Universität des Saarlandes
Naturwissenschaftlich-Technische Fakultät I
Fachrichtung Informatik
Lehrstuhl für Softwaretechnik

Bachelorarbeit

Parallelisierung von Reinforcement Learning zur optimalen Steuerung von Softwareprojekten

Oliver Nalbach

30. September 2011

Betreuer
Dr. Frank Padberg

Eidesstattliche Erklärung

Ich erkläre hiermit an Eides Statt, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Statement in Lieu of an Oath

I hereby confirm that I have written this thesis on my own and that I have not used any other media or materials than the ones referred to in this thesis.

Einverständniserklärung

Ich bin damit einverstanden, dass meine (bestandene) Arbeit in beiden Versionen in die Bibliothek der Informatik aufgenommen und damit veröffentlicht wird.

Declaration of Consent

I agree to make both versions of my thesis (with a passing grade) accessible to the public by having them added to the library of the Computer Science Department.

Saarbrücken, 30.09.2011

_____
(Oliver Nalbach)

Inhaltsverzeichnis

1 Einleitung

Problemstellung Die Leitung eines Softwareprojekts, genauer die Verteilung von Entwicklerteams auf einzelne Komponenten der Software, ist komplex und lässt sich wegen vieler Unwägbarkeiten kaum vorab planen. Umso mehr ist man an einer optimalen Ablaufsteuerung interessiert, um die Entwicklungskosten möglichst gering zu halten. Durch Formalisierung des Projektablaufs [Pad99; Pad02] als *Markovscher Entscheidungsprozess* (engl. Markov Decision Process, MDP), erhält man eine Grundlage, dieses Problem rechnerisch zu lösen.

In einem *Software-MDP* (SW-MDP) sind die Teams verschiedenen Komponenten der Software zugeteilt, welche sie bearbeiten. Wird eine Komponente fertiggestellt oder tritt ein Problem an einer Komponente auf, das einen Eingriff des Projektleiters erfordert, wird anhand einer *Scheduling Strategie* ausgehend von dem aktuellen Stand des Projekts eine neue Zuteilung getroffen; die Entwicklerteams setzen ihre Arbeit fort. Der Vorgang wiederholt sich bis schließlich alle Komponenten abgeschlossen wurden.

Über sogenannte *Reinforcement Learning* (RL) Algorithmen lässt sich eine optimale Scheduling Strategie bestimmen, welche die zu erwartende Dauer des Projekts minimiert. Ein solcher Algorithmus ist *Value Iteration* (VI), die *Dynamische Programmierung* nutzt [Pad05; Pad06]. Jedoch ist VI nur auf „kleine“ Instanzen des Problems anwendbar, da der SW-MDP mit zunehmender Projektgröße sehr schnell wächst. Einen Eindruck gibt Tabelle 1.1 [Wei10, S.93]. Weitere Erläuterungen zu den in der Tabelle verwendeten Instanzbezeichnungen geben wir in Kapitel 3.

Instanz	Zustände
5:3 Min	70.000
6:3 Min	550.000
7:4 Min	28.200.000
11:2 Min	62.000.000

Tabelle 1.1.: Ungefähre Größe des Zustandsraums für einige kleine Instanzen

Aussichtsreicher als Value Iteration sind *Temporal Difference* (TD) Methoden wie der Sarsa(λ) Algorithmus [Wei10; PW11a]. Jedoch lassen sich auch mit Sarsa(λ) aufgrund hohen Zeit- und Platzbedarfs nur mittelgroße Instanzen lösen.

Ziele dieser Bachelorarbeit In dieser Bachelorarbeit untersuchen wir, inwieweit eine *Parallelisierung des Reinforcement Learnings* die Anwendbarkeit auf größere Instanzen oder zumindest schnelleres Lösen von bereits lösbaren Instanzen erlaubt. Durch Parallelisierung von RL Algorithmen konnten bereits für andere praktische Anwendungen signifikante Verbesserungen erzielt werden [Sai+09; LP01].

Als Grundlage für die Parallelisierung dient uns eine von D. Weiss im Zuge seiner Masterarbeit [Wei10] entwickelte, in JAVA geschriebene Software, die mithilfe der Parallelisierungen zukünftig die Möglichkeiten moderner Mehrkernprozessoren nutzen soll.

Im Einzelnen haben wir ...

- bereits vorhandene [BT96; Kre02; Kre03; FE06; GK07; LP01; PEM02; CP05; WS04; Kus+06], aber auch eigene Ansätze zur Parallelisierung von Reinforcement Learning verglichen und ihre Eignung für den SW-MDP und eine mögliche Implementierung im Rahmen der genannten Software und Zielarchitektur bewertet.

- zwei aus unserer Sicht gut geeignete Ansätze, „mehrere Lerner mit gemeinsamer, kompletter Kostentabelle“ (Abschnitt 4.2.2.1) und „mehrere Simulationskerne“ (Abschnitt 4.3.1) in die bestehende Optimierungssoftware integriert.

- die implementierten Ansätze für verschiedene SW-MDP Instanzen getestet sowie eine mögliche Leistungsverbesserung und andere Nebeneffekte experimentell untersucht.

- die beiden Parallelisierungen anhand der Ergebnisse aus den Experimenten hinsichtlich ihrer Effektivität verglichen.

Ergebnisse

- Beide Ansätze schnitten ähnlich gut ab und konnten die Berechnung einer optimalen Strategie teilweise deutlich beschleunigen. Dabei erzielten wir auf dem verwendeten Vierkernprozessor Speedups von bis zu 3,8 (Abb. 8.2g, S.43).

- Die größten Instanzen des SW-MDPs boten auch das größte Potenzial für eine Beschleunigung durch die Parallelisierungen.

- Unerwünschte Nebeneffekte beim Einsatz mehrerer Agenten wie beispielsweise eine schlechtere Effizienz konnten wir – entgegen unserer Erwartungen – nicht feststellen (Abschnitt 8.3, S.48).

- Da der Einsatz von acht Threads mit Hyper-Threading [VGD09] für große Instanzen eine zusätzliche Leistungssteigerung brachte, sehen wir die Möglichkeit einer noch größeren Beschleunigung auf Prozessoren mit noch mehr Rechenkernen.

2
Reinforcement Learning

Dieses Kapitel soll in knapper Form die für das Verständnis der Parallelisierungsansätze nötigen Begriffe erläutern. Die Definitionen der Begriffe stammen ebenso wie die Algorithmen aus dem Standardwerk von Barto & Sutton [SB98]. Eine weitere gute Übersicht über Reinforcement Learning geben Kaelbling, Littman & Moore [KLM96].

Ein *Reinforcement Learning Problem* [SB98, S.51ff] besteht aus einem *Agenten* (bzw. *Lerner*), der mit seiner *Umwelt* interagiert. Zu jedem Zeitpunkt i befindet sich die Umwelt in einem bestimmten Zustand s_i, den der Agent wahrnimmt. Der Agent kann nun aus einer Menge von in diesem Zustand möglichen *Aktionen* $A(s_i)$ eine Aktion a_i wählen, die er durchführt. Die Umwelt reagiert daraufhin mit dem Wechsel in einen (möglicherweise neuen) *Folgezustand* s_{i+1} und einer *Rückmeldung* r_{i+1} in Form von auftretenden Kosten[1] (Abb. 2.1). Der Ablauf wiederholt sich, bis schließlich ein *terminaler Zustand* erreicht wird[2]. Das Ziel des Agenten ist es, einen solchen Zustand mit möglichst geringen Gesamtkosten zu erreichen.

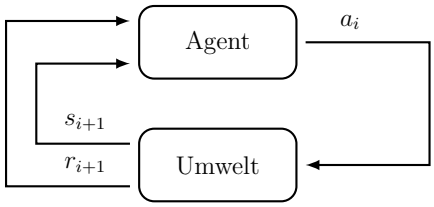


Abbildung 2.1.: Schema eines Reinforcement Learning Problems [SB98, S.52]

Reinforcement Learning Probleme formuliert man oft als sogenannte Markov'sche Entscheidungsprozesse (engl. *Markov Decision Process*, MDP) [SB98, S.66ff]. Ein MDP besitzt eine *Zustandsmenge* S sowie eine *Aktionsmenge* A. Wählt der Agent in Zustand $s \in S$ die Aktion $a \in A(s)$, so erfolgt mit einer gegebenen *Übergangswahrscheinlichkeit* $P(s, a, s')$ eine *Transition* (Zustandswechsel) in den Folgezustand s'. Für jede Transition (s, a, s') ist dabei der Erwartungswert $g(s, a, s')$ ihrer Kosten bekannt. Eine Folge von Transitionen bezeichnen wir auch als *Trajektorie*.

Die Entscheidungen des Agenten beruhen auf einer *Policy* oder *Strategie* π, die, in Abhängigkeit vom Zustand s, den möglichen Aktionen $A(s)$ Wahrscheinlichkeiten $\pi(s, a)$

[1] Analog lassen sich Reinforcement Learning Probleme auch mit Belohnungen statt Kosten charakterisieren; die Formulierung mit Kosten ist jedoch für den SW-MDP passender.

[2] Die Möglichkeit kontinuierlicher Probleme ohne terminalen Zustand [SB98, S.57f] spielt im Rahmen des SW-MDPs keine Rolle, daher wird sie hier ausgespart.

zuordnet, mit denen der Agent sie wählt. π impliziert eine Funktion G^π, die jedem Zustand die erwarteten Kosten bis zum Erreichen eines terminalen Zustands bei Befolgung von π zuordnet. Diese Kosten bezeichnen wir auch einfach als *Cost-to-go*. Analog können wir auch eine Funktion Q^π definieren, die für ein Paar (s, a) aus Zustand und Aktion die zu erwartenden Kosten bei Wahl von a in s und anschließender Befolgung von π angibt.

Damit der Agent sein Ziel minimaler Gesamtkosten erreicht, muss er also eine *optimale Policy* π^* kennen, welche die erwarteten Kosten minimiert. Die Cost-to-go-Funktion G^* von π^* erfüllt die *Bellman-Optimalitätsgleichung* für alle Zustände $s \in S$ [SB98, S.76f]:

$$G^*(s) = \min_{a \in A(s)} \sum_{s' \in S} (P(s, a, s') \cdot (g(s, a, s') + G^*(s')))$$

π^* können wir mittels *Generalisierter Policy Iteration* [SB98, S.105ff] bestimmen: Beginnend mit einer beliebigen Policy und ebenfalls beliebigen initialen Cost-to-go-Werten, wird eine Folge von abwechselnden *Evaluations-* und *Improvementschritten* durchgeführt. Die beiden Phasen bezeichnet man auch als *Prediction* bzw. *Control*. Ein Evaluationsschritt bestimmt für jeden Zustand approximativ die zu erwartenden Kosten unter π, verbessert also die zunächst ungenaue Cost-to-go-Funktion, während der Improvementschritt durch die *greedy Policy* bezüglich der Cost-to-go-Werte eine verbesserte Policy [SB98, S.95] ableitet. Die greedy Policy wählt in s mit Wahrscheinlichkeit 1 die greedy Aktion $greedy(s)$:

$$greedy(s) = \arg\min_{a \in A(s)} \sum_{s' \in S} (P(s, a, s') \cdot (g(s, a, s') + G^\pi(s'))$$

Hierbei liefert arg min das a, für welches die Summe minimal wird. Beide Schritte werden wiederholt, bis die Policy sich nicht weiter verändert. Abbildung 2.2 verdeutlicht das Schema.

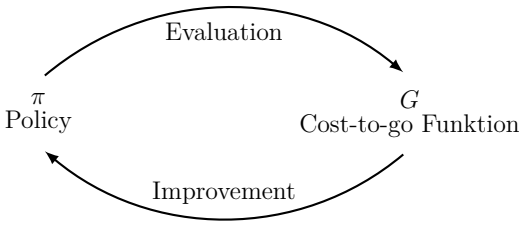


Abbildung 2.2.: Schema der Generalisierten Policy Iteration [SB98, S.105, Fig. 4.7]

Die Cost-to-go-Funktion wird oft – so auch in der erwähnten Optimierungssoftware – mittels einer *Kostentabelle* repräsentiert, die für Zustände deren Cost-to-go-Werte speichert. Eine andere Möglichkeit wäre die sogenannte *Funktionsapproximation*, bei der die Cost-to-go-Funktion nicht explizit mit Paaren aus Zustand und Cost-to-go dargestellt, sondern über eine parametrisierte Funktion approximiert wird.

Es existieren zahlreiche Algorithmen, die nach dem Prinzip der Generalisierten Policy Iteration arbeiten. Für uns ist dabei die Klasse der *Temporal Difference* Methoden [SB98,

S.133ff] von besonderer Bedeutung. Ihre Arbeitsweise lässt sich wie folgt umreißen: Für die Evaluation werden zunächst k Trajektorien simuliert. In der Regel wird dabei eine *ϵ-greedy* Policy genutzt, die für jeden Zustand mit Wahrscheinlichkeit ϵ eine zufällige, ansonsten die greedy Aktion wählt. Bei Einsatz der greedy Policy wären alle Trajektorien gleich, wodurch kein Wissen über momentan nicht als optimal erscheinende Aktionen und die resultierenden Folgezustände gewonnen würde. Für die in den Trajektorien vorkommenden Zustände werden dann die Cost-to-go mit erhaltenen Kostenstichproben und bereits vorhandenen Cost-to-go-Schätzungen approximiert. Die genaue Formel variiert von Algorithmus zu Algorithmus. Schließlich werden die Cost-to-go der besuchten Zustände anhand der neuen Approximation aktualisiert.

```
1  Initialisiere die Kostentabelle Q beliebig
2  Wiederhole:
3      Simuliere k Trajektorien unter einer ε−greedy Policy
4      Für jede Trajektorie t = s_0, a_0, r_0, s_1, a_1, ..., s_{T−1}, a_{T−1}, r_{T−1}, s_T:
5          Initialisiere E mit E(s, a) = 0 für alle (s, a)
6          Für jede Transition (s, a, r, s', a') in t:
7              δ = r + Q(s', a') − Q(s, a)
8              e(s, a) = e(s, a) + 1
9              Für alle Paare (s, a) ∈ S × A:
10                 Q(s, a) = Q(s, a) + α · δ · e(s, a)
11                 e(s, a) = λ · e(s, a)
```

Listing 2.1: Der Sarsa(λ) Algorithmus in Pseudocode

Listing 2.1 zeigt den Sarsa(λ) Algorithmus [SB98, S.179ff] wie wir ihn für den SW-MDP einsetzen in Pseudocode Form. Der Algorithmus arbeitet mit einer Kostentabelle Q für Zustand-Aktions-Paare. Bei der Evaluation wird jede Trajektorie schrittweise durchlaufen und die *One-Step Temporal-Difference* („TD-Fehler“) [SB98, S.164]

$$\delta = r + Q(s', a') - Q(s, a)$$

für jede Transition (s, a, r, s', a')[3] berechnet. Eine sogenannte *Eligibility Trace* E weist jedem Zustand-Aktions-Paar (s, a) eine Gewichtung $E(s, a)$ zu, mit der diese Temporal Difference (TD), vermindert um einen Faktor α, die sogenannte *Schrittweite*, auf dessen Cost-to-go angewendet wird. Zu Beginn hat jedes Zustand-Aktions-Paar die Gewichtung 0. Tritt ein Zustand-Aktions-Paar in der Trajektorie auf, wird seine Gewichtung um 1 inkrementiert. Nach jeder Transition wird die Eligibility Trace für alle Paare mit einem Faktor $\lambda \in [0, 1]$ multipliziert. Je kleiner λ gewählt wird, desto weniger beeinflusst die Temporal Difference also die Kostenschätzungen für weiter zurückliegende Aktionen.

Die Updates in Zeile 10 des Algorithmus' können entweder wie angegeben sofort ausgeführt oder zunächst angesammelt und erst nach der Bearbeitung aller Transitionen einer Trajektorie niedergeschrieben werden. Ersteres bezeichnet man auch als *online*, letzteres als *offline Updates* [SB98, S.166].

[3] Das 5-tupel (s, a, r, s', a') drückt aus, dass die Umwelt von Zustand s zu s' wechselte, nachdem Aktion a gewählt wurde. Dabei traten Kosten r auf und a' war die in s' gewählte Aktion.

3
Der SW-MDP

Im Folgenden beschreiben wir knapp die Modellierung eines Softwareprojektes als MDP mit den zugehörigen Bezeichnungen. Von Padberg wird der SW-MDP in mehreren Artikeln [Pad99; Pad02; Pad06] ausführlicher vorgestellt.

Im SW-MDP besteht die zu entwickelnde Software aus einer Zahl von *Komponenten*, die von verschiedenen *Teams* bearbeitet werden. Dabei ist die Zeit, die eine Komponente zur Fertigstellung benötigt, ebenso unbekannt wie wann und ob Probleme auftreten, deren Behebung zusätzliche Zeit (*Rework*) benötigt. Aufgrund von *Kopplung* der Komponenten untereinander können sich auftretende Probleme auch auf weitere Komponenten auswirken.

Die *Zustände* [Pad99, S.112], die den momentanen Status des Projekts wiederspiegeln, modellieren wir wie folgt: Der *Fortschritt der Komponenten* und die momentan nötige *Überarbeitungszeit* zur Behebung von Problemen werden in beliebigen Zeiteinheiten durch jeweils einen Vektor codiert. Schließlich enthält jeder Zustand noch einen *Countdown*, der die maximale verbleibende Entwicklungszeit angibt. Dies ermöglicht die Modellierung von Projektabbrüchen bei Überschreiten der Frist zur Fertigstellung der Software.

Als Beispiel betrachten wir den Zustand s aus einem Projekt mit vier Komponenten und zwei Teams:

$$s = \left(17, \begin{pmatrix} 4 \\ \infty \\ 3 \\ 0 \end{pmatrix}, \begin{pmatrix} 0 \\ 0 \\ 2 \\ 0 \end{pmatrix}\right)$$

In s verbleiben noch 17 Zeiteinheiten, um das Projekt fertigzustellen. Der erste der beiden Vektoren ist der Fortschrittsvektor. Wir sehen, dass die erste und dritte Komponente des Projekts bereits vier, respektive drei Zeiteinheiten lang bearbeitet wurden. Das Symbol für Unendlichkeit ∞ drückt aus, dass die zweite Komponente bereits fertiggestellt wurde. Der zweite Vektor gibt das Rework an; nur die dritte Komponente muss momentan überarbeitet werden, wofür zu diesem Zeitpunkt noch zwei Zeiteinheiten nötig sind. Die Teamzuteilung selbst ist nicht Teil des Zustands, sie ist durch die Aktionen festgelegt (siehe unten).

Transitionen [Pad99, S.113] finden statt, wenn die bisherige *Projektphase* endet, d.h. wenn mindestens eine der Komponenten fertiggestellt wird oder bei mindestens einer der Komponenten ein Problem auftritt. Die Kosten einer Transition entsprechen jeweils der Zahl der zwischenzeitlich, d.h. seit der letzten Aktion, vergangenen Zeiteinheiten.

Die *Aktionen* [Pad02, S.129] im SW-MDP entsprechen den möglichen *Teamzuteilungen*,

die zu den Entscheidungszeitpunkten nach jeder Transition gewählt werden können. Dabei gelten die Einschränkungen, dass ein Team immer nur an einer Komponente arbeiten kann und jede Komponente von höchstens einem Team bearbeitet wird.

Zur Verdeutlichung des Ablaufs eines Softwareprojekts im SW-MDP zeigt Abbildung 3.1 einen Ausschnitt aus einer möglichen Trajektorie. Für jeden Zustand sind nur der Countdown sowie der Fortschritts- und der Überarbeitungszeit-Vektor angegeben; die Teamzuteilungen stehen über den durch Pfeile dargestellten Transitionen. Das Projekt besteht in diesem Beispiel aus drei Komponenten A, B & C und zwei Teams. Die erste Aktion teilt Team 1 Komponente A und Team 2 Komponente B zu. Nach drei Zeiteinheiten ist Komponente B fertiggestellt; aufgrund eines Problems ist jedoch eine Überarbeitung von A und B nötig geworden. Nach Fortsetzung des Projekts mit identischer Teamverteilung und einer weiteren Zeiteinheit wurde Komponente B überarbeitet und das freigewordene Team 2 kann auf C angesetzt werden.

$$16, \begin{pmatrix}0\\0\\0\end{pmatrix}, \begin{pmatrix}0\\0\\0\end{pmatrix} \xrightarrow[3]{\substack{1 \mapsto A\\2 \mapsto B}} 13, \begin{pmatrix}3\\\infty\\0\end{pmatrix}, \begin{pmatrix}2\\1\\0\end{pmatrix} \xrightarrow[1]{\substack{1 \mapsto A\\2 \mapsto B}} 12, \begin{pmatrix}3\\\infty\\0\end{pmatrix}, \begin{pmatrix}1\\0\\0\end{pmatrix} \xrightarrow[1]{\substack{1 \mapsto A\\2 \mapsto C}} \cdots$$

Abbildung 3.1.: Ausschnitt aus einer Trajektorie im SW-MDP

Eine *Projektinstanz* umfasst eine Menge von Komponenten und Teams. Es gibt vier verschiedene Komponententypen $\mathcal{A}$, $\mathcal{B}$, $\mathcal{C}$ & $\mathcal{D}$, die sich hinsichtlich ihrer Bearbeitungszeit und Fehleranfälligkeit unterscheiden [Pad06, S.81]. Jedes Team kann auf jede der Komponenten *spezialisiert* sein, in welchem Falle es die jeweilige Komponente im Mittel schneller fertigstellt.

Jede Instanz nutzt außerdem eines von vier *Kopplungsmodellen* [Pad06, S.83f], die festlegen, wie sich Fehler bei einer Komponente auf andere Komponenten auswirken. Während sich bei *minimaler* Kopplung Fehler gar nicht fortpflanzen, müssen bei *maximaler* Kopplung stets alle Komponenten überarbeitet werden. Bei *uniformer* Kopplung ist jede Teilmenge der Komponenten mit gleicher Wahrscheinlichkeit von einem Fehler betroffen. Zuletzt besteht die Möglichkeit *asymmetrischer* Kopplung [PW11b, S.6], die eine genauere Modellierung von realistischen Projekten durch Spezifikation einiger zentraler Komponenten, von denen sich Fehler eher ausbreiten als von anderen, erlaubt.

Abbildung 3.2 zeigt, wie wir die verschiedenen Instanzen des SW-MDPs spezifizieren. Zuoberst verrät der Name der Instanz, dass sie elf Komponenten und drei Entwicklerteams beinhaltet. „ASYM“ steht für asymmetrische Kopplung; entsprechend werden die Abkürzungen „MIN“ und „MAX“ für minimale, respektive maximale Kopplung verwendet. Die Tabelle darunter gibt in jeder Spalte für eine der Komponenten deren Typ und die auf diese Komponente spezialisierten Teams an. Zentrale Komponenten im Falle von asymmetrischer Kopplung werden durch einen fett gedruckten Komponententyp kenntlich gemacht.

Welche Transitionen auftreten können und wie hoch ihre Wahrscheinlichkeit ist, wird durch die sogenannten *Basiswahrscheinlichkeiten* [Pad06, S.82f] der Komponententypen festgelegt. Diese geben an, wie hoch die Wahrscheinlichkeit dafür ist, dass nach einer

11:3 ASYM

$\mathcal{A}$	$\mathcal{A}$	$\mathcal{A}$	$\mathcal{A}$	$\mathcal{A}$	$\boldsymbol{\mathcal{D}}$	$\boldsymbol{\mathcal{D}}$	$\boldsymbol{\mathcal{D}}$	$\mathcal{D}$	$\mathcal{D}$	$\mathcal{D}$
1	2	3	1, 2	2, 3	-	-	-	1	2	3

Abbildung 3.2.: Spezifikation der *11:3 ASYM* Instanz

bestimmten Zahl von Zeiteinheiten eine Komponente eines bestimmten Typs fertiggestellt wird, bzw. ein Problem auftritt.

Als Beispiel betrachten wir Abbildung 3.3, welche die Basiswahrscheinlichkeiten für den Komponententyp $\mathcal{C}$ angibt. Eine Komponente des Typs $\mathcal{C}$ wird frühestens nach fünf Zeiteinheiten (in denen sie bearbeitet wurde) fertiggestellt, dies geschieht mit einer Wahrscheinlichkeit von 10%. Wenn noch kein anderes Ereignis eingetreten ist, kann nach sechs vergangenen Zeiteinheiten mit einer Wahrscheinlichkeit von ebenfalls 10% ein Fehler auftreten; in 30% der Fälle wird die Komponente hingegen zu genau diesem Zeitpunkt abgeschlossen.

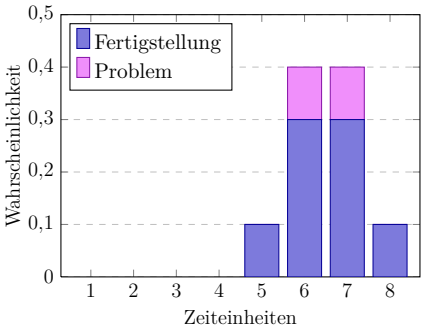


Abbildung 3.3.: Basiswahrscheinlichkeiten für den Komponententyp $\mathcal{C}$

Wir verwenden öfter den Begriff der *„Größe“ einer SW-MDP Instanz*. Konkret meinen wir damit die Größe des Zustandsraums, da diese auch bestimmt, wie aufwendig die Berechnung einer optimalen Policy ist. Die Zahl der Zustände hängt von vielen Faktoren ab [Wei10, S.18]. Im Hinblick auf die Experimente in dieser Arbeit sind davon besonders die Anzahl der Komponenten und das Kopplungsmodell von Bedeutung. Mehr Komponenten erzeugen einen größeren Zustandsraum und die „gemäßigten“ Kopplungsmodelle UNI und ASYM erzeugen meist größere Instanzen als die Extremformen MIN und MAX.

Die Simulation der Trajektorien im SW-MDP ist aufwendig [Wei10, S.79f]. Für die Ermittlung der nächsten Aktion müssen zunächst alle Möglichkeiten zur Verteilung der freien Teams auf freie Komponenten, oder umgekehrt, je nachdem von welchen mehr vorhanden sind, gefunden werden. Danach wird die günstigste darunter durch Nachschlagen in der Kostentabelle ausgewählt. Die Berechnung des Folgezustandes erfolgt durch ein mehrstufiges Zufallsexperiment. Dabei wird der bisherige Zustand wiederholt um eine Zeiteinheit verschoben und für jede aktive Komponente gewürfelt, um zu entscheiden, ob diese fertiggestellt wurde oder ein Problem auftrat. Bei Eintritt eines solchen Ereignisses steht der neue Zustand fest.

4
Stand der Technik

In diesem Kapitel besprechen wir bereits bekannte sowie eigene Ansätze zur Parallelisierung von Reinforcement Learning Algorithmen. Der Fokus liegt dabei auf dem, für den SW-MDP verwendeten, Sarsa(λ) Algorithmus. Eine Auswahl von für unsere Zwecke besonders geeigneten Ansätzen folgt in Kapitel 5.

4.1. Einteilung der Ansätze

Die vorgestellten Ansätze können grob in zwei Gruppen unterteilt werden: Ansätze auf *Ebene der „ganzen Lerner“* bzw. *Agentenebene* und Ansätze auf *Ebene der „einzelnen Verarbeitungsschritte“* eines Reinforcement Learning Algorithmus'.

Bei ersteren bezeichnen wir mit einem *Lerner* bzw. *Agenten* eine ganzheitliche Einheit, die Simulations- und Predictionschritte durchführt. Die generelle Idee zur Parallelisierung besteht nun darin, mehrere Agenten gleichzeitig einzusetzen, die für den selben MDP eine optimale Strategie erlernen und durch Teilen des Gelernten kooperieren. Mehrere Agenten können dabei nicht nur den Lernprozess durch „funktionale“ Parallelisierung beschleunigen, sondern auch dazu genutzt werden, verstärkt noch schlecht erkundete Teile des MDPs zu erforschen. Dies ist besonders interessant, da breite Erkundung des Zustandsraums und genaue Bewertung besuchter Zustände gleichermaßen wichtig sind, um eine optimale Strategie zu erlernen, aber miteinander im Wettstreit um Rechenzeit stehen („exploration-exploitation dilemma“ [SB98, S.4f]).

Auf Ebene der einzelnen Verarbeitungsschritte ist der Gedanke, einzelne Schritte der Reinforcement Learning Algorithmen wie Simulation oder Prediction parallelisiert auszuführen. Die Ansätze auf dieser Ebene sind dabei oft aufgrund der Struktur der Algorithmen recht naheliegend.

4.2. Ansätze auf Agentenebene

4.2.1. Mehrere Lerner mit eigenen Kostentabellen

Bei diesem Ansatz besitzt jeder der Agenten eine eigene Kostentabelle, bzw. im Falle von Funktionsapproximation eine eigene Kostenfunktion, für den gesamten Zustandsraum, in der er die eigenen erlernten Kostenschätzungen speichert. Von Zeit zu Zeit, beispielsweise immer nach einer bestimmten Zahl von Iterationen, tauschen die Agenten

die gelernten Cost-to-go-Werte, bzw. die Parameterwerte ihrer Approximationsfunktionen, untereinander aus. Letztere sind bei einem Austausch natürlich erheblich günstiger. Diese periodische Synchronisation kann das komplette Gelernte oder aber nur besonders wichtige Teile davon umfassen. Abbildung 4.1 verdeutlich das Schema.

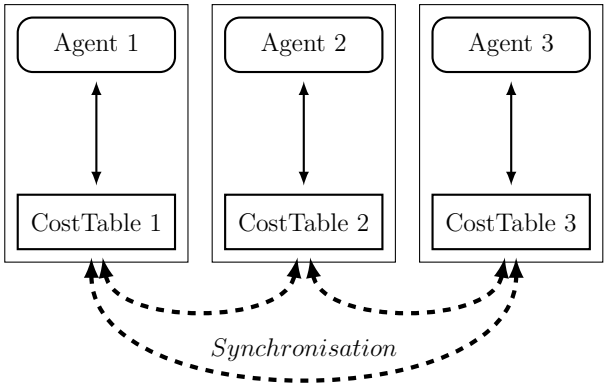


Abbildung 4.1.: Mehrere Agenten mit jeweils eigener Kostentabelle. Die Agenten arbeiten auf logisch getrenntem Speicher (dargestellt durch die Umrandungen). In regelmäßigen Abständen findet ein Austausch des Gelernten statt.

Der Einsatz von mehreren getrennten Kostentabellen mit Synchronisation orientiert sich klar an einer Clusterarchitektur, da auf gemeinsamen Speicher verzichtet wird, der einen direkteren Austausch ermöglichen würde.

4.2.1.1. Komplette Synchronisation

Diese Variante, bei der die Kostentabellen jeweils komplett ausgetauscht werden, wird von M. Kretchmar beschrieben [Kre02; Kre03].

Da die Agenten ihre Aktionen unabhängig voneinander wählen, unterscheidet sich in der Regel die Qualität der Kostenschätzungen für einen bestimmten Zustand zwischen zwei Agenten: Je öfter ein Agent infolge seiner Aktionen einen Zustand besucht hat, desto höher ist die zu erwartende Genauigkeit seiner Schätzung des Cost-to-go-Werts für den jeweiligen Zustand. Daher macht es Sinn, die Anzahl der Updates eines Eintrags in der Kostentabelle als Gewichtung beim Austausch zu nutzen, sodass mehr Updates zu einer stärkeren Gewichtung führen [Kre02, S.3].

Um diese Gewichtung zu realisieren, muss jeder Agent für jeden Cost-to-go-Wert sowohl das Ergebnis des eigenen, als auch das des gemeinsamen Lernens festhalten [Kre02, S.3]. Dadurch verdoppelt sich der Speicherbedarf für die Kostentabelle.

Die Kosten der Synchronisation steigen linear mit der Größe der Kostentabelle. Allerdings untersucht Kretchmar nur einen sehr einfachen[1] MDP mit einem einzigen Zustand [Kre02]. Von Coulom und Preux wird dieser Ansatz auch auf größere MDPs übertragen [CP05], jedoch unter Benutzung von Funktionsapproximation.

[1]Kretchmar wendet die Parallelisierung auf das Problem des *n-armigen Banditen* [SB98, S.26] an.

4.2.1.2. Teilweise Synchronisation

Durch Priorisierung bestimmter Werte kann versucht werden, den teuren Austausch auf das Wichtigste zu beschränken. M. Grounds stellt dafür einen Algorithmus für die Verwendung mit Funktionsapproximation vor [GK07]: Jedem Parameter wird eine umso höhere Priorität zugewiesen, je stärker er sich seit seinem letzten Austausch verändert hat. Die zugrunde liegende Annahme dabei ist, dass stärker schwankende Parameter noch nicht konvergiert sind und daher mehr Aufmerksamkeit benötigen. Es werden immer nur die n wichtigsten Änderungen ausgetauscht.

Eine Übertragung dieses Algorithmus' auf Kostentabellen halten wir für möglich. Dies könnte eine Verbesserung gegenüber der kompletten Synchronisation darstellen, da zumindest der Austausch der Kostenwerte weniger Zeit benötigen würde.

4.2.2. Mehrere Lerner mit gemeinsamer Kostentabelle

Mehrere Lerner lassen sich auch unter Verwendung einer gemeinsamen Kostentabelle einsetzen. Dabei kann ein einzelner Lerner jeweils direkten Zugriff auf die komplette oder aber nur einen Teil der Tabelle erhalten.

4.2.2.1. Komplette Kostentabelle

Bei dieser Version, die von S. Fields vorgestellt wurde [FE06], gibt es keinerlei Abstimmung der Agenten untereinander; alle nutzen gleichzeitig die gesamte Tabelle zum Abrufen und Speichern der Kostenschätzungen. Dadurch findet eine unmittelbare gegenseitige Einflussnahme statt. Abbildung 4.2 soll die Situation verdeutlichen.

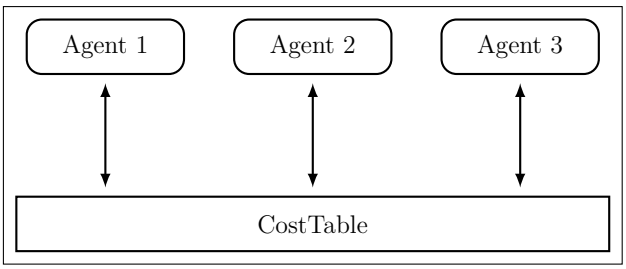


Abbildung 4.2.: Mehrere Agenten mit gemeinsamer Kostentabelle. Die Agenten haben Zugriff auf gemeinsamen Speicher, in dem sich die Kostentabelle befindet.

S. Fields ergänzt den Ansatz mit einer speziellen Hardwarearchitektur für seinen Algorithmus und nennt auch die Möglichkeit, für verschiedene Agenten verschiedene Policys zu nutzen. Letzteres könnte aus unserer Sicht eine einfache Möglichkeit sein, die Erkundung des Zustandsraums zu stärken, indem man greedy Policys neben ϵ-greedy Policys mit relativ großen ϵ einsetzt.

Aufgrund der Auslegung auf einen schnellen, gemeinsamen Speicher erscheint uns dieser Ansatz ideal für den Einsatz auf einer Multicorearchtiektur geeignet zu sein.

4.2.2.2. Partitionierte Kostentabelle

Eine andere Möglichkeit ist, den Zustandsraum des MDPs zu partitionieren und den Lernern verschiedene Teile zuzuweisen. Im Idealfall sollte dann jeder von ihnen auch nur auf die Kostenwerte für Zustände aus seiner Teilmenge zugreifen müssen.

Während die Umsetzung dieser Idee bei manchen RL Algorithmen einfach ist (z.B. für Value Iteration [WS04]), ist die Anwendung auf Temporal Difference Methoden schwieriger, da die Updates hier anhand von Trajektorien erfolgen: Enthält eine Trajektorie viele „fremde“ Zustände, so müssen für diese teure Anfragen an andere Agenten gestellt werden (siehe Abb. 4.3).

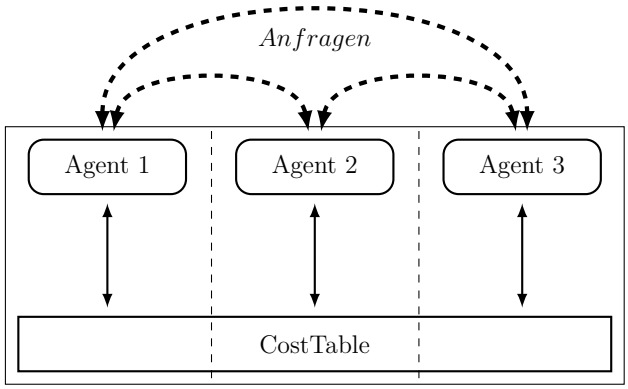


Abbildung 4.3.: Mehrere Agenten mit partitionierter Kostentabelle. Die Agenten arbeiten vorrangig auf einem eigenen Teil der Tabelle. Werden Werte aus einem anderen Teil benötigt, müssen Anfragen an den jeweiligen Besitzer gestellt werden.

Damit geht auch das Problem einher, eine Partition des Zustandsraums mit möglichst wenigen Transitionen zwischen den einzelnen Teilen zu finden, da die Parallelisierung nur unter dieser Voraussetzung effektiv sein kann. Während eine solche Aufteilung für manche MDPs offensichtlich ist, ist sie für den SW-MDP noch unklar. Eine Anwendung dieser Parallelisierungsidee auf einen TD Algorithmus wird z.B. von A. Printista für Q-Learning verfolgt, allerdings auf einem besonders gut geeigneten Labyrinth-MDP) [PEM02].

Der logisch verteilte Speicher lässt sich sowohl für eine Cluster- als auch für eine Multicorearchitektur umsetzen.

4.3. Ansätze auf Ebene der Verarbeitungsschritte

4.3.1. Mehrere Simulationskerne

Einen sehr einfachen Ansatz stellt die Parallelisierung der Simulation der für den Lernprozess nötigen Trajektorien dar. Diese Idee wurde bereits sehr früh von Bertsekas und Tsitsiklis erwähnt [BT96, S.418]. Mehrere „Simulationskerne“ erzeugen parallel Trajektorien; der Evaluationsschritt hingegen, der sowohl parallel als auch abwechselnd mit den

Simulationen erfolgen kann, bleibt ein sequenzieller Block. Die einzelnen Simulatoren arbeiten dabei unabhängig voneinander – es findet keinerlei Abstimmung untereinander statt.

In Abbildung 4.4 präsentieren wir eine mögliche Umsetzung dieser Idee. Mehrere Kerne erzeugen parallel Trajektorien und legen sie in einem Puffer („Trajektorienablage“) ab. Von dort können sie für die Evaluation entnommen werden. Hierbei bezeichnen wir die Einheit des Algorithmus', welche die Cost-to-go-Schätzung anhand der simulierten Trajektorien verbessert, also die Prediction durchführt, als „Predictor“.

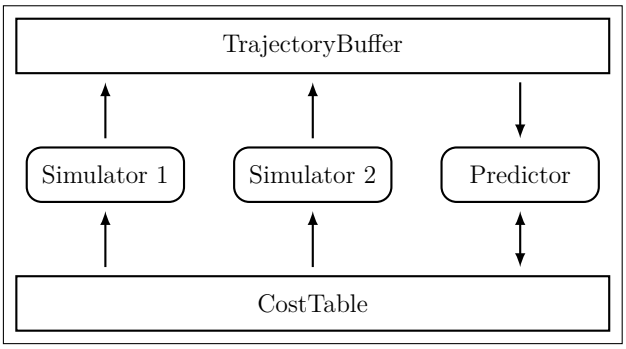


Abbildung 4.4.: Mehrere Simulationskerne. Es werden von mehreren Kernen parallel Trajektorien erzeugt und in einer Ablage gesammelt. Der Evaluationsschritt erhält sie von dort.

4.3.2. Zugriffe auf Kostentabelle im Predictor parallel

Mit der Größe der Kostentabelle werden die Zugriffe auf selbige ein immer größerer Faktor für die Laufzeit. Wir vermuten daher, dass eine Parallelisierung dieser Zugriffe TD Algorithmen im Falle von offline Updates (S.5) beschleunigen könnte. Wie in Abbildung 4.5 angedeutet, könnten die Cost-to-go-Werte der in einer Trajektorie vorkommenden Zustände oder Zustand-Aktions-Paare vor ihrer Verarbeitung parallel gelesen und die akkumulierten Updates am Ende der Verarbeitung parallel zurückgeschrieben werden.

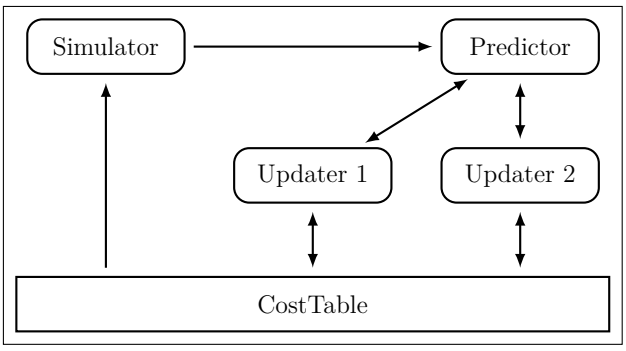


Abbildung 4.5.: Zugriffe auf Kostentabelle im Predictor parallel

4.3.3. arg min-Operator parallel

Die Berechnung der Aktion mit minimalen Kosten für einen bestimmten Zustand in der Simulation beinhaltet u.a. einen oder mehrere Zugriffe auf die Kostentabelle (siehe auch S.8). Verglichen mit der Prediction (Abschnitt 4.3.2) ist die Zahl der Zugriffe insgesamt sogar größer: Während in der Prediction für jede Transition nur ein Wert gelesen und später geschrieben werden muss, ist bei der Simulation eine Abfrage für jede mögliche Aktion notwendig. Es liegt daher nahe, die arg min-Operation zu parallelisieren. Ein derartiger Ansatz wird schon von Bertsekas erwähnt, wenn auch sehr vage als „Parallelisierung der Minimierung über alle Aktionen“ [BT96, S.418].

Da die möglichen Aktionen für den SW-MDP schrittweise berechnet werden, können diese (kombinatorischen) Berechnungen und das Nachschlagen effektiv nebenläufig erfolgen. Dabei könnte ein Kern nacheinander alle möglichen Aktionen bestimmen und an einen oder mehrere Kerne weiterreichen, welche die Aufgabe haben, für ihren Teil der Aktionen diejenige mit den minimalen Kosten zu bestimmen. Aus den resultierenden Minima könnte dann das globale Minimum bestimmt werden (Abb. 4.6).

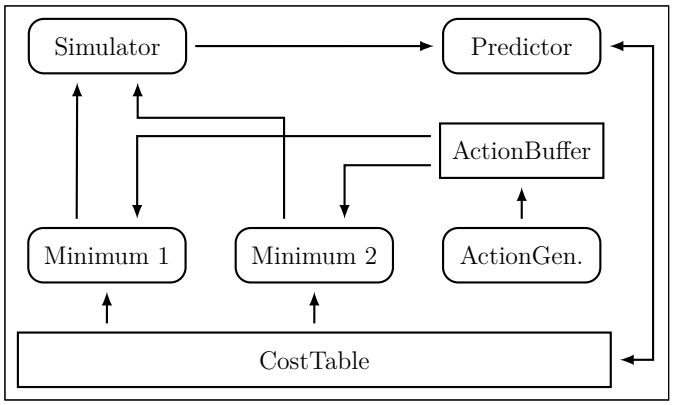


Abbildung 4.6.: *argmin*-Operator parallel. Ein Kern (ActionGenerator) erzeugt die möglichen Aktionen. Mehrere Kerne bestimmen die Aktion mit minimalen Kosten für einen Teil der Aktionen. Schließlich erhält der Simulator die Aktion mit global minimalen Kosten.

4.4. Weitere Möglichkeiten

Schließlich erwähnen wir noch zwei weitere Ansatzpunkte, die aus unserer Sicht für den SW-MDP allerdings nicht in Frage kommen:

Bei Benutzung von Funktionsapproximation können verschiedene Parameter der Funktion parallel optimiert werden [BT96, S.418]. Jedoch ist Funktionsapproximation für den SW-MDP momentan außer Frage: Die Kostenfunktion muss einen Zustand bzw. ein Zustand-Atkions-Paar anhand verschiedener Eigenschaften bewerten. Es ist jedoch noch völlig unklar, welche dies im Falle des SW-MDP sein sollten. Solche, für die zu

erwartenden Projektkosten ausschlaggebende, Eigenschaften herauszufinden, ist gerade das Ziel der aktuellen Forschung.

Desweiteren wäre eine *interne Parallelisierung der Simulationen* im SW-MDP interessant, da diese für den Großteil der Rechenzeit verantwortlich sind, wie der Ausschnitt aus einer Profileranalyse in Tabelle 4.1 zeigt.

Aufgabe	Anteil an Optimierung
Simulation	92%
Nächste Aktion berechnen	39%
Folgezustand berechnen	40%
Prediction	7%

Tabelle 4.1.: Ausschnitt aus einer Profileranalyse

Leider konnten wir aber keine größeren Blöcke von Berechnungen identifizieren, die sich problemlos parallel ausführen ließen. Die Simulation einer Trajektorie im SW-MDP besteht vielmehr aus vielen kleinen Berechnungen, die aufgrund von Datenabhängigkeiten sequenziell erfolgen müssen.

5 Auswahl der Ansätze

Im Folgenden treffen wir eine begründete Auswahl von Ansätzen, die uns für die Nutzung mit dem SW-MDP besonders geeignet erscheinen.

5.1. Besonders geeignete Ansätze

Mehrere Lerner mit gemeinsamer, kompletter Kostentabelle (4.2.2.1) Hierbei handelt es sich um einen Ansatz auf Agentenebene, dessen vergleichsweise einfache Kommunikation der Agenten durch Nutzen eines gemeinsamen Speichers ideal zu Multicoreprozessoren passt. Auch wenn die von Fields [FE06] demonstrierte hohe Effektivität durch eine eigens entworfene Hardwarearchitektur begünstigt wurde, erschien uns eine einfache Software-Implementierung immernoch aussichtsreich. Die Parallelisierung lässt sich scheinbar für viele TD Algorithmen umsetzen, so auch für Sarsa(λ).

Im Rahmen dieser Bachelorarbeit haben wir diesen Ansatz weiterverfolgt und konnten tatsächlich eine Beschleunigung erzielen. Unsere experimentelle Evaluation des Ansatzes findet sich in den Abschnitten 8.2.4 und 8.3 ab Seite 44.

Mehrere Simulationskerne (4.3.1) Auch wenn hier nur ein Teil des Algorithmus' parallelisiert wird, ließ der große Aufwand der Simulation im SW-MDP (Tab. 4.1, S.15) eine besonders gute Eignung dieses Ansatzes vermuten. Wir haben ihn daher implementiert und seine Effektivität für den SW-MDP untersucht. Die von uns erzielten Speedups können in den Abschnitten 8.2.2 und 8.2.3 ab Seite 39 nachgelesen werden.

Trotz seiner Einfachheit und der Erwähnung durch Bertsekas & Tsitsiklis [BT96, S.418] ist eine Auswertung dieses Ansatzes nach unserem Wissen ein Novum.

Zugriffe auf Kostentabelle im Predictor parallel (4.3.2) Profileranalysen legen nahe, dass Zugriffe auf die Kostentabelle besonders für große SW-MDP Instanzen einen starken Einfluss auf die Laufzeit der Optimierungssoftware haben. Zur Verdeutlichung der Verhältnisse haben wir in Tabelle 5.1 die fünf Methoden aufgelistet, die in einem Testlauf mit einer *12:3 UNI* Instanz die größten Anteile an der Laufzeit hatten. Wie zu sehen ist, nahmen Lookups in der Kostentabelle 56% der Zeit in Anspruch.

Deshalb halten wir diesen Ansatz für MDPs mit sehr großem Zustandsraum, also insbesondere für den SW-MDP, für interessant.

Methode	Anteil an der Laufzeit
`kernel.ctab.CountingQTable.get`	56%
`kernel.pol.CountingQTablePolicy.action`	11,7%
`sw.mdp.SWFast_LargeState.equals`	6,5%
`sw.mdp.combi.Permutator$Iterator.next`	3%
`sw.mdp.SW_SimulationIterator.toPrimitive`	1,3%

Tabelle 5.1.: Die fünf aufwendigsten Methoden bei Berechnung einer optimalen Strategie für eine *12:3 UNI* Instanz

arg min**-Operator parallel (4.3.3)** Da es sich um einen zu 4.3.2 analogen Ansatz an anderer Stelle des Algorithmus' handelt, gilt die gleiche Begründung.

5.2. Weniger gut geeignete Ansätze

Mehrere Lerner mit eigenen Kostentabellen (4.2.1) Diese beiden Ansätze orientieren sich eher an einer Clusterarchitektur. Eine Vervielfachung der Kostentabelle erscheint uns unmöglich; für größere SW-MDPs füllt oft bereits eine einzige Tabelle den gesamten Speicher aus. Der Ansatz wurde in den zitierten Quellen auch nur für sehr kleine MDPs untersucht.

Mehrere Lerner mit gemeinsamer, partitionierter Kostentabelle (4.2.2.2) Zunächst würde eine Implementierung dieses Ansatzes eine geeignete Aufteilung des SW-MDPs in Teilmengen mit möglichst wenigen verbindenden Transitionen erfordern. Eine derartige Aufteilung ist uns bislang nicht bekannt und bedürfte weiterer Forschung.

Außerdem müsste ein geeigneter Mechanismus zur Kommunikation der Agenten bei Zugriffen auf fremde Kostenwerte gefunden werden: Der vorhandene Algorithmus von Printista [PEM02] nutzt einen Master mit einer Kopie der gesamten Kostentabelle, der die einzelnen Lerner koordiniert. Da mangelnder Speicherplatz aber eines der Hauptprobleme bei der Arbeit mit dem SW-MDP darstellt, bewerten wir zumindest diese vorhandene Variante als ungeeignet. Insgesamt erscheint der Ansatz auch eher für Cluster gedacht zu sein, da auf gemeinsamen Speicher verzichtet wird.

Jedoch könnte zumindest versucht werden, bei Verwendung von mehreren Lernern auf einer gemeinsamen Kostentabelle, die Agenten so zu konfigurieren, dass sie de facto auf weitesgehend verschiedenen Bereichen des Zustandsraums arbeiten.

6 Analyse

Die Analyse und der Entwurf der parallelen Implementierungen erfolgen auf Grundlage der in Java geschriebenen Optimierungssoftware von D. Weiss. In seiner Masterarbeit werden die wichtigsten Klassen und Methoden erläutert [Wei10, S.65ff].

6.1. Mehrere Simulationskerne

Bisher wechseln sich in der Software die Simulation von Trajektorien und die Verwendung der Trajektorien durch die Predictionphase strikt ab. Mit der Parallelisierung bestehen nun die Möglichkeiten, dies beizubehalten und nur die Simulationsphase nebenläufig auszuführen, oder aber zur gleichen Zeit beides, Simulationen und Prediction, ablaufen zu lassen. Wir haben beide Variante implementiert (Abschnitt 7.3.1 & 7.3.2) und evaluiert (Abschnitt 8.2.2 & 8.2.3).

6.1.1. Simulationen nebenläufig, abwechselnd mit Prediction

Für diese Variante kann die Parallelisierung in eine neue, parallele Version der `Simulator` Klasse gekapselt werden. Diese Klasse ist für die Simulation von Trajektorien unter einer bestimmten Policy zuständig. An der Kommunikation des parallelen Simulators mit dem Predictor muss nichts verändert werden, die Trajektorien können wie bisher in einem Array übergeben werden. Abbildung 6.1 zeigt die Zusammenarbeit der verschiedenen Komponenten der Optimierungssoftware genauer.

An einigen Stellen wird Synchronisation erforderlich. Natürlich muss vor Beginn der Predictionphase auf alle Simulatoren gewartet werden und bei den Zugriffen auf die Trajektorienablage sollte es ebenfalls nicht zu Datenwettläufen kommen.

Für die Simulation der Trajektorien werden sehr viele Zufallszahlen benötigt. Da ein zentraler Generator, wie er momentan in der Optimierungssoftware verwendet wird, synchronisiert werden müsste, was zu langen Wartezeiten führen würde, ist es besser, einen exklusiven Zufallsgenerator pro Thread zu verwenden.

6.1.2. Simulationen und Prediction nebenläufig

Diese Version sollte eine bessere Auslastung der Hardware erzielen, es ist aber auch mehr Implementierungsaufwand zu erwarten; im Gegensatz zu der ersten Variante aus Abschnitt

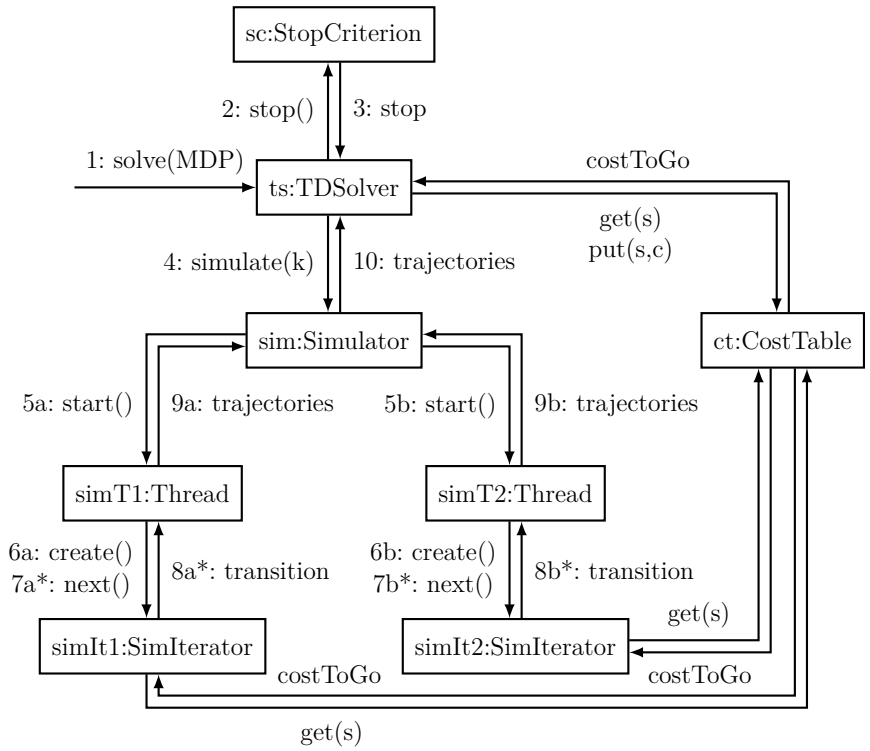


Abbildung 6.1.: Analyse der parallelen Simulation: Ein `TDSolver`-Objekt, das den Ablauf des TD Algorithmus' kapselt, fordert bei einem `Simulator`-Objekt eine Reihe von Trajektorien an (4.). Daraufhin startet der Simulator mehrere Threads, die parallel den Code zur Simulation von Trajektorien mithilfe von `SimulationIterator`-Objekten durchführen (7.-8.). Die simulierten Trajektorien werden in einer gemeinsamen Ablage gesammelt und an den `TDSolver` zurückgegeben, der sie zur Prediction verwendet.

6.1.1 muss die Kommunikation der Trajektorien zwischen Simulatoren und Predictor geändert werden. Dafür erscheint uns ein Produzent-Konsument-Schema geeignet: Die Simulator-Kerne erzeugen wiederholt Trajektorien und legen sie in einem Puffer ab, der Predictor entnimmt die Trajektorien dem Puffer und verwertet sie.

Ein weiterer Unterschied ist, dass sich hier der Zustand der Kostentabelle während der Simulation permanent durch die Arbeit des Predictors ändert. Die Simulatoren nutzen so immer die aktuellsten Cost-to-go-Werte. Da TD Algorithmen als sehr robust bezüglich von Veränderungen der Zeitpunkte, zu denen die Aktualisierungen der Tabelle stattfinden, bekannt sind, sollte diese Änderung kein Problem darstellen.

Neben der schon für 6.1.1 benötigten Synchronisation, ist es auch nötig, den Puffer für die Trajektorien zu synchronisieren.

6.2. Mehrere Lerner mit gemeinsamer, kompletter Kostentabelle

Für die Implementierung dieses Ansatzes ersetzen wir die `TDSolver` Klasse durch einen „Master“, der mehrere Kerne startet, welche die bisherige Arbeit eines `TDSolver`s parallel durchführen. Die Terminierung der Threads wird durch ein gemeinsames *Stopkriterium* gesteuert, bei dem die Agenten vor jeder Iteration nachfragen, ob sie terminieren sollen. Ein solches Stopkriterium kann sich z.B. an der Gesamtzahl der von allen Lernern durchgeführten Iterationen des Algorithmus' orientieren.

Genauso wie bei den beiden vorangegangenen Ansätzen ist es auch hier wichtig, dass ein eigener Zufallszahlengenerator pro Thread verwendet wird. Außerdem kommt eine Reihe von Stellen hinzu, an denen zusätzlich synchronisiert werden muss. Die wichtigsten dieser Stellen haben wir in Abbildung 6.2 zusammengefasst.

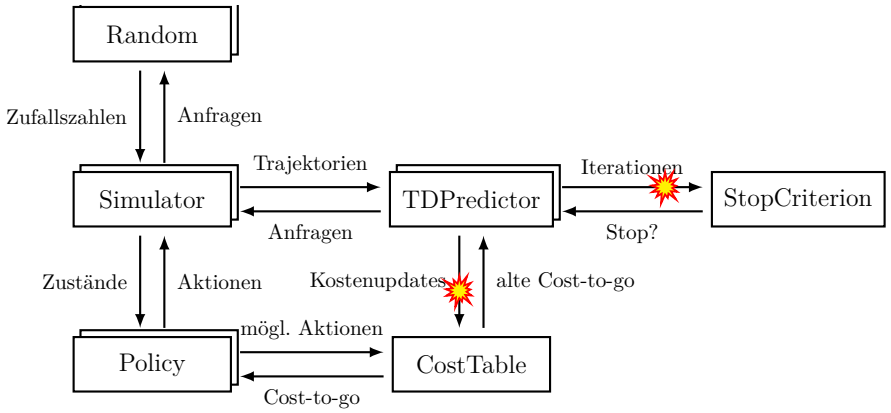


Abbildung 6.2.: Datenfluss in der Software mit kritischen Stellen. Datenwettläufe treten genau dort auf, wo mehrere Threads auf eine gemeinsame Ressource zugreifen und mindestens einer von ihnen schreibend.

Neben den Stopkriterien ist die kritischste dieser Stellen die gemeinsam genutzte Kostentabelle: Wir verwenden in unserer Optimierungssoftware für den Sarsa(λ) Algorithmus offline Updates (S.5), um teure Zugriffe auf die Tabelle einzusparen. Ein Nebeneffekt dieser Updatemethode ist die Vergrößerung der Zeitspanne zwischen Lesen und Schreiben der Cost-to-go-Werte der Zustand-Aktions-Paare. Ohne Synchronisation würden beim Schreibvorgang alle Änderungen durch fremde Agenten, die zwischen Lesen und Schreiben erfolgt sind, verloren gehen. Es käme zu einem klassischen Datenwettlauf.

Zur Lösung dieses Problems sehen wir zwei Alternativen: Auf der einen Seite das *Sperren* eines Eintrags in der Kostentabelle zwischen Lesen des alten und Schreiben des neuen Werts; auf der anderen Seite die Durchführung der Updates über eine *atomare Inkrement-Methode*.

Bei der ersten Option ist eindeutig gewährleistet, dass sich ein bestimmter Eintrag zwischen Lesen und Schreiben durch einen Agenten nicht mehr ändern kann. Da aber alle Trajektorien im selben Startzustand beginnen, wäre dieser permanent gesperrt und es

käme zu einer Sequenzialisierung. Abgesehen davon wäre eine korrekte Implementierung nicht trivial, da es bei MDPs die Zyklen enthalten leicht zu Deadlocks kommen kann, wenn das Sperren der Einträge durch zwei Agenten unkoordiniert erfolgt.

Wir haben daher die zweite Option gewählt. Statt einen neuen absoluten Cost-to-go-Wert in die Kostentabelle zu schreiben, wird der aktuelle Wert – der möglicherweise seit dem Lesen durch den Agenten bereits modifiziert wurde – auf threadsichere Weise um den TD-Fehler angepasst. So müssen Einträge nur kurzzeitig gesperrt werden. Bei dieser Vorgehensweise kann es zwar vorkommen, dass ein Update, das ursprünglich auf Grundlage eines älteren Cost-to-go-Werts berechnet wurde, auf einen inzwischen veränderten Wert angewendet wird; in unseren Experimenten zeigten sich jedoch keine Auswirkungen dadurch (Abschnitt 8.3, S.48).

7
Entwurf & Implementierung

Wie in Kapitel 6 erwähnt, haben wir die Parallelisierungen in eine vorhandene, in Java geschriebene, Reinforcement Learning Software (im Folgenden als „Optimierungssoftware“ oder kurz „Software“ bezeichnet) integriert. In diesem Kapitel stellen wir unsere Umsetzung der Ansätze in Code vor.

7.1. Realisierung der abstrakten Rechenkerne

Die zur Beschreibung der Ansätze genutzten abstrakten „Kerne“ (Kap. 4) realisierten wir in Java durch mehrere parallel laufende Threads. Für deren Erzeugung und Verwaltung nutzen wir einen `ThreadPoolExecutor` aus der Java Standardbibliothek. Dieser Executor nutzt eine feste Zahl von sog. Worker-Threads, in denen er ihm in Form von `Runnable` Objekten übergebene Jobs ausführt.

Solche Worker-Threads minimieren den Mehraufwand durch Threadverwaltung, da sie nur einmal erzeugt werden müssen und danach beliebig viele Jobs ausführen können. Gerade für Ansätze, bei denen die anfallenden Jobs nur kurzlebig sind, z.B. bei paralleler Simulation (Abschnitt 7.3.1), bringt dies eine deutliche Verbesserung gegenüber manueller Threaderzeugung.

7.2. Thread-eigene Zufallsgeneratoren

An vielen Stellen der Software werden Zufallszahlen benötigt, beispielsweise für die zufällige Wahl einer Aktion oder bei der Berechnung des Folgezustands nach Durchführung einer Aktion. Bisher verwendete die Software einen zentralen Zufallsgenerator, der alle benötigten Zahlen lieferte. Bei ersten Versuchen mit einer parallelen Implementierung wurde jedoch sofort sichtbar, dass dies für mehr als zwei Threads bei Synchronisierung des Generators zu einer zu starken Sequenzialisierung führte. Daher entschieden wir uns für thread-eigene Zufallsgeneratoren.

Wir fügten eine neue, von `Thread` abgeleitete, Klasse `ExclusiveRandomThread` hinzu, deren Objekte jeweils ein privates `Random`-Objekt beinhalten. Zusätzlich war eine Fabrik `ExclusiveRandomThreadFactory` zur Threaderzeugung durch den `ThreadPoolExecutor` nötig, um die Worker-Threads als `ExclusiveRandomThread`s starten zu können. Abbildung 7.1 zeigt die beiden Klassen im Kontext ihrer Basisklassen.

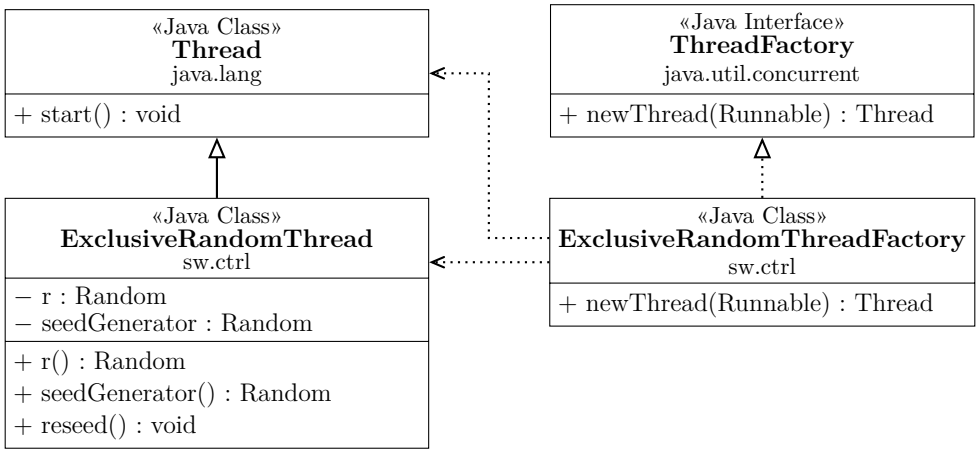


Abbildung 7.1.: Klassen die für thread-eigene Zufallszahlen eine Rolle spielen

Im zweiten Schritt ersetzten wir an mehreren Stellen des Programms die Zugriffe auf den zentralen Zufallsgenerator von der Form

```
1  this.rand = PRNG.r;
```

durch eine entsprechende Fallunterscheidung:

```
1  if (Thread.currentThread() instanceof ExclusiveRandomThread) {
2    this.rand = ((ExclusiveRandomThread)Thread.currentThread()).r();
3  } else {
4    this.rand = PRNG.r;
5  }
```

So wird bei Ausführung des Codes durch einen `ExclusiveRandomThread` der thread-eigene Generator genutzt, ansonsten wie bisher der zentrale.

In Zukunft ist über eine Verwendung der ab Java Version 7 eingebauten Klasse `ThreadLocalRandom` [Thr] nachzudenken, die das gleiche Ziel erreicht aber in der Benutzung sauberer ist.

7.3. Mehrere Simulationskerne

7.3.1. Simulationen nebenläufig, abwechselnd mit Prediction

Für diesen Ansatz implementierten wir eine neue, parallele Version `ParallelSimulator` der `Simulator` Klasse. Abbildung 7.2 zeigt, wie sich diese Klasse einfügt.

Dadurch, dass `ParallelSimulator` von `Simulator` erbt, kann der parallele Simulator sofort an allen Stellen genutzt werden, an denen bisher `Simulator`-Instanzen zur Simulation genutzt wurden. Außerdem können für kleine Werte von k, bei denen sich eine Parallelisierung nicht lohnt, direkt die von `Simulator` geerbten Methoden verwendet werden.

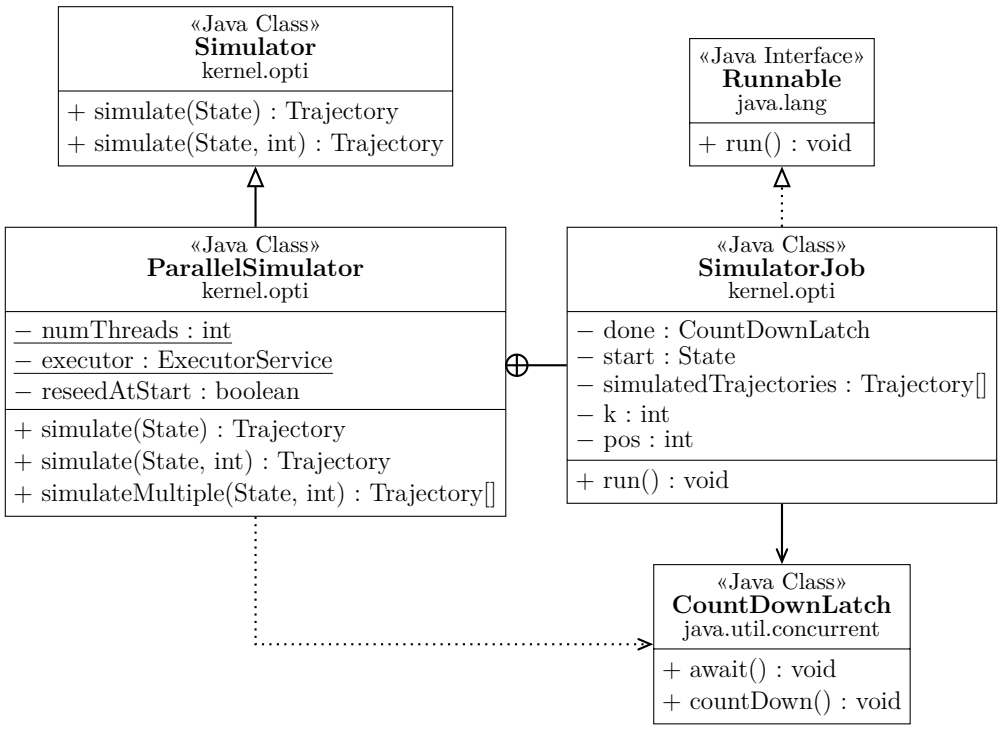


Abbildung 7.2.: An der parallelen Simulation beteiligte Klassen

Die vorhandenen Methoden von `Simulator` erlauben es allerdings nur, jeweils eine einzelne Trajektorie simulieren. Damit die Arbeit unter mehreren Threads aufgeteilt werden kann, müssen aber alle Trajektorien auf einmal angefordert werden. Hierzu fügten wir eine dementsprechende Methode `simulateMultiple(State, int)` hinzu. `simulateMultiple(State, int)` verteilt die k zu erzeugenden Trajektorien auf mehrere `SimulatorJob`-Instanzen, welche die eigentlichen Simulationen durchführen. Gesammelt werden die fertigen Trajektorien in einem (disjunkt adressierten) Array. Eine Barriere `done` stellt sicher, dass die `SimulatorJob`s ihre Arbeit beendet haben, bevor das Trajektorien-Array zurückgegeben wird. Die Definition der Methode ist in Listing 7.1 aufgeführt.

`SimulatorJob` ist eine innere Klasse von `ParallelSimulator`. Die `SimulatorJob`-Objekte sind nicht mehr als `Runnable` Objekt, die den in Listing 7.2 gezeigten Simulations-Code, der zur sequenziellen Version identisch ist, ausführen. Nach Beendigung der Simulation melden sich die `SimulatorJob`s an der Barriere.

Zuletzt mussten wir noch die Klasse `TabularTDSolver`, die den generellen Ablauf der TD-Algorithmen in ihrer Methode `solve` kapselt, so anpassen, dass sie zur Simulation eine Instanz von `ParallelSimulator` nutzt.

7.3.2. Simulationen und Prediction nebenläufig

Im Gegensatz zu 7.3.1 modifizierten wir für diese Variante der parallelen Simulation direkt die `TabularTDSolver` Klasse. Abbildung 7.3 zeigt die weiteren von diesem Parallelisierungsansatz betroffenen Klassen.

```
1  public Trajectory[] simulateMultiple(State start, int k) {
2    // Barriere für die SimulatorJobs
3    final CountDownLatch done = new CountDownLatch(numThreads);
4    // Sammelt die fertigen Trajektorien
5    Trajectory[] simulatedTrajectories = new Trajectory[k];
6    // Verteile die Simulation auf numThreads SimulatorJobs
7    for (int i = 0; i < numThreads; ++i) {
8      int kLocal = (i < k % numThreads) ? (k / numThreads + 1)
9          : (k / numThreads);
10     executor.execute(new SimulatorJob(simulatedTrajectories, done,
11         start, kLocal, i));
12   }
13   // Warte auf Beendigung der Simulationen
14   try {
15     done.await();
16   } catch (InterruptedException e) {
17     e.printStackTrace();
18   }
19   return simulatedTrajectories;
20 }
```

Listing 7.1: Methode `simulate` der Klasse `ParallelSimulator` (Auszug)

```
1  public void run() {
2    for (int i = 0; i < kLocal; ++i) {
3      // Iterator, der die Simulation der Transitionen durchführt
4      TransitionIterator it = mdp.simulationIterator(policy, start);
5      // Simuliere eine Trajektorie
6      Trajectory trajectory = new Trajectory();
7      while (it.hasNext())
8        trajectory.add(it.next());
9      simulatedTrajectories[i * coreCount + pos] = trajectory;
10   }
11   // Melde Fertigstellung
12   done.countDown();
13 }
```

Listing 7.2: Methode `run` der Klasse `SimulatorJob` (Auszug)

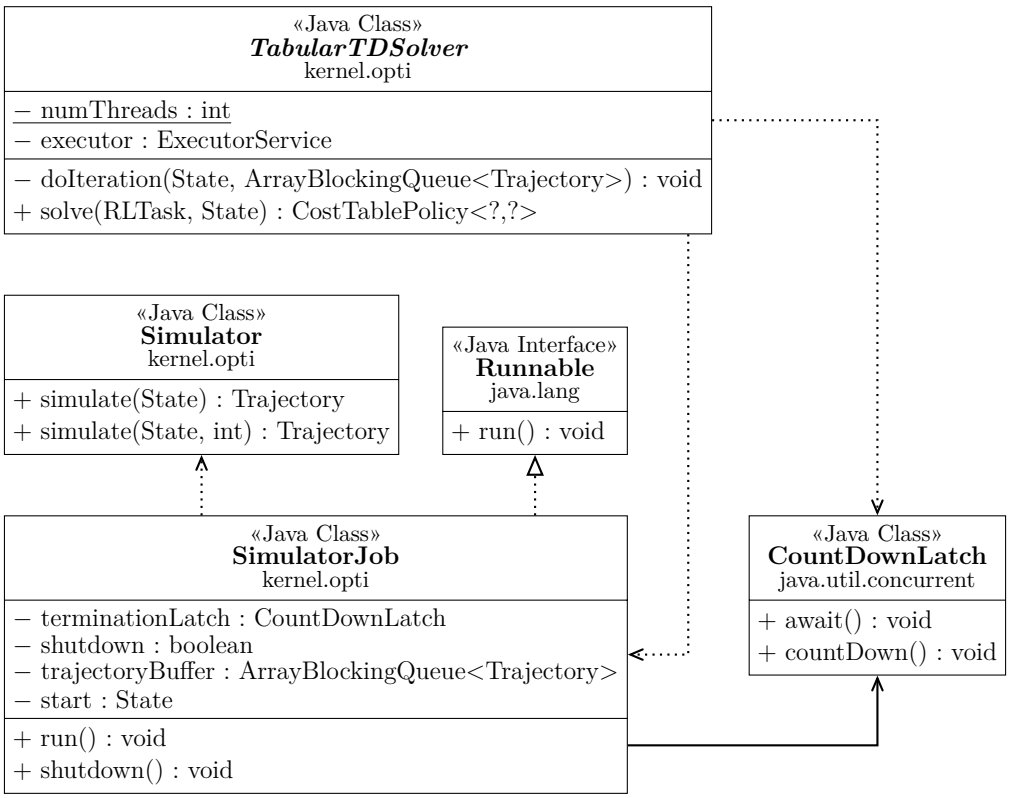


Abbildung 7.3.: An der parallelen Simulation beteiligte Klassen

Bisher bestand der Ablauf der Methode `solve` von `TabularTDSolver` aus abwechselnden Simulations- und Predictionphasen. Dabei wurde für die Simulation jeweils eine neue `Simulator`-Instanz erzeugt und dort die Trajektorien angefordert. Bei dieser Variante des Parallelisierungsansatzes sollen aber Simulation und Prediction nebenläufig erfolgen. Dazu startet der `TabularTDSolver` nun zu Beginn der Optimierung mehrere Simulator-Threads einer Klasse `SimulatorJob`. In einem synchronisierten Puffer („Trajektorienablage“), realisiert durch eine `ArrayBlockingQueue`, werden die fertigen Trajektorien durch die Simulator-Threads gespeichert. `TabularTDSolver` führt wie bisher wiederholt Iterationen des TD-Algorithmus' mittels der Methode `doIteration` durch; im Unterschied zur sequenziellen Version entnimmt der Solver die k Trajektorien nun jedoch direkt der Trajektorienablage, anstatt sie selbst zu simulieren. Sind keine k Trajektorien vorhanden, blockiert der Solver so lange, bis wieder Trajektorien im Puffer verfügbar sind. Der Code von `doIteration` ist in Listing 7.3 dargestellt. Alle beschriebenen Änderungen nahmen wir der Einfachheit halber direkt in der `TabularTDSolver` Klasse vor. Eine Koexistenz der umgestalteten Klasse mit ihrer bisherigen sequenziellen Version wäre aber problemlos möglich.

Ist die Optimierung beendet, müssen auch die laufenden Simulatoren terminiert werden. Dazu besitzt jedes `SimulatorJob`-Objekt ein Feld `shutdown`. Wird dieses durch den Solver auf `true` gesetzt, terminiert der `SimulatorJob` nach Beendigung seiner aktuellen Simulation. Da manche der Simulatoren möglicherweise noch auf das Ablegen einer

```
1  private void doIteration(State start,
2      ArrayBlockingQueue<Trajectory> trajectoryBuffer) {
3    Trajectory[] simulatedTrajectories = new Trajectory[k];
4    // Wähle ein Epsilon für die Simulation (
5    simulationPolicy.setEpsilon(chooseEpsilon(iteration));
6    // Entnehme dem Puffer k Trajektorien
7    for (int i = 0; i < k; ++i)
8      try {
9        simulatedTrajectories[i] = trajectoryBuffer.take();
10     } catch (InterruptedException e) {
11       e.printStackTrace();
12     }
13   // Predictionphase
14   predictor.improvePrediction(simulatedTrajectories);
15 }
```

Listing 7.3: Methode `doIteration` der Klasse `TabularTDSolver` (Auszug)

Trajektorie im (vollen) Puffer warten, wird dieser geleert. Weil unser Puffer mehr Speicherplätze besitzt, als es Simulatoren gibt, kann jeder Simulator seine letzte Trajektorie ablegen und es kommt zu keinem Deadlock.

Die Klasse `SimulatorJob` implementiert wie bei der ersten Variante das `Runnable` Interface. Der Code ihrer Methode `run()` ist in Listing 7.4 dargestellt. Die Simulationen werden mit einer herkömmlichen `Simulator` Instanz durchgeführt und die Trajektorien im gemeinsamen Puffer `trajectoryBuffer` gespeichert. Der Vorgang wird wiederholt, bis vom Solver das Feld `shutdown` gesetzt wurde.

```
1  public void run() {
2    while (!shutdown) {
3      // Periodisches Neu-Seeden des Zufallsgenerators
4      if (simulationCount % 1000 == 0)
5        ((ExclusiveRandomThread) Thread.currentThread()).reseed();
6      // Simuliere eine Trajektorie und schreibe sie in den Puffer
7      try {
8        trajectoryBuffer.put((new Simulator(mdp, simulationPolicy).
             simulate(start)));
9      } catch (InterruptedException e) {
10       e.printStackTrace();
11     }
12     ++simulationCount;
13   }
14   terminationLatch.countDown();
15 }
```

Listing 7.4: Methode `run` der Klasse `SimulatorJob` (Auszug)

Bei diesem Ansatz greifen gleichzeitig die Simulatoren lesend und der Predictor schreibend auf die Kostentabelle zu. Auch wenn nur ein Thread schreibt, kann es ohne

Synchronisation zu Fehlern kommen, beispielsweise während einer internen Vergrößerung der für die Kostentabelle genutzten Hashmap. Aus diesem Grund haben wir die `HashMap` durch eine `ConcurrentHashMap` ersetzt.

7.4. Mehrere Lerner mit gemeinsamer Kostentabelle

Für die Version des Algorithmus' mit mehreren Agenten erweiterten wir die Optimierungssoftware um eine Klasse `ParallelTabularTDSolver`. Um sie neben dem sequenziellen `TabularTDSolver` austauschbar verwenden zu können, war auch eine neue abstrakte Oberklasse `AbstractTabularTDSolver` (siehe auch Abb. 7.4) notwendig.

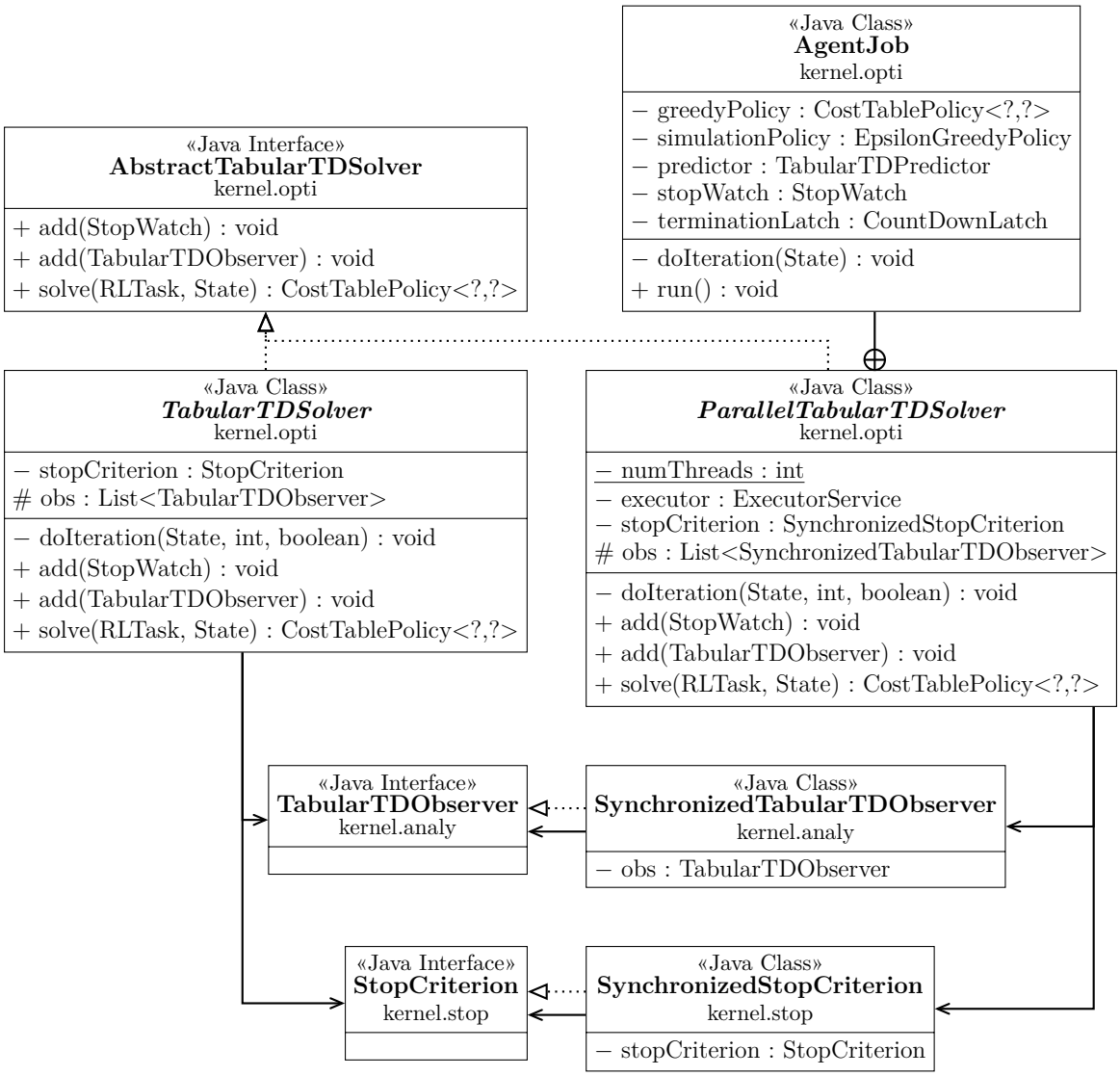


Abbildung 7.4.: Integration der sequenziellen und parallelen Version des Solvers

Analog zur parallelen Simulation nutzten wir auch für diesen Ansatz eine `Runnable` implementierende innere Klasse `AgentJob`, die den eigentlichen Code ausführt. Der `ParallelTabularTDSolver` selbst erzeugt nur die zentrale Kostentabelle und die einzelnen Agenten mit ihren Policys und startet diese, wie Abbildung 7.5 verdeutlicht.

Ein `AgentJob` führt Iterationen des TD-Algorithmus aus. Der Ablauf der Iterationen (Abb. 7.6) unterscheidet sich dabei nicht von der sequenziellen Version. Jeder Agent bestizt eine eigene greedy und Simulations-Policy (in der Regel ist dies eine ϵ-greedy Policy), was den Agenten auch erlaubt, verschiedene Policys zu nutzen. Es ist noch zu erwähnen, dass die Benutzung des Stopkriteriums gemeinsam mit der evenutellen Benachrichtigung der Observer atomar erfolgen muss, da sich durch die Benachrichtigung der Wert des Stopkriteriums ändern kann. Dieser Sachverhalt ist in Abbildung 7.6 durch eine entsprechende Umrahmung der beiden Schritte gekennzeichnet.

Zentralisierung und Synchronisation der Kostentabelle Bisher erlaubte die Software nicht die Benutzung einer geteilten Kostentabelle. Die verschiedenen Klassen die den Predictionschritt implementieren (`TabularTDLambda` & `TabularSarsaLambda`) erzeugten ihre Kostentabelle selbst. Ein neuer Konstruktor für diese Klassen zur Übergabe einer initialisierten Kostentabelle löste das Problem.

Wie im vorangeganenen Kapitel erklärt, muss für die Kostentabelle gegenseitiger Ausschluss bei der Modifikation der Cost-to-go-Werte sichergestellt werden. Dazu legten wir eine neue, threadsichere Version `ConcurrentCountingQTable` der Klasse `CountingQTable` an. `CountingQTable` ist eine von zahlreichen Kostentabellen und speichert neben Kosten für Zustand-Aktions-Paare zusätzlich die Zahl der Updates für jeden Eintrag. Da an einigen Stellen der Software `CountingQTable` direkt verwendet wird, wurde wiederum zusätzlich eine abstrakte Basisklasse `AbstractCountingQTable` angelegt. Abbildung 7.7 zeigt einen Ausschnitt aus der Hierarchie der verschiedenen Implementierungen des `CostTable` Interfaces.

`ConcurrentCountingQTable` nutzt intern eine `ConcurrentHashMap` zur Speicherung der Abbildung von Zustand-Aktions-Paaren auf Cost-to-go-Werte, wodurch die meisten Operationen bereits auf eine effiziente Weise threadsicher implementiert sind. Die in der Analyse angesproche Methode `updateBy(double)` (S.20) für relative Updates von Einträgen wird jedoch nicht a priori von `ConcurrentHashMap` unterstützt. Da es sich um eine zusammengesetzte Operation handelt, mussten wir sie manuell synchronisieren.

Listing 7.5 zeigt unsere Umsetzung der Operation am Beispiel einer der anderen Kostentabellen[1]. Das Muster für eine effiziente Implementierung einer solchen zusammengesetzten Operation auf einer JAVA `HashMap` stammt aus [LSS09] & [Map]. Die Schwierigkeit ist, dass ein Eintrag für einen Zustand (bzw. ein Zustand-Aktions-Paar) gegebenenfalls erst angelegt werden muss, wenn er noch nicht vorhanden ist.

Zunächst wird der Schlüssel `s` in der Hashmap nachgeschlagen. Falls `null` der Rückgabewert ist, war zum Zeitpunkt des `get` Aufrufs noch kein Eintrag vorhanden. Durch

[1] `ConcurrentCountingQTable` enthält eine gesonderte Hashmap für die Aktionen des Startzustands, um deren Kosten schneller abrufen zu können. Dadurch ist `updateBy(double)` für diese Klasse etwas komplizierter.

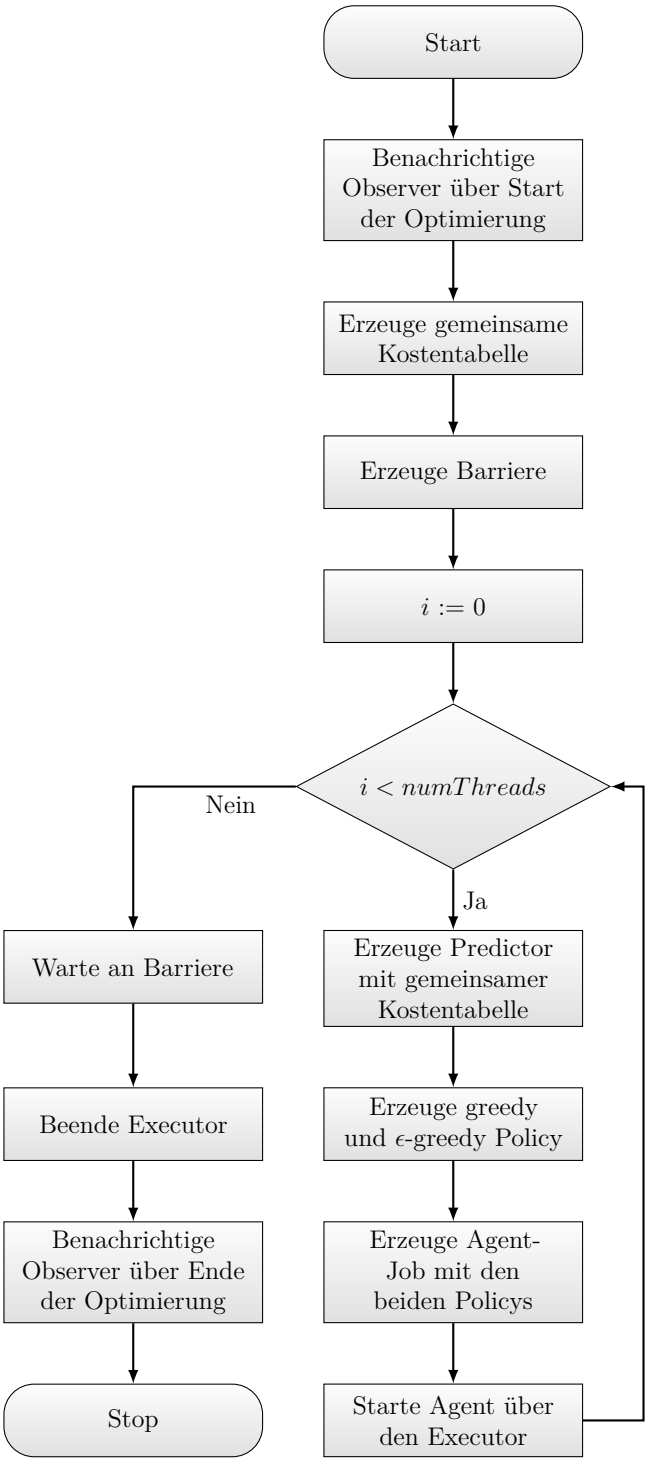


Abbildung 7.5.: Programmablaufplan für die Methode `solve` der Klasse `ParallelTabularTDSolver`

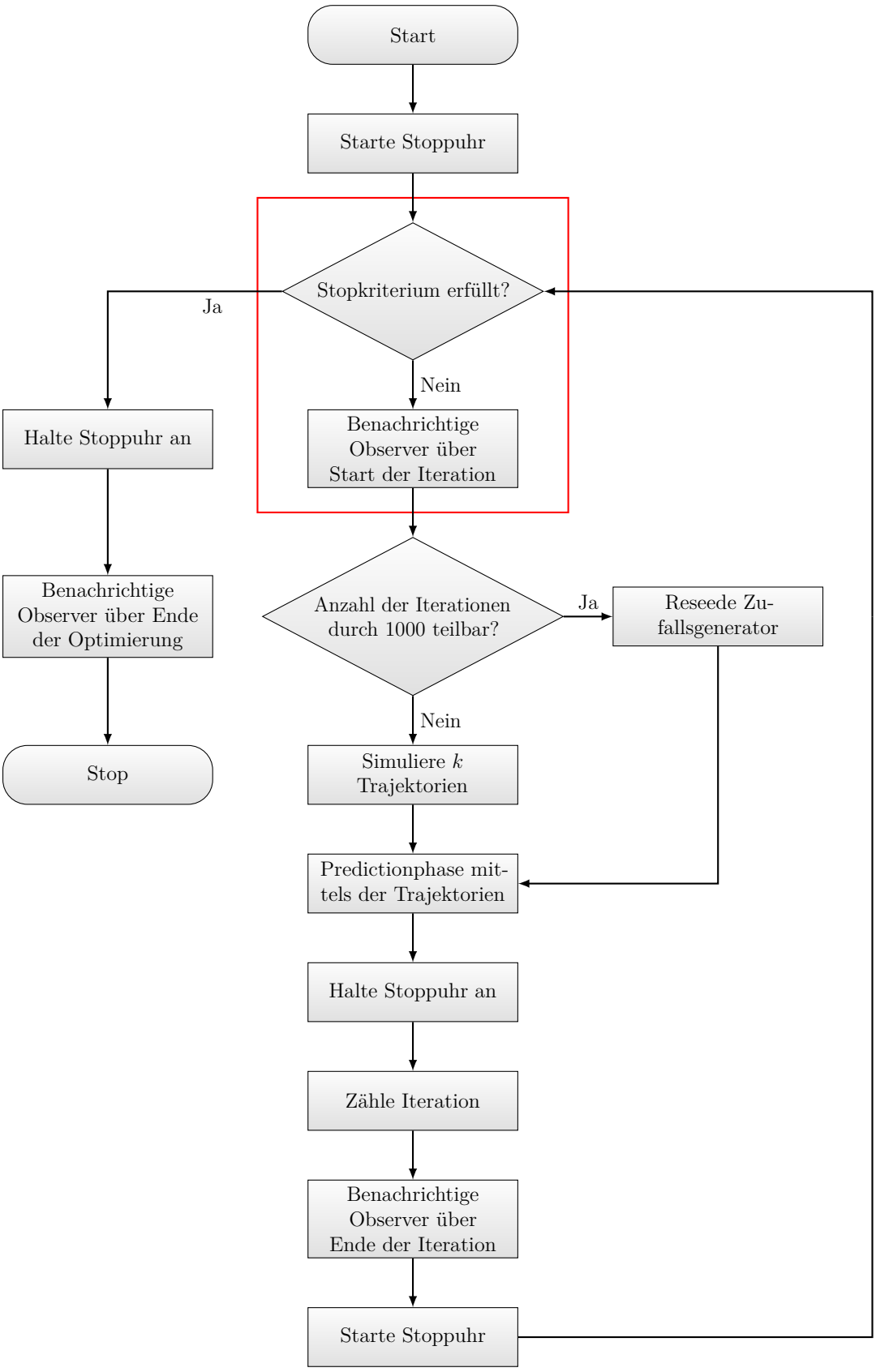


Abbildung 7.6.: Programmablaufplan für die Methode `run` der inneren Klasse `AgentJob` von `ParallelTabularTDSolver`

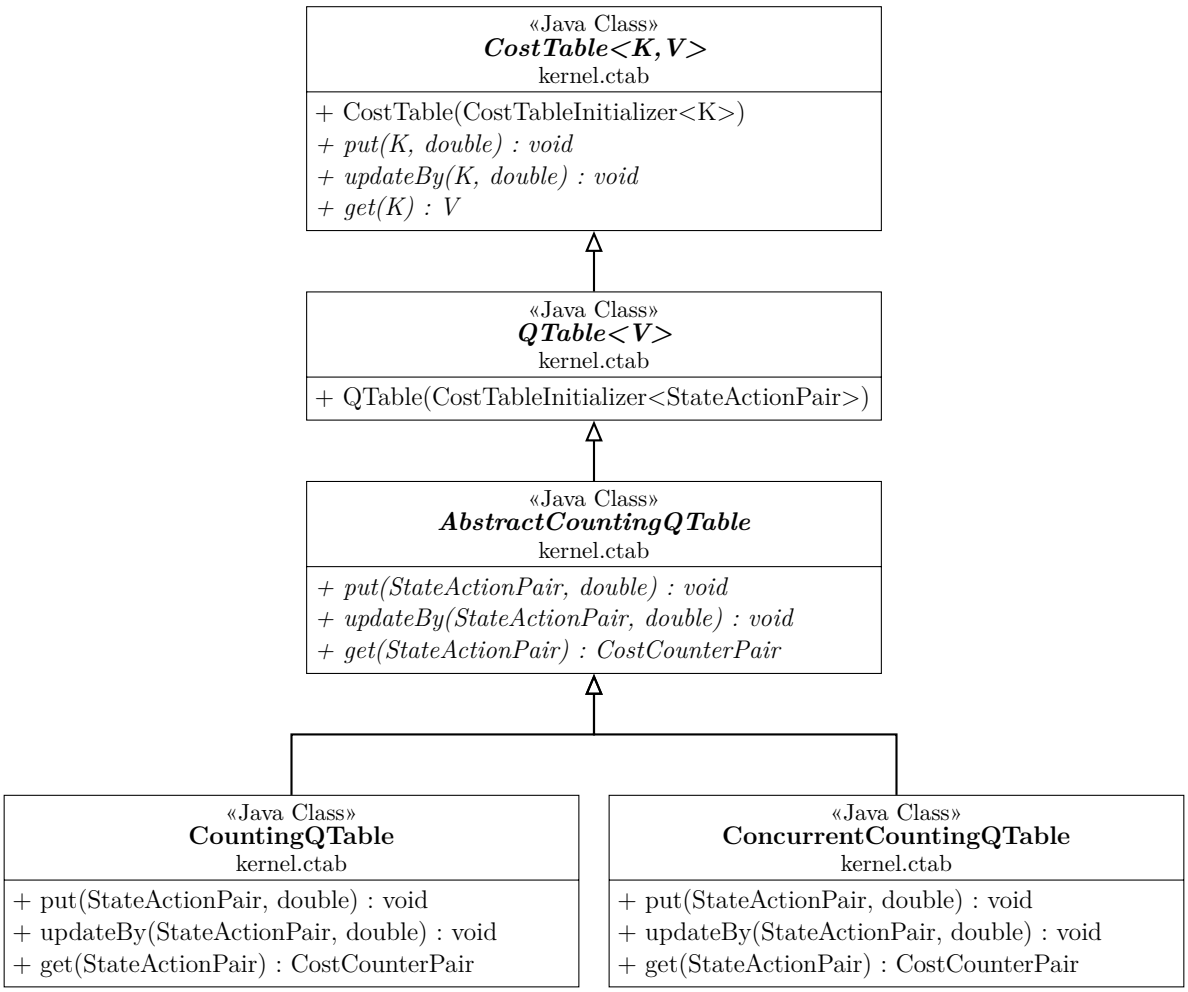


Abbildung 7.7.: Integration der neuen Kostentabelle `ConcurrentCountingQTable`

`putIfAbsent` wird versucht, einen Eintrag (threadsicher) hinzuzufügen. Nach dem Aufruf enthält `value` den Wert für `s` unmittelbar vor diesem Aufruf. Dieser Wert kann auch ungleich `null` sein, falls zwischenzeitlich ein anderer Thread einen Eintrag für `s` erzeugt hat. In diesem Fall oder falls von vornherein ein Eintrag vorhanden war, muss das Update noch angewendet werden. Dafür wird das von der `value` referenzierte Objekt, das nun definitiv den Eintrag für `s` darstellt, gesperrt und aktualisiert.

Synchronisation der Observer des Algorithmus' In der Optimierungssoftware finden zahlreiche Klassen Verwendung, die nach dem Observer-Pattern [Gam+94, S.293] arbeiten, insbesondere die Stopkriterien des TD-Algorithmus. Observer werden u.a. vor und nach jeder Iteration des Algorithmus' benachrichtigt. Beispielsweise zählt die Klasse `NumberOfIterationsCriterion` die durchgeführten Iterationen und stoppt die Optimierung nach einer bestimmten Anzahl. Andere Observer wiederum führen periodisch eine Evaluation der aktuellen Policy durch (Klasse `PeriodicEvaluator`), etc.

Wir nutzen für die verschiedenen Agenten der parallelen Implementierung die sel-

```
1  public void updateBy(State s, double delta) {
2    Cost value = map.get(s);
3    if (value == null) {
4      value = map.putIfAbsent(s, new Cost(delta));
5    }
6    if (value != null) {
7      synchronized (value) {
8        value.updateCost(value.cost() + delta);
9      }
10   }
11 }
```

Listing 7.5: Methode `updateBy` der Klasse `ConcurrentSimpleCostToGoTable`

ben Observer-Instanzen. Dadurch wird eine entsprechende Synchronisation der Observer nötig. Unerwünschte Effekte bei fehlender Threadsicherheit wären beispielsweise durch das `NumberOfIterationsCriterion` nicht mitgezählte Iterationen oder eine stetige Änderung der Policy während ihrer Evaluation durch einen der Agenten in einem `PeriodicEvaluator`.

Für die vorhandenen Observer war es ausreichend, sie mit einer neuen Wrapperklasse `SynchronizedTabularTDObserver` für das Observer-Interface `TabularTDObserver` zu umschließen (Abb. 7.8). Der Wrapper, dessen Methoden durch das `synchronized` Schlüsselwort geschützt sind, delegiert alle Methodenaufrufe an eine unsynchronisierte `TabularTDObserver` Instanz. Nach dem gleichen Muster wurde auch ein Wrapper für das, von den Stopkriterien implementierte, Interface `StopCriterion` hinzugefügt (Abb. 7.9). Abbildung 7.10 zeigt noch einmal einen Gesamtüberblick über den Zusammenhang von Observern und Stopkriterien.

Insbesondere im Hinblick auf unsere Experimente ist noch der sogenannte `StopWatch` Observer gesondert zu nennen, mit dem der reine Zeitverbrauch des TD-Algorithmus ohne die in den Observern verbrachte Zeit gemessen wird. Der Gebrauch dieser Stoppuhr wird in Abbildung 7.6 deutlich. Da verschiedene Agenten zu unterschiedlichen Zeitpunkten TD-Iteration bzw. Observerbenachrichtigungen durchführen, ergibt eine geteilte `StopWatch` Instanz keinen Sinn. Jedem Agent weisen wir daher eine eigene Stoppuhr zu. Für die Zeitmessung bei mehreren Agenten erscheint es uns legitim, einen beliebigen der Agenten für die Zeitmessung auszuwählen, da sich in der Praxis die mit den TD-Iterationen verbrachte Zeit zwischen den Agenten kaum unterscheidet. Nichtsdestotrotz haben wir für die Experimente mit dieser Implementierung jeweils den Zeitverbrauch zwischen den Agenten gemittelt.

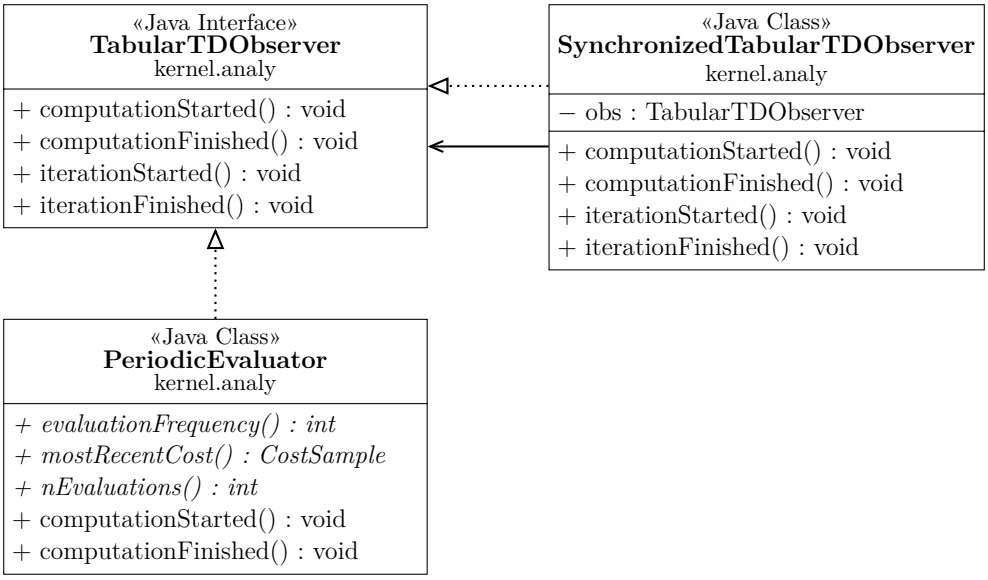


Abbildung 7.8.: Das Interface für alle Observer der verschiedenen Temporal Difference Algorithmen `TabularTDObserver` und die Wrapperklasse `SynchronizedTabularTDObserver`

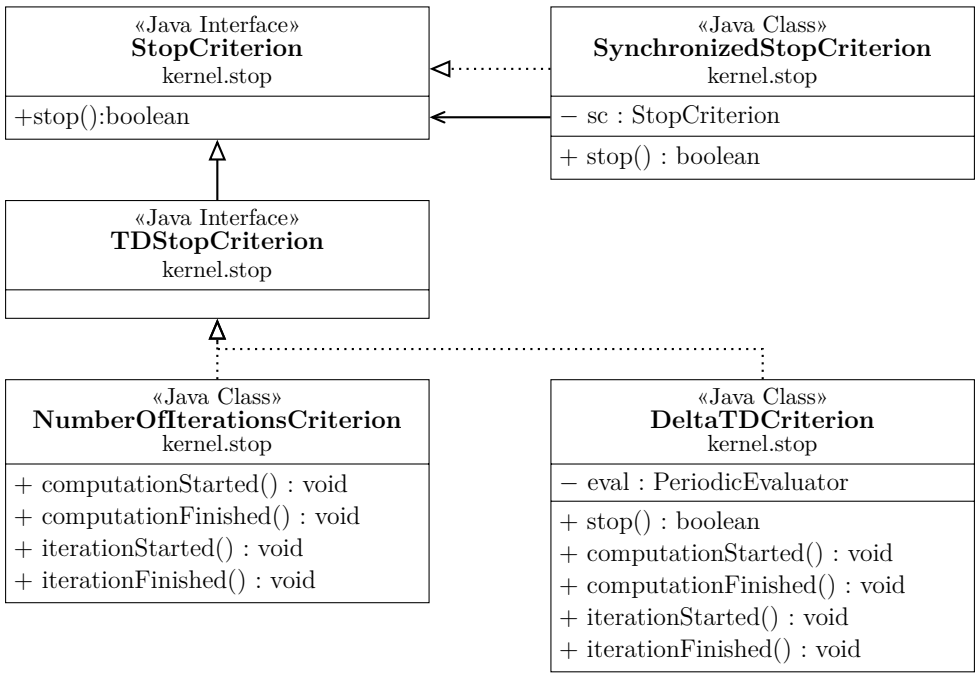


Abbildung 7.9.: Das Interface für die verschiedenen Stopkriterien `StopCriterion` und sein Wrapper `SynchronizedStopCriterion`

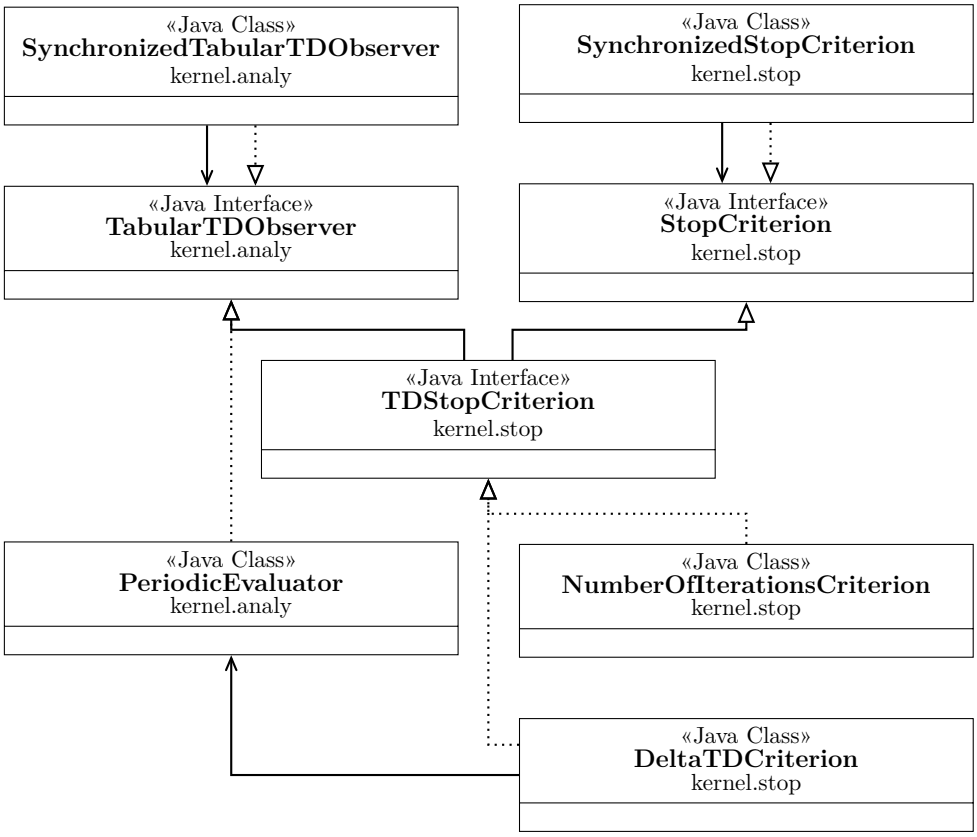


Abbildung 7.10.: Die Stopkriterien für die Temporal Difference Algorithmen (`TDStopCriterion`) müssen den Verlauf des Algorithmus mitverfolgen und erben daher auch die Eigenschaften der Observer

7.5. Testen der Parallelisierungen

Um mehr Sicherheit über die Korrektheit unserer Implementierungen zu gewinnen, testen wir diese auf verschiedenen Ebenen: Einerseits stellten wir auf höherer Ebene sicher, dass die parallelen Versionen der Software in der Lage sind, Policys mit gleichen zu erwartenden Kosten wie die Referenzimplementierung zu berechnen, andererseits prüften wir die durch die Parallelisierungen neu hinzugekommenen Klassen auch mithilfe von Unit-Tests. Teilweise nutzten wir für die Tests einen einfacher als den SW-MDP zu handhabenden Labyrinth-MDP.

Listing 7.6 zeigt beispielhaft einen Test für die erste Variante der parallelen Simulation (7.3.1). Er überprüft, ob die neue `ParallelSimulator` Klasse die korrekte Zahl von Trajektorien simuliert und ob die Trajektorien mit von der sequenziellen `Simulator` Klasse simulierten übereinstimmen. Ein analoger Test wurde auch für die zweite Variante der parallelen Simulation (7.3.2) implementiert. Der aufwendigste Ansatz mit mehreren Agenten bot am meisten Raum für Tests. So testeten wir z.B. die Funktionalität der neuen Kostentabelle und ob das `NumberOfIterationsCriterion` die korrekte Zahl von Iterationen erlaubt.

```
1  @Test
2  public void testSimulateMultiple() {
3    Trajectory expectedTrajectory = simulator.simulate(mdp.
       initialState());
4    Trajectory[] actualTrajectories = parallelSimulator.
       simulateMultiple(mdp.initialState(), 100);
5    for (Trajectory actualTrajectory : actualTrajectories) {
6      assertEquals(expectedTrajectory, actualTrajectory);
7    }
8  }
```

Listing 7.6: Unit-Test für die Methode `simulateMultiple` der Klasse `ParallelSimulator`

8 Experimente

Wir untersuchten sowohl die Leistungssteigerung durch die implementierten Parallelisierungsansätze, als auch mögliche Nebeneffekte, beispielsweise auf die Form der Lernkurven[1] mittels einer Reihe von Experimenten. Dabei interessierte uns auch ein Vergleich mit der sequenziellen Version der Software, ebenso wie ein Vergleich der verschiedenen Parallelisierungsansätze untereinander.

Konkret führten wir Speedupmessungen mit allen drei Ansätzen für eine Reihe von Instanzen des SW-MDP durch und maßen für den Ansatz mit mehreren Agenten zusätzlich die Lernkurven.

8.1. Experiment-Konfiguration

8.1.1. Verwendete Hard- und Software

Alle Experimente führten wir auf einem Acer Aspire M7721 mit Intel Core i7-920 Prozessor und 12GB Hauptspeicher durch. Der i7-920 Prozessor hat eine Taktrate von $4 \times 2{,}67$GHz, wobei die Rechenleistung auf 8 logische Prozessoren mittels Hyper-Threadings [VGD09] verteilt ist. Wir starteten die Optimierungssoftware jeweils aus einer Java Archiv Datei (.jar) heraus in einer unter Windows 7 laufenden Java SE 6 Laufzeitumgebung.

8.1.2. Verwendete Parameterwerte

Für die Parameter des Sarsa(λ) Algorithmus' verwendeten wir durchgehend die selben in Tabelle 8.1 dargestellten Werte. Diese entsprechen weitesgehend den von D. Weiss [Wei10, S.28] ermittelten, für den SW-MDP optimalen Werten. Nur für die Experimente mit der Implementierung von Ansatz 4.3.1 mussten wir die Zahl der Simulationen pro Iteration variieren (siehe S.39). Für noch nicht in der Kostentabelle vorhandene Zustand-Aktions-Paare müssen initiale Werte angenommen werden, hier wählten wir einen Startwert von 0 für alle Zustand-Aktions-Paare.

Wie in Tabelle 8.1 zu sehen ist, nutzten wir für die Schrittweite α keine Konstante, sondern wählten sie umgekehrt proportional zu der Häufigkeit, mit welcher der Wert für ein bestimmtes Zustand-Aktions-Paar in der Kostentabelle bereits aktualisiert wurde.

[1] Mit „Lernkurve“ ist dabei die Entwicklung der zu erwartenden Projektkosten unter der ermittelten Policy mit zunehmender Anzahl von Iterationen des Algorithmus oder Rechenzeit gemeint.

Parameter	Bedeutung	Wert
k	Anzahl der Simulationen pro Iteration	1
λ	Verminderung der Eligibility Trace	0,98
ϵ	Wahrscheinlichkeit für zufällige Aktion	0,09
α	Schrittweite	$\max(\frac{1}{\#Updates}, lowerBound)$

Tabelle 8.1.: Parameterwerte für Sarsa(λ)

Damit die Schrittweite für große Instanzen des SW-MDPs, die sehr viele Iterationen zum Erreichen einer optimalen Policy benötigen, nicht zu schnell abfiel, beschränkten wir die Schrittweite zusätzlich durch eine untere Grenze *lowerBound*. Diese Grenze setzten wir zu Beginn jeweils auf 0,04 und verringerten sie alle 500.000 Iterationen um 0,0035.

8.1.3. Verwendete Instanzen des SW-MDP

Aus der großen Menge möglicher Instanzen, die durch variieren der Anzahl von Komponenten, Teams und des Kopplungsmodells erzeugt werden kann, griffen wir acht mittelgroße bis große Instanzen heraus, die wir bei jeder der Experimentreihen verwendeten. Eine Auflistung und Spezifikation dieser Instanzen findet sich in Anhang A. Die Verteilung der Basiswahrscheinlichkeiten der verschiedenen Komponententypen war die selbe wie bei früheren Experimenten von Padberg [Pad06].

8.1.4. Bestimmung der Projektkosten

Für alle Experimente mussten periodisch die, unter der momentanen greedy Policy zu erwartenden, Projektkosten bestimmt werden. Zwar ist der Wert der greedy Aktion des Anfangszustandes in der Kostentabelle eine Approximation der Projektkosten, er ist aber ungenau, da er in der Regel langsamer kleiner wird als die tatsächlichen Projektkosten [Wei10, S.30]. Daher unterbrachen wir bei unseren Messungen in bestimmten Abständen den TD-Algorithmus und führten sogenannte Monte-Carlo Evaluationen [SB98, S.112ff] durch. Genauer simulierten wir jeweils 200.000 Trajektorien unter der aktuellen Policy und bildeten den Durchschnitt der erhaltenen Projektkosten.

Der Zeitaufwand für diese zusätzlichen Evaluationen ist hoch, je nach Häufigkeit der Evaluationen macht er ein Vielfaches der für den TD-Algorithmus selbst aufgewendeten Zeit aus. Da uns nur die Beschleunigung des TD-Algorithmus' selbst interessierte, wurde der Aufwand für die Monte-Carlo Evaluationen bei den Zeitmessungen nicht berücksichtigt.

8.2. Speedupmessungen

8.2.1. Durchführung der Speedupmessungen

Mithilfe der Speedupmessungen möchten wir die Frage beantworten, ob und wenn ja, wieviel schneller die parallelen Versionen der Software, bei Einsatz einer bestimmten Anzahl n von Threads, im Vergleich zur Originalsoftware sind. Dazu musste die benötigte Rechenzeit für die gleiche verrichtete „Arbeit“ verglichen werden.

Die Anzahl der Iterationen des Algorithmus' ist kein geeignetes Maß für diese Arbeit, da bei jedem der Parallelisierungsansätze Abweichungen bei den zu erwartenden Projektkosten nach gleicher Zahl von Iterationen im Vergleich zur Referenzimplementierung möglich sind.

Gleiche Arbeit sollte also gleiche Qualität, d.h. gleiche zu erwartende Projektkosten der berechneten Policy bedeuten. Daher nutzten wir für die Messungen ein „Delta Kriterium“ (Klasse `DeltaTDCriterion`). Dieses Stopkriterium beobachtet die aktuellen Projektkosten und den Durchschnitt der Kostenstichproben aus den letzten n Evaluationen. Weichen beide Werte um weniger als einen vorgegebenen Wert δ voneinander ab und sind die aktuellen Projektkosten auch höchstens δ größer als das bisherige Minimum aus allen Evaluationen, wird die Optimierung beendet.

8.2.2. Mehrere Simulationskerne, Variante 1 (7.3.1)

Für diese Experimentgruppe mussten wir einen relativ großen Wert für k wählen, um eine Beschleunigung erzielen zu können. Wir entschieden uns für zwei Messreihen mit $k = 24$ und $k = 48$. Bei diesen Werten fallen genug Simulationen am Stück an, sie beeinflussen die Geschwindigkeit, mit welcher der Algorithmus konvergiert, aber noch nicht zu sehr negativ [Wei10, S.24]. Alle 250.000 Iterationen bestimmten wir durch Monte-Carlo Evaluation die Projektkosten.

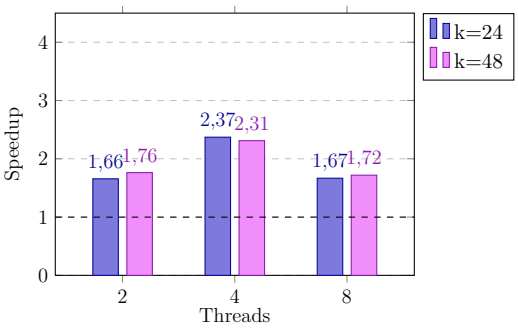
8.2.2.1. Ergebnisse der Messung

Die Resultate der Messungen sind in Abbildung 8.1 dargestellt, dabei bezieht sich die Anzahl der Threads jeweils auf die während der Simulationsphasen genutzten Simulatoren.

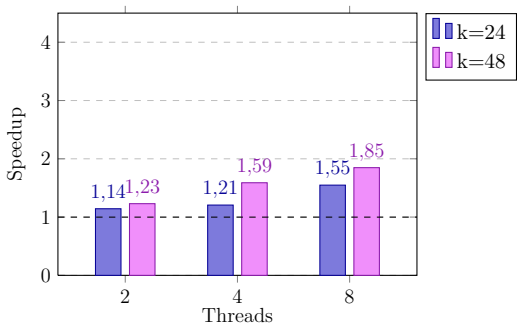
Zunächst stellen wir fest, dass durchgehend Speedup-Werte > 1 erzielt wurden, die sich im Bereich zwischen 1,14 (*10:3 MIN*, Abb. 8.1b) und 2,88 (*11:3 MIN*, Abb. 8.1e) bewegten. Für jede Instanz konnten wir die Berechnungen, bei Nutzung der richtigen Zahl von Threads, jedoch um mindestens 85% beschleunigen. Die parallele Version war also stets schneller als die Referenzimplementierung.

Der Einsatz von vier Simulationskernen lohnte sich immer gegenüber dem Einsatz von nur zweien. Acht Threads führten erst bei den größeren Instanzen *11:3 ASYM* und *12:3 UNI* zu einem deutlichen Gewinn, bei den restlichen waren sie im Vergleich vier Threads in den meisten Fällen leicht unterlegen.

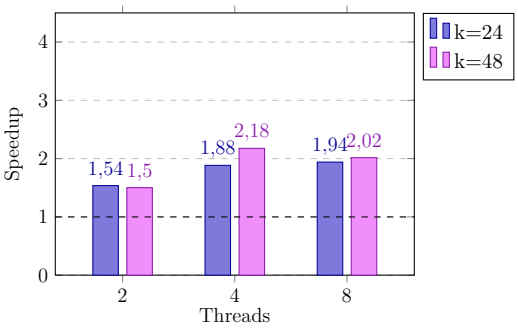
Der Beschleunigungsfaktor schwankte von Instanz zu Instanz stark, auffallend hoch fiel er bei *8:3 MAX* und *11:3 MIN* aus. Insgesamt scheint er mit der Zahl der Komponenten



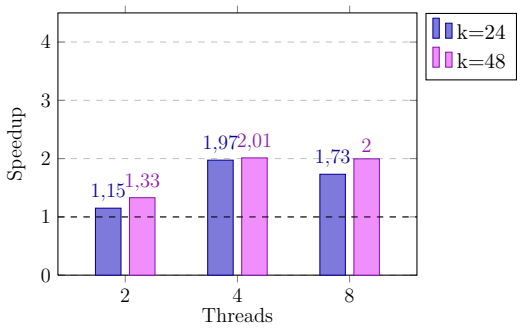
(a) 8:3 MAX



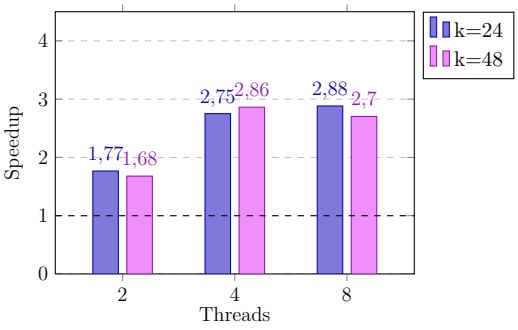
(b) 10:3 MIN



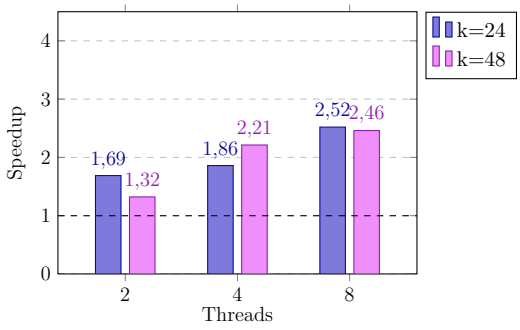
(c) 10:3 MAX



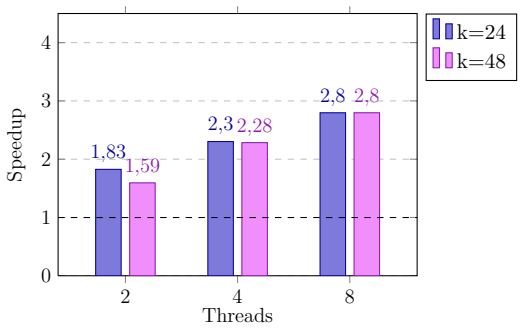
(d) 10:3 UNI



(e) 11:3 MIN



(f) 11:3 ASYM



(g) 12:3 UNI

Abbildung 8.1.: Vergleich des Speedups durch Ansatz 7.3.1 für einige Instanzen

zu wachsen, wie der Vergleich von *10:3 MIN* und *11:3 MIN* oder *10:3 UNI* und *12:3 UNI* nahelegt.

Zumindest für die beiden getesteten Werte $k = 24$ und $k = 48$ scheint der Unterschied in der Anzahl der Simulationen pro Iteration keinen nennenswerten Einfluss auf den Speedup zu haben. Es sind keine Unterschiede sichtbar, die so groß sind, dass sie nicht durch die natürlichen Schwankungen der Lerngeschwindigkeit zustande gekommen sein könnten. Die durchschnittliche absolute Abweichung zwischen den Kosten für die beiden Parameterwerte betrug nur 0,15. Insbesondere aus den Werten für die mit Abstand größte Instanz *12:3 UNI* (Abb. 8.1g) mit der daher vermutlich höchsten Messgenauigkeit ziehen wir den Schluss, dass der Wert von k für die Beschleunigung bei großen Instanzen keine Rolle spielt.

8.2.2.2. Interpretation der Ergebnisse

Da nur die Simulationen parallel erfolgen, hängt die Effektivität der Parallelisierung davon ab, wie groß der Anteil der Simulationen an der Rechenzeit ist. Der Aufwand für die Simulation ist aus vielen Gründen größer als der für die Prediction und wächst mit der Instanzgröße auch schneller. Beispielsweise erfolgen bei der Wahl der greedy Aktion während der Simulation jeweils zahlreiche Zugriffe auf die Kostentabelle; in der Prediction müssen jedoch nurnoch die Kosten für die tatsächlich gewählte Aktion abgerufen werden. Abgesehen von den Zugriffen auf die Kostentabelle muss in der Predictionphase pro Transition der Trajektorie die Temporal Difference (S.5) berechnet werden; der Aufwand dafür ist unabhängig von der Instanz. Die Simulation einer Transition hingegen wird mit zunehmender Zahl an Komponenten immer aufwendiger, beispielsweise nimmt die Zahl möglicher Aktionen zu und bei der Berechnung des Folgezustandes müssen die zusätzlichen Komponenten ebenfalls berücksichtigt werden. Mit der Instanzgröße steigt also der Anteil der Simulationszeit und gleichzeitig das Potenzial zur Beschleunigung.

Etwas überraschend ist für uns, dass die absolute Dauer der Simulationsphasen, die durch größere Werte für k direkt verlängert werden kann, keine sichtbaren Auswirkungen auf die Beschleunigung hat. Bei längeren Simulationsphasen fallen der konstante Mehraufwand durch die Threadverwaltung und die absoluten Unterschiede in der Zeit, welche die einzelnen Simulatorkerne zur Berechnung ihrer Trajektorien benötigen, relativ weniger ins Gewicht. Da sich von $k = 24$ zu $k = 48$ aber keine eindeutige Änderung zeigt, vermuten wir, dass beide Effekte bei $k = 24$ bereits gering sind.

Daher lassen sich auch für alle der getesteten Instanzen vier Threads gegenüber nur zweien gewinnbringend einsetzen. Hyper-Threading ist nur effektiv, wenn die einzelnen Threads, beispielsweise aufgrund von Cache-Misses, einen Prozessor nicht voll auslasten können. Da die Wahrscheinlichkeit von Wartezeiten vor allem mit der Größe der Kostentabelle steigt, lohnt sich der Einsatz von acht Threads für größere Instanzen zunehmend.

8.2.3. Mehrere Simulationskerne, Variante 2 (7.3.1)

Für diese Messreihe führten wir die Monte-Carlo Evaluationen alle 200.000 Iterationen durch.

8.2.3.1. Ergebnisse der Messung

Abbildung 8.2 zeigt die Ergebnisse aus dieser Messreihe. Die angegebene Anzahl von Threads bezieht sich auf die zur Simulation genutzten Threads, die Prediction lief jeweils in einem zusätzlichen Thread. Wir nahmen also Messungen unter Nutzung von insgesamt drei, vier und acht Threads vor. Im Gegensatz zu den anderen beiden Speedupmessungen führten wir die erste Messung also mit mehr als zwei Threads durch, da sonst nur ein Thread für die Simulation verblieben wäre.

Wiederum konnten wir eine annähernd optimale Policy mit der parallelen Implementierung stets schneller bestimmen als mit der sequenziellen Software. Der Beschleunigungsfaktor lag zwischen 1,61 (*8:3 MAX*, Abb. 8.2a) und 3,84 (*12:3 UNI*, Abb. 8.2g). Unter Einsatz von entsprechend vielen Simulatoren war es uns sogar möglich, jede Instanz mehr als doppelt so schnell zu optimieren.

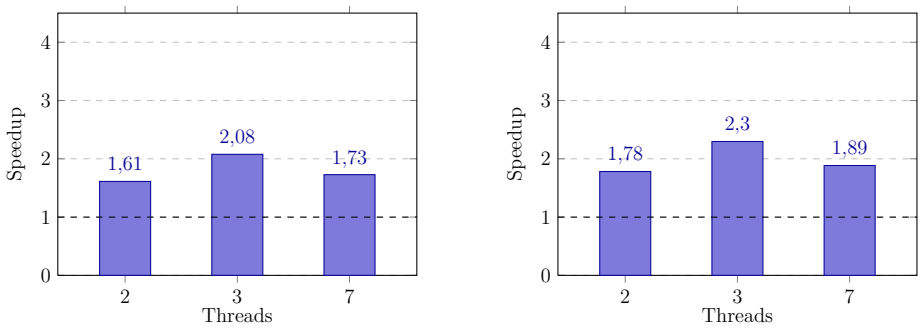
Dabei erreichten drei Simulatoren für alle der gemessenen Instanzen eine bessere Beschleunigung als zwei, der Gebrauch von sieben Threads zur Simulation lohnte sich erst für die größeren Instanzen in bedeutender Weise.

Je größer die Instanz, desto besser fiel auch im Schnitt der Speedup durch die Parallelisierung aus. Beispielsweise entwickelt sich die größte erreichte Beschleunigung für *8:3 MAX*, *10:3 MAX* und *12:3 UNI* von 2,08 über 2,38 zu 3,84. Die Entwicklung lässt sich jedoch nicht für jede Zahl von Threads zwischen jedem Paar von Instanzen nachvollziehen. So stellt z.B. die *10:3 UNI* Instanz (Abb. 8.2d) mit Speedups von 1,8, 2,12 und 2,24 eine Ausnahme dar: Für jede Zahl von Threads fiel hier der Speedup etwas schlechter aus als für die etwa kleinere *10:3 MAX* Instanz (Abb. 8.2c), bei der Beschleunigungen von 1,9 und zweimal 2,38 erreicht wurden.

8.2.3.2. Interpretation der Ergebnisse

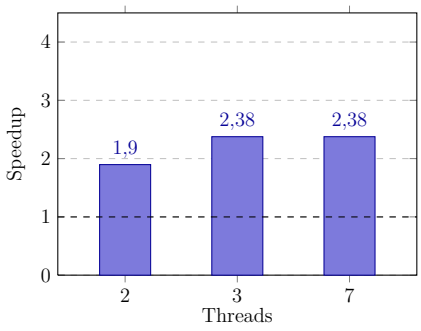
Aufgrund des Schemas aus Produzenten (Simulatoren) und Konsument (Predictor) bestimmt die relative Geschwindigkeit der beiden Seiten die Effizienz dieses Parallelisierungsansatzes. Wie bereits im vorangegangenen Abschnitt beschrieben (S.41), ist die Simulation einer Trajektorie deutlich aufwendiger als ihre Verwendung in der Prediction. Die bessere Effektivität von drei gegenüber zwei Threads ist mit der schnelleren Bereitstellung von Trajektorien für den Predictor durch den zusätzlichen Simulator zu erklären.

Als Grund dafür, dass sieben Simulatoren für *8:3 MAX* und *10:3 MIN* schlechter abschneiden als drei, könnte man vermuten, dass der Predictionschritt in diesen Fällen zum Flaschenhals wird und die Trajektorien nicht schnell genug abarbeiten kann. Eine Profileranalyse zeigte jedoch, dass die Auslastung des Predictors für diese Instanzen auch bei sieben Threads noch relativ gering ist. Für *10:3 MIN* war der Prediction-Thread beispielsweise nur 39% der Laufzeit aktiv, wie Abbildung 8.3 zeigt. Daher erklären

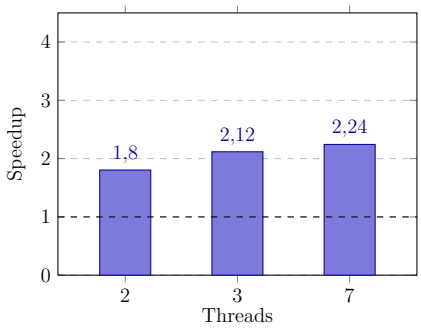


(a) 8:3 MAX

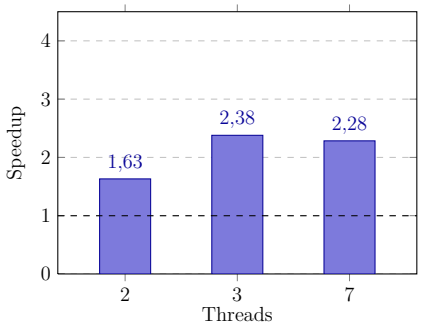
(b) 10:3 MIN



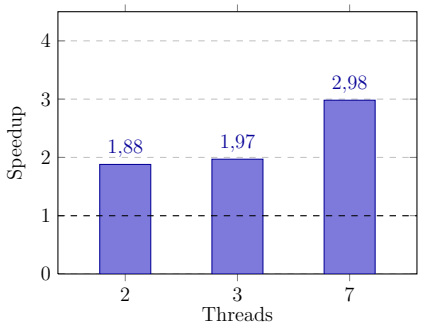
(c) 10:3 MAX



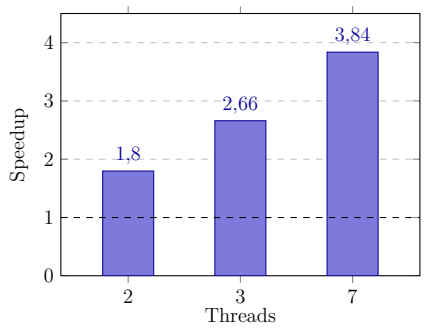
(d) 10:3 UNI



(e) 11:3 MIN



(f) 11:3 ASYM



(g) 12:3 UNI

Abbildung 8.2.: Vergleich des Speedups durch Ansatz 7.3.2 für einige Instanzen

wir die schlechtere Leistung für sieben Simulatoren mit einer schlechten Effizienz des Hyper-Threading bei kleineren Instanzen.

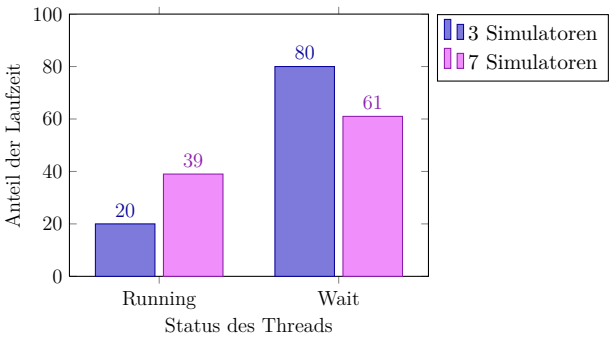


Abbildung 8.3.: Anteile an der Laufzeit die der Prediction-Thread in verschiedenen Zuständen verbringt, gemessen für die 10:3 MIN Instanz.

Dass die Entwicklung des Speedups mit zunehmender Instanzgröße nicht für alle Instanzen eindeutig abzulesen ist, liegt vermutlich schlicht daran, dass die Instanzen teilweise sehr nah beieinander liegen und somit bei dem geringen Stichprobenumfang, den wir verwendeten, Messungenauigkeiten ausreichen um ein klares Bild zu verzerren.

8.2.3.3. Vergleich mit Variante 1 des Ansatzes

Im Vergleich mit der ersten Variante der parallelen Simulation, erkennen wir bei Variante 2 eine meist bessere Nutzung des Multicores. Vor allem bei den großen Instanzen wurden die Unterschiede deutlich. Ausnahmen sind *8:3 MAX* und *11:3 MIN*, für welche Variante 1 besonders gut funktionierte.

Für die bessere Auslastung sehen wir zwei Gründe: Erstens ensteht bei Variante 2 praktisch kein Mehraufwand durch die Threadverwaltung, da die Simulatoren nur einmal zu Beginn der Optimierung gestartet werden müssen. Zweitens können die Simulatoren auch während der Prediction weiterarbeiten, solange in der Trajektorienablage noch Platz ist. Dadurch ist der parallelisierte Teil der Rechenzeit größer als bei Variante 1.

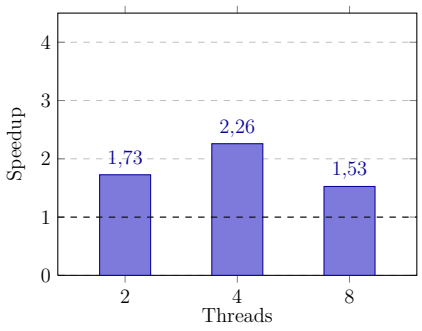
8.2.4. Mehrere Lerner mit gemeinsamer Kostentabelle (7.4)

Für diesen Ansatz evaluierten wir die aktuelle Policy alle 100.000 bis 150.000 Iterationen, je nach Anzahl der nötigen Iterationen zur Berechnung einer annähernd optimalen Strategie.

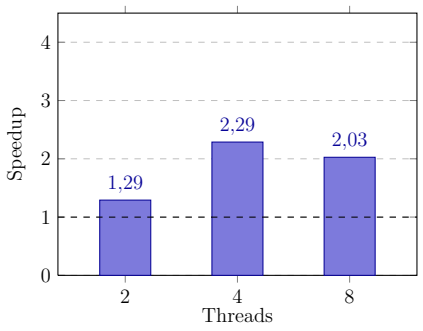
8.2.4.1. Ergebnisse der Messung

Die Ergebnisse der Speedupmessungen sind in Abbildung 8.4 dargestellt; jeder Thread entspricht einem Agenten.

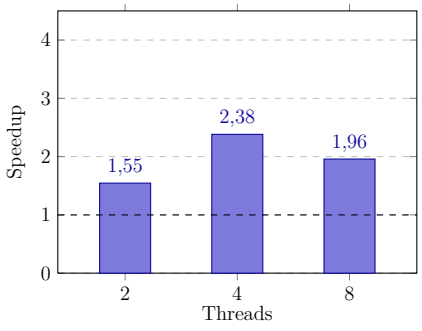
Auch diese Parallelisierung führte bei jeder Messung zu einer Beschleunigung. Während der Speedup für zwei Lerner meist moderat ausfiel und durchschnittlich nur 50% betrug,



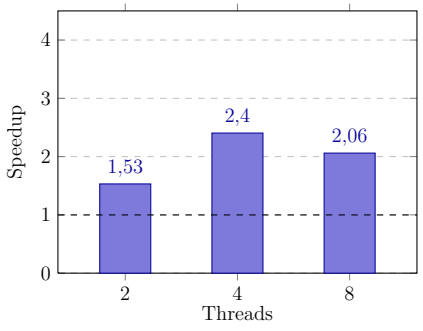
(a) 8:3 MAX



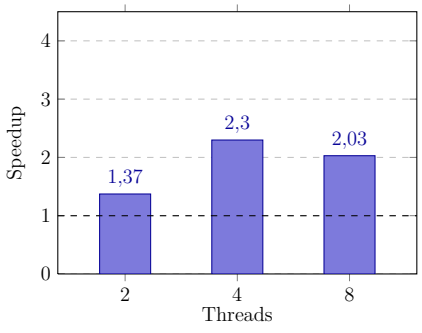
(b) 10:3 MIN



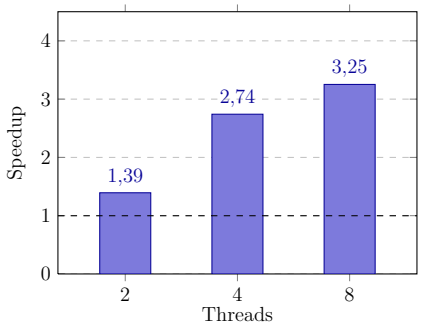
(c) 10:3 MAX



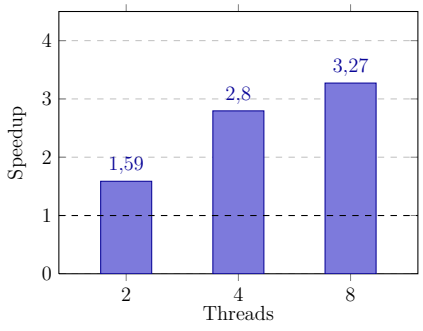
(d) 10:3 UNI



(e) 11:3 MIN



(f) 11:3 ASYM



(g) 12:3 UNI

Abbildung 8.4.: Vergleich des Speedups durch Ansatz 7.4 für einige Instanzen

brachten vier Lerner jeweils eine deutliche Verbesserung mit einer Beschleunigung auf mindestens 2,26-fache Geschwindigkeit. Wie bei den vorangegangenen Ansätzen lohnte sich der Einsatz von 8 Threads auch hier erst für die beiden größten Instanzen *11:3 ASYM* (Abb. 8.4f) und *12:3 UNI* (Abb. 8.4g) deutlich.

Für zwei Agenten lässt sich kein klarer Trend bei zunehmender Instanzgröße ableiten. Beispielsweise wird für die *8:3 MAX* Instanz eine Beschleunigung um den Faktor 1,73 erreicht, während er bei der wesentlich größeren *11:3 ASYM* Instanz nur 1,39 beträgt. Bei vier bzw. acht Lernern nahm der Speedup mit der Instanzgröße hingegen deutlicher zu, die größten Werte lagen hier bei 2,8 bzw. 3,27.

8.2.4.2. Interpretation der Ergebnisse

Prinzipiell können mehr Agenten auch mehr Iterationen des Algorithmus' in der gleichen Zeit durchführen. Dafür, dass der Geschwindigkeitszuwachs nicht das theoretische Maximum erreicht, sind zwei Erklärungen denkbar: Eine geringere Effektivität der durch einen einzelnen Agenten durchgeführten Iterationen aufgrund von redundantem Lernen eines anderen Agenten, wie sie bereits in früherer Forschung beschrieben wurde [FE06, S.2] und Mehraufwand durch die Synchronisation gemeinsam genutzter Ressourcen, vor allem der gemeinsamen Kostentabelle. Ersteres scheint für den SW-MDP allerdings keine große Rolle zu spielen, wie im nächsten Abschnitt erläutert wird (S.48). Dadurch sind lange Wartezeiten bei Benutzung der gemeinsamen Kostentabelle als Ursache am wahrscheinlichsten.

Zwar sind die Zustandsräume der Instanzen sehr groß, jedoch beginnt jede der relativ kurzen Trajektorien im gleichen Startzustand, sodass die Zustand-Aktions-Paare für diesen Zustand stark frequentiert sind. Hinzu kommt ein genereller Overhead durch die aufwendigere Implementierung von `ConcurrentHashMap` gegenüber `HashMap` auch in den Fällen, in denen Zugriffe auf die Kostentabelle gar nicht blockieren.

Mit wachsender Größe des Zustandsraums wird es seltener, dass mehrere Lerner gleichzeitig den Wert des selben Zustand-Aktions-Paars bearbeiten. Daraus resultiert die Zunahme des Speedups für vier und acht Agenten. Umgekehrt werden solche Kollisionen mit zunehmender Zahl der Lerner häufiger, wodurch acht Threads für kleinere Instanzen ineffektiv sind. Da sich für zwei Agenten kein Trend ablesen lässt, kommt es vermutlich schon bei den kleineren Instanzen hier kaum zu gleichzeitigen Zugriffen.

8.2.4.3. Vergleich mit dem ersten Ansatz

Verglichen mit der besseren der beiden Varianten der parallelen Simulation, sind mehrere Agenten leicht unterlegen. Wie begründen dies damit, dass bei der parallelen Simulation zwar auch viele nebenläufige Zugriffe auf die Kostentabelle stattfinden, wobei jedoch jeweils nur einer der Threads schreibt. Die Simulatoren lesen nur aus der Kostentabelle und blockieren nicht. Im Gegensatz dazu greift jeder Agent gleichermaßen schreibend auf die Kostentabelle zu – die Threads befinden sich häufiger im Wartezustand.

8.2.4.4. Vorübergehende Probleme mit mehreren Agenten

Bei der Durchführung der Messreihe stießen wir zunächst auf zwei Probleme mit unserer Implementierung: Zum einen unterlagen die Projektkosten bei einer zunehmenden Zahl von Agenten immer stärkeren Schwankungen, was den automatischen Abbruch der Messungen durch das `DeltaTDCriterion` teilweise unmöglich machte und auch die manuelle Feststellung der Konvergenz erschwerte, zum anderen schien es zu plötzlichen Geschwindigkeitseinbrüchen der Software zu kommen.

Durch Ausprobieren verschiedener Werte für Parameter des Sarsa(λ) Algorithmus fanden wir heraus, dass bei zunehmender Zahl von Agenten immer kleinere Werte für die untere Grenze der Schrittweite α eine deutliche Verminderung der Schwankungen brachten. Aufgrund fehlender Synchronisierung fiel die Schrittweite nicht bei allen Lernern gleichermaßen ab. Infolge der längere Zeit hohen Schrittweite, unterlagen die Kosten der einzelnen Zustand-Aktions-Paare damit auch länger entsprechend starken Veränderungen. Nachdem der Fehler behoben war, traten kaum noch Schwankungen auf.

Die Geschwindigkeitseinbrüche waren auf einen Fehler in der Klasse `DeltaTDCriterion` zurückzuführen, durch den nicht automatisch alle Agenten terminierten, sobald das Kriterium für einen von ihnen erfüllt worden war. Wir modifizierten den Code so, dass das Kriterium nun, nachdem es einmalig erfüllt wurde, bei allen nachfolgenden Aufrufen automatisch `true` zurückliefert.

8.2.5. Probleme mit dem Stopkriterium

Bei der Durchführung der Experimente stellte sich als Problem heraus, dass oftmals Schwankungen der, zu diesem Zeitpunkt praktisch konvergierten, Projektkosten die Terminierung durch `DeltaTDCriterion` verzögerten. Seltener kam es auch zu frühzeitigen Abbrüchen der Optimierung. Um daraus resultierende Ungenauigkeiten auszuschließen, führten wir jede Messung mit zwei verschiedenen Startwerten für den Zufallsgenerator durch, stellten die Zeitpunkte der Konvergenz manuell fest und bildeten den Durchschnitt der beiden erhaltenen Dauern. Zu früh abgebrochene Optimierungen wurden verworfen. Abbildung 8.5 zeigt die Vorgehensweise an einem Beispiel.

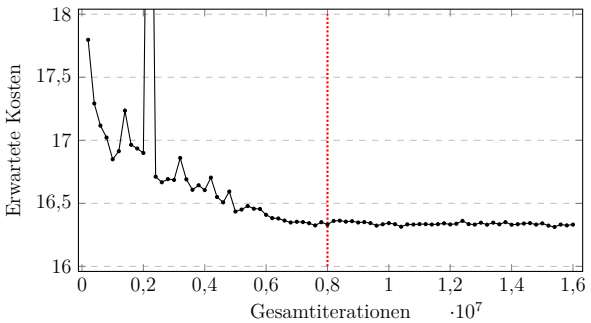


Abbildung 8.5.: Lernkurve der Referenzimplementierung für *10:3 MAX*. Manuell wurden 8 Millionen Iterationen als Meßpunkt ausgewählt (rote Linie).

8.3. Form der Lernkurve bei mehreren Agenten

8.3.1. Bestimmung der Lernkurven

Abhängig von der jeweiligen Instanzgröße, nahmen wir alle 100.000 oder 150.000 Iterationen, über einen Zeitraum von 10 bzw. 15 Mio. Iterationen, eine Stichprobe der Projektkosten mittels Monte-Carlo Evaluation. Den Verlauf der Kosten stellten wir für jede Instanz einmal mit den Gesamtiterationen – also der Summe der Iterationen der einzelnen Agenten – und einmal mit der vergangenen Zeit auf der horizontalen Achse dar. Erstere Darstellung soll eventuelle Auswirkungen auf die Effektivität der einzelnen Iterationen der Lerner aufdecken, die nach früheren Forschungsergebnissen zu erwarten waren [FE06, S.2], letzteres veranschaulicht noch einmal die Performancegewinne.

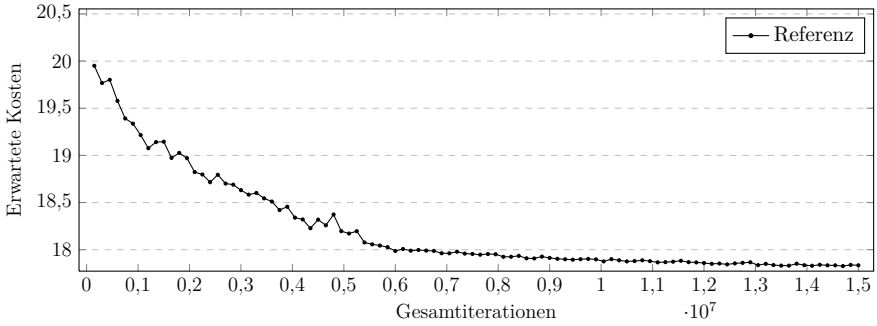
8.3.2. Ergebnisse der Messung

Aus Platzgründen haben wir nur vier beispielhafte Kurven für *12:3 UNI* in einzelnen Diagrammen dargestellt (Abb. 8.6). Die Kurven für die restlichen Instanzen sind jeweils übereinandergelegt (Abb. 8.7 & 8.8).

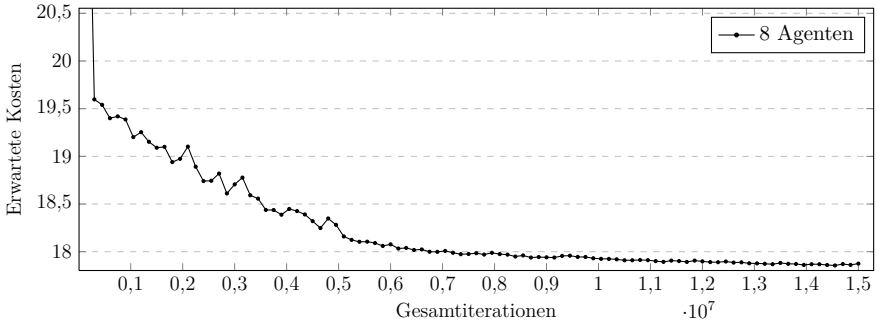
Bei Betrachtung der nach Gesamtiterationen aufgetragenen Lernkurven (Abb. 8.7) können wir praktisch keine Unterschiede zwischen den einzelnen Kurven für eine Instanz feststellen. Unabhängig von ihrer Anzahl scheinen die Agenten die Policy mit gleicher Effektivität zu verbessern.

Die nach der vergangener Rechenzeit skalierten Lernkurven (Abb. 8.8) verdeutlichen die erzielte Beschleunigung. Bei manchen Instanzen, für die die Projektkosten sehr schnell konvergieren (z.B. *11:3 MIN*), lässt sich der Vorteil durch die Parallelisierung weniger gut erkennen, da die Kurven nur in dem hier sehr kleinen Bereich der stärksten Kostenänderungen deutlich auseinanderliegen. Für die Instanzen mit stärkerer Kopplung (vor allem *8:3 MAX*), bei denen die Kosten anfangs stark schwanken, ist der Vorteil durch die Parallelisierung erst im hinteren Teil der Kurven zu erkennen.

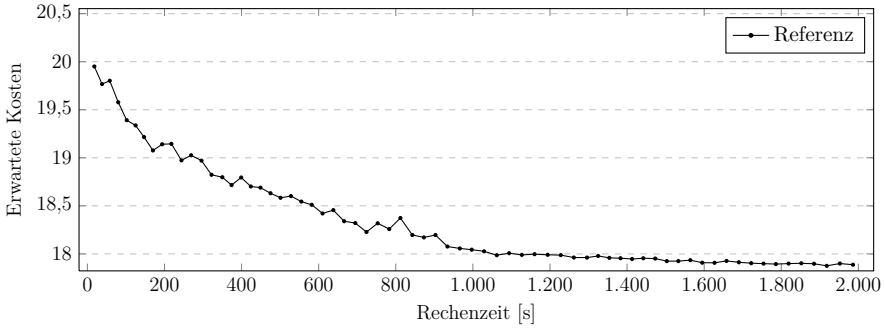
Nach den Ergebnissen von Fields [FE06, S.2] hatten wir sichtbare Abweichungen der Lernkurven erwartet. Diese hätten aufgrund von durch mehrere Agenten gleichzeitig gemachten, überlappenden Erfahrungen entstehen können. Dass keine Abweichungen erkennbar sind, liegt wohl an der Größe des Zustandsraums, ebenso wie an dem schnellen Aktualisieren der Kostentabelle: Durch die große Menge von Zuständen ist es unwahrscheinlich, dass zwei Agenten zur selben Zeit Trajektorien mit einer großen Menge von gemeinsamen Zuständen bearbeiten. Und auch wenn wir offline Updates (S.5) verwenden, verzichten wir auf weitere Verzögerungen der Updates durch Caches, die in der bisherigen Forschung oft eingesetzt wurden.



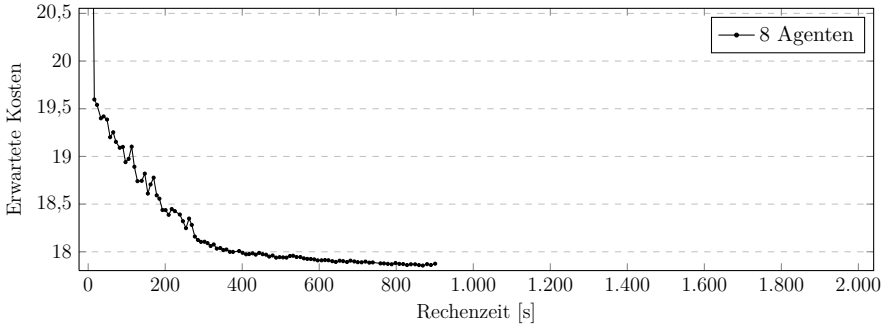
(a) Lernkurve der Referenzimplementierung nach Iterationen



(b) Lernkurve von acht Agenten nach (Gesamt-)Iterationen

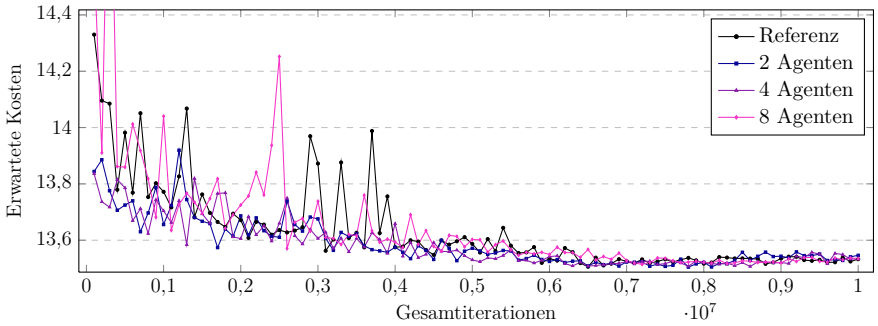


(c) Lernkurve der Referenzimplementierung nach vergangener Zeit

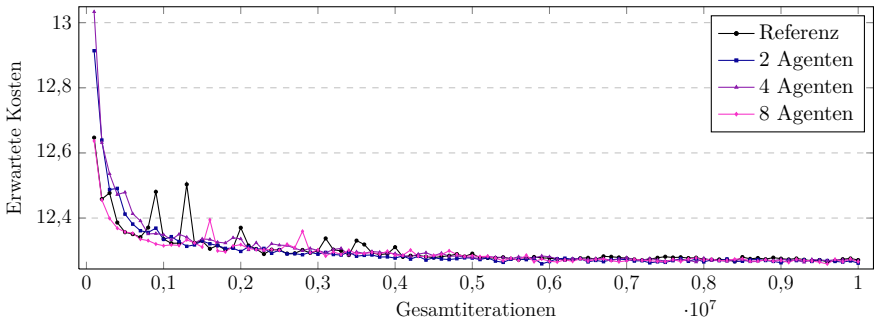


(d) Lernkurve von acht Agenten nach vergangener Zeit

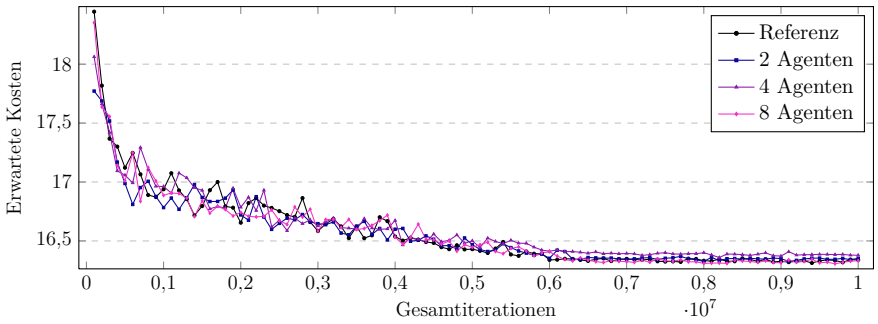
Abbildung 8.6.: Lernkurven für *12:3 UNI*



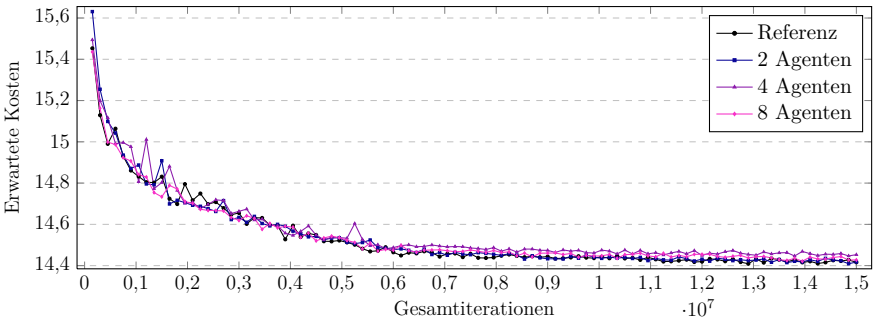
(a) 8:3 MAX



(b) 10:3 MIN

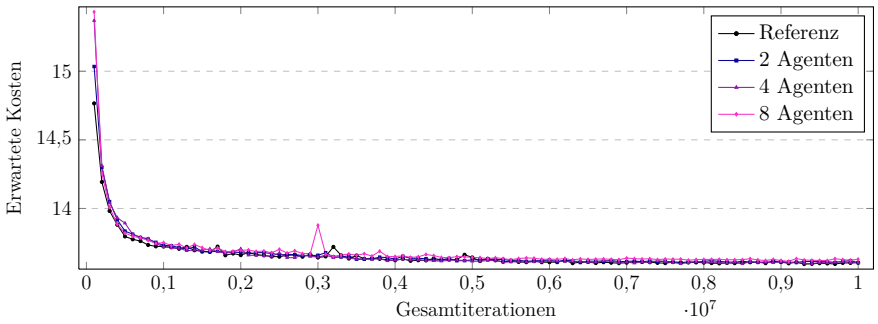


(c) 10:3 MAX

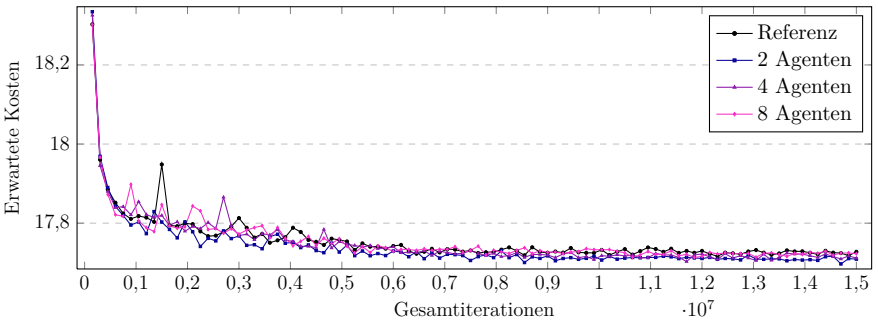


(d) 10:3 UNI

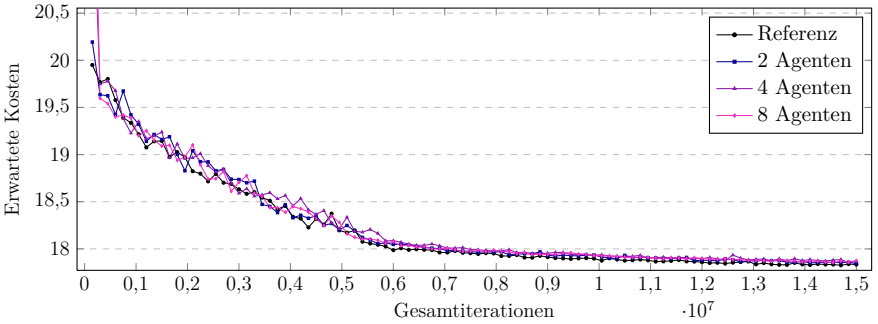
Abbildung 8.7.: Lernkurven für ausgewählte Instanzen, horizontal ist die Gesamtanzahl der durchgeführten Iterationen aufgetragen



(e) 11:3 MIN

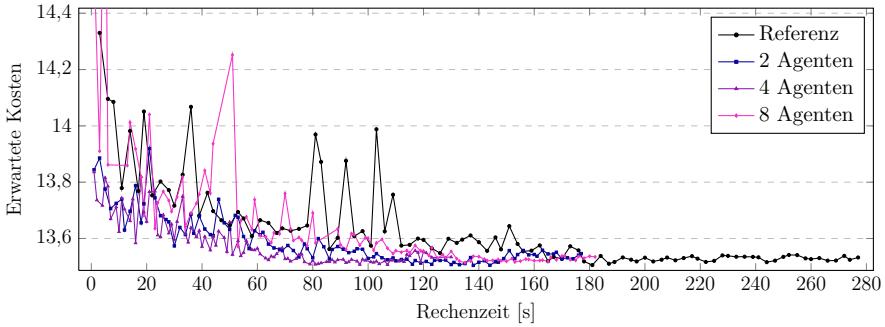


(f) 11:3 ASYM

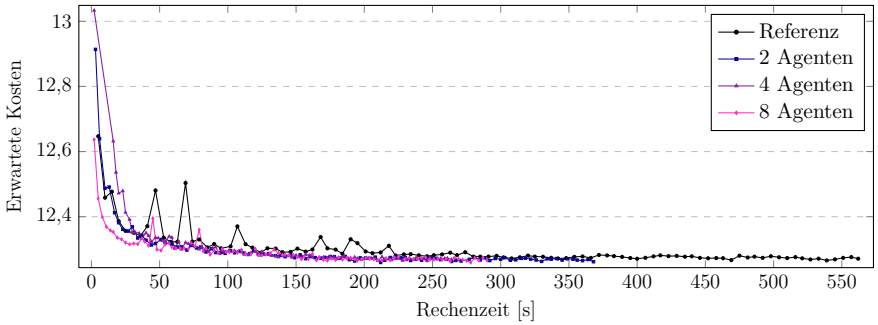


(g) 12:3 UNI

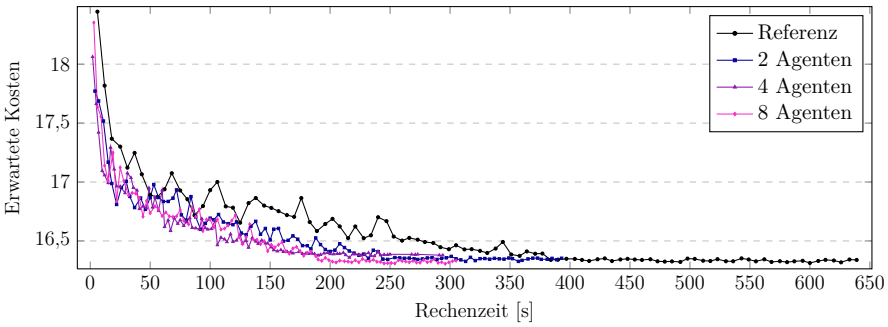
Abbildung 8.7.: Lernkurven für ausgewählte Instanzen, horizontal ist die Gesamtanzahl der durchgeführten Iterationen aufgetragen



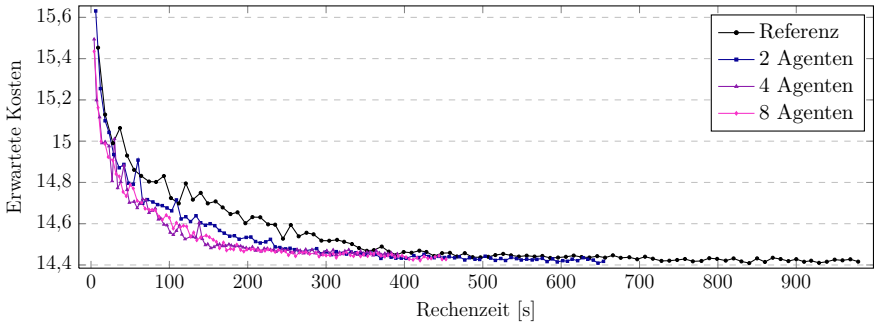
(a) 8:3 MAX



(b) 10:3 MIN

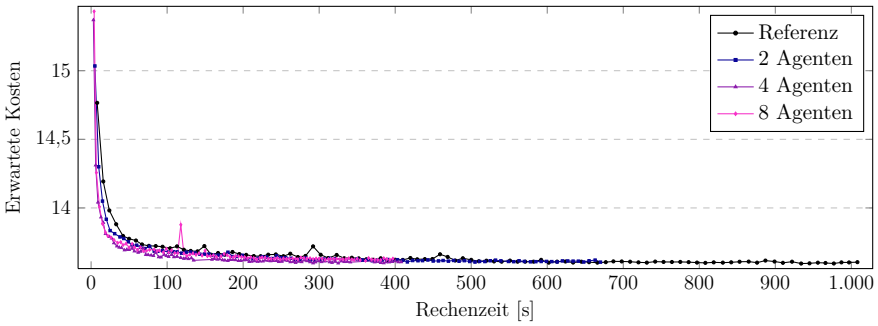


(c) 10:3 MAX

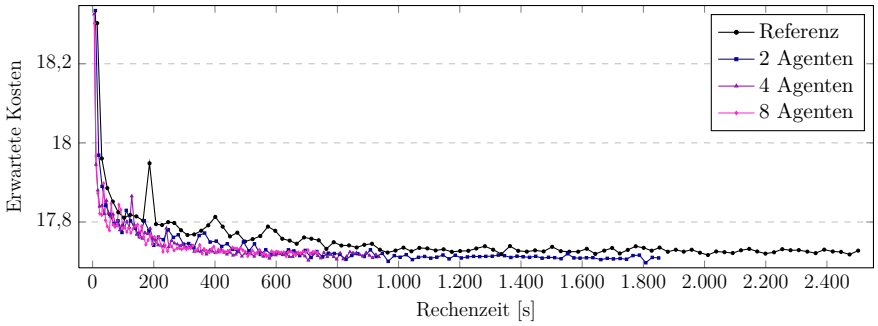


(d) 10:3 UNI

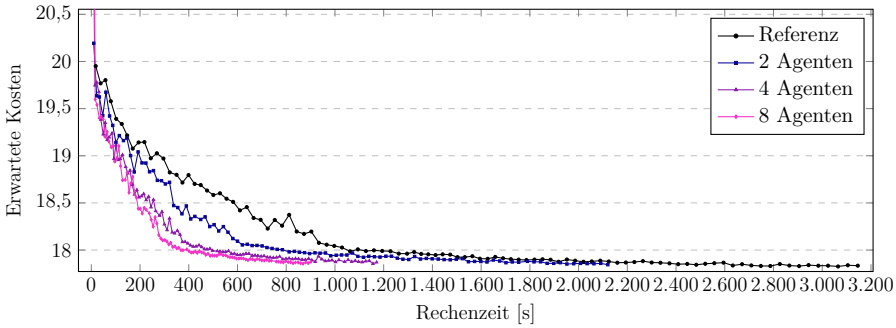
Abbildung 8.8.: Lernkurven für ausgewählte Instanzen, horizontal ist die benötigte Rechenzeit des TD-Algorithmus aufgetragen



(e) 11:3 MIN



(f) 11:3 ASYM



(g) 12:3 UNI

Abbildung 8.8.: Lernkurven für ausgewählte Instanzen, horizontal ist die benötigte Rechenzeit des TD-Algorithmus aufgetragen

9 Ausblick

Auch wenn wir in dieser Arbeit erste Erfolge bei der Parallelisierung von Reinforcement Learning in Anwendung auf den SW-MDP erzielt haben, ist noch Raum für genauere Untersuchungen und Verbesserungen der bestehenden Parallelisierungen sowie für die Implementierung weiterer Parallelisierungsansätze.

9.1. Weitere Experimente

Zunächst sind weitere Experimente zu nennen, auf die wir aufgrund der begrenzten zur Verfügung stehenden Zeit verzichten mussten:

- Dass theoretisch ein optimales Scheduling errechnet werden kann, hilft wenig, wenn in der Praxis nur begrenzte Zeit zur Verfügung steht. Daher wäre eine Untersuchung der *Optimierung unter beschränkten Ressourcen* interessant, um die Frage zu klären ob die parallele Optimierungssoftware nun größere Instanzen als bisher in einem vorgegebenen Zeitraum lösen kann.
- Durch eine höhere Zahl von Durchläufen könnte genauer festgestellt werden, ob sich manche Phänomene, wie aus der Reihe gefallene Speedups, bestätigen. Eventuell ließen sich dadurch weitere Zusammenhänge, beispielsweise zwischen Speedup und Kopplungsmodell, herausstellen.
- Experimente zur *Exploration des Zustandsraums durch mehrere Lerner* wären ebenfalls von Interesse, um genauer zu verstehen, wie sich beim Einsatz mehrerer Lerner diese über den großen Zustandsraum einer SW-MDP Instanz verteilen.

9.2. Weitere Parallelisierungsansätze

Neben den für diese Arbeit tatsächlich implementierten Parallelisierungsansätzen haben wir noch eine Reihe weiterer Ansätze vorgestellt und als für unsere Zwecke geeignet eingestuft (Kapitel 4 & 5). Daher spricht nichts dagegen, sie in der Zukunft weiter zu verfolgen. Auch die Kombination der beiden implementierten Ansätze, die zeitlich nicht mehr möglich war, bleibt weiterhin interessant.

9.3. Verbesserungen an der Optimierungssoftware

Auch die Struktur und Effizienz der Optimierungssoftware bzw. der bestehenden Implementierungen der Ansätze könnte weiter verbessert werden:

- Vor allem bei der Implementierung mehrerer Agenten wurde deutlich, dass die *Architektur der Software* bislang nicht auf Nebenläufigkeit ausgelegt ist. Beispielsweise ist die momentane Verwendung von Observern, bei der sich die Agenten alle Observer teilen, nicht ideal. Zwar funktioniert dies für die vorhandenen Observer, es sind aber auch Observer denkbar, die für jeden Agenten exklusiv sein müssten, z.B. ein Observer zum Messen der Dauer einer Iteration.

- Eventuell ist auch eine weitere *Optimierung des* `DeltaTDCriterion` möglich, mit dem es bei den Speedupmessungen Probleme gab (S.47). Beispielsweise könnte statt des arithmetischen Mittels der Median der letzten Kostenwerte verwendet werden.

- Im Rahmen der Experimente erwähnten wir die Auswertung (S.38) der intermediären Policys. Da diese für eine automatische Terminierung der Optimierung – also in der praktischen Anwendung – zwingend ist, sollte ihr hoher Aufwand nicht außer Acht gelassen werden. Dieser ließe sich durch *Parallelisierung der Monte-Carlo Evaluation* wahrscheinlich sehr gut verringern, da es sich bei der Evaluation um nicht viel mehr als wiederholte Simulation von Trajektorien handelt.

- Für den Ansatz mit mehreren Agenten könnte versucht werden, die *Wartezeit auf gemeinsam genutzte Ressourcen zu verringern*. Womöglich könnte eine eigene Kostentabelle für häufige genutzte Werte pro Agent eine Verbesserung bringen.

- Neben der zeitlichen Dauer einer Optimierung stellt der für die Kostentabelle benötigte Speicherplatz ein noch größeres Problem dar. Dass Instanzen nun schneller gelöst werden können ist nutzlos, wenn der Speicherverbrauch das Lösen von Instanzen realistischer Größe immernoch verhindert. Aus diesem Grund muss über einen geschickteren Umgang mit dem vorhandenen Speicher nachgedacht werden. Beispielsweise wäre eine Auslagerung selten benutzter Werte der Kostentabelle auf die Festplatte denkbar.

9.4. Sonstige weitere Forschung

Abschließend möchten wir noch zwei weitere Punkte anführen:

- Die Frage, ob die durch Parallelisierung modifizierten Algorithmen wirklich noch in jedem Fall gegen eine optimale Strategie konvergieren und stets terminieren, also korrekt sind, wurde von uns außer Acht gelassen. Insbesondere im Falle von mehreren Agenten, wo die Verarbeitung mehrerer Trajektorien mit den resultierenden Veränderungen in der Kostentabelle gleichzeitig erfolgt, muss diese Frage jedoch gestellt werden. Insgesamt beschäftigten sich bisher nur wenige Arbeiten mit

formalen Beweisen der Konvergenz von Algorithmen die mehrere Agenten nutzen [BSP01; San].

- Neben den schon länger handelsüblichen Mehrkernprozessoren, deren Potential wir durch unsere Parallelisierungen nutzbar machten, gewinnen auch Grafikprozessoren immer mehr an Bedeutung, da sie sich zunehmend einfacher auch für allgemeine Berechnungen einsetzen lassen. Es gibt bereits erste Arbeiten zu ihrer Anwendung auf Reinforcement Learning [Pal07]. Daher ist auch eine Anwendung auf den SW-MDP zu prüfen.

A
Instanzen

8:3 MAX

$\mathcal{A}$	$\mathcal{A}$	$\mathcal{B}$	$\mathcal{B}$	$\mathcal{C}$	$\mathcal{C}$	$\mathcal{D}$	$\mathcal{D}$
1	2	3	1	2	3	1	2

10:3 MIN

$\mathcal{A}$	$\mathcal{A}$	$\mathcal{A}$	$\mathcal{B}$	$\mathcal{B}$	$\mathcal{B}$	$\mathcal{C}$	$\mathcal{C}$	$\mathcal{D}$	$\mathcal{D}$
1	2	3	1	2	3	1	2	3	1

10:3 MAX

$\mathcal{A}$	$\mathcal{A}$	$\mathcal{A}$	$\mathcal{B}$	$\mathcal{B}$	$\mathcal{B}$	$\mathcal{C}$	$\mathcal{C}$	$\mathcal{D}$	$\mathcal{D}$
1	2	3	1	2	3	1	2	3	1

10:3 UNI

$\mathcal{A}$	$\mathcal{A}$	$\mathcal{A}$	$\mathcal{B}$	$\mathcal{B}$	$\mathcal{B}$	$\mathcal{C}$	$\mathcal{C}$	$\mathcal{D}$	$\mathcal{D}$
1	2	3	1	2	3	1	2	3	1

11:3 MIN

$\mathcal{A}$	$\mathcal{A}$	$\mathcal{A}$	$\mathcal{B}$	$\mathcal{B}$	$\mathcal{B}$	$\mathcal{C}$	$\mathcal{C}$	$\mathcal{C}$	$\mathcal{D}$	$\mathcal{D}$
1	2	3	1	2	3	1	2	3	1	2

11:3 ASYM

$\mathcal{A}$	$\mathcal{A}$	$\mathcal{A}$	$\mathcal{A}$	$\mathcal{A}$	$\boldsymbol{\mathcal{D}}$	$\boldsymbol{\mathcal{D}}$	$\boldsymbol{\mathcal{D}}$	$\mathcal{D}$	$\mathcal{D}$	$\mathcal{D}$
1	2	3	1, 2	2, 3	-	-	-	1	2	3

12:3 UNI

$\mathcal{A}$	$\mathcal{A}$	$\mathcal{A}$	$\mathcal{A}$	$\mathcal{A}$	$\mathcal{A}$	$\mathcal{A}$	$\mathcal{A}$	$\mathcal{D}$	$\mathcal{D}$	$\mathcal{D}$	$\mathcal{D}$
1	2	3	1, 2	2, 3	1, 3	1, 2, 3	-	1	2	3	-

Abbildung A.1.: Die für unsere Experimente genutzten Instanzen des SW-MDP.

B
Literaturverzeichnis

[BSP01] Bikramjit Banerjee, Sandip Sen und Jing Peng. „Fast Concurrent Reinforcement Learners“. In: *17th International Joint Conference on Artificial Intelligence - Volume 2*. 2001, S. 825–830.

[BT96] Dimitri P. Bertsekas und John N. Tsitsiklis. *Neuro-Dynamic Programming*. Athena Scientific, 1996.

[CP05] Rémi Coulom und Philippe Preux. „Parallelization of the TD(λ) Learning Algorithm“. In: *European Workshop on Reinforcement Learning*. 2005.

[FE06] Scott Fields und I. Elhanany. „Symmetric Multiprocessor Architecture for Multi-Agent Temporal Difference Learning“. In: *IEEE International Midwest Symposium on Circuits & Systems*. 2006, S. 505–509.

[GK07] Matthew Grounds und Daniel Kudenko. „Parallel reinforcement learning with linear function approximation“. In: *International Conference on Autonomous Agents and Multiagent Systems*. 2007.

[Gam+94] Erich Gamma u. a. *Design Patterns: Elements of Reusable Object-Oriented Software*. 1. Aufl. Addison-Wesley Professional, 1994.

[KLM96] Leslie Pack Kaelbling, Michael L. Littman und Andrew W. Moore. „Reinforcement Learning: A Survey“. In: *Journal of Artificial Intelligence Research* 4 (1996), S. 237–285.

[Kre02] R. Matthew Kretchmar. „Parallel Reinforcement Learning“. In: *The 6th World Conference on Systemics, Cybernetics, and Informatics*. 2002.

[Kre03] R. Matthew Kretchmar. „Reinforcement Learning Algorithms for Homogenous Multi-Agent Systems“. In: *Workshop on Agent and Swarm Programming*. 2003.

[Kus+06] Masayuki Kushida u. a. „A Comparative Study of Parallel Reinforcement Learning Methods with a PC Cluster System“. In: *IEEE/WIC/ACM international conference on Intelligent Agent Technology (IAT)*. 2006, S. 416–419.

[LP01] Guillaume Laurent und Emmanueal Piat. „Parallel Q-Learning for a block-pushing problem“. In: *IEEE/RSJ International Conference on Intelligent Robots and Systems*. Bd. 1. 2001, S. 286–291.

[LSS09] Sangjin Lee, Debashis Saha und Mahesh Somani. *Robust and Scalable Concurrent Programming: Lessons from the Trenches. JavaOne*. 2009.
URL: http://www.slideshare.net/sjlee0/robust-and-scalable-concurrent-programming-lesson-from-the-trenches (besucht am 08. 08. 2011).

[Map] 2010.
URL: http://stackoverflow.com/questions/2539654/java-concurrency-many-writers-one-reader/2539761#2539761 (besucht am 08. 08. 2011).

[PEM02] Alicia M. Printista, Marcelo L. Errecalde und Cecilia I. Montoya. „A parallel implementation of Q-Learning based on communication with cache“. In: *Journal of Computer Science & Technology* 1.6 (2002).

[PW11a] Frank Padberg und David Weiss. „Model-Based Scheduling Analysis for Software Projects“. In: *International Workshop on the Quality-Oriented Reuse of Software (QUORS) at IEEE Computer Software and Applications Conference (COMPSAC)*. (erscheint bald). 2011.

[PW11b] Frank Padberg und David Weiss. „Optimal Scheduling of Software Projects Using Reinforcement Learning“. In: *Asia-Pacific Software Engineering Conference (APSEC)*. (erscheint bald). 2011.

[Pad02] Frank Padberg. „A discrete simulation model for assessing software project scheduling policies“. In: *Software Process Improvement and Practice* 7.3-4 (2002), S. 127–139.

[Pad05] Frank Padberg. „On the Potential of Process Simulation in Software Project Schedule Optimization“. In: *International Workshop on Software Cybernetics (II): Software Testing Process Control (IWSC) at IEEE Computer Software and Applications Conference (COMPSAC)*. Bd. 2. 2005, S. 127–130.

[Pad06] Frank Padberg. „A Study on Optimal Scheduling for Software Projects“. In: *Software Process Improvement and Practice* 11.1 (2006), S. 77–91.

[Pad99] Frank Padberg. „A Probabilistic Model for Software Projects“. In: *European Software Engineering Conference ESEC/FSE 7*. 1999, S. 109–126.

[Pal07] Victor Palmer. „Scaling reinforcement learning to the unconstrained multi-agent domain“. Ph.D. Dissertation. Texas A&M University, 2007.

[SB98] Richard S. Sutton und Andrew G. Barto. *Reinforcement Learning*. MIT Press, 1998.

[Sai+09] Priscila Tiemi Maeda Saito u. a. „Parallel implementation of mobile robotic self-localization“. In: *2009 International Conference on Hybrid Information Technology (ICHIT)*. 2009, S. 390–396.

[San] „Evaluating Concurrent Reinforcement Learners“. In: *4th International Conference on MultiAgent Systems (ICMAS)*. 2000, S. 421–422.

[Thr] *Dokumentation von ThreadLocalRandom in der* JAVA *7 API*.
URL: `http://download.oracle.com/javase/7/docs/api/java/util/concurrent/ThreadLocalRandom.html` (besucht am 22. 09. 2011).

[VGD09] Antonio Valles, Matt Gillespie und Garrett Drysdale. *Performance Insights to Intel® Hyper-Threading Technology*. 2009.
URL: `http://software.intel.com/en-us/articles/performance-insights-to-intel-hyper-threading-technology/` (besucht am 16. 09. 2011).

[WS04] David Wingate und Kevin D. Seppi. „P3VI: A Partitioned, Prioritized, Parallel Value Iterator“. In: *21st International Conference on Machine Learning*. 2004, S. 109–116.

[Wei10] David Weiss. „Optimal Scheduling of Software Projects Using Machine Learning“. Masterarbeit. Saarland University, 2010.