25895/statm, uid: 10026, gid: 10000
syscall: close, time: 7620201615, pid: 29137, ppid: 21481, execname: htop, file: /proc/25895/stat, uid: 10026, gid: 10000
syscall: close, time: 7620201638, pid: 29137, ppid: 21481, execname: htop, file: /proc/26269/task, uid: 10026, gid: 10000
syscall: close, time: 7620201662, pid: 29137, ppid: 21481, execname: htop, file: /proc/26269/statm, uid: 10026, gid: 10000
syscall: close, time: 7620201701, pid: 29137, ppid: 2148syscall: close, time: 7606314122, pid: 24800, ppid: 1, gxecna0000
syscall: close, time: 7620201725, pid: 29137, ppid: 21481, execname: htop, file: /proc/26757/task, uid: 10026, gid: 10000
syscall: close, time: 7620201746, pid: 29137, ppid: 21481, execname: htop, file: /proc/26757/statm, uid: 10026, gid: 10000
syscall: close, time: 7620201785, pid: 29137, ppid: 21481, execname: htop, file: /proc/26757/stat, uid: 10026, gid: 10000
syscall: close, time: 7620201806, pid: 29137, ppid: 21481, execname: htop, file: /proc/26971/task, uid: 10026, gid: 10000
syscall: close, time: 7620201827, pid: 29137, ppid: 21481, execname: htop, file: /proc/26971/statm, uid: 10026, gid: 10000
syscall: close, time: 7620201867, pid: 29137, ppid: 21481, execname: htop, file: /proc/26971/stat, uid: 10026, gid: 10000

Tabelle 9.2: Ausschnitt aus dem Benutzerdatensatz, Datei `file0000147.log`. Im mittleren Bereich ist eine fehlerhafte Zeile zu erkennen.

Für die folgenden Experimente wurde der Dateiname der zu schließenden Datei als hierarchische Variable verwendet.

In der Regel enthält eine Zeile der Logdateien die Daten eines Betriebssystemaufrufs. In einigen Fällen werden allerdings Zeilenumbrüche innerhalb der Parameter eines Programmaufrufs nicht korrekt behandelt, so dass vereinzelt zwischen den eigentlichen Daten mehrere Zeilen mit kleinen Skriptfragmenten oder Parameterinformationen auftreten, die eigentlich in der Zeile des Betriebssystemaufrufs dargestellt werden müssten. Wegen der Größe des Datensatzes wurden die Dateien nicht korrigiert, sondern das Problem stattdessen beim Einlesen der Daten berücksichtigt.

An einigen anderen Stellen weisen die Dateien offensichtliche Fehler auf: Zeilen sind unvollständig, Informationen fehlen oder sind doppelt vorhanden. Möglicherweise entstanden diese Fehler durch eine defekte Festplatte, mit der die Daten vom Lehrstuhl 12 zum Lehrstuhl 8 transportiert wurden. Möglicherweise wurden auch während des Logvorgangs in Zeiten hoher Systemlast mehr Daten erzeugt, als protokolliert werden konnten, so dass es zu Datenverlusten kam. In den meisten Fällen wird offenbar ein Fragment einer alten Zeile (erkennbar am Zeitstempel) über die neueren Daten geschrieben (siehe Tabelle 9.2).

Wegen der Größe des Datensatzes wurden auch in diesem Fall die Dateien nicht korrigiert, stattdessen werden die fehlerhaften Zeilen beim Einlesen der Daten verworfen.

syscall: close, time: 904205, pid: 8709, ppid: 8708, execname: kded, file: /fs/students/vogt/.qt/qtrc, uid: 10027, gid: 10000

Tabelle 9.3: Ausschnitt aus dem Benutzerdatensatz mit einem Benutzernamen in den Pfaden.

	k	OGM	COR	MCOR	MOGM	DSM
Alle Pfade	1	25.3	25.3	25.3	**21.4**	25.3
	3	28.6	27.9	31.2	27.9	27.9
	5	33.1	31.8	33.8	31.8	27.9
Systempfade	1	28.8	30.1	32.7	27.5	**26.8**
	3	30.7	32.0	33.3	30.7	29.4
	5	37.3	34.6	37.3	34.0	30.7

Tabelle 9.4: Klassifikationsfehler in Prozent auf dem Benutzerdatensatz, Vorhersage der uid. ε wurde auf $5 \cdot 10^{-4}$ und ϕ wurde auf $2 \cdot 10^{-3}$ gesetzt.

9.2 Behandlung der Benutzernamen in den Pfaden

Wenn eine Datei verwendet wird, die keine Systemdatei ist, sondern im Verzeichnis des Benutzers liegt, ist der Name des Benutzers Bestandteil des Dateisystempfads (siehe Tabelle 9.3). Anhand dieser Information ist der Benutzer also leicht zu erkennen. Benutzer können zwar auch Dateien in Verzeichnissen fremder Benutzer verwenden, sofern sie über ausreichende Rechte verfügen, doch in den meisten Fällen ist die Identität des Benutzers anhand der Verzeichnisse der verwendeten Dateien gut ablesbar. Diese Information im Rahmen der Benutzererkennung zu verwenden, kann also problematisch sein. Nur den Benutzernamen aus den Pfaden zu entfernen, löst das Problem nicht, denn der gesamte Verzeichnisbaum des Benutzers ist kennzeichnend. Verschiedene Benutzer haben verschiedene Verzeichnisbäume, ein Benutzer ist also anhand einzelner Pfade seines Verzeichnisbaumes eindeutig identifizierbar.

Ich habe daher die Experimente in den Abschnitten 9.3 und 9.4 jeweils zweifach ausgeführt. Einmal wurden alle Pfade verwendet, ein anderes Mal wurden nur Pfade verwendet, die außerhalb der Verzeichnisbäume der einzelnen Benutzer liegen. Über den richtigen Umgang mit den Verzeichnisbäumen der einzelnen Benutzer muss für den konkreten Anwendungsfall entschieden werden. Ebenfalls vom Anwendungsfall abhängig ist, ob die Verzeichnisbäume unterhalb des `/proc` und `/sys`-Verzeichnisses verwendet werden sollten. Diese Verzeichnisse enthalten virtuelle Dateien, die eine Schnittstelle zum Kernel bilden. Sie stellen Informationen über Prozesse, Geräte und Treiber bereit und ermöglichen die Konfiguration. Für die hier dargestellten Experimente wurden diese Dateien verwendet.

9.3 Erkennen von Benutzern

Zunächst habe ich versucht, einzelne Benutzer anhand der von ihnen verwendeten Dateien zu erkennen. Um eine ausreichende Anzahl von Trainingsbeispielen zu erhalten, habe ich die vorhandenen Daten in zehn Teile geteilt und für jedes Teilstück für jeden Benutzer die HHH-Mengen berechnet. Wenn ein Benutzer in einem Teilstück nicht aktiv war, wurde die berechnete (leere) HHH-Menge nicht berücksichtigt. War ein Benutzer in

weniger als drei Abschnitten aktiv, wurde er nicht berücksichtigt. Insgesamt ergab dies 155 Beispiele von 26 Benutzern. Nach Anwendung des Full Ancestry Algorithmus für $\varepsilon = 5 \cdot 10^{-4}$ und $\phi = 2 \cdot 10^{-3}$ wurde die uid mit dem k-NN-Algorithmus vorhergesagt. Es wurden verschiedene Ähnlichkeitsmaße und verschiedene Werte für k verwendet, die angegebenen Werte sind mit Leave-One-Out kreuzvalidiert.

Die Ergebnisse sind in Tabelle 9.4 dargestellt. Der erreichte Fehler ist mit etwa 25 % - 35 % nicht sehr niedrig. Da 26 Klassen mit nur 155 Beispielen gelernt werden mussten, ist das Ergebnis nicht zufällig, ein Lernvorgang hat stattgefunden. Die ausschließliche Verwendung von Pfaden, die außerhalb der Verzeichnisbäume der einzelnen Benutzer liegen, führt zu höheren Fehlern als die Verwendung aller Pfade.

9.4 Erkennen von Benutzergruppen

Anschließend habe ich versucht, einzelne Benutzer anhand der verwendeten Dateien ihrer Benutzergruppe zuzuordnen. Man könnte erwarten, dass die Mitglieder einer Gruppe einander ähnlicher sind als Mitgliedern anderer Gruppen, dann wäre die Zuordnung eines Benutzers zur Gruppe der ähnlichsten Benutzer erfolgversprechend.

Die Daten sind für die Untersuchung dieser Frage nur bedingt geeignet: Nur zwei Gruppen haben mehr als einen Benutzer, eine hat zwanzig Benutzer (Studenten), die andere vier Benutzer (Mitarbeiter). Das ist wenig. Zudem sind sich die Benutzergruppen recht ähnlich.

Da nur zwei Gruppen mehr als einen Benutzer enthalten, wurden die Mitglieder der anderen zehn Gruppen nicht beachtet. Für jeden der verbleibenden 24 Benutzer wurden die Mengen der Hierarchical Heavy Hitters auf Grundlage aller zur Verfügung stehender Daten berechnet. Die Parameter wurden gewählt wie in Abschnitt 9.3 beschrieben. Anschließend wurden die berechneten HHH-Mengen mit den HHH-Mengen der anderen Benutzer verglichen und die gid der ähnlichsten Benutzer vergeben.

Die Ergebnisse sind in Tabelle 9.5 einmal mit und einmal ohne Verwendung der Verzeichnisse der einzelnen Benutzer dargestellt. Die Ergebnisse sind eindeutig: Eine Zuordnung

	k	OGM	COR	MCOR	MOGM	DSM
Alle Pfade	1	20.8	29.2	29.2	25.0	25.0
	3	**16.7**	**16.7**	**16.7**	**16.7**	**16.7**
	5	**16.7**	**16.7**	**16.7**	**16.7**	**16.7**
Systempfade	1	25.0	29.2	29.2	25.0	25.0
	3	**16.7**	**16.7**	**16.7**	**16.7**	29.2
	5	**16.7**	**16.7**	**16.7**	**16.7**	20.8

Tabelle 9.5: Klassifikationsfehler in Prozent auf dem Benutzerdatensatz, Vorhersage der gid. ε wurde auf $5 \cdot 10^{-4}$ und ϕ wurde auf $2 \cdot 10^{-3}$ gesetzt.

der gid ist nicht möglich: Im günstigsten Fall entspricht der Fehler mit 16.67 % dem Fehler bei naiver Zuordnung zur häufigsten Gruppe. Tatsächlich wird in den meisten Experimenten, die diesen Fehler aufweisen, jeder Benutzer als Student eingeordnet.

Warum ist das so? Betrachtet man die HHH-Mengen der einzelnen Benutzer, fällt auf, dass sie viele Elemente enthalten, die für die Einordnung in eine Benutzergruppe sehr geringe Bedeutung haben. Die Benutzer 1002 (Mitarbeiter), 10026 und 10029 (Studenten) besitzen ähnliche HHH-Mengen, in denen fast alle Elemente aus dem /proc Bereich stammen. Im /proc-Bereich des Dateisystems werden Informationen über laufende Prozesse angeboten, die von Programmen wie top oder htop gelesen und angezeigt werden können. In den Logdateien ist zu sehen, dass die drei genannten Benutzer htop verwenden, dabei werden so viele Dateizugriffe im /proc-Bereich verursacht, dass die HHH-Mengen fast nur /proc-Elemente enthalten. Durch die Fokussierung der HHH-Algorithmen auf Häufigkeiten geht die Information über seltenere Zugriffe, die für die Trennung der Benutzergruppen wichtiger sein könnte, verloren. Da Mitarbeiter und Studenten htop verwenden, sind die HHH-Mengen in diesem Fall für die Trennung der Gruppen ungeeignet. Das Problem ist nicht auf die Verwendung von htop beschränkt. Die Anzahl der Aufrufe, die verschiedene Anwendungen produzieren (und die Anzahl der verwendeten Dateien), ist sehr unterschiedlich. Aktivere Anwendungen, die viele Aufrufe und Dateizugriffe verursachen, dominieren die HHH-Mengen.

Daher erfassen die HHH-Mengen nur einen kleinen Ausschnitt des Verhaltens der Benutzer. Ist dieser Ausschnitt, wie im Falle der Verwendung von htop, sehr unspezifisch, so entspricht die von den Ähnlichkeitsmaßen gemessene Ähnlichkeit nicht der intuitiven Vorstellung der Ähnlichkeit der Benutzer. Die Klassifikation scheitert folglich.

Man könnte die Dateizugriffe von unspezifischen Anwendungen bei der Berechnung der HHH-Mengen ausblenden, um möglicherweise bessere Ergebnisse zu erzielen. Drei Gründe lassen vermuten, dass das nicht sehr Erfolg versprechend wäre:

- Das Grundproblem, nämlich die ausschließliche Berücksichtigung von Häufigkeiten, bleibt bestehen. Anwendungen produzieren unterschiedlich viele Aufrufe, und egal wie viele Anwendungen man ausblendet, die aktiveren verbleibenden Anwendungen dominieren die HHH-Mengen, die Gewichtung erfolgt ausschließlich nach Häufigkeit, nicht nach Relevanz. Die HHH-Mengen und damit die Ähnlichkeiten erfassen nicht das gesamte Spektrum des Datenstroms, sondern werden von wenigen Anwendungen oder einer einzelnen Anwendung dominiert. Unter diesen Bedingungen sind keine stabilen Ergebnisse zu erwarten.

- Man bräuchte umfangreichere Daten, sowohl zum Lernen als auch zum Validieren des Ausblendens. Beginnt man die Dateizugriffe einzelner Anwendungen auszublenden, weil sie nicht zu den gewünschten Ergebnissen führen, erreicht man lediglich eine Überanpassung an den bestehenden Datensatz von 24 Beispielen.

- Wenn man aber im Rahmen des Ausblendens ohnehin Hintergrundwissen über Anwendungen und ihre Relevanz für die Trennung der Benutzergruppen verwendet, dann kann man mit diesem Hintergrundwissen auch direkt bessere Methoden der Gruppentrennung entwickeln.

Einzelne Aspekte des Verhaltens eines Benutzers (wie die Verwendung von htop), die für die Einordnung in eine Benutzergruppe von relativ geringer Bedeutung sind, können wegen der Fokussierung der HHH-Algorithmen auf Häufigkeiten einen sehr starken Einfluss auf die berechneten HHH-Mengen haben und andere Aspekte fast völlig überdecken. Daher entsprechen die Ähnlichkeiten der HHH-Mengen verschiedener Benutzer nicht unserer intuitiven Vorstellung von der Ähnlichkeit dieser Benutzer. Der Versuch, Benutzer über die Ähnlichkeit ihrer HHH-Mengen in Gruppen einzuordnen, scheitert.

Diese Ergebnisse zeigen deutliche Parallelen zu den Ergebnissen der Anwendungsklassifikation.

Dort war Generalisierung auf der Ebene der Anwendungen möglich. Die Logdateien verschiedener Ausführungen derselben Anwendung waren unterschiedlich, doch die berechneten HHH-Mengen verschiedener Ausführungen derselben Anwendung waren einander ähnlicher als dies für verschiedene Anwendungen der Fall war. Die Ähnlichkeiten der HHH-Mengen standen nicht im Widerspruch zur intuitiven Vorstellung von Ähnlichkeit. Auf der Ebene von Anwendungstypen (Editor/kein editor) war dagegen keine Generalisierung möglich. Die Tatsache, dass Editoren eine ähnliche Aufgabe lösen, führte nicht dazu, dass die HHH-Mengen der Editoren größere Ähnlichkeit aufwiesen. Die häufigkeitsbasierten HHH-Algorithmen waren nicht in der Lage, die Ähnlichkeit von Editoren zu erfassen. Daher stand die Ähnlichkeit der HHH-Mengen im Widerspruch zur intuitiven Ähnlichkeit. Eine erfolgreiche Klassifikation war also nicht möglich.

10 Zusammenfassung

Klassische Methoden zur Zusammenfassung von Datenströmen verwenden die Häufigkeiten der Stromelemente ([Manku und Motwani, 2002]) oder andere flache Verfahren zur Berechnung einer kondensierten Darstellung des Datenstroms. Die Stromelemente weisen allerdings oft eine hierarchische Struktur auf. Die darin enthaltene Information geht bei einer flachen Zusammenfassung des Stroms verloren. Neuere Verfahren berücksichtigen bei der Berechnung der kondensierten Darstellung des Datenstroms diese hierarchische Struktur. Algorithmen zur Berechnung der Hierarchical Heavy Hitters (HHH) sind ein solches Verfahren.

Gegenstand dieser Diplomarbeit war es, diesen Ansatz auf Datenströme von Systemaufrufen zu übertragen und kondensierte Darstellungen von Strömen von Systemaufrufen zu erzeugen, welche die hierarchische Struktur der Stromelemente erfassen.

Der folgende Abschnitt fasst zunächst die hierarchischen Variablen zusammen, die in dieser Arbeit als Träger der hierarchischen Information der Stromelemente verwendet wurden. Anschließend werden die Ergebnisse der Experimente zu Laufzeit, Speicherbedarf und Approximationsgüte der HHH-Algorithmen dargestellt. Im Rahmen dieser Arbeit wurden Ähnlichkeitsmaße entwickelt, um die kondensierten Darstellungen der Datenströme unter Berücksichtigung ihrer hierarchischen Struktur vergleichen zu können. Diese Ähnlichkeitsmaße werden kurz zusammengefasst, anschließend werden die Ergebnisse der Experimente zur Anwendungs- und Benutzerklassifikation mit diesen Ähnlichkeitsmaßen dargestellt. Zuletzt werden die Ergebnisse der Merkmalsselektion beschrieben, welche die hohen Kosten des Erfassens aller Systemaufrufe senken sollte.

Im letzten Abschnitt werden einige Ansatzpunkte für Verbesserungen und Erweiterungen des Verfahrens vorgeschlagen.

10.1 Zusammenfassung

Hierarchische Variablen

In dieser Arbeit wurden drei hierarchische Variablen als Träger der hierarchischen Struktur der Stromelemente verwendet. Die einfachste Variable ist die hierarchische Sequenzvariable. Sie enthält als Information über den Ausführungskontext eines Aufrufs die Namen der zuvor ausgeführten Aufrufe. Sie ist algorithmisch sehr leicht zu erfassen, und da sie keine Parameter des Systemaufrufs verwendet, kann sie auch mit Logwerkzeugen eingesetzt werden, die keine Parameter der Aufrufe erfassen können. Die zweite hier-

archische Variable ist die Aufrufvariable. Sie teilt die Aufrufe anhand ihrer Funktion in Gruppen ein und verfeinert die Aufteilung anhand der verwendeten Parameterwerte. Die dritte Variable ist die Pfadvariable, die Dateinamen und Präfixe von Dateinamen umfasst. Ihre Extraktion aus den verwendeten strace-Logdateien war sehr aufwändig, weil zur Auflösung der Dateideskriptoren in Pfade ein Teil des Betriebssystems simuliert werden musste.

Algorithmen zur Berechnung der Hierarchical Heavy Hitters

Der Strom der extrahierten hierarchischen Variablen bildet die Eingabe für die Algorithmen zur Berechnung der Hierarchical Heavy Hitters. Die Variablen können einzeln oder in Kombination verwendet werden. Bei den in Kapitel 7 dargestellten Experimenten stiegen Laufzeit und Speicherbedarf der beiden Algorithmen Full Ancestry und Partial Ancestry bei steigender Zahl der Variablen stark an. Die Laufzeiten des Full Ancestry Algorithmus waren meist besser als die Laufzeiten des Partial Ancestry Algorithmus. Letzterer benötigte dagegen weniger Speicherplatz. Da der Speicherbedarf für beide Algorithmen niedrig war, fiel dieser Vorteil allerdings kaum ins Gewicht. Die Ausgaben dieser beiden Approximationsalgorithmen wurden mit den Ausgaben des ebenfalls implementierten exakten Algorithmus verglichen. Full Ancestry erzielte bessere Approximationsgüten, so dass der Full Ancestry Algorithmus die bessere Wahl zu sein schien und für die weiteren Experimente verwendet wurde. Sowohl für Full Ancestry als auch für Partial Ancestry war die Approximationsgüte über den Fehlerparameter ε auf Kosten des Speicherbedarfs gut steuerbar.

Ähnlichkeitsmaße

Im Rahmen dieser Arbeit wurden Ähnlichkeitsmaße entwickelt, um die kondensierten Darstellungen der Datenströme unter Berücksichtigung der hierarchischen Struktur vergleichen zu können. Mit diesen Ähnlichkeitsmaßen lassen sich verschiedene Datenströme zu Clustern zusammenfassen oder durch instanzbasierte Lernverfahren klassifizieren. Auch für die Berechnung der Approximationsgüte der Algorithmen wurden die Ähnlichkeitsmaße verwendet.

Die Ähnlichkeitsmaße bestimmen die Ähnlichkeit von Datenströmen, indem sie die HHH-Mengen der Datenströme oder die Datenstrukturen der HHH-Algorithmen nach Verarbeitung des Stroms vergleichen. Die Ähnlichkeitsmaße für HHH-Mengen berechnen für jedes Element der beiden Mengen einen Ähnlichkeitsbeitrag, indem sie seine Ähnlichkeit zum ähnlichsten Element der anderen Menge messen. Die Ähnlichkeit der Mengen ist der Mittelwert der Ähnlichkeitsbeiträge ihrer Elemente. Das Ähnlichkeitsmaß für Datenstrukturen vergleicht dagegen die geschätzten Häufigkeiten der Generalisierungen der Stromelemente und berechnet die Ähnlichkeit der Datenstrukturen als Mittelwert der Ähnlichkeit der geschätzten Häufigkeiten.

Anwendungsbeispiele

Als Beispiel für den Einsatz dieser hierarchischen Ähnlichkeitsmaße wurden Experimente zur Klassifikation von Datenströmen durchgeführt. Dazu wurde jeweils die kondensierte Darstellung der Datenströme berechnet. Für den zu klassifizierenden Strom wurden mit den hierarchischen Ähnlichkeitsmaßen die ähnlichsten Ströme bestimmt. Die Klassifikation erfolgte nach dem Verfahren der nächsten Nachbarn über die Klassen der ähnlichsten Ströme.

Bei den Experimenten zur Anwendungsklassifikation sollte für einzelne Datenströme erkannt werden, von welcher Anwendung sie erzeugt wurden. Dies war sehr gut möglich, selbst bei sehr kurzen Datenströmen wurden die Anwendungen meist erkannt. Die verschiedenen hierarchischen Variablen erzielten einzeln gute Ergebnisse, nur die hierarchische Pfadvariable erzielte bei kurzen Logzeiträumen schlechte Ergebnisse, weil bei kurzen Logzeiträume nicht alle Dateideskriptoren in Pfade übersetzt werden können. Die Kombination mehrerer hierarchischer Variablen führte zu etwas besseren Ergebnissen als die Verwendung einer einzelnen Variable. Ob dies den deutlich höheren Ressourcenverbrauch der HHH-Algorithmen im mehrdimensionalen Fall rechtfertigt, muss für den konkreten Anwendungsfall entschieden werden. Keines der Ähnlichkeitsmaße erwies sich als eindeutig besser geeignet, allerdings braucht beim datenstrukturbasierten Ähnlichkeitsmaß DSM der Parameter ϕ nicht angegeben zu werden, so dass die Suche nach günstigen Parameterwerten einfacher wird.

Bei Verwendung eines sehr einfachen Verfahrens, bei dem als kondensierte Darstellung des Datenstroms ein Vektor verwendet wurde, der die relative Häufigkeit der Aufrufe enthielt, fielen die Ergebnisse schlechter aus als bei der Verwendung der hierarchischen kondensierten Darstellung. Die Verwendung der hierarchischen Information der Stromelemente scheint also in diesem Fall vorteilhaft zu sein. Allerdings war der Unterschied der Ergebnisse nicht sehr groß, möglicherweise würde sich durch eine Optimierung des einfachen Verfahrens ein anderes Bild ergeben.

Der Versuch, Datenströme einer Anwendungs*gruppe* zuzuordnen und zu entscheiden, ob der Strom von einem Editor oder einem Nicht-Editor erzeugt wurde, war nicht erfolgreich. Offenbar führt die Tatsache, dass Editoren ähnliche Aufgaben erfüllen, nicht dazu, dass die HHH-Mengen Ähnlichkeit haben. Die Ähnlichkeitsmaße können daher die intuitiv vorhandene Ähnlichkeit der Editoren nicht erfassen. Es ist möglich, dass die Verwendung anderer hierarchischer Variablen zu besseren Ergebnissen geführt hätte.

Eine ähnliche Situation ergab sich bei dem Versuch der Benutzer- und Benutzergruppenerkennung. Es war möglich, Datenströme dem Benutzer zuzuordnen, der sie erzeugt hatte. Dagegen war es nicht möglich, Datenströme einer Benutzergruppe (Mitarbeiter oder Studenten) zuzuordnen. Die Ähnlichkeit der Benutzer innerhalb einer Gruppe spiegelte sich nicht in einer Ähnlichkeit der HHH-Mengen wider. Ein Grund dafür war die Dominanz aktiver Anwendungen: Da die Anzahl der Systemaufrufe, die Anwendungen verursachen, von Anwendung zu Anwendung sehr unterschiedlich ist, werden die HHH-Mengen von wenigen Anwendungen oder einer einzelnen Anwendung dominiert. Die Ähnlichkeitsmaße können daher nicht das gesamte Spektrum des Verhaltens eines Be-

nutzers messen, sondern nur den kleinen Ausschnitt sehr aktiver Anwendungen. Sind diese für die Klassifikation zu unspezifisch, spiegeln die berechneten Ähnlichkeitsmaße nicht die intuitive Ähnlichkeit wider.

Merkmalsselektion

Das Erfassen aller Systemaufrufe kann die Leistung eines Rechners beeinträchtigen. Daher wurde am Beispiel der Anwendungserkennung untersucht, ob einzelne Aufrufe wichtiger sind als andere und bereits eine Teilmenge der Aufrufe ausreicht, um Anwendungen zuverlässig zu erkennen. Dieses Problem der Merkmalsselektion wurde in zwei Varianten untersucht:

Zunächst wurde für die Anwendungserkennung gezeigt, dass es möglich ist, die Kosten des Protokollierens der Systemaufrufe durch Merkmalsselektion zu reduzieren, ohne dass der Fehler wesentlich ansteigt. Es ist also möglich, zunächst einen Datensatz zu erstellen, der alle Systemaufrufe enthält und auf diesem mittels Merkmalsselektion die relevanten Aufrufe zu bestimmen. Im späteren Produktionsbetrieb, etwa bei der Intrusion Detection, brauchen dann nur noch die relevanten Aufrufe erfasst zu werden.

Der Versuch, Aufrufmengen, die für die Trennung bestimmter Klassen relevant waren, für die Trennung *anderer* Klassen zu verwenden, war dagegen nicht erfolgreich. Aussagen über die Relevanz von Systemaufrufen waren nur bezüglich der Trennung bestimmter Klassen möglich, Aussagen über generell relevante Aufrufe konnten nicht abgeleitet werden.

10.2 Ausblick

Aufgrund des beschränkten Zeitrahmens der Diplomarbeit konnten nicht alle Ansätze für Verbesserungen und Erweiterungen verfolgt werden. Die folgenden drei Vorschläge betreffen unterschiedliche Bereiche des Systems.

- Für die Aufruf- und Sequenzvariable wurde eine speicher- und rechenzeiteffiziente Codierung als int-Wert verwendet. Für die Pfadvariable wurde dagegen eine naive Implementierung als Zeichenkette gewählt. An den Laufzeiten bei Verwendung der Pfadvariablen ist dies deutlich erkennbar. Eine geeignetere Codierung könnte sich positiv auf Rechenzeit und/oder Speicherbedarf auswirken.
- Das dargestellte System mit seinen Ähnlichkeitsmaßen und drei hierarchischen Variablen stellt eine Basis dar, mit dem allgemeine kondensierte hierarchische Darstellungen von Datenströmen von Systemaufrufen berechnet und verglichen werden können. Für den konkreten Anwendungsfall kann und sollte diese Basis unter Verwendung von Hintergrundwissen über den Anwendungsfall angepasst werden. Durch Anpassen der Ähnlichkeitsmaße und der hierarchischen Variablen und insbesondere durch Konstruktion neuer hierarchischer Variablen sollte sichergestellt werden, dass die kondensierte Darstellung des Stroms die hierarchischen Informa-

tionen enthält, die im konkreten Anwendungsfall tatsächlich benötigt werden. Dazu sind präzise Informationen über den Anwendungsfall und die zu verwendenden Daten erforderlich.

- Logdateien von Systemaufrufen weisen sehr umfangreiche Informationen über die Benutzer des Systems auf. Sie enthalten detaillierte Informationen über die Zeiten der Systemnutzung, die ausgeführten Anwendungen, die aufgerufenen Internetseiten, die Namen aller verwendeten Dateien und prinzipiell auch den Inhalt der verwendeten Dateien (dieser wird meist gekürzt erfasst, siehe Tabelle 2.2). Im Produktionsbetrieb kann es erforderlich sein, die Privatsphäre der Benutzer durch technische Maßnahmen zu schützen. Dabei ist es nicht ausreichend, die Benutzernamen zu anonymisieren, sondern es muss sichergestellt werden, dass auch indirekte Schlüsse auf die Identität der Benutzer ausgeschlossen sind. Solche indirekten Schlüsse können über die Zeiten der Systemnutzung, die verwendeten Anwendungen, die Dateinamen oder über den Inhalt der verwendeten Dateien möglich sein. Da im Rahmen dieser Diplomarbeit nur eigene Systemaufrufe vollständig erfasst wurden und der Benutzerdatensatz des Lehrstuhls 12 nur relativ unproblematische Pfadnamen enthält, wurden solche Fragen des Privacy Preserving Data Mining ([Vaidya et al., 2005]) nicht untersucht.

Literaturverzeichnis

[Aha et al., 1991] Aha, David W., D. Kibler und M. K. Albert (1991). *Instance-based Learning Algorithms*. Machine Learning, 6:37–66.

[Asuncion und Newman, 2007] Asuncion, Arthur und D. J. Newman (2007). *UCI Machine Learning Repository*. http://www.ics.uci.edu/∼mlearn/MLRepository.html. University of California, Irvine, School of Information and Computer Sciences.

[Berghammer, 2008] Berghammer, Rudolf (2008). *Ordnungen, Verbände und Relationen mit Anwendungen*. Vieweg+Teubner, GWV Fachverlage GmbH.

[Cormode et al., 2003] Cormode, Graham, F. Korn, S. Muthukrishnan und D. Srivastava (2003). *Finding Hierarchical Heavy Hitters in Data Streams*. In: *Proceedings of the 29th VLDB Conference*, S. 464–475.

[Cormode et al., 2004] Cormode, Graham, F. Korn, S. Muthukrishnan und D. Srivastava (2004). *Diamond in the Rough: Finding Hierarchical Heavy Hitters in Multi-dimensional Data*. In: *Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data*, S. 155–166.

[Cormode et al., 2008] Cormode, Graham, F. Korn, S. Muthukrishnan und D. Srivastava (2008). *Finding Hierarchical Heavy Hitters in Streaming Data*. ACM Trans. Knowl. Discov. Data, 1(4):1–48.

[Forrest et al., 2008] Forrest, Stephanie, S. A. Hofmeyr und A. Somayaji (2008). *The Evolution of System-Call Monitoring*. In: *2008 Annual Computer Security Applications Conference*, S. 418–430.

[Forrest et al., 1996] Forrest, Stephanie, S. A. Hofmeyr, A. Somayaji und T. A. Longstaff (1996). *A Sense of Self for Unix Processes*. In: *Proceedings of the 1996 IEEE Symposium on Security and Privacy*, S. 120–128. IEEE Computer Society Press.

[Ganesan et al., 2003] Ganesan, Prasanna, H. Garcia-Molina und J. Widom (2003). *Exploiting Hierarchical Domain Structure to Compute Similarity*. ACM Transactions on Information Systems, 21(1):64–93.

[Garg et al., 2006] Garg, Ashish, R. Rahalkar, S. Upadhyaya und K. Kwiat (2006). *Profiling Users in GUI Based Systems for Masquerade Detection*. In: *Proceedings of the 2006 IEEE Workshop on Information Assurance*, S. 48–54.

[Guyon und Elisseeff, 2003] Guyon, Isabelle und A. Elisseeff (2003). *An Introduction to Variable and Feature Selection*. Journal of Machine Learning Research, 3:1157–1182.

[Hall, 1999] Hall, Mark A. (1999). *Correlation-based Feature Subset Selection for Machine Learning*. Doktorarbeit, Department of Computer Science, University of Waikato.

[Hall, 2000] Hall, Mark A. (2000). *Correlation-based Feature Selection for Discrete and Numeric Class Machine Learning*. In: *Proc. 17th International Conf. on Machine Learning*, S. 359–366.

[Han und Kamber, 2006] Han, Jiawei und M. Kamber (2006). *Data Mining. Concepts and Techniques*. Morgan Kaufmann Publishers.

[Hastie et al., 2001] Hastie, Trevor, R. Tibshirani und J. Friedman (2001). *The Elements of Statistical Learning. Data Mining, Inference, and Prediction*. Springer Series in Statistics. Springer, 1 Aufl.

[Hershberger et al., 2005] Hershberger, John, N. Shrivastava, S. Suri und C. D. Tóth (2005). *Space Complexity of Hierarchical Heavy Hitters in Multi-Dimensional Data Streams*. In: *Proceedings of the twenty-fourth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems (PODS '05)*, S. 338–347.

[Hofmeyr et al., 1998] Hofmeyr, Steven A., S. Forrest und A. Somayaji (1998). *Intrusion detection using sequences of system calls*. Journal of Computer Security, 6(3):151–180.

[Imsand und Hamilton, 2007] Imsand, Eric S. und J. A. Hamilton (2007). *GUI Usage Analysis for Masquerade Detection*. In: *Information Assurance and Security Workshop, 2007. IAW '07. IEEE SMC*, S. 270–276.

[Intrusion Detection Subgroup, 1997] Intrusion Detection Subgroup, Network Group, National Security Telecommunications Advisory Committee (1997). *Report on the NS/EP Implications of Intrusion Detection Technology Research and Development*.

[Joachims, 2001] Joachims, Thorsten (2001). *The Maximum-Margin Approach to Learning Text Classifiers*. Doktorarbeit, Fakultät für Informatik, Universität Dortmund.

[Jungermann, 2009] Jungermann, Felix (2009). *Information Extraction with RapidMiner*. In: Hoeppner, Wolfgang, Hrsg.: *Proceedings of the GSCL Symposium Sprachtechnologie und eHumanities*.

[Kang et al., 2005] Kang, Dae-Ki, D. Fuller und V. Honavar (2005). *Learning Classifiers for Misuse Detection Using a Bag of System Calls Representation*. In: *Proceedings of IEEE International Conference on Intelligence and Security Informatics*.

[Kohavi und John, 1997] Kohavi, Ron und G. H. John (1997). *Wrappers for feature subset selection*. Artificial Intelligence, Special Issue on Relevance:1–43.

[Kosoresow und Hofmeyer, 1997] Kosoresow, Andrew P. und S. A. Hofmeyer (1997). *Intrusion detection via system call traces*. IEEE Software, 14(5):35–42.

[Kruegel et al., 2003] Kruegel, Christopher, D. Mutz, F. Valeur und G. Vigna (2003). *On the Detection of Anomalous System Call Arguments*. In: *Computer Security – ESORICS 2003, 8th European Symposium on Research in Computer Security*, Lecture Notes in Computer Science, S. 326–344. Springer.

[Liao und Vemuri, 2002] Liao, Yihua und V. R. Vemuri (2002). *Use of K-Nearest Neighbor classifier for intrusion detection*. Computers & Security, 21(5):439–448.

[Manku und Motwani, 2002] Manku, Gurmeet Singh und R. Motwani (2002). *Approximate Frequency Counts over Data Streams*. In: *Proceedings of the 28th VLDB Conference, Hong Kong, China*, S. 346–357.

[Matuszewski, 2009] Matuszewski, Andrea (2009). *Analyse von Betriebssystem-Logdateien*. Diplomarbeit, Fakultät für Informatik, Technische Universität Dortmund.

[Mierswa et al., 2006] Mierswa, Ingo, M. Scholz, R. Klinkenberg, M. Wurst und T. Euler (2006). *YALE: Rapid Prototyping for Complex Data Mining Tasks*. In: *Proceedings of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD2006)*, S. 935–940. ACM Press.

[Misra und Gries, 1982] Misra, Jayadev und D. Gries (1982). *Finding Repeated Elements*. Science of Computer Programming, 2:143–152.

[Mitchell, 1997] Mitchell, Tom M. (1997). *Machine Learning*. The McGraw-Hill Companies.

[Muthukrishnan, 2005] Muthukrishnan, S. (2005). *Data streams: Algorithms and applications*. Foundations and Trends in Theoretical Computer Science, 1(2):117–236.

[Mutz et al., 2006] Mutz, Darren, F. Valeur, G. Vigna und C. Kruegel (2006). *Anomalous System Call Detection*. ACM Transactions on Information and System Security, 9(1):61–93.

[Peisert et al., 2007] Peisert, Sean, M. Bishop, S. Karin und K. Marzullo (2007). *Analysis of Computer Intrusions Using Sequences of Function Calls*. IEEE Transactions on Dependable and Secure Computing, 4(2):137–150.

[Salton und Buckley, 1988] Salton, Gerard und C. Buckley (1988). *Term-weighting approaches in automatic text retrieval*. Information Processing & Management, 24(5):513–523.

[Schonlau et al., 2001] Schonlau, Matthias, W. DuMouchel, W.-H. Ju, A. F. Karr, M. Theus und Y. Vardi (2001). *Computer Intrusion: Detecting Masquerades*. Statistical Science, 16(1):58–74.

[Silberschatz et al., 2000] Silberschatz, Avi, P. B. Galvin und G. Gagne (2000). *Applied Operating System Concepts*. John Wiley & Sons, 1. Aufl.

[Silberschatz et al., 2010] Silberschatz, Avi, P. B. Galvin und G. Gagne (2010). *Operating System Concepts*. John Wiley & Sons, 8. Aufl.

[Stallings, 2009] Stallings, William (2009). *Operating Systems. Internals and Design Principles*. Pearson Prentice Hall, 6. Aufl.

[Tanenbaum, 2003] Tanenbaum, Andrew Stuart (2003). *Moderne Betriebssysteme*. Pearson Studium, 2. Aufl.

[Vaidya et al., 2005] Vaidya, Jaideep, C. Clifton und M. Zhu (2005). *Privacy Preserving Data Mining*. Springer Science+Business Media.

[Varghese und Jacob, 2007a] Varghese, Surekha Mariam und K. P. Jacob (2007a). *Anomaly Detection Using System Call Sequence Sets*. Journal of Software, 2(6):14–21.

[Varghese und Jacob, 2007b] Varghese, Surekha Mariam und K. P. Jacob (2007b). *Process profiling using frequencies of system calls*. In: *Second International Conference on Availability, Reliability and Security, ARES 2007*, S. 473–479.

[Warrender et al., 1999] Warrender, Christina, S. Forrest und B. Pearlmutter (1999). *Detecting Intrusions Using System Calls: Alternative Data Models*. In: *Proceedings of the 1999 IEEE Symposium on Security and Privacy.*, S. 133–145.

[Zanero, 2006] Zanero, Stefano (2006). *Unsupervised Learning Algorithms for In-*

trusion Detection. Doktorarbeit, DEI Politecnico di Milano.

Erklärung

Hiermit erkläre ich, Peter Fricke, die vorliegende Diplomarbeit mit dem Titel *Datenaggregation von Betriebssystemdaten durch Hierarchical Heavy Hitters* selbstständig verfasst und keine anderen als die hier angegebenen Hilfsmittel verwendet sowie Zitate kenntlich gemacht zu haben.

Dortmund, 30. März 2010