Lehrstuhl für Softwaretechnik

Universität Augsburg

Universität
Augsburg
University

Entwicklung eines Lernverfahrens zur intelligenten Laufzeitinstanziierung von abstrakten Testfällen

Bachelorarbeit

im Studiengang Informatik

Manuel Richter

25.02.2018

Inhaltsverzeichnis

1 Einleitung und Motivation

Mit der Entwicklung von Software geht auch der Wunsch nach Verifikation einher, um zuverlässig prüfen zu können, dass das entwickelte System mit seiner Veröffentlichung keine Fehler aufweist. Im Zuge dieser Qualitätssicherung hat sich das Anlegen von Testfällen etabliert. Dabei werden Bedingungen, Mengen von Eingaben und eine Menge von erwarteten Ergebnissen spezifiziert und an das zu prüfende System übergeben, um die erwarteten Ergebnisse mit den tatsächlich erzeugten zu vergleichen. Mit einer geschickten Wahl solcher Testfälle ist es möglich, unser Vertrauen auf einen korrekten Programmablauf zu stärken. Allerdings kann mit diesem Verfahren nicht garantiert werden, dass die entwickelte Software vollständig fehlerfrei ist. Die händische Erstellung von Testfällen beansprucht je nach Größe des zu prüfenden Systems viel Zeit und ist stark anfällig für menschlichen Irrtum. Ein zusätzlicher Faktor, der dieses Problem erschwert, ist die, über die Jahrzehnte hinweg, wachsende Komplexität von Software. Durch diesen Trend gestaltet sich das ausführliche Testen zunehmend als langwieriger und teurer Prozess und zeigt den Bedarf nach zeitsparenden Alternativen auf, die der Qualität der händisch erzeugten Testfälle in nichts nachstehen.

Das Anfertigen eines Modells des realen Systems, welches vom tatsächlichen System abstrahiert und nur die grobe Funktionsweise des eigentlichen Systems abbildet, stellt einen Lösungsansatz für dieses Problem dar. Auf einem solchen Modell lassen sich dann automatisiert sogenannte abstrakte Testfälle generieren, die anschließend auf das reale System überführt werden können. Häufig fällt es entsprechenden Algorithmen aber schwer, diese Testfälle ähnlich geschickt wie ein Mensch zu wählen, da sie meist kein Verständnis für komplexe Zusammenhänge und zeitliche Abhängigkeiten zwischen Modellabläufen entwickeln können. Techniken wie Model-Checking, die in der Lage sind, solche Zusammenhänge abzudecken werden häufig von hohen Laufzeiten begleitet. Je nach Größe des Systems führt dies dazu, dass der Algorithmus nur mit gewissen Beschränkungen ausgeführt werden kann,

mit deren Hilfe eine bessere Laufzeit erzielt wird. Testfälle, die von diesen Algorithmen erzeugt werden, können bei der Ausführung auf dem realen System dann aber unter Umständen nur einen Teil der tatsächlich auffindbaren Fehler erkennen und bieten damit nur bedingt eine solide Basis zur automatischen Generierung von Testfällen.

In dieser Arbeit soll die Frage behandelt werden, ob und wie ein möglichst effizientes Verfahren entwickelt werden kann, das eigenständig lernt abstrakte Testfälle unter Beachtung der Abhängigkeiten und Zusammenhänge des Modells zu generieren. Um dies zu realisieren, werden Konzepte des Reinforcement-Learnings verwendet, um der Testfallgenerierung eine gewisse Intelligenz zu verleihen. Dazu sollen zunächst in Kapitel 2 ähnliche Arbeiten beleuchtet und analysiert werden, um sie mit dem in dieser Arbeit gewählten Ansatz zu vergleichen und abzugrenzen. In Kapitel 3 werden S# und das ZNN-System vorgestellt, welche als Beispiel für das Modell eines größeren Systems dienen sollen. Kapitel 4 setzt sich mit der Definition und Erläuterung verschiedener Ansätze des Reinforcement-Learnings auseinander, die in Kapitel 5 in das betrachtete ZNN-System eingebettet werden. Kapitel 6 dient der Evaluation der Ergebnisse der Lernalgorithmen, die sowohl untereinander als auch mit den Ergebnissen zufällig generierter Testfälle verglichen werden. Kapitel 7 fasst die Ergebnisse dieser Arbeit zusammen und gibt einen Ausblick auf weitere Ansätze zur Lösung dieser Problemstellung.

2 Stand der Technik

Die automatisierte Generierung von sinnvollen Testfällen im Allgemeinen beschäftigt die Forschung bereits seit einiger Zeit. Unter der Vielzahl von Ausarbeitungen, die sich mit diesem Themengebiet auseinandersetzen, existieren jedoch kaum Ansätze, bei denen gleichzeitig die Testfälle auf Basis eines Modells generiert werden und ein Lernalgorithmus verwendet wird um intelligent Testfälle zu entwickeln. Meist wurden Ansätze, die auf Lernalgorithmen basieren, auf dem bestehenden realen System und nicht im Modell ausgeführt und Ansätze, die auf dem Modell eines realen Systems agierten, nutzten keine Lernalgorithmen. Im Rahmen dieser Arbeit sollen diese beiden Ansätze verschmolzen werden, weshalb sie in diesem Kapitel von ähnlichen Arbeiten abgegrenzt und Gemeinsamkeiten hervorgehoben werden sollen.

Rizal Bahaweres et al. stellen in ihrer Arbeit *Analysis of statement branch and loop coverage in software testing with genetic algorithm* [BZKH17] einen Testfallgenerator vor, welcher mit Hilfe eines genetischen Algorithmus Testfälle für ein System entwirft, das die Klassifikation von Dreiecken in die Gruppen *rechtwinklig*, *unregelmäßig*, *gleichschenklig* und *gleichseitig* vornimmt. Um ihren Algorithmus zu evaluieren, messen sie, wie viel Zeit vergeht, bis der gesamte Code des Programms von den generierten Testfällen abgedeckt wurde. Hierbei schneidet der genetische Algorithmus deutlich besser ab, als eine zufällige Generierung von Testfällen. Da es sich bei dem System zur Klassifikation von Dreiecken um ein System geringer Komplexität handelt, kann der genetische Algorithmus problemlos auf dem tatsächlichen System agieren, um die Testfälle zu generieren. Hierbei ist es im Gegensatz zum Ansatz dieser Arbeit also nicht notwendig, ein Modell zu verwenden, das von dem tatsächlichen System abstrahiert.

Im Artikel *Model-Based Test Case Generation for Smart Cards* [PPS+03] hingegen verwenden J. Philipps et al. einen modellbasierten Ansatz, um Testfälle für Chipkarten zu generieren. In dem entsprechenden Modell wird der Austausch von Informationen zwischen Chipkarte und

Lesegerät simuliert. Mit Hilfe dieses Modells können so auf intuitive Art und Weise abstrakte Testfälle generiert werden, welche im Hinblick auf strukturelle Spezifikationen, wie etwa die Abdeckung aller Pfade innerhalb des Modells, getestet und mit der zufälligen Generierung von Testfällen verglichen werden. Um dies zu realisieren, verwenden J. Philipps et al. mehrere Heuristiken mit Beschränkungen im Suchraum und, im Gegensatz zu dieser Arbeit, kein explizites maschinelles Lernverfahren.

Häufiger werden maschinelle Lernverfahren, wie etwa Q-Learning, zum Test der Robustheit verwendet. In der Arbeit *A Reinforcement Learning Approach to Automated GUI Robustness Testing* von Sebastian Bauersfeld et al. [BV12] wird so beispielsweise mit Hilfe von Reinforcement Learning die GUI-Robustheit von Programmen getestet, indem durch Q-Learning Testfälle für ein *Graphical User Interface* (GUI) generiert werden. Diese Testfälle setzen sich aus einer Abfolge von Aktionen zusammen - etwa einem Mausklick gefolgt von einer Texteingabe. Dieser Ansatz ähnelt dem in dieser Arbeit betrachteten sehr, da auch die Testfälle dieser Arbeit sich aus Folgen von Aktionen zusammensetzen werden, was in Kapitel 4.5 noch näher erläutert wird. Allerdings erfolgt die Generierung der Testfälle von Bauersfeld et al. nicht über ein abstraktes Modell des tatsächlichen Systems, sondern wird auf genau dem Graphical User Interface ausgeführt, das getestet werden soll.

Die oben genannten Artikel verfolgen ein ähnliches Ziel wie diese Arbeit, stimmen in ihrem Vorgehen jedoch nicht vollständig mit dem hier gewählten überein. Die Systeme, die in den oben genannten Artikeln getestet werden sollen, können entweder keinen Nutzen aus einem Modell ziehen oder es wird bei der Testfallgenerierung auf die Verwendung von Lernalgorithmen verzichtet. Diese Arbeit soll sich mit einem Modell beschäftigen, für das händisch nur mit Mühe optimale Testfälle gefunden werden können und so eine automatisierte Generierung von Mehrwert ist.

3 Safety-Sharp und das ZNN-Systemmodell

Wenn Testfälle auf Basis eines komplexes Systems generiert wurden und dieses System im Laufe seiner weiteren Entwicklung erweitert oder abgeändert wird, Kostet es viel Zeit die Testfälle wieder auf den aktuellen Stand des komplexen Systems zu bringen. Deshalb werden oft weniger komplexe Modelle dieser Systeme erstellt und daraus *abstrakte Testfälle* abgeleitet. Die Änderungen, die am realen System vorgenommen wurden, können auch leicht am Modell vorgenommen werden. So lassen sich auch am geänderten Modell leicht Testfälle ableiten und es wird weniger Zeit für die Testfallgenerierung aufgewandt. Eine größere Client-Server Landschaft stellt ein Beispiel für ein solches abstrahiertes Modell dar. Im Rahmen dieser Arbeit soll dieses Modell des ZNN-Systems unter Zuhilfenahme des Modellierungsframeworks Safety-Sharp verwendet werden, um Testfälle zur Laufzeit zu instanzieren.

3.1 Safety-Sharp

Safety-Sharp (auch S#) ist ein Sicherheits- und Modellierungsframework, das vom *Institute for Software & Systems Engineering* der Universität Augsburg entwickelt wurde. Es setzt auf der Programmiersprache C# auf und bietet damit unter anderem die Möglichkeit reale Systeme als C#-Programme zu modellieren und dessen Abläufe zu simulieren. Im Rahmen dieser Arbeit soll dies als Grundlage für die modellbasierten Tests dienen, indem ein in S# umgesetztes Modell betrachtet wird. Am Lehrstuhl für Softwaretechnik der Universität Augsburg wurden bereits einige solcher Fallstudien durchgeführt und die zugehörigen Modelle in S# erstellt, darunter auch das Client-Server Modell des ZNN-Systems, auf das sich die Überlegungen dieser Arbeit zum großen Teil stützen werden.

3.2 Das ZNN-System

Das, in der Dissertation *Rainbow: Cost-Effective Software Architecture-Based Self-Adaptation* von Shang-Weng Cheng [Che08], beschriebene

ZNN-System wurde vom Lehrstuhl für Softwaretechnik im Zuge einer Fallstudie der Liste von S#-Modellen hinzugefügt. In diesem Modell wird eine Website von einer Anzahl von Servern bereitgestellt die von mehreren Clients gleichzeitig besucht wird. Die von den Servern bereitgestellten Inhalte lassen sich in die Kategorien *Text*, *kleiner Multimedia-Inhalt* und *großer Multimedia-Inhalt* einordnen. Dabei steht besonders das Load-Balancing-Problem im Vordergrund - also das Problem, viele Anfragen von Clients gleichmäßig unter den Servern aufzuteilen und große Lastspitzen durch eine Verringerung der zurückgegebenen Inhaltsgröße auszugleichen. Zu diesem Zweck entkoppelt, wie in Abbildung 1 zu sehen, ein Proxy-Server die Clients von den Servern, der das Load-Balancing organisieren soll.

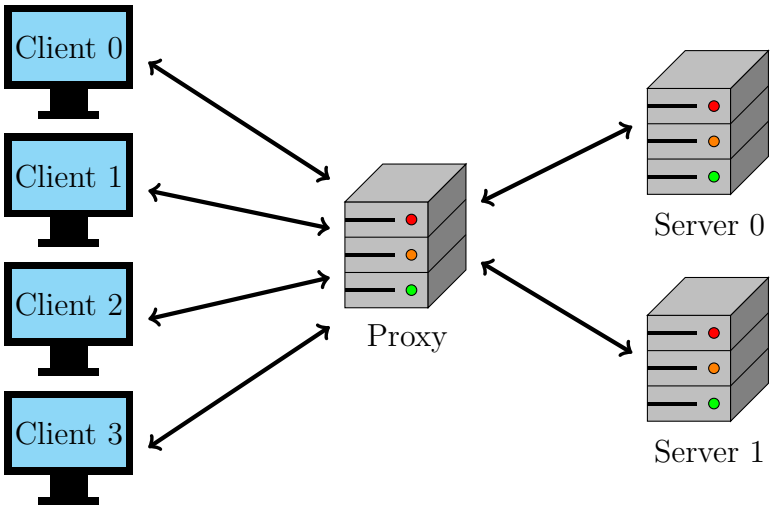


Abbildung 1: Architektur des ZNN-Modells

3.2.1 Client-Proxy-Server Interaktion

Der Proxy-Server (im Folgenden Proxy) ist sowohl für die Vermittlung der Anfragen von Clients an die entsprechenden Server als auch für die Kontrolle der Server zuständig. Der Proxy leitet die Anfragen der Clients derart an die Server weiter, dass sämtliche Server in etwa gleich stark ausgelastet sind und kontrolliert dabei, ob gewisse Richtwerte für die Antwortzeit, Auslastung der Server und der anfallenden Kosten stets eingehalten werden. In Abbildung 2 wird mittels

eines Zustandsdiagramms veranschaulicht, wie der Proxy die Anfrage eines Clients an einen Server weiterleitet und die Antwort des Servers an den Client zurücksendet. Einer der Clients sendet eine Anfrage

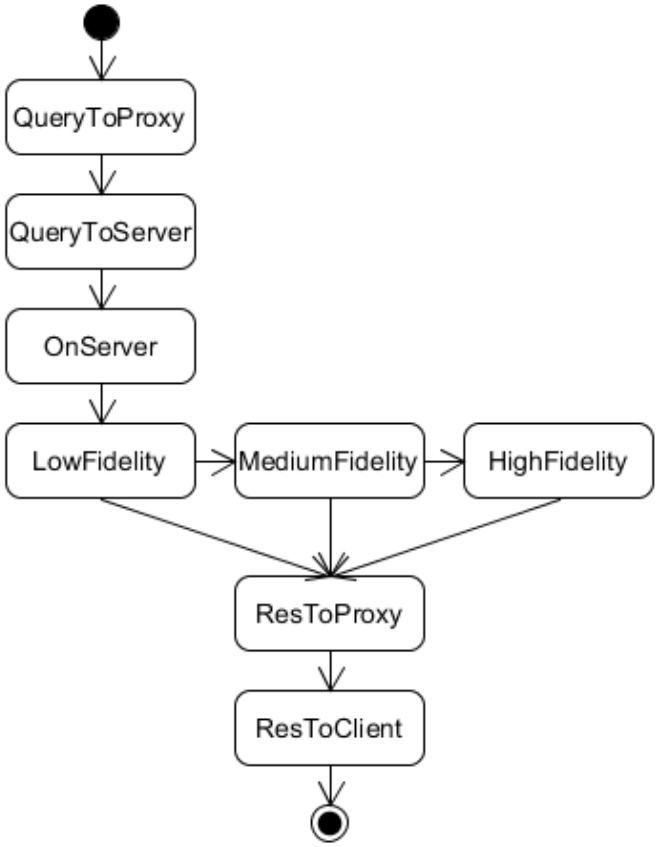


Abbildung 2: Die Verarbeitung eines Anfrage-Antwort-Zyklus

an den Proxy, ohne explizit zu wissen, dass er nicht direkt einen Server anspricht (*QueryToProxy*). Der Proxy verarbeitet die Anfrage des Clients, indem er sie an einen der nicht ausgelasteten Server weiterleitet (*QueryToServer*). Nachdem der Server die Anfrage erhalten hat und mit der Arbeit beginnen kann (*OnServer*), wird der angeforderte Inhalt je nach Serverlast bis zu einem bestimmten Qualitätslevel vorbereitet. Sind die Server stark ausgelastet, wird lediglich Textinhalt (*LowFidelity*) zurückgegeben, bei mittlerer Auslastung dagegen wird Textinhalt zusammen mit einem kleinen Multimediainhalt (*MediumFidelity*) und bei geringer Serverlast der vollständige Inhalt der angeforderten Webseite (*HighFidelity*) zurückgegeben. Nachdem der Server den Inhalt vorbereitet hat, sendet er die Daten zunächst an den

Proxy (*ResToProxy*), welcher anschließend die generierte Antwort an den Client weiterleitet (*ResToClient*).

Stellt jeder der in Abbildung 2 gezeigten Knoten einen Schritt im Ablauf des Modells dar, so benötigt ein Anfrage-Antwort-Zyklus im Regelfall 10 Schritte, da mit Start- und Endzustand bis zu 10 Zustände durchlaufen werden müssen.

3.2.2 Rekonfiguration und Ausfall

Wie in Kapitel 3.2.1 erwähnt, ist der Proxy neben der Weiterleitung von Client-Anfragen und Server-Antworten auch dafür zuständig, dass vordefinierte Richtwerte von den Servern nicht überschritten werden. Überschreitet ein Server einen der Richtwerte, führt dies zu einer *Rekonfiguration*. So muss beispielsweise im Fall, dass die Antwortzeit der Server aufgrund zu hoher Last über einen Grenzwert steigt, vom Proxy ein weiterer Server aktiviert werden, um die Last wieder auszugleichen. Findet diese Rekonfiguration nicht statt, so kann es zu Fehlern oder zur Entstehung unnötiger Kosten kommen. Das ZNN-System umfasst eine Vielzahl dieser Rekonfigurationen. Bei einer zu geringen Serverauslastung ist es dem Proxy auch möglich, einen der aktiven Server zu deaktivieren, um unnötigen Leerlauf und die damit verbundenen Kosten zu reduzieren. Ferner kann, wie in Abbildung 2 dargestellt, die Qualität des angeforderten Inhalts vom Proxy reguliert werden. Bei hoher Last können die Server damit etwa dadurch entlastet werden, dass dem Client nur kleine Multimediainhalte oder sogar nur Textinhalte zur Verfügung gestellt werden. Diese Funktion soll zu Spitzenlastzeiten ein Zusammenbrechen der Serverinfrastruktur und einen etwaigen Verlust von Kundschaft vermeiden.

Um zu prüfen, wie sich die Serverinfrastruktur bei den oben genannten Problemen verhält, werden im ZNN-System *Ausfälle* provoziert. So wird von der Kontrollebene, im Kontext dieser Arbeit von einem Lernalgorithmus, zum Beispiel der Ausfall eines Servers forciert, um die Aktivierung eines neuen Servers zu erzwingen oder die Bearbeitung einer Anfrage zu unterbrechen. Jedem Ausfall wird die *Art* des

Ausfalls, der *Zeitpunkt* zu dem der Ausfall eintritt und der *Server*, auf dem der Ausfall stattfindet, zugeordnet. Die Art des Ausfalls ist dabei stets eine der folgenden:

ServerSelectionFails

Der Proxy kann die Anfrage eines Clients nicht an einen Server weiterleiten. Er wird den Versuch, die Anfrage weiterzuleiten, im nächsten Schritt wiederholen.

SetServerFidelityFails

Der Proxy kann im Falle von zu geringer oder zu hoher Auslastung die Art des zurückgegebenen Inhalts nicht modifizieren. Ein Client könnte etwa lediglich Textinhalt erhalten, obwohl genug Serverkapazitäten vorhanden wären, um den vollständigen Inhalt zurückzugeben. Alternativ könnte der Client keine Antwort erhalten, weil der Server große Multimediainhalte vorbereitet, obwohl er überlastet ist.

ServerDeath

Ein Server fällt komplett aus. Es können keine Anfragen mehr an diesen Server gestellt oder von diesem bearbeitet werden und der Server kann nicht deaktiviert werden, um die Kosten zu minimieren.

CantExecuteQueries

Auf diesem Server können keine Anfragen mehr bearbeitet werden.

ServerCantBeDeactivated

Im Falle von zu geringer Auslastung der aktiven Server kann der betroffene Server nicht mehr deaktiviert werden, um Kosten zu sparen.

ServerCantBeActivated

Im Falle von zu hoher Auslastung der aktiven Server kann der betroffene inaktive Server nicht mehr aktiviert werden, um die Auslastung auszugleichen.

Alle oben genannten Arten von Ausfällen betreffen jeweils einen der Server im System und sollen in einer Menge $Faults$ zusammengefasst werden. Neben dieser sechs Arten von Ausfallarten gibt es jedoch noch eine weitere, die nicht auf einen einzelnen Server beschränkt ist, sondern die Verbindung zwischen Clients und Proxy betrachtet:

ConnectionToProxyFails

Die initiale Verbindung mit dem Proxy schlägt fehl. Der Client kann damit keine Anfragen mehr an den Proxy versenden, womit die Anfragen auch nicht von einem Server bearbeitet werden können.

Basierend auf diesen Arten von Ausfällen lässt sich nun formal mit der Definition der Menge $Faults$ ein Ausfall als Tripel seiner beschreibenden Eigenschaften definieren:

Definition 3.1: Ausfall

Gegeben seien für $T \in \mathbb{N}$, Server $S_1, ..., S_k$ und Clients $C_1, ..., C_l$ die Mengen $Faults$, $Steps = \{0, ..., T\}$, $Server = \{S_1, ..., S_k\}$ und $Clients = \{C_1, ..., C_l\}$.
Dann ist die Menge aller Ausfälle definiert durch die Menge
$\mathcal{A} = \{(f, t, s) | f \in Faults, t \in Steps, s \in Server\}$
$\cup \{(ConnectionToProxyFails, t, c) | t \in Steps, c \in Clients\}$.
Ein Ausfall ist ein Tripel $(f, s, sc) \in \mathcal{A}$.

Diese Ausfälle sollen im Rahmen dieser Arbeit von einem Lernalgorithmus ausgelöst werden und die daraus resultierenden Probleme sollten, wenn möglich, vom Proxy durch eine Rekonfiguration aufgehoben werden. Um durch die Provokation derartiger Ausfälle möglichst viele potentielle Fehler aufzudecken, sollten die gewählten Ausfälle möglichst vielfältig sein und geschickt gewählt werden. Es ist also eine intelligente Laufzeitinstanziierung abstrakter Testfälle erwünscht.

4 Das Lernverfahren

Damit das System intelligent Ausfälle auslösen kann wurde im Rahmen dieser Arbeit mit zwei Ansätzen des *Reinforcement-Learnings* experimentiert. In diesem Abschnitt sollen diese beiden Ansätze näher erläutert werden, weshalb zunächst einige zum Verständnis notwendige Grundlagen definiert werden.

4.1 Definitionen und Grundlagen

Interessiert man sich für maschinelle Lernalgorithmen, so muss man sich zwangsläufig fragen, wann ein Algorithmus ausreichend Erfahrung gesammelt hat, um das definierte Problem zuverlässig lösen zu können. Ein Lernalgorithmus, für den nicht bewiesen wurde das er zur optimalen Lösung konvergiert, ist in realen Systemen offensichtlich ungeeigneter als ein Lernalgorithmus, dessen Konvergenz bewiesen werden kann. Watkins und Dayan bewiesen, dass ein Lernalgorithmus in jedem Falle konvergiert, wenn es sich bei der zu bearbeitenden Aufgabe um einen *Markov-Entscheidungsprozess* handelt [WD92]. Um dieses Ergebnis zu verwenden und damit die Konvergenz des, in dieser Arbeit vorgestellten, Algorithmus zu garantieren, soll zunächst der Begriff der *Markov-Eigenschaft* definiert werden.

Definition 4.1: Markov-Eigenschaft

Für alle $i \in \{0, ..., t\}$ und $j \in \{1, ..., t\}$ mit $t \in \mathbb{N}$ sei

- S_i ein Zustand
- A_i die Aktion, die im Zustand S_i ausgeführt wird
- R_j der Reward für die Aktion A_{j-1} im Zustand S_{j-1}

Die Markov-Eigenschaft liegt vor, wenn für alle r und s' gilt:

$$Pr\{R_{t+1} = r, S_{t+1} = s' | S_0, A_0, R_1, ..., S_{t-1}, A_{t-1}, R_t, S_t, A_t\}$$

$$= Pr\{R_{t+1} = r, S_{t+1} = s' | S_t, A_t\}. \text{ [SB98]}$$

Gilt die Markov-Eigenschaft für ein Problem, so trifft das Lernsystem, das nur seinen aktuellen Zustand kennt, die selbe Entscheidung, wie ein Lernsystem das alle vorhergehenden Zustände und Aktionen kennt. Es muss dem Lernsystem egal sein können wie es in den aktuellen Zustand gelangt ist und muss trotzdem eine Entscheidung treffen können, als wüsste es über die Vergangenen Zustände und Aktionen Bescheid. Die Markov-Eigenschaft liefert damit die Grundlage für die Definition eines Markov-Entscheidungsprozesses, für den wir uns primär interessieren:

Definition 4.2: Markov-Entscheidungsprozess

Eine Reinforcement Learning Aufgabe heißt Markov-Entscheidungsprozess (MDP), wenn sie die Markov-Eigenschaft erfüllt. Falls der Zustands- und Aktionsraum endlich ist, handelt es sich um einen *endlichen* Markov-Entscheidungsprozess. [SB98]

Dabei stellt eine „Reinforcement-Learning Aufgabe“ hier die betrachtete Umgebung dar, in welcher der Lernalgorithmus agieren soll. Sie setzt sich aus einer Menge von Zuständen - dem Zustandsraum, einer Menge von möglichen Aktionen - dem Aktionsraum, einer Menge von Übergangswahrscheinlichkeiten $\mathcal{P}^a_{s,s'} := Pr\{S_{t+1} = s' | S_t = s \wedge A_t = a\}$ und einer Funktion $reward : State \times Action \mapsto \mathbb{R}$ für die Belohnung (engl. reward) einer Aktion in einem Zustand zusammen und repräsentiert damit das zu lösende Problem.

Während der Lernalgorithmus Erfahrungen sammelt, handelt er nicht zwangsläufig vollständig zufällig. Häufig richtet er sich nach einer spezifizierten *Strategie*, mit deren Hilfe er die nächste Aktion in einem fixen Zustand deterministisch wählen kann. Diese Strategie wird während des Lernprozesses mit Hilfe der erhaltenen Belohnungen weiter ausgebaut und verfeinert, um schließlich gegen die *optimale Strategie* zu konvergieren, die das Problem optimal löst. Neben der Frage, wie man die Strategie modifiziert, um sie möglichst schnell gegen die optimale Strategie konvergieren zu lassen, kann man sich auch fragen, wann

welcher konkrete Teil einer Strategie verbessert wird. Zu dieser Frage existieren zwei unterschiedliche Ansätze:

Definition 4.3: On-/Off-Policy

Beim *On-Policy*-Ansatz wird die aktuell verfolgte Strategie gleichzeitig optimiert und zur Entscheidungsfindung herangezogen. Der *Off-Policy*-Ansatz dagegen optimiert eine Strategie, diese muss aber nicht dieselbe sein auf der auch die Entscheidungen basieren. [SB98]

4.2 Q-Learning

Die Strategie eines lernenden Systems wird durch den Lernalgorithmus spezifiziert. Dieser soll im Falle des ZNN-Systems intelligent Ausfälle auslösen und so eine optimale Reihenfolge der Ausfälle finden, sodass ein möglichst großer Teil des Gesamtsystems geprüft werden kann. Die Umgebung des ZNN-Systems lässt sich als Markov-Entscheidungsprozess formulieren, wie in Kapitel 4.4 bis 4.6 erläutert wird. Damit konvergiert ein Lernalgorithmus, der auf dieser Umgebung agiert, in jedem Falle [WD92] was eine nützliche Grundlage zur Anwendung des Q-Learning-Algorithmus von Christopher Watkins [Wat89] darstellt.

Der lernende Teil eines Lernsystems wird Agent genannt. Der Agent ist in der Lage, innerhalb eines Zustands eine Auswahl von Aktionen auszuführen und die Reaktion der Umgebung durch eine folgende Belohnung wahrzunehmen. Die Belohnung, die im Folgenden auch Reward genannt wird, ist stets abhängig von dem aktuellen Zustand, in dem sich das System befindet und der Aktion, die der Agent in diesem Zustand ausgewählt hat und kann den Agenten für seine Aktion belohnen (positiver Zahlenwert), bestrafen (negativer Zahlenwert) oder neutral behandeln. Der Reward wird nicht von dem Agenten selbst generiert, sondern von der Umgebung des Agenten, welche die Nützlichkeit der gewählten Aktion durch eine reelle Zahl ausdrückt. Q-Learning verbessert seine Strategie, indem eine Q-Wert Tabelle aufgebaut wird,

in welcher für jeden Zustand und jede Aktion gespeichert wird, wie vielversprechend das Zustands-Aktions-Paar ist. Um dem Algorithmus eine gewisse Weitsicht zu ermöglichen, wird daher der Reward mittels der folgenden Aktualisierungsformel in die Q-Wert-Tabelle übertragen:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha[R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t)]$$

$Q(S_t, A_t)$ beschreibt dabei den Q-Wert des aktuellen Zustands mit der Aktion, die aktuell gewählt wurde. R_{t+1} ist stellvertretend für den Reward, der auf dieses Zustands-Aktions-Paar folgt und $\max_a Q(S_{t+1}, a)$ ist der größte Q-Wert, der im Folgezustand erreicht werden kann. α und γ spielen dabei die Rolle von Parametern, die das Lernen des Agenten je nach Wert beschleunigen oder verlangsamen können. Die Lernrate α kontrolliert den Einfluss, den ein neuer Reward auf den Q-Wert hat und der Diskontwert γ gewichtet schnellere Lösungsstrategien, also solche mit möglichst wenigen Aktionen, stärker, sodass für diese Lösungswege größere Q-Werte entstehen.

4.2.1 Der Algorithmus

Da die Aktualisierungsformel bereits auf bestehende Einträge der Q-Wert-Tabelle zurückgreift, muss diese zuvor initialisiert worden sein. Dafür gibt es verschiedene Ansätze, welche die Q-Werte mit festen Zahlenwerten initialisieren. So wird beispielsweise bei optimistischen Q-Werten ein Zahlenwert verwendet, der größer ist als der maximal zu erreichende Reward. Damit ist der Agent gezwungen, jedes Zustands-Aktions-Paar mindestens ein Mal auszuprobieren, da die Q-Wert-Tabelle ihm für einen unbesuchten Zustand stets den größten Wert verspricht.

Nach der Initialisierung wird dann ausgehend vom aktuellen Zustand die beste Aktion, also die Aktion mit dem höchsten Q-Wert, gewählt und ausgeführt. Nach der Aktualisierung dieses Q-Wertes wird der Folgezustand wahrgenommen und basierend auf diesem wiederum die nächste Aktion gewählt. Das Vorgehen soll im Folgenden anhand des Algorithmus nach dem Buch von Sutton und Barto [SB98] veranschaulicht werden:

Algorithm 1: Q-Learning

$\forall s \in \mathcal{S}, a \in \mathcal{A}(s)$ *Initialize* $Q(s,a)$ *arbitrarily*

$Q(terminal\text{-}state, \cdot) = 0$

while *episodes left* **do**

  Initialize S

  repeat

    Choose A from S (e.g., ϵ-*greedy*)

    Take action A, observe R, S'

    $Q(S,A) \leftarrow Q(S,A) + \alpha[R + \gamma max_a Q(S',a) - Q(S,A)]$

    $S \leftarrow S'$;

  until S *is terminal*;

end

Dabei stellt $Q(S,A)$ den Eintrag der Q-Wert-Tabelle dar, der mit dem Zustands-Aktions-Tupel (S,A) übereinstimmt. Weiterhin ist der Reward mit R und der Folgezustand von S nach Anwendung der Aktion A mit S' bezeichnet. Die Wahl der Aktion A in S kann auf verschiedene Weisen erfolgen. Die bisher beschriebene Wahl der Aktion, die im aktuellen Zustand den höchsten Q-Wert besitzt, kann dazu führen, dass die erste Folge von Aktionen, die der Agent ausprobiert hat, auch für alle weiteren Durchläufe verwendet wird. Das kann zu unerwünschtem Verhalten führen und ist gänzlich ungeeignet für Umgebungen, die sich dynamisch verändern können. Aus diesem Grund soll ein alternatives Verfahren zur Wahl einer Aktion beleuchtet werden, das auch in dynamischen Umgebungen die optimale Strategie entwickeln kann.

4.2.2 ϵ-Greedy

Das in Kapitel 4.2.1 vorgestellte Verfahren zur Wahl einer Aktion wird *greedy* genannt. Mit diesem Verfahren wählt der Agent stets die vielversprechendste Aktion aus der Q-Wert-Tabelle aus, von der er sich den größten Reward erhofft. Mit diesem Vorgehen ist allerdings die Gefahr groß, dass der Agent in einem lokalen Optimum feststeckt und deshalb nicht in Betracht zieht, eine Aktion zu wählen, die ihn zum globalen Optimum führen könnte. Abbildung 3 zeigt, wie der Agent dann stets eine Aktion wählt, die weit vom Optimum entfernt ist. Es ist also

wünschenswert, dass der Agent stellenweise Aktionen wählt, die laut den Daten in der Q-Wert-Tabelle nicht den höchsten Reward versprechen, bei denen es sich aber trotzdem um ein lokales Optimum handeln könnte. Er sollte seine Aktionen jedoch nicht vollständig zufällig wählen, da er andernfalls kein echtes Lernverhalten aufweist und vielversprechende Aktionen nur durch Zufall ausführt. Gewünscht ist also ein Mittelweg, der das Beste aus beiden Vorgehen verwenden kann und möglichst schnell gegen die optimale Strategie konvergiert. Dazu kann der Agent mit einer gewissen Wahrscheinlichkeit dazu gezwungen werden, eine zufällige Aktion auszuführen, um so potentiell bessere Aktionen zu finden. Beim ϵ-Greedy-Ansatz stellt das $\epsilon \in [0, 1]$ die Wahrscheinlichkeit dar, eine zufällige Aktion zu wählen - also zu erkunden (*explore*). Damit ist $1 - \epsilon$ die Wahrscheinlichkeit, den bisher besten erlernten Lösungsweg zu verfolgen, also auszunutzen (*exploit*).

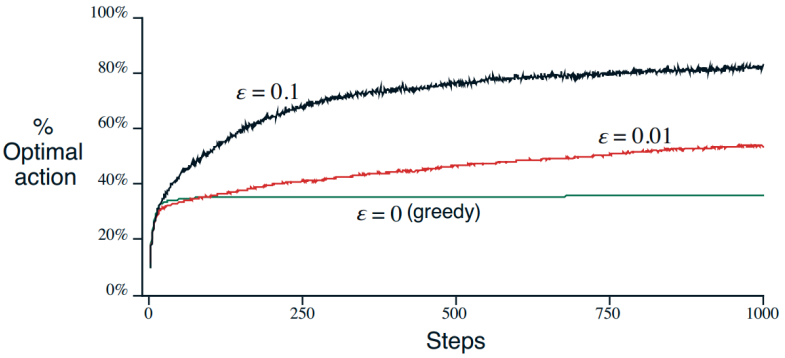


Abbildung 3: ϵ-Greedy auf dem n-armed bandit-Problem [SB98]

Bei dem Q-Learning-Algorithmus handelt es sich um einen Off-Policy Ansatz, da die Aktualisierung des aktuellen Q-Wertes sich auf die bestmögliche Aktion bezieht, die im Folgezustand möglich ist. Wird die Aktion aber mit Hilfe des ϵ-greedy Verfahrens gewählt, ist nicht immer gegeben, dass diese Aktion auch tatsächlich im nächsten Zustand verwendet wird. Der Agent verbessert seine Strategie also nicht gemäß des Pfades, den er tatsächlich verfolgt, sondern gemäß des (vermeintlich) Bestmöglichen.

4.3 SARSA

Um im Zuge dieser Arbeit auch einen On-Policy Lernalgorithmus evaluieren zu können wurde, aufgrund der Nähe zum Q-Learning, der SARSA-Ansatz ausgewählt. Der SARSA-Algorithmus von G. A. Rummery und M. Niranjan [RN94] hat starke Ähnlichkeiten zum Off-Policy Q-Learning-Algorithmus, verfolgt allerdings eine andere Strategie zur Berechnung der Q-Werte. SARSA geht nicht davon aus, dass stets die beste Aktion gewählt wird, sondern verbessert genau die Strategie, die im Augenblick verfolgt wird. Dazu wählt der Agent zuerst die nächste Aktion aus und berechnet erst danach den neuen Q-Wert der zuvor gewählten Aktion:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha[R_{t+1} + \gamma Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t)]$$

Um eine einheitliche Schreibweise zu gewährleisten, wurde die Notation aus dem Buch *Reinforcement Learning: An Introduction* von Sutton und Barto [SB98] übernommen.

4.3.1 Der Algorithmus

Algorithmus 2 soll den SARSA-Ansatz auf programmiertechnischer Ebene veranschaulichen. Nach dem Beginn einer Episode, also eines Lernabschnitts, wird eine initiale Aktion gewählt und ausgeführt.

Algorithm 2: SARSA

$\forall s \in \mathcal{S}, a \in \mathcal{A}(s)$ *Initialize* $Q(s,a)$ *arbitrarily*

$Q(terminal\text{-}state, \cdot) = 0$

while *episodes left* **do**

  Initialize S

  Choose A from S (e.g., ϵ-*greedy*)

  repeat

    Take action A, observe R, S'

    Choose A' from S' (e.g., ϵ-*greedy*)

    $Q(S,A) \leftarrow Q(S,A) + \alpha[R + \gamma Q(S',A') - Q(S,A)]$

    $S \leftarrow S'$;

    $A \leftarrow A'$;

  until S *is terminal*;

end

Im Folgezustand wird dann erneut eine Aktion gewählt und basierend auf der gewählten Aktion der Q-Wert aktualisiert. Die Folgeaktion wird nach Beginn der nächsten Episode ausgeführt und das Vorgehen wiederholt sich, bis ein finaler Zustand erreicht wird. Die Variablen α, γ und $Q(S,A)$ werden wie im Q-Learning-Algorithmus in Kapitel 4.2.1 definiert und verwendet.

4.4 Wahl des Zustandsraums

Um die oben genannten Algorithmen umsetzen zu können, müssen einige Parameter festgelegt werden, die bisher vernachlässigt wurden. Eine große Herausforderung stellt dabei die korrekte Wahl eines Zustands- und Aktionsraumes für den Lernalgorithmus dar. Enthalten diese zu viele Informationen über die Umgebung, so führt dies zu einem starken Wachstum der Q-Wert-Tabelle und somit zu einer drastischen Verlängerung der Laufzeit. Enthalten sie dagegen zu wenige Informationen, führt dies zu schlechtem Lernverhalten und zu unerwünschten Effekten.

Der Zustandsraum sollte demnach so klein wie möglich sein, aber alle für das Lernen notwendigen Informationen enthalten. Damit der

Agent in der Lage ist, eine optimale Reihenfolge von Ausfällen zu erlernen, liegt es nahe, dass der Zustandsraum die Informationen darüber enthält, zu welchem Zeitpunkt welcher Ausfall ausgelöst wurde. Damit würde der Lernalgorithmus aber auf Informationen aus vergangenen Schritten zurückgreifen und der Zustandsraum folglich die Markov-Eigenschaft verletzen. Es ergibt sich also, dass diese Information für den Zustandsraum ungeeignet ist, wenn man weiterhin die Konvergenz garantieren möchte. Stattdessen lässt sich jedem der Ausfälle aus $Faults \cup \{ConnectionToProxyFails\}$ ein Wahrheitswert zuordnen, der aussagt, ob der jeweilige Ausfall in der Vergangenheit bereits eingetreten ist oder nicht. Der Zustandsraum spannt sich in einen Raum der Form $(Fault_1 \times Fault_2 \times Fault_3 \times Fault_4 \times Fault_5 \times Fault_6 \times Fault_7)$ auf, bei dem jedes $Fault_i$ durch einen Wahrheitswert (0 oder 1) repräsentiert wird. Auch ohne das Wissen über den genauen Ausfallzeitpunkt ergibt sich so pro Episode eine Sequenz von Ausfällen, die der Agent verfolgen und optimieren kann. Es gilt jedoch zu beachten, dass durch eine solche Kodierung des Zustandsraums neben dem Zeitpunkt auch die Information, auf welchem Server der Ausfall aufgetreten ist, verloren geht. Andernfalls hätte dies zur Folge, dass sich der Zustandsraum polynomiell vergrößert. Diese Unschärfe wird vom Lernalgorithmus ausgeglichen, da er mit dem Erlernen einer Reihenfolge von Ausfällen auch gleichzeitig die Abhängigkeiten zwischen den einzelnen Servern erlernt.

Eine Abfolge von Ausfällen entsteht, indem der Agent etwa aus dem Zustand (0000000) heraus die bisher beste Aktion dieses Zustands wählt und dadurch im nächsten Schritt den Folgezustand (0001000) erreicht. Dort wählt er erneut die beste Aktion des Zustands und so ergibt sich eine Kette aus generierten Ausfällen pro Episode, die der Agent mit Hilfe des Lernalgorithmus optimieren kann. Diese Sequenz von Ausfällen einer Episode stellt hier einen *abstrakten Testfall* dar.

Aus den zuvor definierten sieben Arten von Ausfällen ergibt sich gemeinsam mit der oben erläuterten Verwendung von Wahrheitswerten ein Zustandsraum der Größe $2^7 = 128$. Diese Größenordnung ist für

einen Rechner leicht zu verarbeiten und sollte eine solide Grundlage für den Zustandsraum bilden.

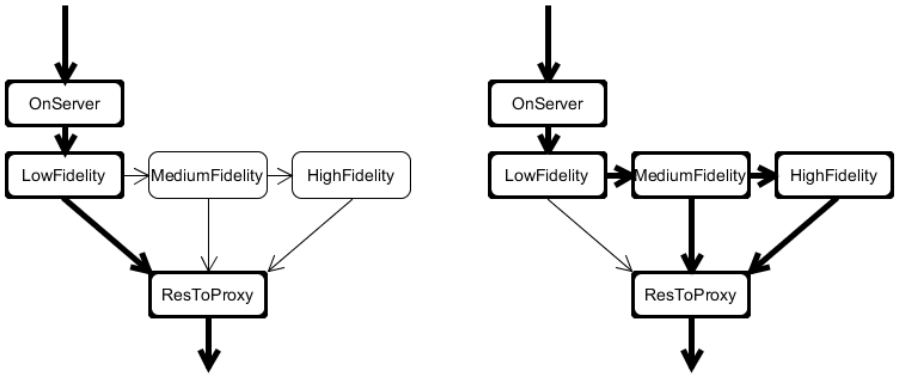
4.5 Wahl des Aktionsraums

Da der Agent die Ausfälle der Server intelligent und selbstständig auslösen soll, muss es ihm möglich sein, über die Art und den Ort des Ausfalls frei zu bestimmen. Es liegt also nah, den Aktionsraum als $Faults \times Server \cup \{ConnectionToProxyFails\} \times Clients$ zu wählen, um diese Entscheidungsfreiheit zu gewährleisten. Die Mengen sind dabei so definiert, wie in Kapitel 3.2.2 beschrieben. Löst der Agent nun etwa die Aktion $(ConnectionToProxyFails, C_2)$ im Zustand (0000000) aus, geht er in den Zustand (1000000) über, da ein Ausfall der Art $ConnectionToProxyFails$ auf dem Client C_2 aufgetreten ist. Die Darstellungen dieser Aktionen lassen sich vereinfachen, indem man Faults, Clients und Server jeweils beginnend bei 0 durchnummeriert und so etwa die oben genannte Aktion in $(0, 2)$ übersetzen kann. Eine solche Kodierung führt zu einer Aktionsraumgröße von $|Faults| \cdot |Server| + |Clients|$, was für eine realistische Anzahl von Servern und Clients ebenfalls eine überschaubare Basis bildet. Um den Aktionsraum darüber hinaus klein zu halten, entfällt der Parameter *Step* aus Kapitel 3.2.2 vollständig, da Ausfälle stets in dem Schritt aktiviert werden, in dem sie vom Agenten ausgewählt werden.

4.6 Rewardkriterien

Analog zur Frage nach der konkreten Form des Zustands- und Aktionsraums müssen Kriterien für den Reward für lohnende Aktionen des Agenten definiert, sowie der Begriff der „lohnenden“Aktionen spezifiziert werden. Eines der üblichen Verfahren des modellbasierten Testens stellt das Finden eines Testfalls dar, der möglichst viele Bereiche des Modells abdeckt. Wird etwa, wie in Abbildung 4a dargestellt, im Kontext des ZNN-Systems in jedem Schritt nur Textinhalt (LowFidelity) von den Servern zur Verfügung gestellt, weil der Testfall bereits früh höherwertigen Inhalt durch einen oder mehrere Ausfälle unterbindet, so wird nur ein kleiner Teil der möglichen Verzweigungen des Modells

durchlaufen. Ein Testfall, der lange die Möglichkeit zulässt, dass unterschiedliche Inhalte von den Servern zurückgegeben werden, kann dagegen wesentlich mehr Verzweigungen abdecken, wie Abbildung 4b veranschaulicht. Je mehr Verzweigungen des Modells durch einen Test-



(a) geringe Branch Coverage (b) hohe Branch Coverage

Abbildung 4: Branch Coverage im Modell

fall abgedeckt werden, desto höher ist die Wahrscheinlichkeit, mit diesem Testfall im echten System einen Fehler zu erkennen. Die Idee, einen Testfall anhand der abgedeckten Verzweigungen (*Branch Coverage*) im System zu evaluieren wird auch ohne Lernalgorithmus häufig zur Generierung von Testfällen verwendet [PPS+03]. Auch im Rahmen dieser Arbeit soll die Branch Coverage zur Evaluierung der Nützlichkeit eines Testfalls dienen und wird daher zur Bestimmung des Rewards genutzt, die der Agent für die Wahl eines Testfalls erhält. Im Folgenden ist die Branch Coverage durch eine Kennzahl beschrieben, die den Anteil zwischen besuchten und nicht besuchten Codepfaden des Modells darstellt. Allgemein lässt sich die Branch Coverage pro Episode also durch den folgenden Ausdruck beschreiben:

$$coverage_{episode} = \frac{\text{branches taken}}{\text{branches to take}}$$

Interessiert man sich für die Branch Coverage zu einem bestimmten Zeitschritt t, so lässt sich der folgende Ausdruck zur Beschreibung verwenden:

$$coverage_t = \frac{b_t - b_{t-1}}{c}$$

Dabei ist b_t die Anzahl der bis zum Schritt $t \in \{0, ..., T\}$ abgedeckten Verzweigungen und c die Anzahl aller möglichen Verzweigungen im Programmcode. Dieser Term dient pro Schritt als Indikator für die „Güte“ einer gewählter Kombination aus Ausfällen und Serverzuweisungen. Das optimale Ergebnis wird folglich mit $coverage_{episode} = \sum_{t=0}^{T} coverage_t = 1$ bei T Schritten in einer Episode erreicht, was der vollen Abdeckung aller Pfade im System entspricht. Da sich im Rahmen dieser Arbeit der Reward direkt aus der Branch Coverage pro Schritt ergibt, werden diese beiden Begriffe im Weiteren als synonym betrachtet.

4.7 Horizont

Bei dem Horizont handelt es sich um die Zeit, die dem Agenten zur Verfügung steht, um eine Lösung zu erlernen. Nach Ablauf eines Horizonts beginnt eine neue Episode. Ist der Horizont zu kurz gewählt, kann es für den Agenten damit unmöglich sein, das korrekte Verhalten zu erlernen. Ein zu großer Horizont dagegen wirkt sich maßgeblich negativ auf die Laufzeit des lernenden Systems aus.

Im Falle des ZNN-Systems ist der Horizont variabel, allerdings zeigte sich bereits in Abbildung 2, dass ein Durchlauf von der Anfrage eines Clients an den Server bis zu seiner Antwort im Regelfall 10 Zeitschritte dauert. Damit sollte der Horizont des hier betrachteten Algorithmus nicht unter 10 Schritte fallen, um dem Agenten das Lernen zu ermöglichen.

5 Weitere Implementierungsdetails

Neben den zuvor genannten Prinzipien, denen die automatische Testfallgenerierung im ZNN-System folgt, sollen im Folgenden weitere Details erläutert werden, die sich im Zusammenhang mit der konkreten Implementierung ergeben haben.

5.1 Randbedingungen

Wie bereits in Kapitel 4.7 beschrieben, ist die Anzahl der Schritte einer Episode frei wählbar, solange sie mehr als 10 Schritte umfasst. Zudem ist auch die Anzahl der Episoden beliebig wählbar, sollte aber groß genug gewählt werden, dass der Agent genügend Zeit hat, um eine optimale Lösung zu erlernen. Neben Episodenzahl und -länge kann in jedem simulierten ZNN-Modell eine Anzahl von Servern und Clients festgelegt werden. Dadurch variiert die Anzahl der möglichen Ausfälle, die auftreten können - so können etwa bei drei verfügbaren Servern auch nur maximal drei Totalausfälle auftreten. Gleichermaßen kann ein einzelner Client nicht drei verfügbare Server voll auslasten, sodass zwei davon stets deaktiviert bleiben. Um das Problem möglichst deterministisch zu halten, soll sich keiner der verwendeten Parameter, wie etwa Lernrate oder Episodenzahl, zur Laufzeit des Systems verändern.

5.2 Branch-Coverage

Jeder Branch, der in einer Episode durchlaufen wurde, wird nach den Kriterien aus Kapitel 4.6 verrechnet und erzeugt bei erneuter Abdeckung in derselben Episode keinen weiteren zusätzlichen Reward. Weiterhin existieren auch Branches, die nur äußerst selten abgedeckt werden, aber zur Gesamtzahl an vorhandenen Branches hinzugezählt werden. Damit ist es unwahrscheinlich, eine Abdeckung von 100% zu erreichen, was sich in der nun folgenden Evaluation des Systems widerspiegeln wird.

6 Evaluation

Dieses Kapitel dient dem Vergleich der implementierten Algorithmen mit verschiedenen Alternativen, wie etwa der zufälligen Auswahl von Ausfällen. Dazu soll zunächst die Größe des ZNN-Systems grob abgeschätzt werden und anschließend die verschiedenen implementierten Algorithmen untereinander verglichen werden.

6.1 Codemetriken

Das vorliegende Modell des ZNN-Systems umfasst etwa 1522 Codezeilen (LOC) und besitzt eine zyklomatische Komplexität von 53. Die Anzahl der logischen Codezeilen (SLOC-L) beträgt dabei 398 und spiegelt wider, wie viele Anweisungen tatsächlich in den 1522 Codezeilen enthalten sind. Gemessen wurden diese Werte mit Hilfe dem Code-Metrikwerkzeug *LocMetrics* [loc07]. Bei der zyklomatischen Komplexität handelt es sich um die Zahl der unabhängigen Pfade durch den Code des Programms. Damit benötigt das ZNN-System etwa 53 Schritte, um jeden möglichen Codepfad abdecken zu können. Da allerdings ein kleiner Teil dieser unabhängigen Pfade nicht erreichbar ist, weil etwa die Anzahl der Clients zur Laufzeit konstant bleibt, das Modell aber auch wechselnde Clientzahlen zur Laufzeit unterstützt, fällt die Abschätzung auf weniger als 50 Schritte ab.

6.2 Q-Learning versus SARSA versus Zufall

6.2.1 Konfiguration

Die implementierten Lernverfahren wurden nacheinander auf dem Modell ausgeführt und über die 2D-Plotting Pythonbibliothek MatPlotLib [Hun07] in graphischer Darstellung visualisiert. Um die zufällige Auswahl von Parametern für Testfälle umzusetzen, wurde der Explorationsparameter ϵ des Q-Learning-Algorithmuses auf 1 gesetzt. Damit wählt dieser stets zufällig die Belegungen für das Aktionstupel und es ergeben sich pro Schritt die für den folgenden Vergleich notwendigen $coverage_t$ Werte.

Einige der Parameterwerte konnten vorab bereits restringiert werden. So sollte die Anzahl der Clients bzw. Server nicht kleiner als 2 sein, um zu vermeiden, dass die Episode durch den Ausfall des einzigen Servers im System vorzeitig endet. Ferner reicht ein einzelner verbundener Client nicht aus, um den Server stark genug auszulasten, sodass ein zweiter Server aktiviert werden müsste. In beiden Fällen könnte nur ein Bruchteil des gesamten Modells abgedeckt werden.

Weiterhin muss der Agent mindestens $|Zustandsraum| * \#Aktionen = 2^7 * 7 = 896$ Schritte oder 90 Episoden mit jeweils 10 Schritten durchlaufen haben, um jede Aktion in jedem Zustand ausgeführt zu haben. Um allerdings auch jede Aktion auf jedem Server simuliert zu haben, muss der Lernalgorithmus $|Zustandsraum| * \#Aktionen * \#Server = 2^7 * 7 * 3 = 2688$ Schritte beziehungsweise 270 Episoden mit jeweils 10 Schritten durchlaufen haben. Diese Maßzahlen sollen im Rahmen dieser Arbeit als untere Schranke für die Wahl der Episodenanzahl dienen.

Da ein einzelner Durchlauf von der Anfrage des Clients bis zur Antwort des Servers etwa 10 Schritte benötigt, sollte eine Episode ein vielfaches dieser Zeit betragen, damit mehr als ein Server-Client-Austausch stattfinden kann. Die Lernrate bestimmt, wie stark sich neue Informationen auf alte auswirken. Mit einer Lernrate von $\alpha = 1$ wird jede zuvor erlernte Information verworfen und durch die neu erlernte Information ersetzt, mit einer Lernrate von $\alpha = 0$ lernt der Agent überhaupt nicht und lässt die Informationen so, wie sie initialisiert sind. Für ein ausgewogenes Lernverhalten wurde aus diesem Grund eine Lernrate von $\alpha = 0.5$ gewählt.

Parameter	Beschreibung	Belegung
Clients	Anzahl der anfragenden Clients	3
Server	Anzahl vorhandener Server	3
E	Anzahl zu durchlaufender Episoden	300
T	Schrittzahl pro Episode	50
α	Lernrate	0,5

Diese Parameter wurden sowohl für Q-Learning, als auch für SARSA verwendet, um die beiden Lernverfahren untereinander besser Vergleichen zu können.

6.2.2 Zufällige Parameterwahl

Bei den Ergebnissen aus Abbildung 5-7 handelt es sich um die über die Episodenanzahl gemittelte kumulative Abdeckung, welche pro Schritt erreicht wurde. Bereits bei dem Vergleich zwischen Q-Learning und der Wahl zufälliger Testcases in Abbildung 5 zeigt sich, dass die zufällig gewählten Testfälle im Durchschnitt nur etwa 60 % der Verzweigungen abdecken können. Grund dafür ist die in Kapitel 3.2 erwähnte zeitliche Abhängigkeit der Ausfälle. Im Mittel löst hier der Zufall zu früh Ausfälle aus, die eine weitere Abdeckung neuer Codepfade verhindern.

In Schritt 0 bis 2 werden die meisten Verzweigungen abgedeckt, da hier, neben der Initialisierung, auch die erste Anfrage des Clients an den Proxy und die Weiterleitung dieser Anfrage an entsprechende Server durchgeführt wird. Ein weiterer starker Anstieg der Coverage kann in Schritt 9 bis Schritt 10 beobachtet werden. Zu diesem Zeitpunkt leitet der Proxy die Antwort des Servers an den Client weiter. Nachdem einer dieser Client-Server-Zyklen mit einer Dauer von 10 Schritten durchlaufen wurde, treten nur noch marginale Verbesserungen der Abdeckung auf.

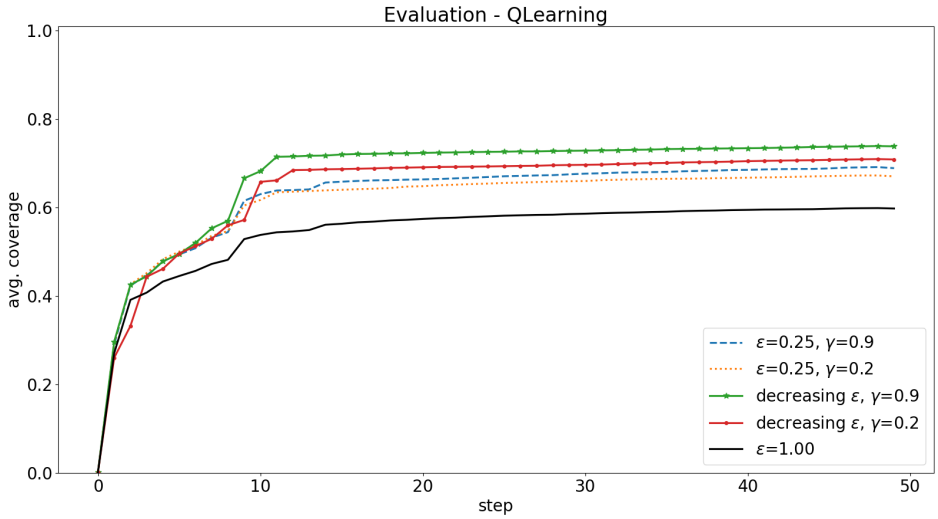


Abbildung 5: Q-Learning vs Zufall

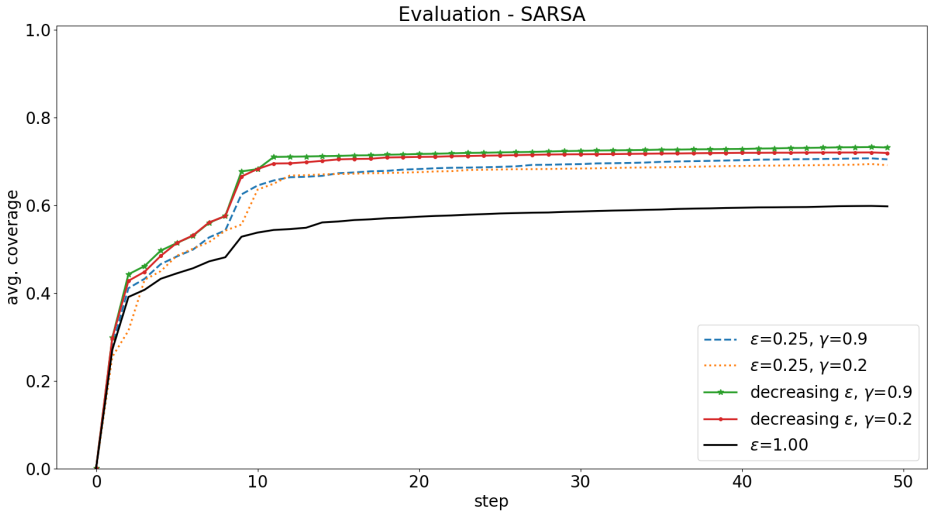


Abbildung 6: SARSA vs Zufall

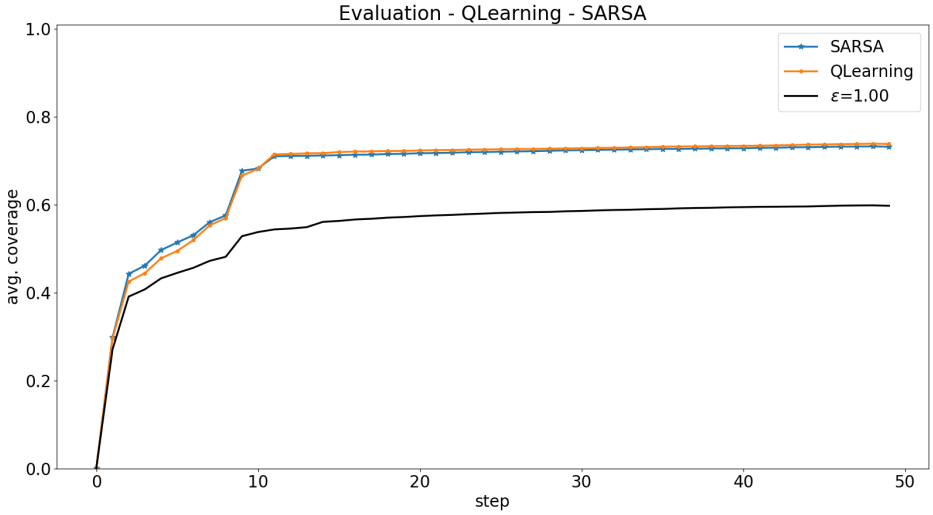


Abbildung 7: Q-Learning vs SARSA vs Zufall

6.2.3 Q-Learning

Im direkten Vergleich schneidet der Q-Learning-Ansatz besser ab. Es wurden vier Testläufe mit je unterschiedlichen Epsilon- und Gammabelegungen gestartet. So konvergiert der Algorithmus mit einer Belegung von $\epsilon = 0,25$ und $\gamma = 0,2$ bei einer durchschnittlichen Abdeckung von 67,33 %, wohingegen mit einer Belegung von $\gamma = 0,9$ sogar 69,19 % Abdeckung erreicht werden können . Nach C. Watkins Artikel *Q-Learning* [WD92] führt ein klein gewähltes Gamma dazu, dass der Agent nur eine kurze Spanne an Aktionen beachtet, was in diesem Fall negative Auswirkungen auf die Zweigabdeckung hat, da der Agent bis zu 10 Schritte „in die Zukunft"blicken muss, um den richtigen Zeitpunkt für eine Aktion finden zu können.

Zu einer weiteren Verbesserung führt die dynamische Anpassung der Explorationsstrategie. Wird die Explorationsrate ϵ pro Episode Schrittweise von $1,00$ auf $0,01$ mit einer Schrittweite von $0,02$ verringert, so wird in Schritt 50 lediglich mit einer Wahrscheinlichkeit von 1% exploriert. In Verbindung mit einem Gamma von $0,2$ kann damit eine Abdeckung von 70% erreicht werden. Die höchste bislang beobachtete durchschnittliche Abdeckung mit einer Coverage von 75% kann mit einem Gammawert von $\gamma = 0,9$ erreicht werden. Auch hier wirkt sich ein kleines Gamma negativ auf das Ergebnis aus.

6.2.4 SARSA

In Abbildung 6 wurde der Lernalgorithmus SARSA mit verschiedenen Belegungen getestet. Hier liegen die Abdeckungswerte sehr nah beieinander, allerdings ergibt sich ein Muster, welches mit den Ergebnissen der Q-Learning-Auswertung vergleichbar ist. Mit fixem Epsilon deckt ein Gamma von $0,9$ auch hier mehr Verzweigungen ab, als ein Gamma von $0,2$. Es lässt sich aber beobachten, dass SARSA mit diesen beiden Werten etwas besser abschneidet, als der Q-Learning-Algorithmus.
Auch beim SARSA-Algorithmus führt ein dynamisches Anpassen der Explorationsstrategie, zu einem verbesserten Ergebnis. Mit einem Gam-

ma von $0,2$ kann so eine $72,11\%$-ige Abdeckung erreicht werden. Die höchste bisher beobachtete durchschnittliche Abdeckung zeigt sich bei einem Gamma von $0,9$ mit $73,54\%$.

6.2.5 Algorithmenvergleich

In Abbildung 7 ist der Q-Learning-Algorithmus im Vergleich mit dem SARSA-Algorithmus aufgetragen. Da in beiden Fällen die Belegung mit dynamisch abnehmendem Epsilon und einem Gamma von 0,9 die besten Ergebnisse zeigten, wurden diese Werte jeweils für den Vergleich verwendet. Hier liegt der SARSA-Ansatz mit $73,54\%$ im 50. Schritt nur knapp hinter dem Q-Learning-Ansatz, welcher eine Code-Abdeckung von $74,00\%$ aufweist . Zu Beginn konvergiert SARSA schneller gegen die optimale Abdeckung, konvergiert danach aber insgesamt langsamer als Q-Learning. Grund hierfür sind die unterschiedliche Strategien, welche die beiden Algorithmen verfolgen. Während der Off-Policy Ansatz von Q-Learning stets auf den langfristig maximalen Reward hinarbeitet, versucht SARSA als On-Policy Algorithmus seine Strategie zu optimieren und löst deshalb, gerade in den ersten Schritten, nur kleine Ausfälle aus. Schließlich liegt SARSA um $13,60\%$ und Q-Learning um $14,06\%$ über der vom Zufall maximal erreichten Zweigabdeckung.

7 Zusammenfassung

In dieser Arbeit wurde ein Lernverfahren entwickelt, das zur Laufzeit automatisiert abstrakte Testfälle generiert. Diese basieren auf den Reinforcement Learning-Algorithmen Q-Learning und SARSA und werden mit Hilfe des Frameworks S# simuliert und ausgeführt. Das Modell der Server-Client-Architektur des ZNN-Systems bildet dabei die Grundlage für die Umgebung der Lernalgorithmen. Als einheitliches Rewardkriterium wurde die Branch Coverage pro Schritt gewählt. Bei der Evaluation der Ergebnisse zeigte sich, dass eine, mit jedem Schritt fallende, Explorationsrate zusammen mit einem Diskont-Wert von $0,9$ die besten Resultate, sowohl für Q-Learning als auch für SARSA, erzeugen konnte. Dabei schnitt der Q-Learning-Ansatz etwas besser ab als der SARSA-Algorithmus. Unabhängig davon führten die von den Lernalgorithmen generierten Testfälle stets zu einer höheren Zweigabdeckung als die durch den Zufall hervorgebrachten. Grund dafür ist, dass der Lernalgorithmus in der Lage ist, die Reihenfolge, in der die Ausfälle generiert und ausgelöst wurden, zu beachten.

7.1 Fazit

Wie diese Arbeit gezeigt hat, kann, unter der Annahme, dass ein betrachtetes Modell zeitlichen Abhängigkeiten während des Testens unterliegt, durch die Anwendung von *Reinforcement Learning*-Ansätzen wie Q-Learning oder SARSA noch eine signifikante Verbesserung erwirkt werden. Damit wurde erneut das Potential und die vielseitigen Anwendungsmöglichkeiten des maschinellen Lernens belegt.

7.2 Ausblick

Die Folgenden Ansätze könnten noch zu einer Verbesserung des Ergebnisses führen, wurden im Rahmen dieser Arbeit jedoch nicht behandelt:

- **Explorationsrate:** Um die Erkundung weiter zu optimieren kann die fixe Explorationsrate Beispielsweise durch den *Counter-Based Exploration*-Ansatz aus der Arbeit von S. Thrun [Thr92] ersetzt werden.

- **Dynamische Lernrate:** Ähnlich zur Explorationsrate lässt sich auch die Lernrate pro Schritt dynamisch verringern und kann möglicherweise zu einem verbesserten Ergebnis führen.
- **Evaluation:** Eine Auswertung der Dauer, bis eine maximal mögliche Zweigabdeckung erreicht wurde, könnte weitere Schlüsse auf das Lernverhalten zulassen.

8 Literatur

[BV12] Bauersfeld, Sebastian ; Vos, Tanja: A reinforcement learning approach to automated gui robustness testing. In: *Fast Abstracts of the 4th Symposium on Search-Based Software Engineering (SSBSE 2012)*, 2012, S. 7–12

[BZKH17] Bahaweres, R. B. ; Zawawi, K. ; Khairani, D. ; Hakiem, N.: Analysis of statement branch and loop coverage in software testing with genetic algorithm. In: *2017 4th International Conference on Electrical Engineering, Computer Science and Informatics (EECSI)*, 2017, S. 1–6

[Che08] Cheng, Shang-Wen: *Rainbow: Cost-Effective Software Architecture-Based Self-Adaption*, Carnegie Mellon University, dissertation, 2008

[Hun07] Hunter, J. D.: Matplotlib: A 2D graphics environment. In: *Computing In Science & Engineering* 9 (2007), Nr. 3, S. 90–95. `http://dx.doi.org/10.1109/MCSE.2007.55`. – DOI 10.1109/MCSE.2007.55

[loc07] `http://www.locmetrics.com/index.html`

[MSB11] Myers, Glenford J. ; Sandler, Corey ; Badgett, Tom: *The art of software testing*. 2. John Wiley & Sons, 2011

[PPS+03] Philipps, J. ; Pretschner, A. ; Slotosch, O. ; Aiglstorfer, E. ; Kriebel, S. ; Scholl, K.: Model-Based Test Case Generation for Smart Cards Support by the BMBF (project EMPRESS) is gratefully acknowledged. In: *Electronic Notes in Theoretical Computer Science* 80 (2003), S. 170 – 184. – ISSN 1571–0661. – Eighth International Workshop on Formal Methods for Industrial Critical Systems (FMICS'03)

[RN94] Rummery, Gavin A. ; Niranjan, Mahesan: *On-line Q-learning using connectionist systems*. Bd. 37. University of Cambridge, Department of Engineering, 1994

[SB98] Sutton, Richard S. ; Barto, Andrew G.: *Reinforcement Learning: An Introduction*. MIT Press, 1998

[Thr92] Thrun, Sebastian B.: Efficient Exploration In Reinforcement Learning. 1992. – Forschungsbericht

[Wat89] Watkins, Christopher John Cornish H.: *Learning from Delayed Rewards*. Cambridge, UK, King's College, Diss., May 1989

[WD92] Watkins, Christopher J. ; Dayan, Peter: Q-learning. In: *Machine learning* 8 (1992), Nr. 3-4, S. 279–292