Bachelorstudiengang Informatik

Bachelorarbeit

Grundlagen der Selbstorganisationsmechanismen „Agent Discovery“ und „Termination Detection“ im Kontext eines agentenbasierten Einsatzplanungsverfahrens

vorgelegt von

Sönke Martens

Betreuender Gutachter: Prof. Dr. Michael Sonnenschein
Zweiter Gutachter: M.Sc. Christian Hinrichs

Oldenburg, den 31. Juli 2013

Inhaltsverzeichnis

Abkürzungsverzeichnis

CTA	Computation-Tree-Algorithmus
STA	Static-Tree-Algorithmus
SVM	Splitting-Vektor-Methode

Formelverzeichnis

N	Anzahl der Agenten
E	Kantenzahl
M	Nachrichtenzahl
G	Maximalgrad des Graphen
D	Durchschnitt des Graphen (Maximaler Abstand aller Knoten eines Graphen)

Zusammenfassung. Im Rahmen dieser Bachelorarbeit wird gezeigt, wie ein agentenbasiertes Einsatzplanungsverfahren um Terminierungserkennung und Verfahren zur verteilten Berechnung der Agentenzahl erweitert werden kann. Die Funktionsweise der untersuchten Algorithmen für die Terminierungserkennung und Berechnung der Agentenzahl werden erläutert und verschiedene Kriterien, wie z.B Nachrichtenkomplexität, analysiert. Aufbauend auf den theoretischen Analysen werden die Algorithmen praktisch evaluiert, indem sie in ein Multiagentensystem mit einem Einsatzplanungsverfahren integriert werden und in unterschiedlichen Szenarien getestet werden. Zusätzlich bietet die vorliegende Arbeit einen Vergleich und Überblick unterschiedlicher verteilter Algorithmen zur Terminierungserkennung und Berechnung der Agentenzahl. Die untersuchten Algorithmen sind über die Anwendung in verteilten Einsatzplanungsverfahren hinaus interessant, weil sie in beliebigen verteilten Systeme mit einer zusammenhängenden, bidirektionalen Topologie, beliebiger Nachrichtenverzögerung und asynchroner Kommunikation eingesetzt werden können. Die theoretischen und praktischen Abschätzungen bzgl. Nachrichtenkomplexität und Dauer der Algorithmen dienen dabei als Orientierung zur Auswahl geeigneter Verfahren für Terminierungserkennung und für die Berechnung der Agentenzahl.

1. Einleitung

Durch verteilte Systeme - wie Agentensysteme - ist es möglich komplexe, verteilte Probleme zu lösen. Ein Beispiel hierfür sind Multiagentensysteme, die für verschiedene Anwendungsfälle eingesetzt werden, wie Simulationen [14], Überwachungssysteme [17] oder Prozess-Scheduling [19]. Dabei können Multiagentensysteme Selbstorganisation und Emergenz ermöglichen, d.h. die einzelnen Agenten erfüllen durch Anpassung an andere Agenten oder ihre Umgebung eine globale Funktion. [14] Das agentenbasierte Einsatzplanungsverfahren COHDA, das im Rahmen dieser Bachelorarbeit erweitert wird, kann beispielsweise mittels Selbstorganisationsmechanismen eine Gruppe Stromverbraucher so steuern, dass sie kooperativ ihren Verbrauch anhand der erzeugten Strommenge planen. [5]

Der Nutzen und die Nutzungsmöglichkeiten verteilter Systeme wird durch ihre erschwerte Verwaltung und Synchronisation getrübt. Beispielsweise liegen benötigte Informationen über das System nicht immer an einer zentralen, abfragbaren Stelle vor. Das Einsatzplanungsverfahren COHDA könnte zum Beispiel optimiert werden, wenn die Agenten die Gesamtzahl an Agenten im System ermitteln könnten. Mögliche Verfahren zur verteilten Berechnung der Agentenzahl sind unter anderem

Echo-Algorithmen und Gossip-Ansätze. (vgl. [13] und [1]) Aufgrund des Fehlens eines gemeinsamen Oberbegriffs, werden diese Verfahren hier unter den Begriff Agent Discovery Algorithmen zusammengefasst. Ein Ziel dieser Bachelorarbeit ist, Agent Discovery Algorithmen zu untersuchen und zu vergleichen, die im Kontext des Einsatzplanungsverfahrens COHDA als Erweiterung einsetzbar sind.

Ein weiteres Ziel ergibt sich aus einem weiteren Beispiel für die erschwerte Synchronisation, nämlich dem Erkennen des Endes einer verteilten Berechnung in einem Multiagentensystem. Dieses Problem wird als *Distributed Termination Detection* bezeichnet. [10] Termination Detection ist essentiell für terminierende Berechnungen, die ein Endergebnis liefern sollen. Denn ohne Termination Detection können die Ergebnisse nicht genutzt werden, weil zwischen Endergebnissen und Zwischenergebnissen nicht unterschieden werden kann. Erst wenn der Algorithmus terminiert ist und dieser Zustand erkannt wird, ist sicher, dass gefundene Ergebnisse endgültig sind und somit verwendet werden können. Deshalb soll das Einsatzplanungsverfahren COHDA um Termination Detection erweitert werden, so dass die Agenten kooperativ erkennen können, wann die Berechnung von COHDA beendet ist. Terminierungserkennung in verteilten Systemen ist nicht trivial. Denn eine Überprüfung, ob alle Agenten ihre Berechnung beendet haben, genügt nicht, da jederzeit ein Agent durch eine empfangene Nachricht wieder aktiv werden kann. [13, S. 121] Im Rahmen dieser Bachelorarbeit werden deshalb neben den Agent Discovery Algorithmen zusätzlich Algorithmen vorgestellt, die Termination Detection ermöglichen und COHDA um die Terminierungserkennung erweitern können. Dabei werden die Algorithmen auf verschiedene Kriterien wie Nachrichtenkomplexität und Speicheraufwand theoretisch und praktisch überprüft und Vor- und Nachteile der betrachteten Algorithmen erläutert.

In Abschnitt 2 wird zunächst das Einsatzplanungsverfahren COHDA beschrieben und speziell auf die Anforderungen für die Erweiterungen Agent Discovery und Termination Detection eingegangen. Aufbauend auf den Anforderungen werden Agent Discovery Algorithmen in Abschnitt 3 und Termination Detection in Abschnitt 4 betrachtet. Dabei wird jeweils ein Überblick über verschiedene Algorithmen geboten und für die Erweiterung von COHDA geeignete Algorithmen im Detail erläutert und verglichen. Anschließend wird in Abschnitt 5 ein Konzept entworfen, mit welchem die Integration der Algorithmen in das Multiagentensystem von COHDA ermöglicht wird. Die untersuchten Algorithmen wurden implementiert und in zahlreichen Testläufen mit unterschiedlichen Szenarien evaluiert. Die Ergebnisse der Testläufe werden in Abschnitt 6 analysiert und den theoretischen Erwartungen gegenüber gestellt. Abschließend wird in Abschnitt 7 aus den gewonnenen Ergebnissen ein Fazit gezogen, so dass unter anderem Empfehlungen möglich sind, welche Algorithmen sich als Erweiterung von COHDA eignen.

2. Das Einsatzplanungsverfahren: COHDA

Die verteilte Heuristik COHDA wird von Christian Hinrichs im Rahmen seiner Dissertation erarbeitet und wurde in [5] und [6] vorgestellt. Ein mögliches Einsatzgebiet von COHDA ist das Demand Side Management im Energiebereich. Anders als bei der traditionellen Einsatzplanung, bei welcher der Ausgleich zwischen erzeugter und benötigter Strommenge ausschließlich durch steuerbare Erzeuger geregelt wird, können beim Demand Side Management zusätzlich auch die Verbraucher geregelt werden. Dies ist besonders sinnvoll, wenn einerseits schwer steuerbare Erzeuger vorhanden sind, wie regenerative Anlagen, und andererseits Verbraucher zur Verfügung stehen, die in einem bestimmten Rahmen frei regelbar sind, wie z.B. Kühlschränke, Klimaanlagen etc. Um die Einsatzplanung vorzunehmen, muss eine große Anzahl an verschiedenen Verbrauchern und Erzeugern so geregelt werden, dass das globale Ziel Stromausgleich zu jeder Zeit erfüllt ist, ohne dabei lokale Bedingungen, wie Temperaturschwellen eines Kühlschranks, zu verletzen. Die Heuristik COHDA löst dieses Problem mittels Selbstorganisationsmechanismen, indem die einzelnen Systemteilnehmer in einem verteilten System kooperativ eine Lösung suchen. [5]

Auf die Funktionsweise von COHDA wird in Abschnitt 2.1 näher eingegangen und anschließend werden in Abschnitt 2.2 die Voraussetzungen an das System sowie der Ablauf von COHDA und den Erweiterungen erläutert.

2.1. Funktionsweise

COHDA wird durch ein Multiagentensystem realisiert, d.h. es gibt mehrere (Software-)Agenten die miteinander interagieren, um eine optimale Lösung zu ermitteln. Beim Beispiel des Demand Side Management sind die Agenten einzelne Verbraucher und Erzeuger, die über ein Netzwerk kommunizieren. Jeder Agent hat unterschiedliche Modi, die er wählen kann. Insgesamt müssen die Modi der einzelnen Agenten so gewählt werden, dass das globale Ziel und die lokalen Bedingungen der Agenten erfüllt sind. Jeder Agent kennt dabei nur das globale Ziel und seine eigene lokale Bedingung, so dass die Agenten miteinander kommunizieren müssen, um eine Lösung zu finden. Eine Möglichkeit wäre es, wenn alle Agenten ihre lokalen Bedingungen einem zentralen Agenten übermitteln, der aus diesen dann eine optimale Lösung berechnet. Dieser Ansatz ist nicht immer wünschenswert oder möglich. (vgl. [6])

Deshalb wird in COHDA eine Lösung mit Hilfe von Selbstorganisationsmechanismen gesucht, indem Agenten auf die Entscheidungen ihrer Nachbarn reagieren. Die Nachbarn eines Agenten sind dabei die Agenten, mit denen er direkt über Nachrichtenkanäle verbunden ist. Das Multiagentensystem bildet einen zusammenhängenden Graphen, in dem die Agenten die Knoten und die Nachrichtenkanäle zu den Nachbarn die Kanten des Graphen bilden. Bevor erklärt werden kann, wie sich die Agenten

an ihre Nachbarn anpassen, werden zunächst einige Eigenschaften der Agenten und ihrer Modi beschrieben.

Die Modi M_{ij} eines Agenten A_i besitzen zwei Eigenschaften: Einen Berechnungswert w_{ij} und einen Strafwert p_{ij}. Der Berechnungswert fließt bei der Erreichung des globalen Ziels z ein. Existieren beispielsweise nur zwei Agenten A_1 mit den Berechnungswerten $w_{11} = 1, w_{12} = 2, w_{13} = 6$ und A_2 mit $w_{21} = 4, w_{22} = 1$, dann versuchen die Agenten Berechnungswerte zu wählen, welche in der Summe dem globalen Ziel möglichst nah kommen. Bei einem globalen Ziel $z = 3$ würde A_1 w_{12} und A_2 w_{22} wählen. Der Strafwert der Berechnungswerte beschreibt die lokalen Bedingungen der Agenten. Je kleiner der Strafwert ist, umso weniger verletzt der gewählte Modus die lokalen Bedingungen. Aus diesem Grund bevorzugen Agenten bei der Suche nach einer Lösung Modi mit möglichst geringem Strafwert. In Gleichung 1 wird die Berechnung der Qualität der Lösung für einen Agenten A_i gezeigt. Dabei ist g_i die Differenz der Summe der gewählten Berechnungswerte zum globalen Ziel und p_i ist der Strafwert des gewählten Berechnungswert von A_i. α bestimmt dabei, wie egoistisch ein Agent ist. Ist beispielsweise α niedrig, sind für den Agenten die lokalen Bedingungen wichtiger als das globale Ziel. Insgesamt ist die Qualität der Lösung q_i für einen Agenten A_i umso besser, je niedriger q_i ist. Ein großes q_i dagegen bedeutet eine große Abweichung vom globalen Ziel oder einen großen Strafwert für den gewählten Modus.

$$q_i = \alpha_i \cdot g_i + (1 - \alpha_i) \cdot p_i \tag{1}$$

Damit ein Agent einen geeigneten Modus wählen kann, muss er g_i berechnen können und dafür die gewählten Berechnungswerte der anderen Agenten kennen. Dazu besitzen alle Agenten eine Menge Σ, in der sie für jeden Agenten im System speichern, welcher Berechnungswert dieser Agent nach ihrem Kenntnisstand momentan hat und von welchem Zeitpunkt diese Information ist. Da die Agenten zu Beginn noch keine Nachrichten von anderen Agenten erhalten haben, ist Σ zunächst leer. Mit Σ_i kann der Agent A_i g_i berechnen, indem die Differenz vom globalen Ziel z und der Summe der Berechnungswerte aus Σ_i gebildet wird.

$$g_i = \left|\left| z - \sum_{w \in \Sigma_i} w \right|\right|_1 \tag{2}$$

Neben Σ besitzt jeder Agent zusätzlich eine Menge Σ^*, in welcher die bisher beste Menge Σ gespeichert wird, also die Menge die dem globalen Ziel am nächsten war. Zu Beginn wählt ein Agent A_i einen neuen Modus M_{ij}, fügt den neuen Berechnungswert w_{ij} zu seinem bisher leerem Σ_i hinzu und initialisiert das bisher nicht vorhandene Σ_i^* mit Σ_i. Anschließend sendet A_i seinen Nachbarn in einer Nachricht sowohl Σ_i als

auch Σ_i^*.

Ein Agent A_j der eine Nachricht von einem Agenten A_i empfängt, aktualisiert seinen Kenntnisstand, indem er alle veralteten Werte aus seinem Σ_j mit neueren Werten aus dem empfangenen Σ_i ersetzt. Außerdem ersetzt er sein Σ_j^* mit Σ_i^*, wenn dieses für ihn bzgl. Gleichung 1 eine bessere Lösung bietet. Nun überprüft A_j, ob durch die Wahl eines anderen Berechnungswertes w_{jk} eine bessere Lösung möglich ist. Falls dies der Fall ist, ersetzt er den alten Berechnungswert in Σ_j mit w_{jk} und überschreibt Σ_j^* mit Σ_j, wenn die Qualität vom neuen Σ_j besser als die von Σ_j^* ist. Falls keine Verbesserung möglich ist, wird der Modus mit dem in Σ_j^* gespeicherten Berechnungswert gewählt und der alte Berechnungswert in Σ_j mit diesem ersetzt. Wenn sich durch Empfang der Nachricht von A_i Σ_j oder Σ_j^* verändert haben, sendet A_j eine Nachricht mit Σ_j und Σ_j^* an seine Nachbarn, die dann beim Empfang der Nachricht wie beschrieben reagieren.

Insgesamt suchen die Agenten auf diese Weise kooperativ eine Lösung. Wählt ein Agent einen neuen Modus, passen sich die anderen Agenten wenn notwendig an und durch Σ^* wird verhindert, dass die beste Lösung vergessen wird. Somit werden Schritt für Schritt bessere Lösungen entdeckt, bis kein Agent mehr eine Verbesserung erzielen kann und der Algorithmus terminiert. COHDA bietet also eine Möglichkeit in einem verteilten System mit verschiedenen Systemteilnehmern ein globales Ziel dezentralisiert zu erreichen ohne dabei lokale Bedingungen zu verletzen.

2.2. Voraussetzungen und Ablauf

Damit COHDA in möglichst vielen verschiedenen Systemen einsetzbar ist, sind die nötigen Voraussetzungen gering. Beispielsweise ist es für beliebige zusammenhängende Topologien verwendbar. Der Graph, der das Multiagentensystem beschreibt, kann also beliebige Formen haben, solange er zusammenhängend ist. Deshalb existiert nicht unbedingt ein zentraler Agent, welcher mit allen anderen Agenten direkt verbunden ist. Agenten können mit ihren Nachbarn über bidirektionale Nachrichtenkanäle kommunizieren und der dabei erfolgende Nachrichtenaustausch darf asynchron sein, d.h. COHDA verlangt nicht, dass die Agenten Nachrichten gleichzeitig senden. Zusätzlich ist COHDA robust gegen verzögerte Nachrichten und es ist somit nicht erforderlich, dass Nachrichten in der Reihenfolge ankommen, wie sie losgeschickt wurden. [6]

Die beiden Erweiterungen, die in den nächsten Kapiteln vorgestellt werden, sollen die gleichen Anforderungen wie COHDA unterstützen, damit sie die Einsatzmöglichkeiten von COHDA nicht einschränken. Die Erweiterungen müssen also bei beliebigen Topologien, asynchroner Kommunikation und Nachrichtenverzögerung funktionieren. Die erste Erweiterung - Agent Discovery - soll den Agenten die Anzahl der Agenten im System mitteilen. Diese kann zur Optimierung des Algorithmus von COHDA

benutzt werden. Beispielsweise können die Agenten mit der Agentenzahl erkennen, wann eine Menge Σ Berechnungswerte für alle Agenten beinhaltet. Die zweite Erweiterung - Termination Detection - bietet den Agenten die Möglichkeit kooperativ die Terminierung von COHDA festzustellen, so dass die Agenten wissen, wann das Ergebnis final ist und verwendet werden kann.

Allgemein soll der Gesamtablauf mit den Erweiterungen wie folgt ablaufen: Bevor der Algorithmus von COHDA startet, stehen die Nachbarschaftsbeziehungen bereits fest. Die Agenten kennen also ihre Nachbarn. Außerdem besitzt jeder Agent eine eindeutige ID, die bei den Erweiterungen benötigt wird. Irgendwann wird allen Agenten von außen das globale Ziel für COHDA übermittelt. Diese Mitteilung dient gleichzeitig als Startsignal für die erste Erweiterung zur Ermittlung der Agentenzahl. Da die Agenten nicht wissen können, ob andere Agenten bereits die Erweiterung gestartet haben, muss diese auch mit mehreren Initiatoren funktionieren.

Nachdem die Agenten die Agentenzahl ermittelt haben, können die Agenten COHDA starten. Erwähnenswert ist, dass COHDA parallel bei allen Agenten startet und es somit keinen zentralen Agenten gibt, von dem der Algorithmus startet. Gleichzeitig mit COHDA startet die zweite Erweiterung, welche den Ablauf von COHDA untersucht, um die Terminierung festzustellen. Wenn die Terminierung eintritt, werden alle Agenten darüber von der Erweiterung in Kenntnis gesetzt und jeder Agent weiß somit, dass die gesuchte Lösung gefunden ist.

3. Agent Discovery

Eine häufige Aufgabe verteilter Systeme ist, globale Informationen kooperativ zu ermitteln. Beispiele für globale Informationen sind die Anzahl der Teilnehmer oder verbrauchter Speicherplatz im System. [8] Um solche Informationen zu ermitteln, müssen diese aus den Teilinformationen der Systemteilnehmer gewonnen werden. Anhand des Beispiels der Anzahl der Teilnehmer, werden in diesem Kapitel Algorithmen gesucht, die es den Agenten ermöglichen, diese Information kooperativ zu ermitteln. Diese Algorithmen werden im Folgenden als Agent Discovery Algorithmen (kurz AD-Algorithmen) bezeichnet und meinen damit verteilte Verfahren zur Ermittlung der Agentenzahl in einem Multiagentensystem. Weiterhin sollen die gesuchten AD-Algorithmen COHDA erweitern können, so dass, wie in Abschnitt 2 erwähnt, den Agenten die Agentenzahl zur Verfügung steht und dabei die oben beschriebenen Voraussetzungen von COHDA erfüllt werden. Insbesondere ist dabei wichtig, dass es keinen zentralen Agenten gibt, der alle Agenten in seiner Nachbarschaft hat. Somit ist nicht möglich, dass ein Agent zentral die Anzahl ermittelt, indem er seine Nachbarn durchzählt. Weiterhin wird angenommen, dass eine bestimmte Zeit vor dem Start von COHDA die Nachbarschaftsbeziehungen der Agenten bekannt sind und

sich nicht mehr ändern, so dass dann ein AD-Algorithmus gestartet werden kann.

Das Problem ist beispielsweise lösbar, indem jeder Agent seine ID an alle Nachbarn schickt. Ein Nachbar leitet die empfangene ID ebenfalls an alle Nachbarn weiter, wenn er nicht bereits zuvor die gleiche ID empfangen hat. Mit diesen beiden einfachen Regeln hat jeder Agent nach einer gewissen Zeit alle IDs empfangen und weiß somit, wie viele Agenten im System existieren. Leider hat diese einfache Lösung, auch Flooding-Technik [13, S. 30] genannt, zwei entscheidende Nachteile. Zum einen braucht der Ansatz eine große Anzahl an Nachrichten, nämlich mindestens die Anzahl der Agenten N mal die Anzahl der Kanten E. Denn jede ID wird im besten Fall einmal über jede Kante geschickt. Zum anderen kann kein Agent feststellen, ob er bereits alle IDs empfangen hat. Er weiß also nicht, ob die Anzahl der Agenten endgültig und korrekt ist. Um dies zu erreichen, müsste der Ansatz noch mit einer Terminierungserkennung überlagert werden. (vgl. Abschnitt 4) Da eine Terminierungserkennung zusätzlich Kontrollnachrichten benutzt, würde die Anzahl benötigter Nachrichten steigen. Es gibt jedoch Möglichkeiten, das Problem effizient zu lösen. Dazu werden in den folgenden Abschnitten Algorithmen mit unterschiedlichen Eigenschaften vorgestellt.

In Abschnitt 3.1 wird ein eigener Ansatz vorgestellt, der versucht die Nachrichtenmenge zu verringern, indem nach und nach Agenten aus dem AD-Algorithmus ausscheiden. Am Ende bleibt ein Agent übrig, der aus der Anzahl der ausgeschiedenen Agenten die Agentenanzahl schließen kann. Einen anderen Ansatz verfolgen Gossip Algorithmen. Bei diesen wählt jeder Agent wiederholt einen seiner Nachbarn und teilt diesem eine Information mit, welcher in der nächsten Iteration wiederum ebenfalls die Information einem Nachbarn sendet. Somit verbreitet sich die Information über das gesamte System ähnlich zu einem Gerücht, daher auch der Name Gossip (engl. für Klatsch) Algorithmus. [15, S. 1] Gossip Algorithmen können benutzt werden, um Agent Discovery zu ermöglichen. In [1] besitzt zu Beginn jeder Agent ein Token und übermittelt diesen an einen Nachbarn. Dabei werden Techniken eingesetzt, die dafür sorgen, dass am Ende ein Agent alle Token hat und somit die Agentenanzahl kennt. Das Verfahren ermöglicht es insbesondere darauf zu reagieren, wenn neue Agenten oder ganze Gruppen von Agenten während der Berechnung hinzugefügt oder entfernt werden. Diese Flexibilität ist aber durch eine erhöhte Komplexität des Algorithmus erkauft. Da angenommen wurde, dass sich die Anzahl der Agenten während des AD-Algorithmus nicht ändert, ist dieser Mehraufwand nicht nötig. Ein einfacher Gossip-Ansatz für Agent Discovery wurde von Jelasity, Montresor und Babaoglu in [8] vorgestellt und wird in Abschnitt 3.2 näher beschrieben.

Auf die zuvor erwähnte Flooding-Technik baut der Echo-Algorithmus von Ernest J. H. Chang auf. [2] Dieser Algorithmus wird in Abschnitt 3.3 mittels kleiner Änderungen für Agent Discovery angepasst. Der Echo-Algorithmus ist dabei ein Konzept,

welches nicht nur für Agent Discovery interessant ist, sondern auch bei vielen anderen Problemen eingesetzt wird. Beispielsweise kann er dafür verwendet werden, einen Spannbaum eines Graphen zu finden. (vgl. [13, S. 30f.] und Abschnitt 4.4)

3.1. Join Ansatz

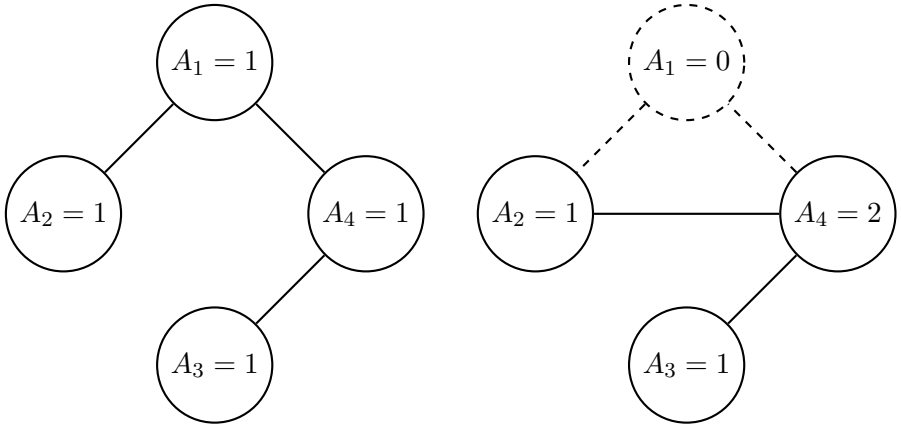
Der Join Ansatz versucht, die Nachrichtenzahl zu senken, indem die Anzahl der Teilnehmer des AD-Algorithmus stetig sinkt und somit potentiell weniger Agenten Nachrichten senden. Ein Agent der nicht mehr am Algorithmus teilnimmt wird als **entfernt** bezeichnet. Damit ist nicht gemeint, dass dieser aus der zu Grunde liegenden Nachbarschaft entfernt wird, sondern lediglich, dass der Agent selbst und seine Kanten als entfernt markiert werden. Somit wissen die anderen Agenten, dass der entfernte Agent am AD-Algorithmus nicht mehr teilnimmt. Neben dem entfernt Zustand kann ein Agent auch **beschäftigt**, **anfragend** oder **ruhend** sein. Jeder Agent ist zunächst ruhend und besitzt eine mit 1 initialisierte Zählvariable.

Ein ruhender Agent A_i versucht einen Partner unter seinen Nachbarn zu finden, um diesem seine Zählvariable mitzuteilen. Dazu fragt er einen beliebigen Nachbarn A_j an und der Agent A_i wechselt somit in den Zustand anfragend. Abhängig vom Zustand des Nachbarn, sind nun unterschiedliche Abläufe möglich. Ist A_j beschäftigt, wird die Anfrage abgelehnt, so dass A_i wieder ruhend wird und nach einer bestimmten Zeit wieder bei einem Nachbarn anfragt. Wenn A_j anfragend ist und der Agent bei dem A_j anfragt wiederum A_i ist, dann werden beide Agenten Partner. Fragt A_j jedoch bei einem anderen Agenten A_k ($k \neq i$) an, wird die ID der beiden Agenten betrachtet. Ist die ID von A_i kleiner als die von A_j, wird die Anfrage behandelt, als ob A_j beschäftigt ist und die Anfrage wird abgelehnt. Ist allerdings die ID von A_i größer, wird die Anfrage von A_j an A_k beendet und A_i und A_j werden Partner. Durch diese Regeln ist gewährleistet, dass nicht alle Agenten anfragend sind, ohne dass Partner entstehen. Mindestens der Agent mit der höchsten ID findet einen Partner. Es kommt also nicht zu einem Dead-Lock. Ist A_j ruhend, werden A_i und A_j ebenfalls Partner. Um den Wahlprozess zu verbessern, wäre es denkbar, dass Agenten nur bei Agenten mit einer niedrigeren ID anfragen, dazu müssen die Agenten jedoch die ID der Nachbarn kennen. Sind zwei Agenten Partner, sind sie beschäftigt und können keine anderen Partner haben, so lange sie beschäftigt sind.

Sind zwei Agenten Partner, passiert folgendes: Der Agent mit der geringeren Kantenzahl wird als entfernt markiert und die Zählvariable des Agenten wird zu der Zählvariable des Partners dazu addiert. Haben beide die gleiche Kantenzahl, wird der Agent mit der niedrigeren ID entfernt. Außerdem werden ebenso die ausgehenden Kanten des entfernten Agenten als entfernt markiert. Damit es nicht passieren kann, dass der Graph durch das Entfernen der Kanten nicht mehr zusammenhängend ist, müssen neue Kanten eingeführt werden. Wurde der Agent A_i entfernt, wird für jeden

Nachbarn von A_i, eine Kante zwischen diesem und dem Partner A_j von A_i erstellt, wenn diese Kante nicht bereits vorhanden ist. Es werden also alle Nachbarn vom entfernten A_i zum Partner A_j hinzugefügt und A_j wird zu den Nachbarschaftsmengen der Nachbarn von A_i hinzugefügt. Der Name Join (engl. zusammenfügen) Ansatz, soll darauf hindeuten, dass die beiden Partner prinzipiell zu einem Knoten mit summierter Zählvariable und vereinigeter Nachbarschaft zusammengefügt werden.

In Abbildung 1 ist ein Entfernungs-Schritt nach dem Start des Algorithmus dargestellt. Sind beispielsweise A_1 und A_4 Partner und A_1 wird entfernt, werden seine Kanten als entfernt markiert und die Zählvariable von A_1 zur Zählvariable von A_4 hinzuaddiert. Außerdem wird eine Nachbarschaftsbeziehung zwischen dem Partner A_4 und dem Nachbarn A_2 von A_1 erzeugt.



(a) Graph vor dem Entfernen von Knoten A_1

(b) Graph nach dem Entfernen von Knoten A_1

Abbildung 1: Beispiel mit vier Agenten

Mit der Zeit werden auf diese Weise alle Agenten bis auf einen entfernt, welcher die Terminierung des Algorithmus daran erkennt, dass er nur mit entfernten Nachbarn verbunden ist. Seine Zählvariable enthält dann die Anzahl der Agenten, welche er an alle mit der Flooding-Technik verschickt. Dazu muss er die Anzahl an alle Nachbarn schicken, welche diese ebenso an ihre Nachbarn weiterleiten. Im besten Fall liegt die Anzahl der Nachrichten des Algorithmus in $O(N \cdot G + E)$ und im Worst-Case in $O(N^2 + N \cdot G + E)$. (vgl. Beweis A.1 im Anhang) Im Durchschnitt sollte die Nachrichtenzahl aber deutlich besser als im Worst-Case sein, da dieser nur auftritt, wenn zu jeder Zeit alle Agenten gleichzeitig anfragend sind und nur der mit der höchsten ID einen Partner findet. In diesem Fall würde immer nur ein Agent zur Zeit entfernt werden. Eine genaue Aussage über die Effizienz des Algorithmus kann nur durch eine Evaluation bestimmt werden, da sie auch stark von der Implementierung abhängt. Beispielsweise hängt die Effizienz davon ab, wie viele Nachrichten benötigt

werden, damit zwei Agenten Partner werden.

Außerdem wurde bisher die Frage vernachlässigt, wie neue Kanten hinzugefügt werden. Ist das Multiagentensystem in einer vollvermaschten Topologie realisiert, können Kanten direkt hinzugefügt werden, indem die physikalischen Verbindungen genutzt werden. Ansonsten können Kanten nicht direkt realisiert werden. Alternativ könnten sich die Agenten zu neu erstellten Kanten den Pfad merken, über den sie verbunden sind. Im vorherigen Beispiel müssten sich die Agenten A_2 und A_4 für die zwischen ihnen erzeugte Kante merken, dass Nachrichten auf dieser Kante über A_1 versendet werden. Diese Anpassung hat jedoch zur Folge, dass in nicht vollvermaschten Topologien die Anzahl der Nachrichten stark steigen kann. Wenn beispielsweise A_4 eine Nachricht an A_2 schickt, leitet A_1 eine Kopie der Nachricht an A_2 weiter. Somit wurden zwei Nachrichten anstatt von einer Nachricht benötigt. Weil der Pfad einer neu erzeugten Kante über mehrere Knoten laufen kann, kann die Anzahl benötigter Nachrichten in anderen Fällen noch höher sein. Um die Anzahl neu erzeugter Kanten gering zu halten, wird der Knoten mit weniger Kanten entfernt.

3.2. Gossip Algorithmus

In diesem Unterabschnitt wird zunächst auf die allgemeine Vorgehensweise des Gossip Ansatzes in [8] eingegangen und anschließend die von den Autoren von [8] vorgeschlagene Anpassung zum Zählen von Knoten verdeutlicht.

Jeder Agent besitzt einen Wert, in dem eine beliebige Zahl gespeichert wird, und das Ziel des allgemeinen Ansatzes ist es, einen globalen Durchschnittswert über diese Werte zu bilden. Dazu besitzt jeder Agent zwei Threads. Einen Thread zum Senden seines Wertes an einen Nachbarn und einen Thread zum Empfangen dieser Informationen. Der Sende-Thread eines Agenten A_i macht folgendes: Er wählt einen der Nachbarn A_j des Agenten A_i zufällig aus und sendet diesem seinen Wert W_i. Der Empfänger der Nachricht empfängt diese mit dem Empfangen-Thread, welcher an A_i eine Antwort mit dem Wert W_j sendet. Beide Agenten weisen ihren Werten dann den Durchschnitt der beiden Werte W_i und W_j zu. D.h. es werden folgende Zuweisungen ausgeführt:

$$W_i \leftarrow \frac{W_i + W_j}{2} \text{ und } W_j \leftarrow \frac{W_i + W_j}{2}$$

Nachdem der Sende-Thread einen neuen Nachbarn gewählt hat, wird der neu zugewiesene Wert versendet. Mit der Zeit werden sich die Werte aller Agenten ähnlicher und ab einen bestimmten Zeitpunkt konvergieren die Werte zum globalen Durchschnittswert. Dieses Vorgehen kann man leicht für Agent Discovery anpassen. Wenn alle Agenten mit dem Wert 0 und ein Agent mit dem Wert 1 initialisiert werden, dann konvergieren alle Werte zum Durchschnittswert, welcher bei N Agenten $\frac{1}{N}$ ist. Die Anzahl der Agenten ist dann der Kehrwert des gespeicherten Werts.

Im Vergleich zum Join Ansatz ist die Kommunikation und der Ablauf des Algorithmus sehr einfach. Es müssen beispielsweise keine neuen Kanten hinzugefügt werden oder Agenten entfernt werden. Anders als beim Join Ansatz, lässt sich jedoch keine theoretische Anzahl der Nachrichten bestimmen, da diese davon abhängt wie der Algorithmus implementiert ist. Senden zum Beispiel bei einer schlechten Implementierung die Agenten ihren Wert immer wieder zum gleichen Nachbarn, wird im schlimmsten Fall der Wert nie konvergieren. Es wurde aber in [8] anhand von Evaluationen gezeigt, dass bei vielen Topologien der Algorithmus bei einer sinnvollen Implementierung effizient ist.

Der Gossip Ansatz hat jedoch zwei Probleme. Beispielsweise muss ein Agent ausgewählt werden, dessen Wert mit 1 initialisiert wird. Dies ist in verteilten Systemen nicht trivial, kann aber mit Election-Algorithmen gelöst werden. Diese Algorithmen wählen einen *Leader* unter den Agenten aus, welcher dann im Fall des Gossip Algorithmus mit 1 initialisiert werden kann. [13, S. 35-101] In [8] wird eine weitere Möglichkeit aufgezeigt dieses Problem geschickter zu lösen, auf welche aber im Rahmen dieser Arbeit nicht näher eingegangen wird. Das zweite Problem ist, dass analog zur Flooding-Technik auch hier die Agenten nicht erkennen können, wann der Gossip Algorithmus beendet ist. Anders als bei der Flooding-Technik lässt sich dies nicht lösen, indem der Algorithmus mit einer Terminierungserkennung überlagert wird. Denn auch wenn der Gossip Algorithmus konvergiert ist, ist er nicht terminiert. Da die Agenten nicht wissen können, dass alle Agenten den gleichen Wert haben, werden sie weiterhin Werte austauschen. Der Algorithmus läuft also weiter. Es muss also in bestimmten Zeitabständen überprüft werden, ob alle Werte gleich sind und falls dies zutrifft, muss der Gossip Algorithmus gestoppt werden. Beides ist nur mit großem Aufwand möglich.

3.3. Echo-Algorithmus

Die hier vorgestellte Version des Echo-Algorithmus wurde von Friedemann Mattern in [13, S. 30f.] beschrieben und ist gegenüber der ursprünglichen Version von Chang [2] einfacher und effizienter. Auch im Fall des Echo-Algorithmus wird zunächst der allgemeine Ansatz betrachtet, bevor er für Agent Discovery mit kleinen Änderungen angepasst wird. Außerdem wird zunächst angenommen, dass es nur einen Initiator des Algorithmus gibt, d.h. nur ein Agent startet den Echo-Algorithmus. Später wird diese Voraussetzung gelockert und gezeigt, dass auch Varianten für mehrere Initiatoren möglich sind.

Jeder Agent des Systems benötigt für den Algorithmus eine mit 0 initialisierte Variable C, welche die eingehenden Nachrichten zählt. Der Initiator versendet an alle Nachbarn sogenannte Explorer. Empfängt ein Agent einen Explorer wird seine Variable C um 1 erhöht. Ist der empfangene Explorer der erste Explorer, der den Agenten

erreicht hat, wird außerdem der Nachbar, von dem der Explorer kam, als **Vorgänger** gespeichert. Anschließend schickt der Agent allen Nachbarn außer seinem Vorgänger ebenfalls einen Explorer. Die Explorer durchlaufen also analog zur Flooding-Technik alle Agenten. Beim Echo-Algorithmus existieren zusätzlich noch Echo-Nachrichten, welche von einem Agenten zu seinem Vorgänger geschickt werden, wenn seine Variable C gleich der Anzahl seiner Nachbarn ist und er nicht der Initiator ist. Auch beim Empfang eines Echos wird C um 1 erhöht. Jeder Agent mit Nachfolgern - also Agenten die Vorgänger eines anderen Agenten sind - können folglich erst ein Echo zum Vorgänger schicken, wenn er von allen Nachfolgern ein Echo erhalten hat, da keine Explorer von Nachfolgern zu ihrem Vorgänger geschickt werden. Während die Explorer-Nachrichten vom Initiator immer weiter entfernte Knoten durchlaufen, nehmen die Echos den umgekehrten Weg zurück zum Initiator. Ist beim Initiator der Wert von C gleich der Anzahl seiner Nachbarn, weiß der Initiator, dass alle Echos bei ihm angekommen sind und der Echo-Algorithmus terminiert ist.

Der Echo-Algorithmus kann für verschiedene Aufgaben genutzt werden. Beispielsweise kann der Initiator den Explorern eine Information mitteilen, welche dann an alle Agenten verteilt wird oder im umgekehrten Fall können die Agenten den Echos Informationen mitgeben, so dass der Initiator am Ende von allen Agenten diese Information erhalten hat. Der zweite Fall kann für Agent Discovery genutzt werden. Dazu erhält jeder Agent eine weitere Zählvariable, die mit 1 initialisiert wird. Schickt ein Agent ein Echo los, wird diese Zählvariable mit dem Echo zum Vorgänger übertragen und der Vorgänger addiert die Zählvariable im Echo zu seiner Zählvariable hinzu. Wenn auch der Vorgänger von allen Nachbarn eine Nachricht erhalten hat, weiß er, dass keine weiteren Nachrichten mehr bei ihm ankommen werden und schickt auch seine Zählvariable mit einem Echo zu seinem Vorgänger. Da jeder Agent genau eine Echo Nachricht losschickt, wenn bereits die Zählvariablen von seinen Nachfolgern zu seiner Zählvariable hinzugefügt wurden, ist die Zählvariable des Initiators am Ende gleich der Anzahl der Agenten. Dies wird im Folgenden an einem Beispiel verdeutlicht.

Der Echo-Algorithmus soll in einem Multiagentensystem mit vier Agenten durchgeführt werden. Das Multiagentensystem wird dabei durch den Graph in Abbildung 2 beschrieben. Dort sind auch die einzelnen Schritte des Echo-Algorithmus dargestellt. Der Initiator des Echo-Algorithmus ist A_1, welcher zuerst an seine Nachbarn A_2 und A_4 Explorer versendet. Dadurch wird C bei beiden erhöht und A_1 als Vorgänger von A_2 und A_4 markiert. A_2 und A_4 schicken wiederum nun auch Explorer an ihre Nachbarn. Im 3. Schritt kann man erkennen, dass die Nachrichten, die sich A_2 und A_4 gegenseitig geschickt haben, die Variablen C erhöht haben. Die Kante zwischen A_2 und A_4 wurde aber nicht als Kante zum Vorgänger gekennzeichnet, da sowohl A_2 als auch A_4 bereits einen Vorgänger haben. Bei A_3 wurde A_2 als Vorgänger markiert,

d.h. der Explorer von A_2 hat A_3 zuerst erreicht. Dies zeigt auch, dass der Ablauf des Algorithmus nicht deterministisch ist, sondern von Übertragungsgeschwindigkeiten abhängt. A_3 schickt nun einen Explorer an A_4, da A_4 kein Vorgänger ist. Außerdem schickt A_3 ein Echo mit dem Inhalt seiner Zählvariable zu seinem Vorgänger, da er bereits von allen Nachbarn Nachrichten empfangen hat (C ist also gleich der Nachbarzahl von A_3). Die Zählvariable im Echo wird zur Zählvariable von A_2 hinzugefügt und in 4. können nun auch A_2 und A_4 ihre Zählvariablen mit Echos an A_1 schicken. Da somit auch A_1 von allen Nachbarn Nachrichten empfangen hat, weiß er, dass der Echo-Algorithmus terminiert ist und dass vier Agenten im System vorhanden sind.

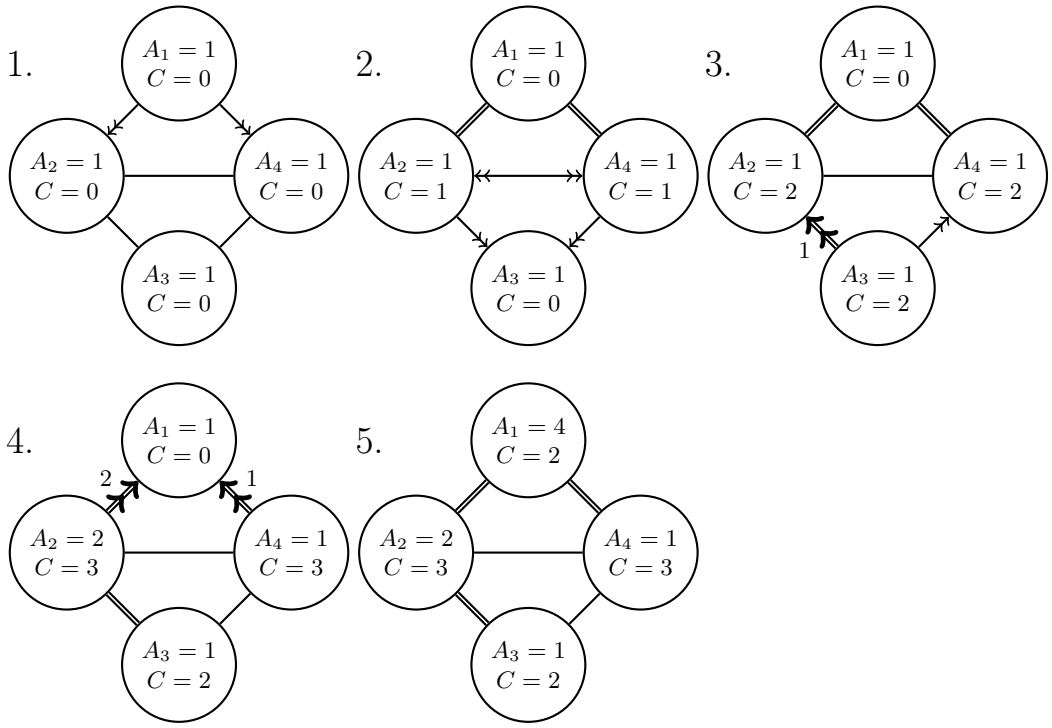


Abbildung 2: Schritte des Echo-Algorithmus: Dabei stellen kleine Doppelpfeile die Explorer dar, große Doppelpfeile sind Echos und doppellinige Kanten sind Vorgängerkanten

An dem Beispiel lässt sich gut erkennen, dass die Kanten zu den Vorgängern einen Spannbaum des Graphen bilden. Diese Erkenntnis wird später noch im Abschnitt 4.4 für die Terminierungserkennung benötigt. Der Spannbaum ist dabei nicht eindeutig, da die Agenten abhängig von den Übertragungsgeschwindigkeiten unterschiedliche Vorgänger haben können.

Damit auch die anderen Agenten die Agentenzahl erfahren, muss nun der Initiator die Agentenzahl beispielsweise mit der Flooding-Technik den anderen Agenten mitteilen. Somit benötigt der Algorithmus maximal $4 \cdot E$ Nachrichten bei einer Kantenzahl E. Jede Kante wird von zwei Explorern durchlaufen oder im Fall einer Kan-

te zu einem Vorgänger von einem Explorer und einem Echo. Zusätzlich wird am Ende jeder Kante von ein bis zwei weiteren Nachrichten bei der Flooding-Technik benutzt. Damit hat der Algorithmus eine im Vergleich zu den vorher betrachteten Algorithmen sehr effiziente Nachrichtenkomplexität von $O(E)$. Dabei wurde jedoch noch nicht der Fall betrachtet, wenn mehrere Initiatoren zu Beginn vorhanden sind. In dem folgenden Unterabschnitt werden mögliche Varianten vorgestellt, die dieses Problem lösen.

3.3.1. Varianten für mehrere Initiatoren

Der Echo-Algorithmus für einen Initiator lässt sich leicht für mehrere Initiatoren erweitern. Dazu besitzt jeder Initiator A_i eine mit 0 initialisierte Variable C_i zum Zählen der Nachrichten und eine mit 1 initialisierte Zählvariable $Agents_i$ zum Zählen der Agenten. Nun wird für jeden Initiator parallel ein Echo-Algorithmus gestartet. Die versendeten Explorer Nachrichten enthalten dabei die ID des Initiators. Kommt ein Explorer mit dem Initiator A_i bei einem Agenten A_j an, wird für diesen Agenten eine mit 1 initialisierte Variable C_i, eine mit 1 initialisierte Variable $Agents_i$ und eine Variable $Pred_i$ für den Vorgänger erstellt. Jeder Agent speichert also für die parallel ablaufenden Echo-Algorithmen eigene Variablen. Explorer und Echo Nachrichten die von einem Agenten ausgehen, beinhalten ebenfalls die ID des Initiators und eine von einem Agenten A_j versendete Echo-Nachricht mit der ID i überträgt auch die entsprechende von A_j gespeicherte Zählvariable $Agents_i$ zum im $Pred_i$ gespeicherten Vorgänger. Insgesamt werden also für k Initiatoren k unabhängige Echo-Algorithmen gestartet und somit liegt die Anzahl an Nachrichten in $O(E \cdot k)$. Für viele Initiatoren ist die Anzahl an Nachrichten also entsprechend hoch.

Die Anzahl lässt sich jedoch reduzieren, (vgl. [13, S. 76f.]) indem jeder Agent nur Echo und Explorer für den Initiator mit der höchsten ID versendet. Dazu besitzt jeder Agent eine Variable Max, in der die höchste ID der empfangenen Explorer gespeichert wird. Explorer und Echos mit einer ID kleiner Max werden dann vom Agenten ignoriert. Von den k laufenden Echo-Algorithmen stoppen also alle Algorithmen mit der nicht höchsten ID nach einer Weile und nur der Initiator mit der höchsten ID erfährt die Agentenzahl. Dieser benachrichtigt dann wieder alle anderen Agenten mit der Flooding-Technik über die Agentenzahl. Die mittlere Nachrichtenkomplexität kann durch die Vereinfachung auf $O(E \cdot \log k)$ reduziert werden. [13, S. 77] Darüber hinaus wird nur eine Variable C, $Agents$ und $Pred$ für jeden Agenten benötigt. Erreicht ein Explorer mit einer höheren ID, als in Max gespeichert ist, einen Agenten, werden C und $Agents$ auf 1 zurückgesetzt, Max auf ID gesetzt und $Pred$ mit dem neuen Vorgänger überschrieben.

Mattern stellt in [13, S. 76-101] ein weiteres bzgl. der Nachrichtenkomplexität leicht effizienteres Verfahren vor: Das Adoptionsverfahren. Bei diesem Verfahren lau-

fen ebenfalls mehrere Echo-Algorithmen parallel. Existieren beispielsweise zwei Initiatoren A_i und A_j mit $i > j$ und trifft ein Explorer von A_i auf einen Knoten, der bereits von einem Explorer von A_j besucht wurde, wird dieser nicht einfach durchlaufen. Stattdessen wird der gesamte Bereich, der von Explorern von A_j besucht wurde, adoptiert, d.h. alle Echos die in diesem Bereich entstehen werden zu A_i umgeleitet. Dadurch werden anders als in der vorherigen Variante diese Bereiche von Initiatoren mit niedriger ID genutzt und nicht verworfen.

Insgesamt kann so die Nachrichtenzahl verringert werden, die Nachrichtenkomplexität liegt im Mittel jedoch ebenfalls in $O(E \cdot \log k)$. [13, S. 100f.] Ein Nachteil des Adoptionsverfahren ist, dass es langsamer als das einfache Verfahren ist. [13] Aus diesem Grund und der nicht signifikant verbesserten Nachrichtenkomplexität wird das Adoptionsverfahren im Rahmen der Bachelorarbeit nicht näher betrachtet.

3.4. Vergleich

In den letzten Abschnitten wurde gezeigt, dass unterschiedliche Ansätze existieren, Agent Discovery zu ermöglichen. Die drei vorgestellten theoretischen Ansätze lösen das Problem dabei auf unterschiedliche Weise und besitzen daher auch verschiedene Eigenschaften. Beispielsweise ist der Gossip-Ansatz von Jelasity, Montresor und Babaoglu eine sehr einfache Variante, bei der die Agenten durch die Bildung eines Durchschnittswerts Agent Discovery erfüllen können. Ein jedoch entscheidender Nachteil ist, dass der Gossip-Ansatz nicht terminiert. Dadurch ist dieser nicht als Erweiterung von COHDA geeignet. Denn beim Start von COHDA kann nicht festgestellt werden, ob die gefundene Agentenzahl bereits konvergiert und somit endgültig ist.

Beim Join-Ansatz ist dieses Problem nicht vorhanden, da am Ende des Algorithmus nur ein Agent übrig ist, der daran die Terminierung des Algorithmus erkennen kann. Dafür wurde eine Methode verwendet, bei der unter den Agenten Partner gebildet werden und jeweils einer der Partner ausscheidet. Die Nachrichtenkomplexität des Algorithmus ist im besten Fall sehr effizient und liegt in $O(N \cdot G + E)$. Der Worst-Case liegt dagegen bei $O(N^2 + N \cdot G + E)$. Dabei steigt die Nachrichtenkomplexität in nicht vollvermaschten-Topologien zusätzlich, da durch das Entstehen neuer Kanten Nachrichten über mehrere Knoten gesendet werden müssen. Wie effizient der Algorithmus genau ist, lässt sich also anhand theoretischer Überlegungen nicht ohne Weiteres feststellen.

Eine einfache und effiziente Methode ist der Echo-Algorithmus. Durch geschicktes Durchlaufen des Graphen kann dieser mit Explorer- und Echo-Nachrichten Informationen von allen Agenten ermitteln. Es wurde gezeigt, dass der Echo-Algorithmus mit wenigen Änderungen für Agent Discovery eingesetzt werden kann. Die Nachrichtenkomplexität hängt dabei von der Anzahl der Kanten ab und liegt somit in $O(E)$.

Weiterhin lässt sich der Echo-Algorithmus auch bei mehreren Initiatoren ermöglichen und bei geschickter Umsetzung hat dieser Ansatz bei k Initiatoren eine mittlere Nachrichtenkomplexität von $O(E \cdot \log k)$.

Die gezeigten Ansätze sind somit effizienter als der intuitive Ansatz mit Flooding-Technik. Zusätzlich erkennen Join-Ansatz und Echo-Algorithmus das Ende der Agent Discovery ohne überlagerte Terminierungserkennung selbstständig.

4. Termination Detection

In diesem Abschnitt werden verschiedene Algorithmen vorgestellt, die COHDA um Terminierungserkennung erweitern können. Die betrachteten Verfahren zur Terminierungserkennung sind allgemeingültig, also unabhängig von COHDA anwendbar, weshalb im Weiteren allgemein von Basisalgorithmus gesprochen wird und damit die verteilte Berechnung meint, deren Terminierung überwacht werden soll. Im Gegensatz dazu wird der Algorithmus, der die Terminierung überwacht und feststellt, DTD-Algorithmus genannt, was für *Distributed Termination Detection*-Algorithmus steht.

Bevor einige DTD-Algorithmen genauer betrachtet werden können, werden zunächst in Abschnitt 4.1 nötige Definitionen und Eigenschaften von *Termination Detection* eingeführt. Danach wird in Abschnitt 4.2 ein Überblick über verschiedene DTD-Algorithmen geboten und auf ihre Eignung untersucht, COHDA zu erweitern. Drei besonders geeignete DTD-Algorithmen werden in einzelnen Unterkapiteln (4.3, 4.4, 4.5) detailliert erklärt.

4.1. Definition

Angenommen alle Agenten sind unabhängig voneinander, d.h. keine Nachrichten werden zwischen Agenten ausgetauscht, dann ist der Basisalgorithmus terminiert, wenn die Berechnung jedes einzelnen Agenten terminiert ist. Denn fehlt der Nachrichtenaustausch, kann keine Nachricht eine neue Berechnung auslösen. Um Terminierung festzustellen, wäre es ausreichend zu überprüfen, ob alle Agenten mit ihrer Berechnung abgeschlossen sind - in anderen Worten **passiv** sind. Komplizierter wird *Termination Detection* durch den Austausch von Nachrichten. Empfängt ein Agent eine Nachricht, löst dies eine erneute Berechnung aus und der Agent wird wieder **aktiv**. Solange Nachrichten auf den Kanälen zwischen den Agenten vorhanden sind, ist der Basisalgorithmus also nicht terminiert. Es ist folglich nicht ausreichend, ausschließlich festzustellen, ob die Agenten passiv sind. [9] Da nicht direkt überprüfbar ist, ob Kanäle **leer** sind, also keine Nachrichten versenden, müssen DTD-Algorithmen durch geschickte Techniken dieses indirekt ermitteln, indem sie z.B. die Anzahl empfangener und gesendeter Nachrichten vergleichen. [13] Die gemachten Beobachtungen

über Terminierung werden in Definition 1 zusammengefasst.

Definition 1 (Terminierung) *Eine verteilte Berechnung ist terminiert gdw. alle Agenten* ***passiv*** *sind und alle Kanäle* ***leer*** *sind.*

Ein wichtiger Aspekt ist die Nachrichtenkomplexität von DTD-Algorithmen. DTD-Algorithmen benötigen **Kontrollnachrichten**, um ihre Aufgabe zu erfüllen. Je weniger Kontrollnachrichten benötigt werden, umso weniger werden Kanäle beansprucht und der Basisalgorithmus verlangsamt, der die gleichen Kanäle nutzt. Zur Unterscheidung von den Kontrollnachrichten werden die Nachrichten des Basisalgorithmus im Folgenden **Basisnachrichten** genannt. Die notwendige Anzahl der Kontrollnachrichten C hängt unter anderem von der Anzahl der Basisnachrichten M ab. Ein DTD-Algorithmus wird dabei als optimal bzgl. Nachrichtenkomplexität bezeichnet, wenn für die Anzahl der Kontrollnachrichten C gilt: $C = O(M)$. [10]

4.2. Überblick

Seit Anfang der Achtziger Jahre wurden verschiedene DTD-Algorithmen vorgeschlagen und untersucht. (vgl. [10]) Die meisten frühen Ansätze benötigen jedoch eine synchrone Nachrichtenübermittlung und sind deshalb für das asynchrone COHDA ungeeignet. (vgl. [10]) Andere DTD-Algorithmen wie z.B. [4] setzen FIFO-Kanäle voraus. FIFO-Kanäle arbeiten nach dem First-In-First-Out-Prinzip, d.h. Nachrichten kommen in der Reihenfolge an, wie sie losgeschickt werden. Dies verletzt jedoch die Bedingung, dass die Erweiterung auch funktionieren muss, wenn Nachrichten beliebig verzögert sind. (siehe Abschnitt 2.2) Dessen ungeachtet ist der Ansatz [4] interessant, da er einiger der wenigen DTD-Algorithmen ist, die symmetrisch aufgebaut sind, d.h. jeder Agent handelt bzgl. der Terminierungserkennung gleich und jeder Agent kann unabhängig voneinander die Terminierung ermitteln.

Dies ist nicht selbstverständlich. Beispielsweise setzt der Algorithmus [3] voraus, dass nur ein Agent den Basisalgorithmus startet, also nur ein Agent zu Beginn aktiv ist, welcher auch als einziger die Terminierung feststellen kann. Deshalb muss dieser die anderen Agenten über die Terminierung am Ende benachrichtigen. Da bei COHDA nicht nur ein Agent am Anfang aktiv ist, müsste der DTD-Algorithmus dementsprechend angepasst werden. Das dies möglich ist, wurde in [16] gezeigt. Der angepasste Algorithmus in [16] erfüllt außerdem alle Voraussetzungen, die in Abschnitt 2.2 gefordert wurden, und funktioniert dementsprechend bei asynchroner Kommunikation, beliebigen Topologien und verzögerten Nachrichten.

Die meisten Algorithmen stellen jedoch bestimmte Bedingungen an die Kommunikationstopologie. Ein Beispiel hierfür sind *Credit*-Algorithmen, wie [11], [7] und [18]. Bei diesen Algorithmen besitzen Basisnachrichten und aktive Prozesse einen Credit. Insgesamt beträgt die Credit Summe des gesamten Systems 1. Schickt ein Agent eine

Basisnachricht, übergibt er einen Teil seines Credits an diese Nachricht, welche bei Ankunft ihren Credit an den Zielagenten gibt. Wird ein Agent passiv, überträgt er seinen gesamten Credit an einen zentralen Agenten oder an die Umgebung. Hat der zentrale Agent bzw. die Umgebung den gesamten Credit des Systems, also 1, ist der Basisalgorithmus terminiert. Auch wenn diese Variante sehr geschickt und einfach ist und der Credit-Algorithmus in [18] selbst bei Ausfällen von Agenten zuverlässig funktioniert, wird ein zentraler Agent benötigt, der zu allen anderen Agenten eine Verbindung hat. Da COHDA jedoch für beliebige Topologien entwickelt wird, sollen die gewählten DTD-Algorithmen auch für beliebige Topologien verwendbar sein. Ein zentraler Agent oder eine Verbindung zur Umgebung kann also nicht vorausgesetzt werden. Wieder andere Algorithmen setzen eine Ring-Topologie voraus, (vgl. [10]) welche aus diesem Grund auch für COHDA ungeeignet sind.

Neben [16] erfüllt auch ein in [9] von Mahapatra und Dutt vorgestellter Algorithmus die geforderten Voraussetzungen. Bei diesem DTD-Algorithmus wird über den Graph des Multiagentensystems ein Spannbaum gebildet, mit dessen Hilfe die Terminierung festgestellt werden kann. Die Kinderknoten im Spannbaum melden dazu dem Elternknoten, wenn sie passiv werden und alle losgeschickten Nachrichten bei ihrem Ziel angekommen sind. Bei der Terminierung erfährt somit der Wurzelknoten von seinen Kinderknoten, dass alle Knoten passiv sind und alle Nachrichtenkanäle leer sind.

Eine andere mögliche Lösung für Termination Detection ist die Vektor-Methode von Friedemann Mattern (vgl. [13, S. 184-194] und [12]). Bei dieser Methode durchwandert ein Vektor alle Agenten, stoppt bei aktiven Agenten und zählt und vergleicht eingehende und ausgehende Nachrichten. Auf diese Weise kann festgestellt werden, ob die Kanäle leer sind und der Algorithmus terminiert ist. Eine Variante für beliebige Topologien ist die Splitting-Vektor-Methode, welche wie der Algorithmus von Mahapatra und Dutt auf einem Spannbaum arbeitet. [13, S. 192f.]

Um einen genaueren Vergleich zwischen den möglichen Algorithmen zu bieten, werden in den folgenden Unterabschnitten die DTD-Algorithmen aus [16], [9] und [13, S. 192f.] vorgestellt und dabei die Vor- und Nachteile der unterschiedlichen Algorithmen beleuchtet. Neben der Nachrichtenkomplexität wird der Speicheraufwand, die Nachrichtengröße und die theoretische Verzögerung der Terminierungserkennung besonders betrachtet, d.h. wie viel Zeit der DTD-Algorithmus benötigt, um die Terminierung zu erkennen.

4.3. Computation-Tree-Algorithmus

Der hier vorgestellte DTD-Algorithmus von Shavit und Francez [16] erweitert den DTD-Algorithmus [3] von Dijkstra und Scholten, so dass mehrere Agenten zu Beginn aktiv sein können. Im Folgenden wird hierfür der Begriff Initiator eingeführt.

Definition 2 (Initiator) *Ein Agent, der zu Beginn des Basisalgorithmus* ***aktiv*** *ist, ist ein* ***Initiator****. Agenten, die zu Beginn* ***passiv*** *sind, sind Nicht-Initiatoren.*

4.3.1. Computation-Tree-Algorithmus mit einem Initiator

Um den Algorithmus verständlicher zu machen, wird zunächst angenommen, dass wie beim Algorithmus von Dijkstra und Scholten nur ein Initiator existiert und dieser keine Basisnachrichten empfangen kann. Jede Basisnachricht wird im Laufe des Basisalgorithmus mit einer Kontrollnachricht durch den DTD-Algorithmus beantwortet, so dass am Ende die Anzahl der Antworten identisch mit der Anzahl der Basisnachrichten ist. Das bedeutet, wenn der Agent A_j eine Basisnachricht an den Agenten A_i über den Kanal s übermittelt, muss A_i irgendwann eine Antwort an A_j schicken. Jeder Agent A_i besitzt für jeden Kanal s eine Variable c_{is}, die zählt wie viele Basisnachrichten von dem Agenten A_i nicht beantwortet wurden. Zusätzlich existiert eine Variable C_i, die zählt, wie viele Basisnachrichten der Agent insgesamt noch beantworten muss. C_i ist also die Summe der c_{is} für einen Agenten A_i. Analog gibt es für ausgehende Basisnachrichten die Variable D_i, welche zählt, wie viele bisher unbeantwortete Basisnachrichten der Agent A_i gesendet hat. Ein Agent A_1 mit $C_1 = 2$ und $D_1 = 3$ muss also noch zwei Basisnachrichten beantworten und drei Antworten erhalten.

Wenn für alle Agenten $C_i = 0$ und $D_i = 0$ gilt, gibt es keine unbeantworteten Basisnachrichten und folglich sind alle Kanäle leer. Daher lässt sich aus der Definition von Terminierung folgendes Lemma ableiten.

Lemma 1 (Terminierung) *Der Basisalgorithmus ist terminiert gdw.*

$$\forall A_i : C_i = 0 \wedge D_i = 0 \wedge passiv$$

Bisher wurde die Frage offen gelassen, wann ein Agent Antworten auf Basisnachrichten sendet. Ein Agent darf eine Basisnachricht beantworten, wenn Regel 1 gilt.

Regel 1 *Jeder Nicht-Initiator A_i darf eine Antwort senden, gdw.*

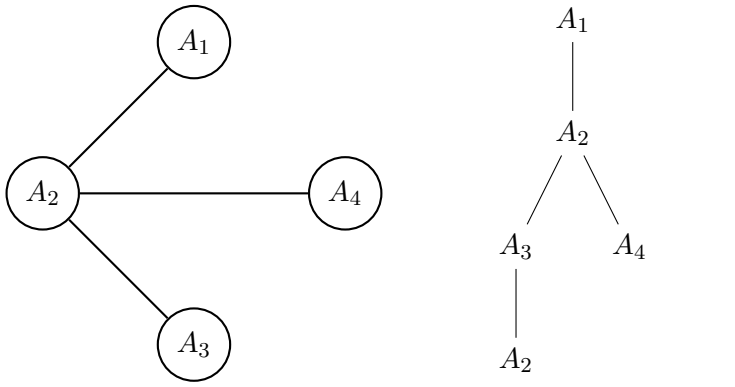
$$(C_i > 1) \vee (C_i = 1 \wedge D_i = 0 \wedge passiv)$$

Das heißt ein Nicht-Initiator darf immer dann eine Antwort auf eine Basisnachricht senden, wenn er mehr als eine Nachricht beantworten muss oder selbst passiv ist, selbst keine unbeantworteten Basisnachrichten gesendet hat und nur eine Basisnachricht beantworten muss. Gilt dabei die erste Bedingung, also dass ein Agent mehrere Basisnachrichten beantworten muss, beantwortet er immer die Nachrichten

zuerst, die als letztes empfangen wurden. (siehe Regel 2) Die älteste Nachricht wird folglich als letztes beantwortet.

Regel 2 *Jeder Agent beantwortet zuerst die neuste, unbeantwortete Basisnachricht.*

Durch diese beiden Regeln ist gewährleistet, dass Lemma 1 genau dann erfüllt ist, wenn der Basisalgorithmus terminiert ist. [16] Eine weitere Folge der beiden Regeln ist, dass die Basisnachrichten des Initiators als letztes beantwortet werden. In diesem Fall sind bereits alle anderen Agenten passiv und alle Kanäle leer. Falls der Initiator ebenfalls passiv ist, ist der Basisalgorithmus terminiert. Der Initiator kann also die Terminierung feststellen, wenn alle seine Basisnachrichten beantwortet wurden.



(a) Graph (b) Berechnungsbaum

Abbildung 3: Beispiel mit vier Agenten

Im Folgenden soll der Verlauf des DTD-Algorithmus anhand eines Beispiels verdeutlicht werden. Die Kanäle und Agenten werden durch die Abbildung 3a dargestellt. Sei A_1 der Initiator eines beliebigen Basisalgorithmus. A_1 sendet eine Basisnachricht zu A_2, dadurch wird $D_1 = 1$ und $C_2 = 1$. A_1 wird passiv, da $D_1 = 1$ ist, weiß der Agent jedoch, dass noch eine Antwort aussteht und der Basisalgorithmus nicht terminiert ist. A_2 ist nun aktiv und sendet jeweils eine Basisnachricht zu A_3 und A_4, somit ist $D_2 = 2$ und A_3 und A_4 aktiv. Auch A_2 ist mit der Berechnung fertig und wird passiv. Dennoch darf A_2 keine Antwort zu A_1 schicken, da $D_2 > 0$ und $C_2 \leq 1$ ist. A_3 wiederum schickt nun eine Basisnachricht an A_2, so dass $C_2 = 2$ ist. A_2 führt nun seine Berechnung durch, will keine Basisnachrichten verschicken und wird danach wieder passiv. Da $C_2 = 2$ gilt, darf A_2 nun eine Antwort schicken. Ohne Regel 2 dürfe A_2 aussuchen, ob er nun eine Antwort an A_3 oder A_1 schickt. Würde A_2 jetzt eine Antwort an A_1 schicken, würde dieser glauben, dass der Algorithmus terminiert ist, obwohl A_4 noch aktiv ist. Daher erzwingt Regel 2, dass zunächst die neuere Nachricht von A_3 beantwortet werden muss. Darauf kann A_3 eine Antwort an A_2 schicken und nachdem A_4 passiv wird, würde er ebenso eine Antwort an A_2

schicken. A_2 kann die letzte Antwort an A_1 schicken und A_1 stellt durch den Erhalt der letzten Antwort die Terminierung fest. An dem Beispiel wird auch deutlich, wieso diese Art des DTD-Algorithmus Computation-Tree-Algorithmmus genannt wird. Denn betrachtet man den Berechnungsbaum des Basisalgorithmus in Abbildung 3b, ist zu erkennen, dass der DTD-Algorithmus die Basisnachrichten von den Blättern aus gesehen beantwortet. Zuerst wurden die Antworten von A_2 an A_3 und A_4 an A_2 gesendet und als letztes die Basisnachricht von A_1 beantwortet.

Um die Voraussetzung zu entfernen, dass der Initiator keine Basisnachrichten empfangen darf, muss zusätzlich Regel 3 eingeführt werden.

Regel 3 *Ein Initiator A_i darf eine Antwort senden, wenn $(C_i > 0)$ erfüllt ist.*

Mit dieser Regel wird der DTD-Algorithmus für Basisalgorithmen mit einem Initiator vollständig beschrieben. Da der Algorithmus auf jede Basisnachricht genau einmal antwortet, ist die Anzahl der Kontrollnachrichten mit der Anzahl der Basisnachrichten identisch. Der DTD-Algorithmus hat also optimale Nachrichtenkomplexität. Ein Nachteil des Algorithmus ist, dass es lange dauern kann, bis der DTD-Algorithmus die Terminierung erkennt. Denn ist der Berechnungsbaum des Basisalgorithmus sehr tief und beinhaltet der längste Pfad viele Agenten, ist es möglich, dass nachdem der letzte Agent passiv wird, eine große Anzahl an Antworten geschickt werden muss. Dies verzögert die Terminierungserkennung. Im Worst-Case erkennt die Wurzel die Terminierung mit einer Verzögerung von $O(N)$, falls der längste Pfad des Berechnungsbaums alle Agenten beinhaltet. Die Verzögerung wächst dabei über N nicht hinaus, da jeder Agent bereits bei $(C_i > 1)$ Nachrichten schicken darf.

Der Speicheraufwand ist ohne Optimierung hoch. Denn jeder Agent muss für jede Kante eine Variable c_{is} speichern und eine Variable D_i. (C_i ergibt sich aus der Summe der c_{is}) Darüber hinaus muss er speichern, in welcher Reihenfolge alle Nachrichten angekommen sind. Der Speicheraufwand lässt sich stark reduzieren, wenn man bedenkt, dass ein Agent immer Antworten schicken darf, wenn $C_i > 1$ gilt. Daraus folgt, dass sich der Agent nur merken muss, von welcher Kante die älteste Nachricht gekommen ist. Alle anderen Nachrichten kann er sofort beantworten. Jeder Agent benötigt also nur D_i und eine Variable c_{old}, in welcher der Agent gespeichert wird, von dem die erste Basisnachricht kam. Der Speicheraufwand ist folglich für alle Systeme und alle Agenten konstant. Ebenso ist die Nachrichtengröße klein und konstant, da Kontrollnachrichten nur die Information beinhalten müssen, dass sie eine Antwort sind.

4.3.2. Computation-Tree-Algorithmus mit mehreren Initiatoren

Legt man fest, dass der DTD-Algorithmus eine beliebige Anzahl an Initiatoren haben kann, kann kein Agent ohne weiteres erkennen, dass der Basisalgorithmus terminiert

ist. Um den DTD-Algorithmus anzupassen, müssen wir die Sichtweise auf den Basisalgorithmus ändern. Wenn mehrere Initiatoren vorhanden sind, existiert nicht nur ein Berechnungsbaum, sondern für jeden Initiator einer. Jeder von einem Initiator aus gestartete Berechnungsbaum lässt sich als eigener Teilalgorithmus des Basisalgorithmus betrachten. Sind alle Teilalgorithmen terminiert, ist auch der Basisalgorithmus terminiert. Jeder Initiator stellt dabei die Terminierung seines Teilalgorithmus fest. Ist ein Initiator A_i passiv und wurden alle seine Basisnachrichten beantwortet ($D_i = 0$), dann ist sein Teilalgorithmus terminiert. Von diesem Moment an, kann der Agent A_i wieder als Nicht-Initiator betrachtet werden, so dass für ihn wieder Regel 1 gilt.

Regel 4 *Ein Initiator A_i, für den gilt $D_i = 0 \wedge passiv$, wird zu einem Nicht-Initiator.*

Wenn also keine Initiatoren mehr existieren, sind alle Teilalgorithmen terminiert und somit ist auch der Basisalgorithmus terminiert. Anders als beim DTD-Algorithmus für einen Initiator können in diesem Fall die Initiatoren jedoch nicht die Terminierung des Basisalgorithmus feststellen, wenn alle ihre Basisnachrichten beantwortet wurden. Denn ein Initiator kann nicht wissen, ob noch andere aktive Teilalgorithmen laufen.

Hierfür schlagen Shavit und Francez eine einfache Lösung vor, welche die Agentenzahl des Systems benötigt. Sie verwerfen diese Idee jedoch, weil die Agentenzahl nicht in allen Systemen gegeben ist. Da im Kontext von COHDA den Agenten durch die Erweiterung Agent Discovery die Agentenzahl bekannt ist, kann diese Möglichkeit jedoch verwendet werden. Jeder Agent A_i sendet dazu an alle Nachbarn eine $Finished_i$-Nachricht, wenn er zum Nicht-Initiator wird, wobei die $Finished_i$-Nachricht die eindeutige ID von A_i erhält. Wie in Abschnitt 2 erwähnt, ist solch eine ID vorhanden. Agenten merken sich, dass sie eine $Finished_i$ Nachricht erhalten haben und falls sie nicht bereits vorher eine $Finished_i$ Nachricht erhalten haben, leiten sie diese an alle Nachbarn weiter. Somit weiß jeder Agent mit einer gewissen Verzögerung, dass A_i kein Initiator mehr ist. Nehmen wir an, dass alle Agenten Initiatoren sind, erhält jeder mit der Zeit von jedem anderen Agenten eine $Finished$-Nachricht. Da jeder Agent die Anzahl der Agenten kennt, weiß er dann, dass der Basisalgorithmus terminiert ist.

Um dieses möglich zu machen, muss jeder Agent als Initiator betrachtet werden, auch wenn er zu Beginn passiv ist. Diese Änderung der Definition von Initiator ist möglich, da durch die Regel 4 jeder passive Initiator sofort zu einem Nicht-Initiator wird. Dadurch ist gewährleistet, dass am Ende jeder Agent von jedem anderen Agenten eine $Finished$-Nachricht erhält.

Durch die Anpassungen hat man einen DTD-Algorithmus, der nicht nur für beliebige Topologien funktioniert, sondern auch mehrere Initiatoren ermöglicht. Zusätzlich ist der DTD-Algorithmus symmetrisch. Jeder Agent führt also letztendlich den gleichen Quellcode des Algorithmus aus. Die Kehrseite der Anpassungen ist, dass der DTD-Algorithmus nicht mehr optimale Nachrichtenkomplexität hat. Ein Agent schickt zu allen anderen Agenten *Finished*-Nachrichten, dabei wird jeder Kanal einmal benutzt. Bei N Agenten und E Kanälen werden insgesamt also $N \cdot E$ Finished Nachrichten verschickt. Die Nachrichtenkomplexität liegt also in $O(M+N \cdot E)$. Ebenso ist der Speicheraufwand nicht mehr konstant, da jeder Agent sich merken muss, ob er bereits eine $Finished_i$-Nachricht erhalten hat. Der Speicheraufwand liegt also in $O(N)$, wobei N die Anzahl der Agenten ist.

Die Verzögerung der Teminierungserkennung ist $O(N + D)$, wobei D der Durchschnitt des Graphen ist, d.h. der größte Abstand zwischen allen Knoten des Graphen. Im Worst-Case muss dieser Abstand von den *Finished*-Nachrichten am Ende durchlaufen werden.

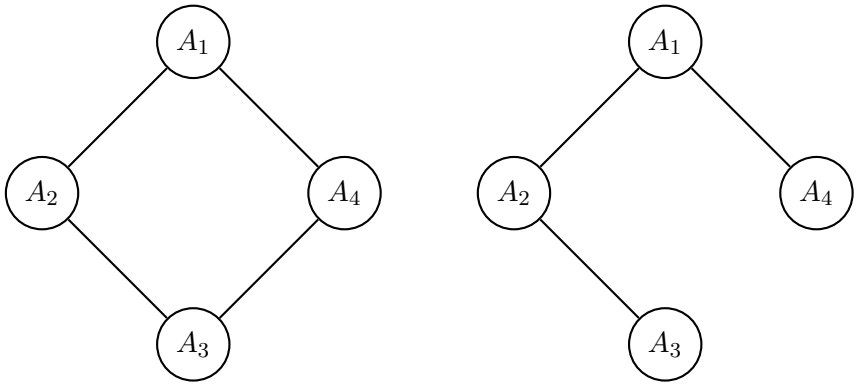
4.4. Static-Tree-Algorithmus

Der Static-Tree-Algorithmus [9] wurde von Mahapatra und Dutt entwickelt. Anders als der Computation-Tree-Algorithmus kann dieser nicht ohne Vorbereitung starten, da er einen Spannbaum als Kontrollstruktur benötigt. Der Spannbaum kann z.B. mittels eines Echo-Algorithmus aufgebaut werden (vgl. Abschnitt 3.3), indem der Initiator des Echo-Algorithmus als Wurzel markiert wird und die Kanten zu den Vorgängern die Kanten des Spannbaums bilden. Beispielsweise ist ein Spannbaum zum Graph in Abbildung 4a in Abbildung 4b dargestellt. Der Basisalgorithmus kann jedoch immer noch Basisnachrichten über Kanäle schicken, die nicht im Spannbaum existieren. Der Spannbaum wird nur für den DTD-Algorithmus benötigt. Im Folgenden sind mit Kinder und Eltern eines Agenten die Kinder- und Elternknoten im Spannbaum gemeint.

Wie in dem Computation-Tree-Algorithmus gibt es auch in diesem Algorithmus Antwort-Kontrollnachrichten. Ein passiver Agent wird dabei als **frei** bezeichnet, wenn alle seine Nachrichten beantwortet wurden.

Definition 3 (Frei) *Ein passiver Agent, der keine unbeantworteten Basisnachrichten hat, ist **frei**.*

Zunächst wird der Fall betrachtet, dass keine Basisnachrichten versendet werden, also auch keine Antworten notwendig sind. In diesem Fall sind alle passiven Agenten auch frei. Ziel ist es nun, dass der Wurzelknoten, Terminierung feststellen kann. Dazu wird die Kontrollnachricht *Stop* eingeführt. Immer wenn ein Agent frei ist und von allen Kindern *Stop*-Nachrichten erhalten hat, schickt er eine *Stop*-Nachricht an



(a) Graph (b) Spannbaum

Abbildung 4: Beispiel mit vier Agenten

seinen Elternknoten. Für Blätterknoten ist es ausreichend, wenn sie frei werden, da sie keine Kinder haben. Ohne Basisnachrichten wäre dies ausreichend, damit der Wurzelknoten Terminierung feststellen kann. Denn wenn der Wurzelknoten frei ist und von allen Kindern *Stop*-Nachrichten erhalten hat, bedeutet dies, dass auch alle Kinder frei sind. Der Basisalgorithmus ist also terminiert.

Damit der Algorithmus auch mit Basisnachrichten funktioniert, wird die oben erwähnte Antwort-Kontrollnachricht und eine zusätzliche *Resume*-Kontrollnachricht eingeführt. Schickt ein Agent A_i eine Basisnachricht zu A_j und ist A_j nicht frei, hat also noch keine *Stop*-Nachricht zum Elternknoten geschickt, dann schickt A_j sofort eine Antwortnachricht an A_i. Ist A_i passiv, wird er dann also frei, wenn keine weiteren Antworten ausstehen. Durch die Antwortnachrichten wird also erreicht, dass A_i keine *Stop*-Nachrichten schicken kann, bevor seine Basisnachricht erfasst wurde. Ansonsten könnte es passieren, dass der Wurzelknoten Terminierung feststellt, obwohl noch eine Basisnachricht vorhanden ist.

Schwieriger ist der Fall, wenn A_j bereits frei ist. Durch die Basisnachricht wird A_j nämlich wieder aktiv und somit nicht frei. In diesem Fall darf A_j nicht direkt eine Antwort an A_i schicken, sondern A_j schickt zunächst eine *Resume*-Nachricht an seinen Elternknoten. Die *Resume*-Nachricht hebt eine zuvor geschickte *Stop*-Nachricht von A_j auf und der Elternknoten merkt sich, dass noch eine *Stop*-Nachricht von A_j aussteht. Ist der Elternknoten selbst schon als frei markiert, schickt er wiederum eine *Resume*-Nachricht zu seinen Elternknoten. Diese *Resume*-Nachricht wird dann solange von einem Knoten zum nächsten Elternknoten geschickt, bis ein nicht freier Agent gefunden wurde. Dabei hebt er alle zuvor auf diesem Weg gesendeten *Stop*-Nachrichten auf. Wurde ein nicht freier Agent gefunden, wird eine Antwort-Nachricht auf den gleichen Weg abwärts im Baum zu A_j geschickt, welcher die Antwort zu A_i weiterleitet. Dadurch wird gewährleistet, dass ein Elternknoten nicht fälschlicherwei-

se denkt, dass alle Kinderknoten passiv sind.

Zum Beispiel könnte der Agent A_4 in Abbildung 4a eine Basisnachricht an A_3 schicken, nachdem dieser schon frei geworden ist. A_3 wird wieder nicht frei und benachrichtigt A_2 mittels *Resume*-Nachricht, dass ein vorher gesendetes *Stop* aufgehoben werden muss. Angenommen A_2 ist nicht frei, dann schickt er die Antwort für A_4 an A_3, welcher diese an A_4 weiterleitet.

Wie oben bereits geschrieben, erkennt der Wurzelknoten Terminierung daran, dass er selbst frei ist und von allen Kindern *Stop*-Nachrichten erhalten hat. Nun schickt er eine Terminierungsnachricht an seine Kinder, welche diese wiederum ebenfalls weiterleiten, bis auch alle Blätter wissen, dass der Basisalgorithmus terminiert ist.

Die Worst-Carse Nachrichtenkomplexität des Algorithmus ist $O(M \cdot D + N)$, wobei M die Anzahl der Nachrichten und N die Anzahl der Agenten ist. [9] D ist der größte Abstand zwischen den Knoten eines Graphen. Beispielsweise gilt beim Graph in Abbildung 4a $D = 2$ und beim Spannbaum in Abbildung 4b ist der größte Abstand zwischen A_3 und A_4, also $D = 3$. Die Verzögerung des Terminierungsalgorithmus hängt hier von der Entfernung vom Wurzelknoten zu den Blättern ab. Im schlimmsten Fall ist dies der größte Abstand im Graph, also der Durchschnitt D des Graphen. Wird der Blattknoten in diesem Fall passiv, muss die *Stop*-Nachricht über alle Kanten hoch propagiert werden. Die Worst-Case Verzögerung liegt also in $O(D)$. Welches eine sehr gute $Worst-Case$ Verzögerung ist, weil die Anzahl der Kanten in den meisten Fällen verglichen mit der Anzahl an Basisnachrichten klein ist. Der Speicheraufwand ist in vielen Topologien gering, da er vom Grad der Knoten abhängt. Denn ein Agent muss für jeden Kinderknoten speichern, ob dieser bereits ein *Stop* gesendet hat. Bei einem Maximalgrad des Graphen von G liegt der Speicheraufwand also in $O(G)$.

Dagegen liegt die Worst-Case Nachrichtengröße in $O(D)$, da sich $Resume-$ und Antwort-Nachrichten merken müssen, welchen Pfad sie genommen haben, bzw. zurücklaufen müssen. Dieser Pfad ist im ungünstigsten Fall D lang. Alternativ würde es reichen, wenn in Kontrollnachrichten nur der ursprüngliche Absender gespeichert wird. In diesem Fall müssten die Agenten aber wissen, welche Agenten sich in welchem Teilbaum ihrer Kinderknoten befinden, so dass die Kontrollnachrichten den richtigen Pfad zurücknehmen. Diese Informationen müssten jedoch ermittelt werden und würden den Speicheraufwand erhöhen.

4.5. Splitting-Vektor-Methode

Damit die Splitting-Vektor-Methode einsetzbar ist, müssen den Knoten einige Informationen über die Struktur des Graphen bekannt sein. Daher wird zunächst in Abschnitt 4.5.1 beschrieben, wie die nötigen Informationen ermittelt werden können, bevor anschließend in Abschnitt 4.5.2 die eigentliche Terminierungserkennung

erläutert wird.

4.5.1. Vorbereitung

Analog zum Static-Tree-Algorithmus benötigt die Splitting-Vektor-Methode einen Spannbaum der über die Struktur das Multiagentensystems überlagert ist. Weiterhin braucht jeder Agent einen lokalen Vektor V mit einem Eintrag für jeden Agenten im System und eine Nummer, die seine Stelle im Vektor anzeigt. In diesem Vektor wird später gespeichert, zu welchem Agenten der Knoten Nachrichten geschickt hat. Dazu muss er die Nummern aller Nachbarn kennen, damit er die Anzahl der Nachrichten an der richtigen Stelle speichern kann. Die Agenten müssen zusätzlich wissen, welche Nummern die Agenten im Teilbaum eines Kinderknoten besitzen. (vgl. [13, S. 192f.])

Diese Informationen können ebenso wie der Spannbaum mit einem Echo-Algorithmus ermittelt werden. Mit Hilfe des Agent Discovery Echo-Algorithmus kann für jeden Knoten im Spannbaum ermittelt werden, wie groß der Teilbaum eines Kindes ist, d.h. aus wie vielen Agenten der Teilbaum eines Kindes besteht. Denn wenn ein Knoten ein Echo von einem Kind empfängt, entspricht die übertragene Zählvariable der Größe des Teilbaums. (vgl. 3.3) Anders als bei der ursprünglichen Version des Echo-Algorithmus für Agent-Discovery speichert ein Agent jede Zählvariable, die er von einem Echo empfängt. In Abbildung 5 ist dies an einem Beispiel dargestellt. Nach dem Durchlauf des Echo-Algorithmus sind im zweiten Schritt die Größen der Teilbäume für die Kinder gespeichert. Gleichzeitig kann, wie in Abschnitt 4.4 beschrieben, mit dem Echo-Algorithmus der benötigte Spannbaum gebildet werden.

Jedem Agenten muss nun eine Nummer von 1 bis zur Agentenzahl N zugeordnet werden. Die Agentenzahl wurde dabei durch den Echo-Algorithmus ermittelt und ist der Wurzel bekannt. Zunächst teilt die Wurzel sich die Nummer 1 zu. Die restlichen $N-1$ Nummern werden in K Intervalle geteilt, wobei K die Anzahl der Kinder der Wurzel ist. Die Größe der Intervalle entspricht dabei der Größe des Teilbaums eines Kindes. In Abbildung 5 hat die Wurzel beispielsweise zwei Kinder mit den Größen 2 und 1, so dass die Wurzel dem ersten Kind das erste Intervall $[2, 3]$ zuordnet und dem zweiten Kind das Intervall $[4, 4]$. Danach sendet die Wurzel den Kinderknoten ihre jeweiligen Intervalle und die Anzahl der Agenten. Die Kinder selbst ordnen sich die erste Nummer des Intervalls zu und teilen das restliche Intervall wieder entsprechend der Anzahl der Kinder und der Größe der Teilbäume auf. Dann werden die Intervalle und die Anzahl der Agenten wiederum zu den Kindern gesendet. Dies wird solange wiederholt, bis allen Knoten Nummern zugeordnet wurden.

Da durch diesen Vorgang jeder Agent die Agentenzahl N kennt, kann jeder den benötigten Vektor V mit N Einträgen erstellen. Den Agenten fehlen folglich nur noch die Nummern ihrer Nachbarn. Deshalb schickt jeder Agent, sobald ihm eine Nummer zugeordnet wurde, seinen Nachbarn seine Nummer. Somit lassen sich alle

nötigen Informationen für die Splitting-Vektor-Methode ermitteln.

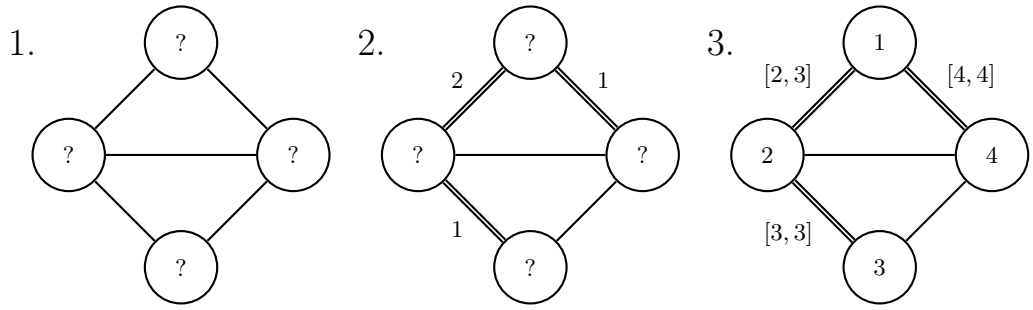


Abbildung 5: Die Abbildung stellt die Schritte zur Ermittlung der Nummern und Intervalle dar. Der oberste Knoten ist der Wurzelknoten. Der Spannbaum ist mit doppelten Linien eingezeichnet. Die Kantenbeschriftung in Schritt 2 beziffert die Größen der Teilbäume und in Schritt 3 die Intervalle. Schritt 1: Vor dem Echo-Algorithmus, Schritt 2: Nach dem Echo-Algorithmus, Schritt 3: Nach dem Verteilen der Nummern

4.5.2. Algorithmus

Nachdem die nötigen Strukturen mit den Methoden aus dem vorherigen Abschnitt aufgebaut worden sind, kann Matterns Splitting-Vektor-Methode aus [13, S. 192f.] für die Terminierungserkennung eingesetzt werden. Wie oben erwähnt, besitzt jeder Agent einen Vektor V zum Zählen der Basisnachrichten. Immer wenn ein Agent mit der Nummer i eine Basisnachricht an einen Agenten j schickt, erhöht der Agent i die j.-Stelle seines Vektors um eins. Empfängt der Agent j eine Basisnachricht, erniedrigt er unabhängig vom Sender der Basisnachricht die j.-Stelle seines Vektors. Im linken Graph in Abbildung 6 hat z.B. der Agent 1 jeweils eine Nachricht an Agent 2 und Agent 4 geschickt. Da beide Nachrichten bereits angekommen sind, ist sowohl bei Agent 4 eine -1 an der 4. Stelle seines Vektors als auch bei Agent 2 eine -1 an der 2. Stelle eingetragen. Die Nachricht von Agent 3 an Agent 4 ist dagegen noch nicht angekommen, so dass die 4. Stelle bisher nicht auf -2 erniedrigt wurde.

Wenn der Wurzelknoten vermutet, dass der Basisalgorithmus terminiert ist, startet er einen Terminierungstest. Anders als bei den beiden vorherigen DTD-Algorithmen startet die eigentliche Terminierungserkennung also verzögert und es werden vor dem Start des Terminierungstests keine Kontrollnachrichten benötigt. Die Basisnachrichten müssen jedoch von Anfang an in den Vektoren gezählt werden. Es sind mehrere Kriterien denkbar, an denen die Wurzel erkennen kann, dass ein geeigneter Zeitpunkt für ein Terminierungstest gekommen ist. Beispielsweise könnte er die Dauer messen, wann die letzte Basisnachricht bei ihm angekommen ist und übertrifft diese Dauer einen bestimmten Schwellenwert, startet die Wurzel einen Terminierungstest.

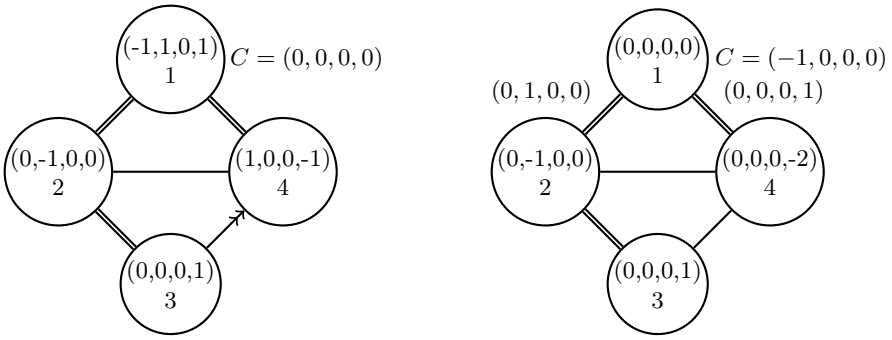


Abbildung 6: Terminierungstest mit der Splitting-Vektor-Methode. Doppelpfeile zeigen Basisnachrichten und doppelinige Kanten sind die Kanten des Spannbaums.

Ein anderes Kriterium im Kontext von COHDA wäre zum Beispiel die Qualität der besten Lösung. Denn solange die Qualität nicht einen bestimmten Wert erreicht hat, ist eine Terminierung unwahrscheinlich. Bei solchen Kriterien muss darauf geachtet werden, dass es nicht passieren darf, dass die Wurzel niemals einen Terminierungstest startet.

Der Terminierungstest erfolgt folgendermaßen: Neben seinen lokalen Vektor V besitzt die Wurzel zu Beginn einen weiteren Vektor C mit der gleichen Größe wie V, der bei Beginn des Basisalgorithmus als Nullvektor initialisiert wurde. Der Vektor C soll den Spannbaum des Graphen durchlaufen und alle gespeicherten Informationen in den Vektoren der Agenten sammeln. Dazu addiert die Wurzel seinen Vektor V zum Vektor C hinzu und setzt V auf den Nullvektor. Anschließend wird der Vektor C den Kindern geschickt. Dabei sind jedoch alle Stellen des gesendeten Vektors 0, die nicht in den vorher ermittelten Intervallen der Kinder liegen. Der Vektor C wird also in mehrere Vektoren aufgeteilt, welche die Werte der Intervalle übertragen. Der Teil von C mit der eigenen Nummer des Agenten bleibt beim Agenten zurück. Dieser erste Schritt ist im rechten Graph in Abbildung 6 für das vorherige Beispiel dargestellt.

Empfängt ein Knoten j den Vektor C von seinem Elternknoten, wartet C so lange bei diesem Knoten, wie der Knoten j aktiv ist oder die Summe der j. Stelle von C und V größer als 0 ist. Denn im ersten Fall ist der Algorithmus noch nicht terminiert, da der Agent aktiv ist, und im zweiten Fall ist noch eine Basisnachricht zum Agenten j unterwegs. Sind beide Bedingungen falsch, wird V zu C hinzuaddiert und V mit dem Nullvektor überschrieben. Besitzt der Agent Kinder, teilt er C für die Intervalle der Kinder auf und schickt den Kindern den entsprechenden Teil des Vektors. Sind keine Kinder vorhanden, sendet der Agent den gesamten Vektor zum Elternknoten. Ein Agent, der von allen Kindern C empfangen hat, addiert den zu-

rückgebliebenen und die empfangenen Vektoren zu einem Vektor C zusammen und schickt diesen Vektor zu seinem Elternknoten, falls er nicht der Wurzelknoten ist. Ist der aufsummierte Vektor C am Ende bei der Wurzel der Nullvektor, bedeutet dies, dass alle gesendeten Nachrichten empfangen wurden und alle Agenten passiv sind. Der Basisalgorithmus ist also terminiert und die Wurzel kann die anderen Agenten über die Terminierung informieren. Falls C nicht der Nullvektor ist, lässt sich keine Aussage über die Terminierung machen und der Terminierungstest muss zu einem späteren Zeitpunkt erneut gestartet werden.

Ein großer Vorteil des Algorithmus ist die geringe Anzahl an Kontrollnachrichten eines Terminierungstests. Denn für einen Terminierungstest durchlaufen die Vektoren alle Kanten des Spannbaums zweimal und somit ist die Anzahl der Kontrollnachrichten unabhängig von der im Normalfall großen Anzahl der Basisnachrichten. Die Gesamtzahl der Kontrollnachrichten hängt von der Anzahl der Terminierungstests ab und die Anzahl der Terminierungstests hängt wiederum von den gewählten Kriterien ab. Werden die Kriterien so gewählt, dass der Terminierungstest sehr häufig durchgeführt wird, kann die Anzahl der Kontrollnachrichten stark steigen. Wählt man jedoch Kriterien, die selten einen Terminierungstest auslösen, ist die Verzögerung bis zur Terminierungserkennung in der Regel höher. Somit lässt sich durch geschickte Wahl der Kriterien steuern, wie das Verhältnis der Nachrichtenkomplexität zur Verzögerung der Terminierungserkennung ist. Der Algorithmus lässt sich also flexibel anpassen. Jedoch besitzt die Splitting-Vektor-Methode besonders in großen Systemen zwei entscheidende Nachteile: Sowohl der Speicheraufwand für die Vektoren als auch die Größe der Kontrollnachrichten hängen von der Agentenanzahl ab.

4.6. Vergleich

Die drei unterschiedlichen Algorithmen: Computation-Tree-Algorithmus (CTA), Static-Tree-Algorithmus (STA) und die Splitting-Vektor-Methode (SVM) werden im Folgenden verglichen und bewertet. Dabei werden die Eigenschaften benötigte Informationen, Nachrichtengröße, Speicheraufwand, Nachrichtenkomplexiät und Verzögerung der Terminierungserkennung betrachtet. Bei dem Vergleich wird davon ausgegangen, dass die Anzahl der Basisnachrichten größer als die der Agenten und Kanten ist. Diese These ist wohl für die meisten Algorithmen zutreffend, da davon ausgegangen werden kann, dass jeder Teilnehmer mindestens eine Basisnachricht an einen Nachbar schickt. Ein Überblick über die Vor- und Nachteile der Algorithmen zeigt Tabelle 1. Im Weiteren werden die einzelnen Aspekte detailliert erläutert.

Beim Computation-Tree-Algorithmus muss im Gegensatz zu den anderen Algorithmen kein Spannbaum oder andere Informationen ermittelt werden. Seine Nachrichtengröße ist sehr klein und konstant. Der Speicheraufwand bei CTA liegt in $O(N)$ und hängt damit von der Größe des Systems ab. Ebenso hängt die Verzögerung der

	Informationen	Nachrichtengröße	Speicheraufw.	Nachrichtenkompl.	Verzögerung
CTA	+	+	−	?	−
STA	∘	∘	+	?	+
SVM	−	−	−	?	?

Tabelle 1: Vergleich der Algorithmen

Terminierungserkennung von der Anzahl der Agenten ab und liegt im Worst-Case in $O(N + D)$. Durch die Erweiterung von CTA auf mehrere Initiatoren ist die Nachrichtenkomplexität nicht mehr optimal, also $O(M)$, sondern liegt in $O(M + N \cdot E)$.

Dagegen hat der Static-Tree-Algorithmus eine Worst-Case-Nachrichtenkomplexität von $O(M \cdot D + N)$. Da es sich hierbei aber um eine Abschätzung nach oben handelt, ist ein genauer Vergleich zwischen der Nachrichtenkomplexität von STA und CTA mit mehreren Initiatoren schwer möglich. Ein Vorteil von STA ist die kurze Verzögerung zwischen Terminierung und Terminierungserkennung. Denn die Worst-Case Verzögerung liegt in $O(D)$. Auch bietet STA einen vergleichsweise kleinen Speicheraufwand von $O(G)$ und eine Nachrichtengröße von $O(D)$. Um zu funktionieren, benötigt STA einen Spannbaum, der deshalb bereits vor dem Start des Basisalgorithmus ermittelt werden muss.

Die Splitting-Vektor-Methode benötigt ebenfalls einen Spannbaum und darüber hinaus noch zusätzliche Informationen über die Kinderknoten. Ein Vergleich zwischen der Nachrichtenkomplexität von SVM zu den anderen Algorithmen ist schwer möglich, da er von der Anzahl der Terminierungstests abhängt. Auch die Verzögerung der Terminierungserkennung lässt sich schwer einschätzen. Im besten Fall wird ein Terminierungstest exakt bei der Terminierung gestartet und braucht dann wie STA eine Verzögerung von $O(D)$, da der Vektor den Spannbaum durchlaufen muss. In den meisten Fällen wird die Verzögerung jedoch länger sein, da der exakte Terminierungszeitpunkt im Voraus nicht feststellbar ist. Anders als bei den beiden anderen Algorithmen ist bei SVM die Nachrichtengröße nicht in allen Systemen gleich, sondern entspricht der Anzahl der Agenten. Der Speicheraufwand von $O(N)$ entspricht dem Speicheraufwand von CTA.

Insgesamt hat kein Algorithmus in allen Eigenschaften den Vorzug. Besonders die Nachrichtenkomplexität lässt sich in der Theorie schwer vergleichen und benötigt eine nähere praktische Untersuchung, um ein abschließendes Urteil über die Eignung der Algorithmen zu bilden. Auch lässt sich schwer vorhersagen, ob sich im Kontext von COHDA Kriterien für die Splitting-Vektor-Methode finden lassen, so dass sowohl Nachrichtenkomplexität als auch Verzögerung der Terminierungserkennung gering sind.

5. Entwurf

In den vorherigen Abschnitten wurde gezeigt, dass Algorithmen existieren, die COHDA um Agent Discovery und Termination Detection erweitern können. Jedoch wurde die Frage unbeantwortet gelassen, wie diese Algorithmen in das Multiagentensystem integriert werden können, in welchem COHDA implementiert ist. Daher wird in diesem Abschnitt ein Konzept für eine mögliche Implementierung der Erweiterungen vorgestellt. Ein Ziel des Konzepts ist keine großen Änderungen an der bestehenden Implementierung von COHDA und dem verwendeten Multiagentensystem vorzunehmen, so dass COHDA unabhängig von den Erweiterungen weiterentwickelt werden kann. Zusätzlich kann durch die Trennung von COHDA und den Erweiterungen einerseits der flexible Einsatz der Erweiterungen gezeigt werden, d.h. dass sie auch für andere Algorithmen ohne viele Anpassungen eingesetzt werden können, und andererseits ist durch die Trennung die Evaluation der einzelnen Erweiterungen einfach möglich. Bevor das Konzept näher beschrieben wird, wird in Abschnitt 5.1 das bereits vorhandene Multiagentensystem und die Implementierung von COHDA erläutert, auf die das Konzept aufbaut. Anschließend wird der Aufbau und die Funktionsweise der Erweiterungen in Abschnitt 5.2 verdeutlicht. Abschließend wird in Abschnitt 5.3 eine mögliche Umsetzung der implementierten Agent Discovery und Termination Detection Algorithmen beschrieben. Insgesamt bildet das Konzept die Basis für die im nächsten Kapitel folgende Evaluation.

5.1. Multiagentensystem

Das verwendete Multiagentensystem ist in Python implementiert und kann ein Netzwerk mit einer großen Anzahl von verteilten Prozessen simulieren. Nachrichten, die von einem Prozess zu einem anderen Prozess versendet werden, haben eine simulierte Verzögerungszeit und bilden somit reelles Verhalten nach. Im Vergleich zu realen Systemen bietet die Simulation durch das Festlegen eines Startwerts für den Zufallszahlengenerator den Vorteil von deterministischen Verhalten. Dadurch lässt sich ein Simulationsdurchgang mit exakt gleichen Ablauf wiederholen. [6] Dies erleichtert zum Beispiel den Vergleich zwischen verschiedenen DTD-Algorithmen, da diese unter gleichen Bedingungen getestet werden können. Weiterhin bietet das Multiagentensystem die Möglichkeit beliebige Topologien mit unterschiedlicher Agentenzahl zu simulieren. Das Multiagentensystem erstellt dazu beim Start eine vorher festgelegte Anzahl von Agenten, startet für jeden Agenten einen Prozess mit einer ID und einer Adresse und teilt den Agenten ihre Nachbarschaft mit. Welchen Code die Agenten ausführen, kann ebenfalls vor dem Start festgelegt werden.

Im Fall von COHDA wird das Verhalten von der Klasse *AgentCOHDA* mit den Methoden *notify*, *update* und *run* definiert. (siehe Klassendiagramm in Abbildung 7)

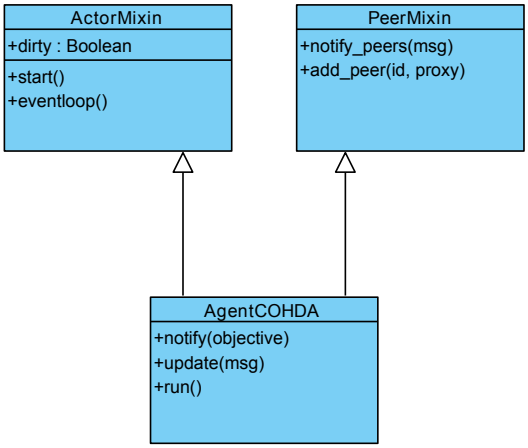


Abbildung 7: Agentenklassen, die COHDA realisieren

notify wird ausgeführt, wenn der Agent das globale Ziel von der Umgebung empfängt. Dahingegen nimmt die *update*-Methode alle Nachrichten von anderen Agenten entgegen und aktualisiert mit den empfangenen Informationen Σ und Σ^*. Das restliche Verhalten, d.h. das Suchen und Versenden einer besseren Lösung, wird in der Methode *run* definiert.

Die Klasse *AgentCOHDA* besitzt die beiden Oberklassen *ActorMixin* und *PeerMixin*. *ActorMixin* steuert die Ausführung der *run*-Methode. Dazu besitzt sie eine Methode zum Starten einer Dauerschleife, die bei der Erstellung des Agenten ausgeführt wird. Die Dauerschleife ruft immer dann *run* auf, wenn der boolesche Wert dirty von der *update*-Methode auf wahr gesetzt worden ist. Dies ist der Fall, wenn sich durch den Empfang einer Nachricht Σ oder Σ^* geändert hat. Die *run*-Methode setzt dirty bei Ausführung wieder auf falsch. Durch dieses Vorgehen ist die Methode zum Empfang von Nachrichten und zum Versenden von Nachrichten getrennt. Vereinfacht ausgedrückt ist die update-Methode ein Nachrichtenpuffer, welcher die Nachrichten empfängt und die run-Methode der Prozess des Agenten, der in bestimmten Zeitintervallen überprüft, ob sich was geändert hat. Die run-Methode reagiert somit mit einer gewissen Verzögerung auf Änderungen. Dies soll das Verhalten von technischen Geräten simulieren, die ebenfalls nicht ohne Verzögerung reagieren können.

Die zweite Oberklasse besitzt Methoden, die Agenten zur Nachbarschaft des Agenten hinzufügen oder Nachrichten an Nachbarn versenden. Ersteres wird wie oben beschrieben bereits beim Start des Multiagentensystems genutzt und letzteres wird von der *run*-Methode verwendet.

Neben den Agenten existiert eine zentrale Visualisierungskomponente, die mittels Observer Pattern die Variablen der Agenten überwacht, auswertet und an den Nutzer ausgibt. Dadurch kann der Algorithmus untersucht und evaluiert werden. Eine weitere Funktion der Visualisierungskomponente ist, die Anzahl aktiver Agenten zu

zählen und an das Multiagentensystem weiterzugeben. Dieses stoppt die Simulation, wenn alle Agenten passiv sind und keine weiteren Nachrichten im Umlauf sind. Dieses Vorgehen soll durch einen der oben eingeführten dezentralen DTD-Algorithmen ersetzt werden.

5.2. Erweiterung - Konzept

Für die Implementierung der Erweiterungen sind die abstrakten Klassen *AgentPreparation* und *AgentDTD* vorgesehen, welche sich an den Aufbau von *AgentCOHDA* orientieren. Die erste Klasse dient sowohl als Oberklasse für die AD-Algorithmen als auch als Oberklasse für die Vorbereitung der DTD-Algorithmen, wie beispielsweise die Vorbereitung der Splitting-Vektor-Methode (siehe Abschnitt 4.5.1). Analog zu *AgentCOHDA* besitzt *AgentPreparation* die Methoden *notify*, *update* und *run*. Die Methode *notify* wird zu Beginn von der Umgebung aufgerufen. Anders als bei *AgentCOHDA* benötigt *notify* jedoch keinen Parameter, welcher das globale Ziel übermittelt, da dieses für Agent Discovery und den DTD-Vorbereitungsalgorithmen irrelevant ist, sondern dient ausschließlich als Startsignal für den Algorithmus. Die *update*-Methode wird beim Empfang von Nachrichten aufgerufen und auch der Aufruf von *run* erfolgt analog zu dem von *AgentCOHDA*. Bei *AgentPreparation* ist somit ebenfalls eine Verzögerung zwischen *update*- und *run*-Methode vorhanden, so dass eine technische Reaktionszeit simuliert wird.

Die zweite abstrakte Klasse *AgentDTD* bietet einen Rahmen für die DTD-Algorithmen. Der Aufruf der *run*-Methode funktioniert ebenfalls analog zu *AgentCOHDA*. Anders als bei *AgentCOHDA* wird *notify* nicht von der Umgebung aufgerufen, sondern der Aufruf erfolgt, wenn auch COHDA startet. Ein weiterer Unterschied ist, dass die Klasse zwei *update*-Methoden vorsieht. Während die *update_dtd_msg* Methode Kontrollnachrichten, wie die Antwort-Kontrollnachricht von CTA, empfängt, überwacht die Methode *update_basic_msg* den Nachrichtenempfang von COHDA. Letzteres ist notwendig, da alle vorgestellten DTD-Algorithmen auf dem Empfang von Basisnachrichten reagieren. So erhöht CTA beispielsweise den Zähler C bei Empfang einer Basisnachricht. Da die DTD-Algorithmen ebenso auch auf das Senden von Basisnachrichten reagieren, existiert die Methode *send_basic_msg*, die beim Versenden von Basisnachrichten aufgerufen wird.

Die beiden Klassen sind Unterklassen von der Klasse *ActorMixin*. Im Unterschied zu *AgentCOHDA* haben *AgentPreparation* und *AgentDTD* jedoch nicht *PeerMixin* als Oberklasse, sondern die Klasse *ExtendedPeerMixin* (siehe Abbildung 8). Da *PeerMixin* nur Nachrichten an alle Nachbarn versenden kann, genügt sie nicht für die Erweiterungen. Daher bietet *ExtendedPeerMixin* zusätzlich Funktionen, die Nachrichten an einen Agenten oder Nachrichten an alle Agenten bis auf einen Agenten versenden können. Ersteres ist notwendig, wenn es sich beispielsweise um das Ver-

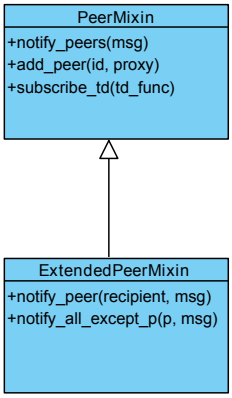


Abbildung 8: Klasse zum Versenden von Nachrichten

senden einer Antwort-Kontrollnachricht an einen Agenten handelt. Letzteres wird benötigt, wenn z.B. bei SVM ein Vektor an alle Kinder des Agenten verschickt wird.

Die Klassen *AgentCOHDA*, *AgentPreparation* und *AgentDTD* müssen so verschaltet werden, dass der in Abschnitt 2.2 beschriebene Ablauf erfolgt. Das bedeutet, dass zuerst die Vorbereitung und Agent Discovery gestartet werden, und erst nachdem diese beendet sind, soll COHDA und der überwachende DTD-Algorithmus ausgeführt werden. Die Verschaltung ist mit der in Abbildung 9 gezeigten Klasse *AgentAddon* möglich. Diese Klasse hat drei Attribute, in denen frei konfigurierbar ist, welcher AD/Vorbereitungs-Algorithmus, Basisalgorithmus und DTD-Algorithmus verwendet werden soll. Der gewählte AD/Vorbereitungs-Algorithmus muss dazu die Oberklasse *AgentPreparation* besitzen und der DTD-Algorithmus die Oberklasse *AgentDTD*. Durch diese flexible Konfiguration ist es ohne viel Aufwand möglich andere Algorithmen für Agent Discovery und Termination Detection zu benutzen. Ebenso kann der zu überwachende Basisalgorithmus ausgetauscht werden, wenn der neue Algorithmus die gleichen Schnittstellen wie *AgentCOHDA* verwendet. Diese Möglichkeiten erfüllen die oben geforderte Unabhängigkeit von COHDA und den Erweiterungen und erleichtert somit die Evaluation der Algorithmen.

Dennoch sind kleinere Anpassungen an den Klassen *AgentCOHDA* und der Klasse *PeerMixin* erforderlich. Eine Änderung betrifft die *notify*-Methode von *AgentCOHDA*. Diese wurde um einen Parameter *preparation* erweitert, über den der Agent die Anzahl der Agenten - also das Ergebnis der Agent Discovery - erfährt und bietet somit eine Schnittstelle zur Agent Discovery an. Das *PeerMixin* muss so angepasst werden, dass wie oben beschrieben die Methode *send_basic_msg* des DTD-Algorithmus das Versenden von Basisnachrichten überwachen kann. Dazu besitzt die Klasse *PeerMixin* eine neue Methode *subscribe_td*, in der dem Basisalgorithmus mitgeteilt wird, welcher DTD-Algorithmus beim Versenden einer Nachricht informiert werden soll.

Damit ist der Aufbau der einzelnen Teilkomponenten von *AgentAddon* geklärt.

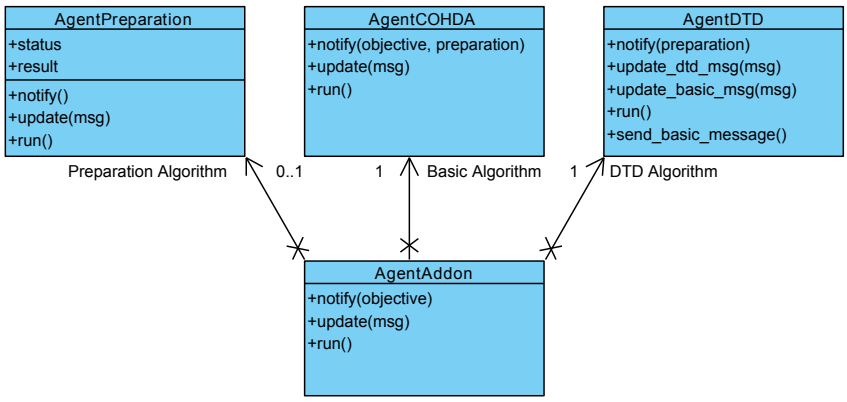


Abbildung 9: Klassendiagramm: COHDA mit Erweiterungen

Im Folgenden wird verdeutlicht, wie der gewünschte Ablauf von COHDA mit Erweiterungen durch *AgentAddon* realisiert werden kann. Zu Beginn des Ablaufs erstellt das Multiagentensystem für jeden Agenten ein Prozess, dessen Verhalten durch *AgentAddon* bestimmt wird. Beim Erstellen der Agenten werden im Konstruktor von *AgentAddon* ebenfalls die Teilkomponenten - Agent Discovery/Vorbereitung, COHDA und Termination Detection - initialisiert. Nach der Initialisierung startet das Multiagentensystem über die *notify*-Methode von *AgentAddon* den Agenten. Die *notify*-Methode speichert darauf das übermittelte globale Ziel und startet die Agent Discovery Erweiterung durch den Aufruf der *notify*-Methode von *AgentPreparation*. Die *run*- und *update*-Methode von *AgentAddon* überwachen und steuern daraufhin den Verlauf der Teilkomponenten.

Die *run*-Methode wird aufgerufen, wenn seit letzter Ausführung von *run* eine oder mehrere neue Nachrichten empfangen wurden. Dahingegen werden die *run*-Methoden der Teilkomponenten nicht mehr wie zuvor beschrieben in einer Dauerschleife aufgerufen, sondern in der *run*-Methode von *AgentAddon*, wenn die *dirty*-Variable der Teilkomponente wahr ist. Dadurch dass *AgentAddon* die *run*-Methoden der Teilkomponenten steuert, kann gewährleistet werden, dass COHDA und die DTD-Komponente erst ausgeführt werden, wenn die Vorbereitungskomponente über ihre *status*-Variable signalisiert hat, dass die Vorbereitung abgeschlossen ist. Wenn dies geschieht, startet *AgentAddon* in der *run*-Methode *AgentCOHDA* und *AgentDTD* über ihre *notify*-Methode und übergibt dabei das Ergebnis der Vorbereitung über den Parameter *preparation*.

Ebenso werden Nachrichten, die durch die Teilkomponenten verschickt werden, nicht mehr direkt durch die *update*-Methode der jeweiligen Teilkomponente bearbeitet, sondern zuerst über die *update*-Methode von AgentAddon empfangen. Diese leitet sie dann an die jeweilige Teilkomponente weiter. Empfängt *AgentAddon* beispielsweise eine DTD-Kontrollnachricht leitet AgentAddon diese an die *upda-*

te_dtd_msg-Methode von *AgentDTD* weiter. Wird jedoch eine Basisnachricht empfangen, wird diese sowohl an die *update*-Methode von *AgentCOHDA* weitergeleitet als auch die Methode *update_basic_msg* aufgerufen. Durch das Weiterleiten der Nachrichten mittels *AgentAddon*, wird folglich die Überwachung der empfangenen Basisnachrichten durch den DTD-Algorithmus ermöglicht.

Für Evaluationszwecke existiert auch für *AgentAddon* eine Visualisierungskomponente, welche die Variablen von *AgentAddon* und seinen Teilkomponenten überwacht. Die Visualisierungskomponente stellt außerdem fest, ob alle Agenten mittels DTD-Algorithmus die Terminierung erkannt haben. Falls dies der Fall ist, signalisiert sie dem Multiagentensystem, dass die Simulation beendet werden kann.

Zusammenfassend bietet *AgentAddon* die Möglichkeit die vorgestellten AD- und DTD-Algorithmen im Kontext von COHDA einzusetzen und zu evaluieren. Dabei können ohne zusätzlichen Programmieraufwand einzelne Teilkomponenten ausgetauscht werden. Ein weiterer Vorteil ist, dass die Variablen- und Methodenbezeichner der einzelnen Teilkomponenten identisch sein können, da sie unabhängig voneinander sind und die Verknüpfung der Teilkomponenten duch *AgentAddon* erfolgt. Trotz der Vorteile von *AgentAddon*, sind nur geringe Änderungen an *AgentCOHDA* und *PeerMixin* und keine Änderungen am eigentlichen Multiagentensystem nötig.

5.3. Erweiterung - Algorithmen

Das oben vorgestellte Konzept beschreibt die Integration der beiden Erweiterungen Termination Detection und Agent Discovery in das Multiagentensystem von COHDA. In diesem Abschnitt wird darauf aufbauend der Entwurf um eine mögliche Umsetzung der einzelnen AD- und DTD-Algorithmen vervollständigt. Dabei liegt der Schwerpunkt auf der Integration der Algorithmen und auf Besonderheiten der Umsetzung. Für eine detaillierte Beschreibung der Algorithmen sei auf Kapitel 3 und Kapitel 4 verwiesen. Die Umsetzung der Algorithmen - d.h. Join Ansatz, Echo Algorithmus, Computation-Tree-Algorithmus, Static-Tree-Algorithmus und Splitting-Vektor-Methode - werden im Folgenden einzeln vorgestellt. Der zuvor vorgestellte Gossip Algorithmus wird in dieser Arbeit nicht weiter betrachtet, da dieser, wie in Abschnitt 3.4 beschrieben, im Kontext von COHDA für Agent Discovery ungeeignet ist.

5.3.1. Join Ansatz

Der Join Ansatz wird in zwei Varianten implementiert. Die Klasse *AgentJoinSimple* repräsentiert den einfachen Ansatz, in dem davon ausgegangen wird, dass neue Kanten im Multiagentensystem eingeführt werden dürfen. Dagegen stellt die Klasse *AgentJoin* eine Variante dar, bei der keine neuen Kanten eingeführt werden, sondern

ausschließlich Nachrichten über bereits vorhandene Kanten verschickt werden.

Die Klasse *AgentJoinSimple* ist eine Unterklasse von *AgentPreparation* und kann somit für die Erweiterung von COHDA um Agent Discovery verwendet werden. In der Methode *notify* wird der Join Prozess für einen Agenten gestartet und eine Kopie der Nachbarschaft angelegt. Alle Veränderungen der Nachbarschaftsbeziehungen durch den Join Ansatz werden ausschließlich auf die Kopie angewendet. Die *update*-Methode, die bei Ankunft von Nachrichten aufgerufen wird, legt ausschließlich eingegangene Nachrichten entsprechend ihres Nachrichtentyps in unterschiedliche Listen ab und setzt *dirty* auf *True*. Die eigentliche Funktionalität ist in der *run*-Methode implementiert. Hier wird entsprechend der verschiedenen Zustände (ruhend, anfragend, beschäftigt, entfernt) das in Abschnitt 3.1 beschriebene Verhalten in einzelnen Methoden durchgeführt. Da in Abschnitt 3.1 von der eigentlichen Kommunikation mit Nachrichten abstrahiert wurde, soll diese hier näher beschrieben und auf unterschiedliche Probleme und Lösungen eingegangen werden.

Für die Partnersuche sind drei Nachrichtentypen vorgesehen: Anfrage, Akzeptanz und Ablehnung. Bei einer erfolgreichen Partnersuche antwortet der Nachbar auf eine Anfrage mit einer Akzeptanz-Nachricht und wechselt den Status zu beschäftigt. Ist er dagegen bereits vorher beschäftigt oder hat er eine weitere Anfrage von einem Agenten mit einer höheren ID erhalten, sendet er eine Ablehnung. Problematisch wird die Kommunikation erst, wenn der anfragende Agent A_i eine Nachricht von einem Agenten A_j mit höherer ID erhält, bevor er eine Akzeptanz von seinem Nachbar A_h erhalten hat. Dann muss er die Anfrage zu A_h abbrechen, da ein Deadlock möglich wäre, wenn alle Agenten gleichzeitig anfragend sind und keiner seine Anfrage aufheben würde. Das Problem ist, dass A_h bereits die Anfrage von A_i akzeptiert haben könnte und somit denkt sie wären Partner. Deshalb muss A_i die Akzeptanz von A_h mit einer Ablehnung beantworten, um diesen mitzuteilen, dass sie keine Partner sind. Dies führt aber mit der Nachrichtenverzögerung zu einem weiteren Problem. Denn eine stark verzögerte Ablehnung zu einer nicht mehr aktuellen Anfrage kann den Algorithmus zu fehlerhaften Ergebnissen führen. Nehmen wir an, ein Agent erhält diese veraltete Ablehnung, nachdem er eine Akzeptanz zu einer aktuellen Anfrage versendet hat. Er würde denken, die Partnerschaft wäre beendet, wohingegen der anfragende Agent durch die empfange Akzeptanz denkt, dass die beiden Agenten sich zusammenfügen können. Deshalb müssen Anfragen, Akzeptanz-Nachrichten und Ablehnungen zusätzlich zur ID des Empfängers eine ID der Anfrage besitzen, so dass veraltete Nachrichten von alten Anfragen beim Empfang in *update* aussortiert werden können und nur aktuelle Nachrichten im Algorithmus Berücksichtigung finden.

Nachdem ein anfragender Agent A_i eine Akzeptanz-Nachricht von seinem Partner erhalten hat, kann er sich mit dem Partner A_j zusammenfügen. Damit nicht in einer weiteren Nachricht angefragt werden muss, wie viele Kanten der Partner hat, ist

diese Information bereits in der Akzeptanz-Nachricht vorhanden. Der anfragende Agent entscheidet dann mit dieser Information, welcher Agent von beiden Partnern entfernt werden soll. Soll der Partner A_j entfernt werden, sendet er diesem eine *kill*-Nachricht. Ansonsten entfernt der anfragende Agent sich selbst. Wenn ein Agent eine *kill*-Nachricht erhalten hat oder sich selbst entfernt, sendet er seinem Partner mit einer *join*-Nachricht seine Zählvariable und die IDs und Adressinformationen seiner Nachbarn. Weiterhin sendet er seinen anderen Nachbarn eine Wechsle_Nachbar-Nachricht mit der ID und den Adressinformationen seines Partners. Bei Erhalt der Nachrichten tauschen die Nachbarn A_i mit A_j in ihrer Nachbarschaftsmenge aus, wohingegen A_j seine Zählvariable und die Zählvariable von A_i aufsummiert und die Nachbarschaftsmengen vereinigt. Jedoch benötigt auch das Zusammenfügen der Agenten zusätzliche Anpassungen, um in allen Situationen korrekt zu funktionieren.

Beispielsweise können Probleme entstehen, wenn mehrere benachbarte Agenten sich zeitnah zusammenfügen. Wenn ein Agent bereits entfernt wurde, bevor er von einem ebenfalls entfernten Nachbarn die Wechsle_Nachbar-Nachricht empfangen hat, dann sendet er seinem Partner eine nicht mehr aktuelle Nachbarschaftsmenge. Deshalb müssen alle entfernten Agenten, Wechsle_Nachbar-Nachrichten zu ihrem ehemaligen Partner weiterleiten, damit Nachbarschaftsbeziehungen aktuell gehalten werden. Ein anderes Problem entsteht durch die Nachrichtenverzögerung. Nehmen wir an, es entstehen zwei Wechsle_Nachbar-Nachrichten. Die ältere Nachricht besagt, dass A_1 mit A_2 getauscht werden soll, während durch die neuere Nachricht A_2 mit A_3 ausgetauscht werden soll. Empfängt ein Nachbar von A_1 zuerst die neuere Nachricht und dann die ältere, würde er A_1 nur mit A_2 austauschen, obwohl A_2 bereits entfernt ist. Die neuere Nachricht konnte er jedoch nicht anwenden, da ihm A_2 noch unbekannt war. Aus diesem Grund speichert der Agent alle nicht anwendbaren Wechsle_Nachbar-Nachrichten und ruft diese erneut ab, wenn sich seine Nachbarschatfsmenge ändert. Weiterhin können *join*- und Wechsle_Nachbar-Nachrichten dazu führen, dass bereits entfernte Agenten wieder hinzugefügt werden. Um dies zu verhindern, müssen sich die Agenten alle entfernten Agenten merken.

Am Ende des Algorithmus, wenn alle Agenten zusammengefügt wurden, wird mit der Flooding-Technik die Agentenzahl versendet. (siehe Abschnitt 3.1) Dabei signalisiert jeder Agent über die Status-Variable, dass er mit dem Join-Ansatz fertig ist.

AgentJoin kann direkt auf *AgentJoinSimple* aufbauen und als Oberklasse benutzen. Es müssen dabei nur kleine Änderungen beim Senden und Empfangen von Nachrichten erfolgen, damit der Algorithmus ohne neue (physikalische) Kanten funktioniert. Die Klasse kann dazu die Methoden zum Empfangen und Senden von Nachrichten überschreiben. Zusätzlich wird ein neuer Paket-Nachrichtentyp eingefügt, in der zu versendende Nachrichten mit der ID des Zielagenten verschickt werden. Agenten, die eine Paketnachricht erhalten, leiten diese zum nächsten Agenten auf dem Weg

zum Zielagenten weiter, wenn sie nicht selbst der Zielagent sind. Damit dies funktioniert, speichern die Agenten den nächsten Knoten auf dem Pfad zu anderen Agenten, indem sie sich merken, von wem sie eine *join*- oder Wechsle_Nachbar-Nachricht erhalten haben, der sie als erstes von diesem Agenten informiert hat. Dieser Nachbar ist dann entweder direkt mit dem Zielagenten verbunden oder hat die Information über den Zielagenten seinerseits von einem Nachbarn erhalten. D.h. es wird induktiv ein Pfad beim Empfang von join- oder Wechsle_Nachbar-Nachrichten aufgebaut. Durch dieses induktive Vorgehen entstehen Pfade, die zum Zielagenten führen und nicht zyklisch durch den Graphen verlaufen. Die einfache Variante lässt sich also ohne großen Aufwand anpassen, um den join-Ansatz bei beliebigen Topologien einsetzen zu können.

5.3.2. Echo Algorithmus

Der Echo-Algorithmus lässt sich durch das Implementieren einer Unterklasse in das Multiagentensystem integrieren, welche die Oberklasse *AgentPreparation* implementiert. Der implementierte Algorithmus soll dabei den effizienteren Ansatz für mehrere Initiatoren aus Abschnitt 3.3.1 umsetzen, bei dem zwar alle Agenten einen eigenen Echo-Algorithmus starten, jedoch nur der Echo-Algorithmus beendet wird, der von dem Agenten mit der höchsten ID gestartet wurde.

Jeder Agent startet beim Aufruf von *notify* einen Echo-Algorithmus, falls der Agent nicht bereits zuvor von einem Echo-Algorithmus eines anderen Agenten durchlaufen wurde. In diesem Fall hätte die Variable *Max* bereits einen Wert. Ansonsten werden die Variablen *C* mit 0, *Max* mit der eigenen ID, *Agents* mit 1, *dirty* mit *True* und die Status-Variable mit *gestartet* initialisiert. Da *dirty* gleich *True* ist, wird *run* danach ausgeführt. Die Methode *run* wird in dieser Umsetzung ausschließlich zum Versenden von Nachrichten verwendet. Anhand der Status-Variable wird entschieden, welche Art von Nachricht versendet wird. Beim Status *gestartet* wird ein Explorer mit der in *Max* gespeicherten ID zu allen Nachbarn außer dem Vorgänger geschickt und der Status zu *Explorer versendet* gewechselt. Wenn bisher kein Vorgänger vorhanden ist, wird eine Nachricht an alle Nachbarn gesendet. Im Zustand *Explorer versendet* wird überprüft, ob die Variable *C* der Anzahl an Nachbarn entspricht. Wenn dies der Fall ist und der Agent nicht der Initiator des Echo-Algorithmus ist, sendet der Agent seine Zählvariable *Agents* mit einem Echo an sein Elternknoten.

Die *update*-Methode nimmt die empfangenen Explorer und Echos entgegen und setzt bei Empfang *dirty* auf *True*. Dabei werden alle Nachrichten ignoriert, die eine niedrigere ID besitzen, als in *Max* gespeichert ist. Empfängt *update* dahingegen eine Nachricht mit höherer ID wird *Max* mit dieser ID überschrieben, der Absender der Nachricht als Vorgänger gemerkt, *C* auf 1 gesetzt und alle anderen Variablen wieder mit den ursprünglichen Wert überschrieben. Insbesondere ist der Status wieder *gest-*

artet und *run* versendet Explorer mit der neuen ID von *Max* an die Nachbarn. Durch dieses Vorgehen wird erreicht, dass sich wie gewünscht nur der Echo-Algorithmus vom Agenten mit der höchsten ID durchsetzt. Alle Nachrichten, deren ID der ID von *Max* entsprechen, bewirken Anpassungen der Variablen C und *Agents*, wie in Abschnitt 3.3 beschrieben.

Wenn der Initiator des Echo-Algorithmus alle Echos erhalten hat, versendet er das Ergebnis an alle Kinder, welche das Ergebnis ihrerseits an alle Kinder verschicken. Dies unterscheidet sich von der Beschreibung in Abschnitt 3.3, insofern als der Agent hier das Ergebnis nur zu den Kindern schickt und nicht zu allen Nachbarn. Dieses Vorgehen verringert die Anzahl der benötigten Nachrichten. Der Agent muss dafür jedoch seine Kinder kennen. Dies ist einfach umsetzbar, indem er bei Erhalt eines Echos den Absender als Kind in einer Liste abspeichert. Daraus ergibt sich zusätzlich die Möglichkeit den Echo-Algorithmus als Vorbereitung für den STA DTD-Algorithmus zu benutzen, bei welchem ebenfalls die Agenten ihre Kinder kennen müssen.

5.3.3. Computation-Tree-Algorithmus

Der Computation-Tree-Algorithmus für mehrere Initiatoren soll als Unterklasse von *AgentDTD* implementiert werden, damit er COHDA als Teilkomponente von *AgentAddon* um Terminierungserkennung erweitern kann. Der Algorithmus lässt sich dabei ohne große Abweichung von der Beschreibung des Algorithmus in Abschnitt 4.3 umsetzen. So erhöhen oder erniedrigen die Methoden *update_dtd_msg*, *update_basic_msg* und *send_basic_msg* die Variablen C und D, wie oben beschrieben. Bei Empfang einer Basis- oder DTD-Nachricht wird dirty *True* und damit die *run*-Methode aufgerufen. In dieser werden die Regeln für das Versenden der Nachrichten überprüft und falls diese erfüllt sind, D erniedrigt und dem letzten Absender eine Antwort geschickt. Dazu werden in *update_basic_msg* alle Absender zu einem Stack hinzugefügt. Um die Regeln zum Versenden von Antworten zu überprüfen, muss der DTD-Algorithmus wissen, ob der Agent passiv ist. Dies ist leicht überprüfbar, da der Agent nur aktiv ist, wenn der *dirty*-Wert vom Basisalgorithmus *True* ist. Denn das Senden von Nachrichten erfolgt bei AgentCOHDA ausschließlich in der Methode *run*, die nur aufgerufen wird, wenn *dirty* gleich *True* ist.

In *run* wird darüber hinaus geregelt, wann der Agent kein Initiator mehr ist. Beim Zustandswechsel zum Nicht-Initiatoren versendet der Agent an alle Nachbarn eine *finished*-Nachricht mit seiner ID. Wie in Abschnitt 4.3 erläutert, leiten die Agenten empfangene *finished*-Nachrichten weiter und erkennen die Terminierung daran, dass sie eine *finished*-Nachricht von jedem Agenten im Multiagentensystem erhalten haben. Um dies überprüfen zu können, wird jedem Agenten die Agentenzahl zu Beginn über *notify* mitgeteilt.

5.3.4. Static-Tree-Algorithmus

Der Static-Tree-Algorithmus soll ebenfalls als Unterklasse von *AgentDTD* umgesetzt werden. Damit der Algorithmus funktioniert, muss der Agent mittels notify über seine Kinder und seinen Elternknoten informiert werden. Dies ist beispielsweise mit dem Echo-Algorithmus aus Abschnitt 5.3.2 möglich.

Jeder Agent besitzt für die Durchführung des Algorithmus den booleschen Wert *stopped* und die Zählvariablen *stopped_children* und *unanswered_msg*. Mit dem Wert *stopped* merkt sich der Agent, ob er dem Elternknoten bereits mit einer *Stop*-Kontrollnachricht mitgeteilt hat, dass er frei ist und ein *Stop* von allen Kindern erhalten hat. In der Variable *stopped_children* wird die Anzahl der Kinder gezählt, die eine *Stop*-Nachricht gesendet haben. Dabei muss der Agent sich nicht merken, welcher Agent *Stop*-Kontrollnachrichten geschickt hat, da es nicht passieren kann, dass ein Agent eine weitere *Stop*-Kontrollnachricht schickt ohne vorher eine *Resume*-Kontrollnachricht zu schicken, die den Wert *stopped_children* verringert. Die letzte Variable *unanswered_msg* wird für die Überprüfung benötigt, ob der Agent frei ist. In ihr wird die Anzahl unbeantworteter Basisnachrichten und unbeantworteter *Resume*-Kontrollnachrichten gezählt. Ein Agent ist dann frei, wenn er passiv ist und $unanswered\_msg = 0$ erfüllt ist. Dies weicht von der Definition von frei in Abschnitt 4.4 insofern ab, als hier auch *Resume*-Kontrollnachrichten beantwortet werden müssen, wodurch einerseits ein Problem mit der Nachrichtenverzögerung gelöst wird und andererseits eine einfache Abarbeitung von Antwort-Nachrichten möglich ist. Denn durch die Änderung, können direkte Antworten auf eine Basisnachricht und das Weiterleiten einer Antwortnachricht auf dem Pfad des Spannbaums - d.h. das Beantworten von Resume-Nachrichten - gleich behandelt werden. Jede *Antwort*-Nachricht enthält in einer Liste den Pfad, den die Antwort zurück zum Zielagenten laufen muss. Die Agenten in der Liste wurden durch die *Resume*-Nachricht beim Durchlaufen des gleichen Pfades auf umgekehrten Weg Schritt für Schritt hinzugefügt und ist bei direkten Antworten auf Basisnachrichten leer. Empfängt nun ein Agent eine *Antwort*-Nachricht, erniedrigt er *unanswered_msg* und entfernt, wenn vorhanden, den letzten Agenten in der Liste und schickt diesem eine *Antwort* mit der Restliste. Zusätzlich löst die erweiterte Definition von *frei* ein mögliches Problem, das durch die Nachrichtenverzögerung entsteht. Dadurch dass ein Agent nun erst ein *Stop* schicken kann, wenn alle *Resume*-Nachrichten beantwortet wurden, kann ein *Stop* ein älteres *Resume* nicht überholen.

Anders als bei den anderen Algorithmen, ist es beim Static-Tree-Algorithmus schwierig die Logik in einem Teil für den Empfang und einem Teil für das Senden von Nachrichten zu trennen, da bei den Kontrollnachrichtentypen sehr unterschiedlich reagiert werden muss. Daher wird in den beiden update-Methoden direkt entschieden, welche Kontrollnachrichten versendet werden. Um dennoch die Trennung

zwischen Empfang und Senden von Nachrichten zu behalten und somit eine technische Verzögerung zu simulieren, speichert die *update*-Methode Kontrollnachrichten in einer Liste, deren Inhalt die *run*-Methode anschließend versendet. Wie auf die Nachrichten in den beiden *update*-Methoden reagiert wird, entspricht dabei bis aus die oben erwähnten Abweichungen und Ergänzungen der Erklärung in Abschnitt 4.4.

Wenn die Wurzel durch Empfang der *Stop*-Nachrichten ihrer Kinder die Terminierung erkennt, sendet sie eine *Finish*-Kontrollnachricht an ihre Kinder, welche diese ebenfalls weiterleiten, bis auch alle Blätter wissen, dass der Basisalgorithmus terminiert ist.

5.3.5. Splitting-Vektor-Methode

Analog zu den anderen DTD-Algorithmen soll die Splitting-Vektor-Methode auch als Unterklasse von *AgentDTD* implementiert werden. Die nötige Vorbereitung wird durch die Klasse *AgentEchoTreeID* realisiert, welche *AgentEcho* als Oberklasse hat und den Algorithmus mit den Ideen aus Abschnitt 4.5.1 anpasst, so dass jeder Agent zu Beginn mit *notify* über seine zugeordnete Nummer, die Nummern seiner Nachbarn und die Intervalle der Kinder informiert werden kann. Wie in Abschnitt 4.5.2 beschrieben, wird bei der Splitting-Vektor-Methode ein Vektor erstellt, bei dem die Anzahl empfangener und gesendeter Basisnachrichten gespeichert wird. Das Eintragen der Nachrichtenanzahl in den Vektor erfolgt in der Umsetzung in *update_basic_msg* und *send_basic_msg*. Dies ist deshalb möglich, da der Agent die Nummern seiner Nachbarn durch die Vorbereitung kennt und somit bei gesendeten Nachrichten die richtige Stelle erhöhen kann. Die Vektoren werden fortlaufend aktualisiert und irgendwann startet der Wurzelknoten einen Terminierungstest und ein Kontrollvektor summiert die einzelnen Vektoren zusammen. Hier stellt sich die Frage, anhand welcher Kriterien der Wurzelknoten entscheidet, wann ein Terminierungstest sinnvoll ist.

Denkbar wäre beispielsweise die Qualität der momentanen Lösung von COHDA zu überprüfen und ab einem bestimmten Schwellenwert einen Terminierungstest zu starten. Es ist jedoch schwierig einen geeigneten Schwellenwert zu wählen, da schwer abgeschätzt werden kann, in welchem Bereich die Lösung liegt. Wird daher der Schwellenwert zu niedrig gewählt, wird sehr häufig ein Teminierungstest begonnen. Wird der Schwellenwert dagegen zu hoch gewählt, kann es passieren, dass die Qualität der Lösung niemals diesen Wert erreicht und demnach niemals ein Terminierungstest gestartet wird. Sinnvoller ist es, wenn die Wurzel die seit der letzten empfangenen Basisnachricht vergangene Zeit misst und mit einer Wartezeit vergleicht. Mit dieser Vorgehensweise ist sicher, dass spätestens eine bestimmte Zeit nach der Terminierung ein Terminierungstest ausgeführt wird. Doch auch in diesem Fall ist die Wahl eines geeigneten Wertes für eine Wartezeit im Voraus nicht trivial, da die Zeit zwischen

zwei Basisnachrichten nicht konstant ist.

Sinnvoller ist es, die Wartezeit dynamisch zu bestimmen. Dazu besitzt die Wurzel eine Variable t_{max} in der sie das längste Zeitintervall zwischen zwei empfangenen Basisnachrichten misst. Wenn die Wurzel eine neue Basisnachricht empfängt, berechnet sie die seit der letzten empfangenen Basisnachricht vergangene Zeit t_{neu}. Wenn eine gemessene Zeitspanne größer als die aktuelle Belegung von t_{max} ist, wird diese mit t_{neu} überschrieben. Anschließend wird der Wartezeit das Produkt von t_{max} und einer Konstante α zugewiesen. (siehe Gleichung 3) Folglich orientiert sich die Wurzel an der maximalen Dauer zwischen zwei Basisnachrichten und merkt sich den größten Wert als Erfahrungswert. Dadurch besitzt die Wurzel eine Möglichkeit abzuschätzen, wann der Basisalgorithmus terminiert ist. Um zu vermeiden, dass bei einer leichten Überschreitung der Wartezeit ein Terminierungstest ausgelöst wird, wird α mit einem Wert größer als 1 initialisiert. Welcher Wert für α sinnvoll ist, um eine geringe Verzögerungszeit mit wenigen Terminierungstests zu erreichen, wird in Kapitel 6 geklärt werden.

$$t_{wartezeit} := \alpha \cdot t_{max} \tag{3}$$

Um die benötigten Zeitintervalle messen zu können, muss der Agent eine Möglichkeit besitzen, den aktuellen Zeitpunkt zu bestimmen. Im realen Einsatz kann ein Agent hierfür die aktuelle Zeit der Systemuhr ablesen, indem der Agent beispielsweise die Systemzeit mit dem Pythonpaket *time* misst. Bei einer Simulation würde dies jedoch zu stark schwankenden Zeitintervallen führen, da das Multiagentensystem abhängig von der Simulationslast unterschiedlich schnell simuliert wird. Aus diesem Grund wird im Fall der Simulation für die Zeitmessung der Schrittzähler des Simulationsframeworks genutzt, der den aktuellen Simulationsschritt angibt. Der Simulationsschritt wird beim Aufruf von der Methode *update_basic_msg* vom Wurzelknoten abgelesen. Anschließend wird von diesem Wert der Simulationsschritt der vorherigen Basisnachricht subtrahiert und mit dem ermittelten Zeitintervall die Wartezeit berechnet.

Die Wurzel überprüft in der *run*-Methode, ob $t_{wartezeit}$ überschritten wurde. Bisher wurde die *run*-Methode jedoch nur dann aufgerufen, wenn *update dirty* auf *True* gesetzt hat und anschließend wurde *dirty* zurück auf *False* gesetzt. Beim Wurzelknoten in SVM wird *dirty* nach Ausführung von *run* nicht wieder auf *False* gesetzt, damit die run-Methode immer wieder überprüft, ob bei Überschreitung der Wartezeit ein Terminierungstest gestartet werden muss.

Der Terminierungstest erfolgt analog zur Beschreibung in Abschnitt 4.5.2. Dazu empfangen die Agenten die Kontrollvektoren mit der Methode *update_dtd_msg*. Auf dem Weg von der Wurzel zu den Blättern addieren die Agenten ihre Vektoren zu den Kontrollvektoren, teilen den Kontrollvektor in Teilvektoren und leiten diesen

an ihre Kinder mit der *run*-Methode weiter, sobald der Agent passiv ist. Dies ist der Fall, wenn *dirty* vom Basisalgorithmus *False* ist. (siehe Abschnitt 5.3.3) Beim umgekehrten Weg fassen die Agenten die empfangenen Kontrollvektoren ihrer Kinder wieder zusammen und senden den aufsummierten Kontrollvektor an ihren Elternknoten ebenfalls mit der *run*-Methode. Der Wurzelknoten summiert die Kontrollvektoren von den Kindern zusammen und zählt anschließend die Einträge des Vektors, die 0 sind. Entspricht die Anzahl dieser Einträge der Anzahl der Agenten, ist der Algorithmus terminiert und die Wurzel teilt den anderen Agenten dieses analog zum Static-Tree-Algorithmus mit, indem er diese Information über den Spannbaum flutet. (vgl. Abschnitt 5.3.4)

6. Evaluation

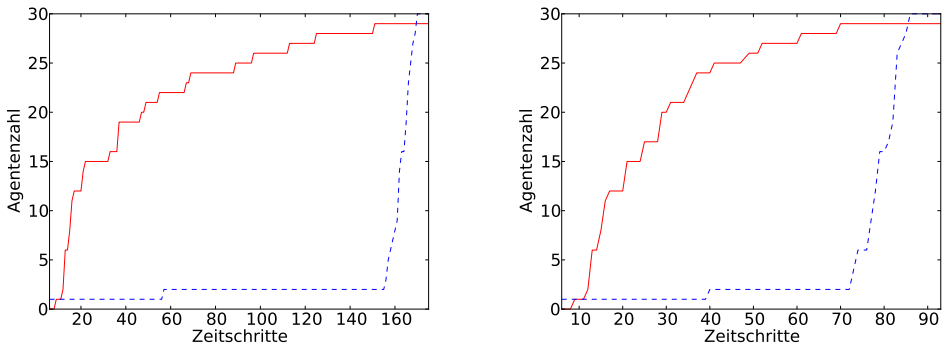
Im Folgenden werden die oben vorgestellten Agent Discovery Algorithmen und Termination Detection Algorithmen evaluiert. Dazu wurden die einzelnen Algorithmen nach dem Entwurf von Kapitel 5 in einem Multiagentensystem implementiert und in mehreren Durchläufen mit unterschiedlichen Szenarien getestet. Die Ergebnisse dieser Testdurchläufe werden hier präsentiert und dienen zusätzlich zum theoretischen Vergleich in Abschnitt 3.4 und Abschnitt 4.6 zur Bewertung der einzelnen Algorithmen. Auf der Grundlage dieses Kapitels können dann anschließend Empfehlungen zum Einsatz der Erweiterungen gegeben werden.

6.1. Agent Discovery

In den Abschnitten 6.1.1 und 6.1.2 werden die Ergebnisse der Testdurchläufe für den Join Ansatz und den Echo-Algorithmus getrennt vorgestellt. Als Testszenario dient dabei, wenn nicht anders beschrieben, ein System mit 30 Agenten und einer Small-World Topologie, bei der die Agenten in einem Ring angeordnet sind. Anders als bei der Ring-Topologie existieren jedoch $\phi \cdot N$ zusätzliche Verbindungen zwischen auf dem Ring nicht benachbarten Agenten. [6] (N ist die Anzahl der Agenten und ϕ eine Konstante) Die Small-World Topologie wird verwendet, da die Funktionsweise von COHDA mit dieser Topologie ausführlich untersucht wurde und gezeigt werden konnte, dass COHDA bei einer Belegung von ϕ mit 0,5 ein gutes Verhältnis von Nachrichten und Simulationsdauer hat. (vgl. [6]) Die Simulationsdauer wird im Folgenden in Zeitschritten gemessen. Ein Zeitschritt hat dabei die Länge der minimalen Verzögerungszeit einer Nachricht. Die Testszenarien wurden mit einer minimalen Verzögerungszeit von 1 und, wenn nicht anders beschrieben, einer maximalen Verzögerung von 2 durchgeführt. Nachdem die beiden AD-Algorithmen separat analysiert wurden, werden diese in Abschnitt 6.1.3 für unterschiedliche Topologien mit verschiedenen Agentenzahlen verglichen.

6.1.1. Join Ansatz

In Abbildung 10 ist der Verlauf einer Durchführung für beide Varianten des Join Ansatzes zu sehen. Bei beiden Verläufen ist erkennbar, dass zu Beginn des Algorithmus viele Agenten entfernt werden. Die Rate, in der Agenten entfernt werden, nimmt aber im Verlauf erheblich ab. Dies lässt sich damit erklären, dass mit der Zunahme entfernter Agenten weniger Agenten vorhanden sind, die Partner eines anderen Agenten werden können. Sind alle Agenten zusammengefügt - d.h. alle Agenten bis auf einen entfernt - wird die Information über die korrekte Agentenzahl verteilt. Dies ist an der starken Erhöhung des Durchschnitts der Variable *Agents* sichtbar. Die Form des Verlaufs unterscheidet sich bei *AgentJoin* und *AgentJoinSimple* zwar nicht, jedoch ist die Dauer eines Durchlaufs bei *AgentJoin* länger. Denn die Kommunikation zwischen Agenten ist bei *AgentJoinSimple* durch das Einführen neuer Kanten über kürzere Pfade möglich und benötigt dementsprechend weniger Zeitschritte. Aus dem gleichen Grund ist die Nachrichtenzahl bei *AgentJoinSimple* niedriger. Der Unterschied von Simulationsdauer und Nachrichtenzahl zwischen den beiden Varianten wächst dabei mit der Anzahl der Agenten. Dies wird in Abbildung 11 deutlich.



AgentJoin AgentJoinSimple

Abbildung 10: Beispielverlauf einer Durchführung. Der durchgezogene Graph zeigt die Anzahl entfernter Agenten. Der gestrichelte Graph zeigt den Durchschnitt der Variable *Agents*

Während bei fünf Agenten kaum ein Unterschied vorhanden ist, ist beispielsweise bei fünfzig Agenten die Simulationsdauer von *AgentJoin* ungefähr doppelt so hoch wie die von *AgentJoinSimple*. Auch dies lässt sich mit den kürzeren Kommunikationspfaden bei *AgentJoinSimple* erklären. Denn bei *AgentJoinSimple* kommunizieren die Agenten nur mit ihren aktuellen Nachbarn, während bei *AgentJoin* die Kommunikationspfade mit der Anzahl der Agenten wachsen können. Somit hat die Agentenzahl einen größeren Einfluss auf *AgentJoin* als auf *AgentJoinSimple*. Dennoch nimmt bei beiden Varianten das Wachstum der Simulationsdauer mit der Agentenzahl ab,

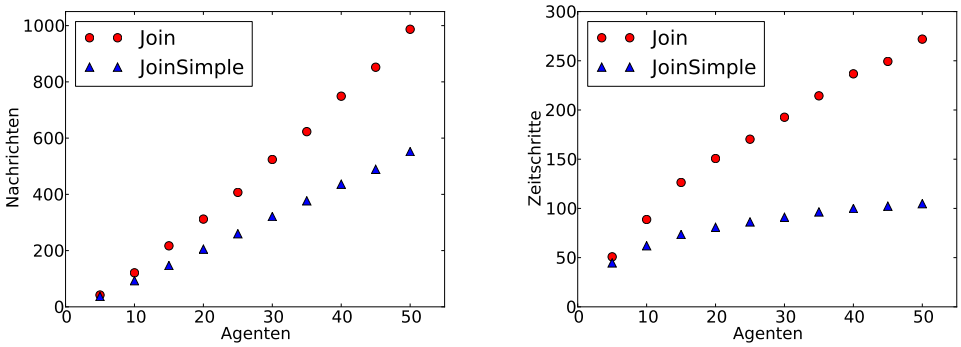


Abbildung 11: Einfluss von Agentenzahl auf Nachrichtenzahl und Simulationsdauer (Messwerte gemittelt über 100 Durchläufe)

was auf ein logarithmisches Wachstum der Simulationsdauer hindeuten kann. Für eine genaue Aussage über das Wachstumsverhalten sind jedoch weitere Simulationsdurchläufe auch mit größeren Agentenzahlen notwendig. Bei dem Wachstum der Nachrichtenmenge lässt sich ebenfalls nur schwer bestimmen, ob es sich um lineares oder potentielles Wachstum handelt. Es lässt sich jedoch an den Messwerten von *AgentJoinSimple* beobachten, dass die Nachrichtenmenge deutlich unter der theoretischen Worst-Case-Nachrichtenzahl liegt. (vgl. Abschnitt A.1)

Lässt man die Agentenzahl konstant und erhöht die maximale Nachrichtenverzögerung, verlängert sich ebenfalls die Simulationsdauer bei beiden Varianten, da die Kommunikation zwischen den Agenten mehr Zeit in Anspruch nimmt. (siehe Abbildung 18 im Anhang) Die Simulationsdauer wird dabei in etwa um den gleichen Faktor größer wie die durchschnittliche Nachrichtenverzögerung. Die Nachrichtenverzögerung hat also keinen entscheidenden Einfluss auf das Verhalten des Algorithmus. Dies wird auch daran deutlich, dass sich die Nachrichtenzahl bei Erhöhung der Nachrichtenverzögerung nur geringfügig ändert.

6.1.2. Echo-Algorithmus

Der Echo-Algorithmus wurde mit dem Entwurf in Abschnitt 5.3.2 umgesetzt und getestet. Analog zum Join-Algorithmus wird auch hier der Einfluss von Agentenzahl und Nachrichtenverzögerung auf den Algorithmus untersucht. Ersteres wird in Abbildung 12 gezeigt. Die Anzahl der Nachrichten steigt bei Erhöhung der Agentenzahl etwas stärker als linear. Dies ist vergleichsweise wenig, wenn man bedenkt, dass zu Beginn jeder Agent einen separaten Echo-Algorithmus startet. Die niedrige Nachrichtenzahl begründet sich dadurch, dass zu Beginn sehr viele Echo-Algorithmen von Algorithmen mit höherer ID verdrängt werden. Dies ist an dem Verlauf in Abbildung 13 gut erkennbar. Während zu Beginn jeder der dreißig Agenten einen unterschiedlichen Wert in der Variable *Max* gespeichert hat und somit dreißig Echo-Algorithmen par-

allel laufen, nimmt diese Zahl rapide ab, bis sich ein Echo-Algorithmus durchgesetzt hat.

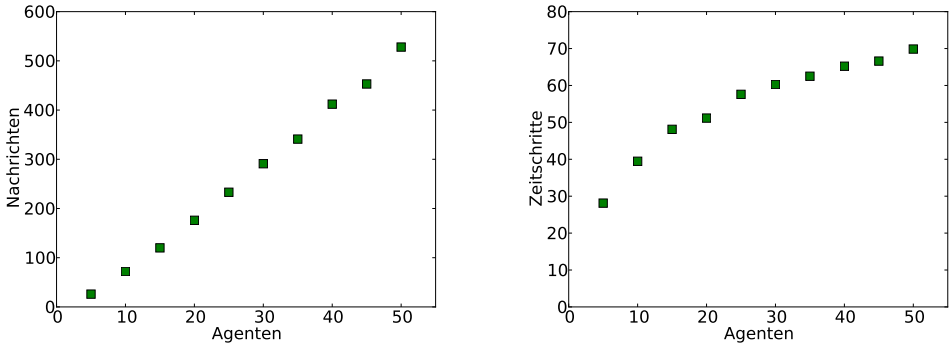


Abbildung 12: Einfluss von Agentenzahl auf Nachrichtenzahl und Simulationsdauer (Messwerte gemittelt über 100 Durchläufe)

Im Gegensatz zur Nachrichtenzahl nimmt das Wachstum der Simulationsdauer bei steigender Agentenzahl ab. Denn beim Echo-Algorithmus beschleunigt sich mit jedem Zeitschritt die Verbreitung des Algorithmus und immer mehr Agenten werden parallel durchlaufen. Da bei einer höheren Agentenzahl diese Parallelität besser ausgenutzt wird, steigt die Simulationsdauer weniger stark an.

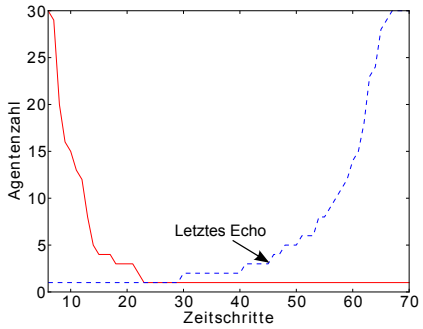


Abbildung 13: Beispielverlauf einer Durchführung. Der durchgezogene Graph zeigt die Anzahl unterschiedlicher Einträge in den *Max*-Variablen. Der gestrichelte Graph zeigt den Durchschnitt der Variable *Agents*

Eine Veränderung der maximalen Nachrichtenverzögerung hat dagegen keine entscheidenden Einflüsse auf den Echo-Algorithmus. So sinkt auch beim Echo-Algorithmus die Nachrichtenzahl nur schwach. (siehe Abbildung 19 im Anhang) Denn durch die Verzögerung erhalten manche Agenten erst später das Startsignal und können somit von anderen Explorern durchlaufen werden, bevor sie einen Echo-Algorithmus starten konnten. Deswegen werden weniger Echo-Algorithmen parallel ausgeführt und die Zahl der Explorer sinkt geringfügig. Die Simulationsdauer steigt aufgrund der längeren Übertragungszeit von Nachrichten.

6.1.3. Vergleich

Im theoretischen Vergleich in Abschnitt 6.1.3 konnte nicht abschließend geklärt werden, ob der Join-Ansatz oder der Echo-Algorithmus im Durchschnitt eine geringere Nachrichtenkomplexität hat. Dies ist durch die Ergebnisse der praktischen Evaluation im direkten Vergleich möglich. Zusätzlich erfolgt ein Vergleich zwischen der Dauer der beiden Algorithmen bei unterschiedlichen Szenarien mit verschiedenen Topologien. Die Abbildungen zu den Messergebnissen befinden sich im Anhang. (Abbildung 20 bis Abbildung 25)

Bei der Ring-Topologie zeigt sich, dass der Join-Ansatz verglichen mit dem Echo-Algorithmus mehr Nachrichten und Zeit für die Erfüllung der Aufgabe benötigt und dieser Mehraufwand mit steigender Agentenzahl wächst. Der einfache Join-Ansatz ist dagegen in diesem Szenario ähnlich effizient wie der Echo-Algorithmus. Bei höherer Anzahl von Agenten ist der einfache Join-Ansatz sogar effizienter. Der Grund, warum *AgentJoinSimple* so viel effizienter als *AgentJoin* ist, liegt auch in der Ring-Topologie begründet. In der Ring-Topologie ist der Abstand zwischen vielen Agenten groß, wodurch die Kommunikationspfade bei *AgentJoin* besonders lang werden. Werden zusätzliche Kanten im Ring eingeführt, indem ϕ erhöht wird, verringert sich der Unterschied bei der Nachrichtenzahl zwischen den Algorithmen. (siehe Abbildung 21 bis Abbildung 23) Der Unterschied zwischen *AgentJoin* und *AgentJoinSimple* wird geringer, da bei höherer Anzahl der Kanten mehr Agenten dichter beieinander liegen, so dass der Nachteil von *AgentJoin* verringert wird. Der gleiche Effekt bewirkt auch, dass sich bei erhöhter Kantenzahl die Simulationsdauern von *AgentJoin* und *AgentJoinSimple* angleichen. Dies ist besonders bei den Ergebnissen zum vollvermaschten Netzwerk in Abbildung 24 sichtbar. Denn hier sind alle Agenten direkt verbunden und somit gibt es bei den Kommunikationswegen keinen Unterschied mehr. Daher ist die Simulationsdauer und Nachrichtenzahl bei beiden Varianten des Join-Ansatzes bei der vollvermaschten Topologie identisch.

Beim Echo-Algorithmus wächst die Nachrichtenzahl mit ϕ aufgrund der größeren Kantenzahl deutlich, wodurch die Abhängigkeit von Nachrichtenzahl und Kantenzahl bestätigt wird. (vgl. Abschnitt 3.3.1) Dahingegen verändert sich die Simulationsdauer des Echo-Algorithmus bei Erhöhung der Kantenzahl weniger deutlich, so dass zwischen der einzelnen Messergebnisse der Smallworld-Szenarien kaum ein Unterschied feststellbar ist. Sichtbar ist, dass der Echo-Algorithmus bei einer Ring-Topologie mit hoher Agentenzahl länger braucht als bei einer gleich großen Smallworld-Topologie. Dies lässt sich dadurch erklären, dass die in Abschnitt 6.1.2 erwähnte parallele Abarbeitung im Ring nicht effizient möglich ist. Denn im Ring können sich die Explorer vom Initiator aus nur in zwei Richtungen ausbreiten, da keine Abzweigungen vorhanden sind.

Als weiteres Szenario wurde eine Gitter-Topologie getestet, in der die Agenten in

einem zweidimensionalen Gitter angeordnet sind. Auch hier lösen die Algorithmen Agent-Discovery bezüglich Nachrichtenkomplexität und Dauer unterschiedlich effizient. Der Echo-Algorithmus benötigt bei der Gitter-Topologie ebenfalls am wenigsten Zeit, um die Agentenzahl zu ermitteln. Auffälliger ist, dass hier die Nachrichtenkomplexität des Echo-Algorithmus sichtbar über der Nachrichtenkomplexität des einfachen Join-Ansatzes liegt. Dessen ungeachtet ist *AgentEcho* effizienter als *AgentJoin*.

Insgesamt lässt sich an den Messergebnissen ablesen, dass der Echo-Algorithmus für die meisten getesteten Szenarien sowohl eine niedrigere Nachrichtenkomplexität als auch eine niedrigere Zeitdauer benötigt als die Varianten des Join-Ansatzes. Der einfache Join-Ansatz ist in manchen Fällen ähnlich effizient wie der Echo-Algorithmus. Jedoch muss bedacht werden, dass dieser die Anforderung hat, neue Kanten einführen zu dürfen. Ist diese Anforderung nicht erfüllt, ist nur die Variante *AgentJoin* einsetzbar. Da *AgentJoin* bei den getesteten Szenarien mit Ausnahme des vollvermaschten Netzwerks vergleichsweise hohe Nachrichtenkomplexität und Zeitdauer aufweist, ist der Echo-Algorithmus in den meisten Fällen die beste Wahl.

6.2. Termination Detection

Im Folgenden wird die Evaluation der DTD-Algorithmen vorgestellt. Wie bei Agent Discovery wurden auch für Termination Detection verschiedene Testszenarien verwendet, um die Funktionsfähigkeit und Effizienz der Algorithmen zu überprüfen. Wenn nicht anders erwähnt, beziehen sich die Testergebnisse auf ein Szenario mit einer Smallworld-Topologie ($\phi = 0{,}5$) und einer Nachrichtenverzögerung zwischen 1 und 2. In den Abschnitten 6.2.1 bis 6.2.3 werden die Messergebnisse zur Nachrichtenkomplexität und Verzögerung der Terminierungserkennung der einzelnen DTD-Algorithmen getrennt vorgestellt. Zusätzlich werden auch die Nachrichtenzahl und Simulationsdauer von COHDA aufgezeigt, damit die Größenordnung der Messergebnisse besser eingeordnet werden kann. Am Ende des Abschnitts werden die Algorithmen miteinander verglichen, um auch für Termination Detection entscheiden zu können, welche der untersuchten DTD-Algorithmen sich als Erweiterung von COHDA am besten eignen.

6.2.1. Computation-Tree-Algorithmus

Beim Computation-Tree-Algorithmus spiegeln die Messergebnisse für die Nachrichtenkomplexität die theoretischen Überlegungen aus Abschnitt 4.3 wieder. In den Messergebnissen in Abbildung 14 ist sichtbar, dass die Anzahl der Kontrollnachrichten von CTA schneller wächst und höher ist als die Anzahl der Basisnachrichten. Zum einen besitzt CTA Antwort-Kontrollnachrichten, deren Anzahl der Menge an Basisnachrichten entspricht. Denn jede Basisnachricht wird genau einmal beantwor-

tet. Deshalb ist die Anzahl der Antworten unabhängig von der Agentenzahl oder Topologie und folglich immer identisch mit der Anzahl der Basisnachrichten. Der zusätzliche Anteil an Kontrollnachrichten ergibt sich aus den Finished-Nachrichten, mit denen jeder Agent beim Wechsel zum Nicht-Initiatoren das Multiagentensystem flutet. Weil dieser Anteil der Kontrollnachrichten nicht unabhängig von der Anzahl der Agenten ist, wächst die Nachrichtenkomplexität von CTA stärker als die von COHDA.

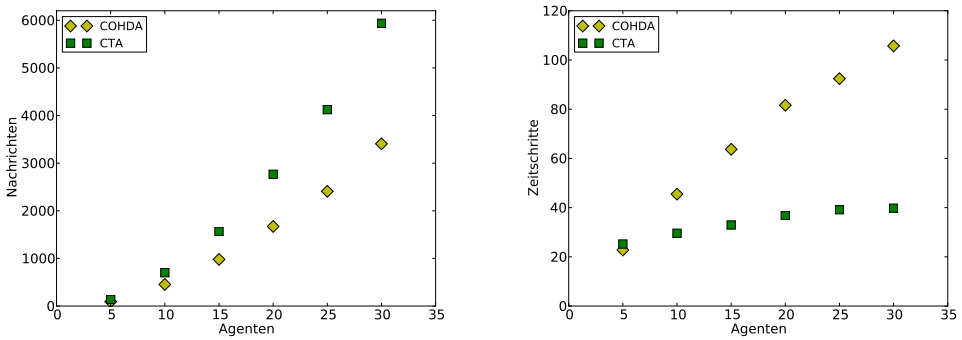


Abbildung 14: Vergleich zwischen Nachrichtenzahl von COHDA und CTA und zwischen Verzögerung der Terminierungserkennung und der Zeitdauer von COHDA (Messwerte gemittelt über 50 Durchläufe)

Neben der Nachrichtenkomplexität wurde die Verzögerung der Terminierungserkennung betrachtet. Diese gibt an, wie viel Zeit benötigt wird, bis alle Agenten die Terminierung erkennen. Bei CTA kann beobachtet werden, dass die Verzögerung mit der Agentenzahl zunimmt. Denn mit einer größeren Anzahl von Agenten wächst die Wahrscheinlichkeit, dass die Agenten nach der Terminierung auf mehrere Antwort-Nachrichten warten müssen. Da zusätzlich bei einer größeren Anzahl der Agenten auch der Durchschnitt des Graphen wächst, müssen zusätzlich Finished-Nachrichten längere Pfade überwinden, um alle Agenten zu erreichen.

6.2.2. Static-Tree-Algorithmus

Der Static-Tree-Algorithmus weist beim Testszenario eine im Vergleich zum theoretischen Worst-Case von $O(M \cdot D + N)$ gute Nachrichtenkomplexität auf. Auch bei STA wird jede Basisnachricht mit einer Kontrollnachricht beantwortet. Zusätzlich existieren Stop-, Resume- und Finished-Kontrollnachrichten. Betrachtet man die Messergebnisse in Abbildung 15, fällt auf, dass die Anzahl an Kontrollnachrichten mit der Anzahl an Basisnachrichten fast übereinstimmt. Die Anzahl der Kontrollnachrichten wächst leicht schneller und liegt geringfügig über der Anzahl der Basisnachrichten. Neben der großen Anzahl an Antwort-Kontrollnachrichten werden also kaum andere Kontrollnachrichten in diesem Testszenario für die Terminierungserkennung benö-

tigt. Die Verzögerung der Terminierungserkennung wächst wie die Nachrichtenkomplexität mit der Anzahl der Agenten. Dies liegt daran, dass die Nachrichtenzahl bei STA von den Abständen im Spannbaum, die mit der Anzahl der Agenten wachsen, abhängt.

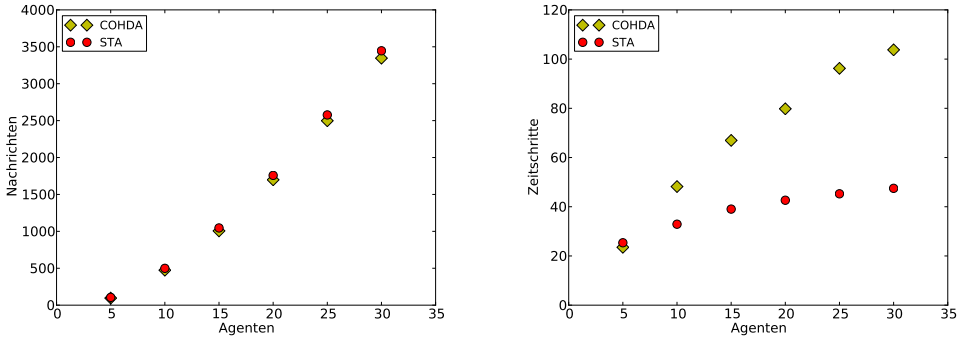


Abbildung 15: Vergleich zwischen Nachrichtenzahl von COHDA und STA und zwischen Verzögerung der Terminierungserkennung und der Zeitdauer von COHDA (Messwerte gemittelt über 50 Durchläufe)

6.2.3. Splitting-Vektor-Methode

Bei der Splitting-Vektor-Methode hängt die benötigte Anzahl an Kontrollnachrichten direkt von der Anzahl der Terminierungstests ab. Je mehr Terminierungstests gestartet werden, desto höher ist also die Nachrichtenzahl. Für die Verzögerung der Terminierungserkennung spielen zwei Faktoren eine Rolle: Zum einen wie viel Zeit nach der Terminierung vergeht, bis ein Terminierungstest gestartet wird, zum anderen wie lange ein Terminierungstest dauert. Die Länge des Terminierungstests hängt dabei von der Tiefe des Spannbaums ab, der für die Splitting-Vektor-Methode verwendet wird. Die in Abschnitt 5.3.5 eingeführte Konstante α hat dabei Einfluss, wie lange der Wurzelknoten wartet, bis ein Terminierungstest gestartet wird. Folglich beeinflusst α auch, wie viele Terminierungstests gestartet werden und wie lange es dauert, bis die Wurzel nach der Terminierung einen Terminierungstest startet.

Im Folgenden wird daher überprüft, welchen Einfluss α auf die Nachrichtenkomplexität und Verzögerung der Terminierungserkennung hat. Zusätzlich werden auch der Einfluss der Agentenzahl und der Nachrichtenverzögerung überprüft. Die Nachrichtenverzögerung ist dabei bei der Splitting-Vektor-Methode besonders interessant, da sich die Wurzel bei der Wartezeit für einen Terminierungstest an den Zeitabständen zwischen zwei empfangenen Basisnachrichten orientiert.

In Abbildung 16 sind die Messergebnisse zur Nachrichtenmenge für unterschiedliche Parameter zu sehen. Anhand der Ergebnisse ist sichtbar, dass mit der Agentenzahl die Nachrichtenzahl steigt. Dies lässt sich darauf zurückführen, dass der

Kontrollvektor bei erhöhter Agentenzahl mehr Agenten durchlaufen muss. Der Einfluss von α ist dabei geringer, als der Einfluss der Agentenzahl. Dies lässt darauf schließen, dass bei diesem Szenario selbst bei einem niedrigen α in den meisten Fällen nur ein Terminierungstest ausgeführt wird. Durch eine Erhöhung von α kann die durchschnittlich benötigte Nachrichtenzahl leicht gesenkt werden, da in manchen Fällen weniger Terminierungstests benötigt werden. Jedoch kann die Nachrichtenzahl nicht beliebig weit gesenkt werden. Denn es muss mindestens ein Terminierungstest durchgeführt werden. Daher ist beispielsweise die Nachrichtenmenge bei einem α von 4 und 8 und bei einer niedrigen maximalen Nachrichtenverzögerung gleich. Bei einer Erhöhung der maximalen Nachrichtenverzögerung lässt sich nur bei niedrigem α ein Wachstum der Nachrichtenzahl feststellen. Durch ein niedriges α ist wahrscheinlicher, dass bei einer kurzen Wartezeit ein Terminierungstest gestartet wird. Deswegen werden in manchen Fällen bei großer Nachrichtenverzögerung und kleinem α Terminierungstests zu früh gestartet. Eine Erhöhung von α macht SVM folglich robuster gegen große Nachrichtenverzögerungen.

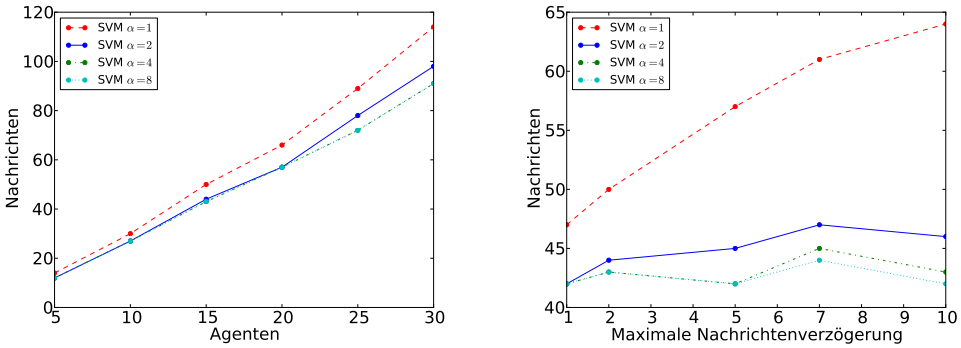


Abbildung 16: Einfluss von α, Agentenzahl und Nachrichtenverzögerung auf die Nachrichtenzahl von SVM. (Messwerte gemittelt über 50 Durchläufe; Einfluss der Nachrichtenverzögerung bei 15 Agenten gemessen)

Die Messergebnisse zur Verzögerung der Terminierungserkennung sind in Abbildung 17 zu sehen. Wie bei der Nachrichtenzahl, steigt auch hier die Verzögerung der Terminierungserkennung mit der Agentenzahl. Denn mit der Agentenzahl wächst die Tiefe des Spannbaums, den der Kontrollvektor durchlaufen muss. Die Verzögerung wächst dabei nicht gleichmäßig mit der Agentenzahl. Dies ist wahrscheinlich damit erklärbar, dass auch die Tiefe des Spannbaums nicht linear mit der Agentenzahl wächst. Die Erhöhung der Nachrichtenverzögerung bewirkt ebenfalls einen Anstieg der Verzögerung der Terminierungserkennung, weil der Kontrollvektor entsprechend länger für das Durchlaufen des Spannbaums braucht. Außerdem lässt sich auch bei einem größer gewählten α feststellen, dass sich die Terminierungserkennung stärker verzögert. Denn, wie oben bereits erläutert, wartet der Wurzelknoten bei einem

großen α länger, bis er einen Terminierungstest startet.

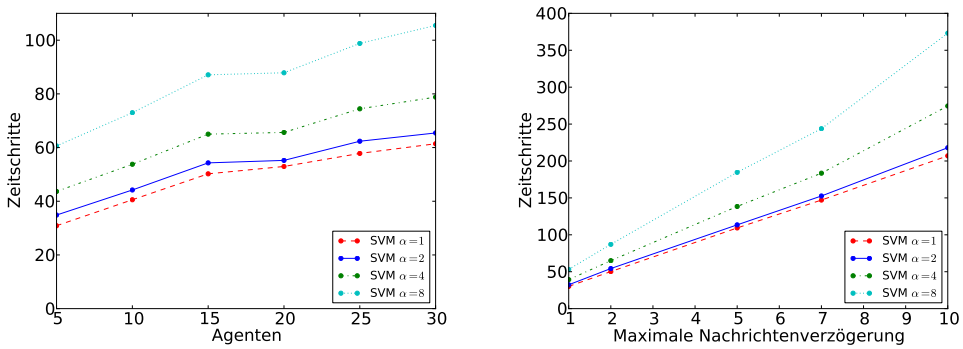


Abbildung 17: Einfluss von α, Agentenzahl und Nachrichtenverzögerung auf die Verzögerung der Terminierungserkennung von SVM. (Messwerte gemittelt über 50 Durchläufe; Einfluss der Nachrichtenverzögerung bei 15 Agenten gemessen)

6.2.4. Vergleich

Im Vergleich in Abschnitt 4.6 erfolgte bereits eine Gegenüberstellung der einzelnen Kriterien mit Hilfe der theoretischen Abschätzungen. Im Folgenden sollen zusätzlich die Ergebnisse der praktischen Evaluation verglichen werden und geklärt werden, inwiefern diese mit den theoretischen Abschätzungen übereinstimmen. Analog zur Evaluation der AD-Algorithmen wurden auch die DTD-Algorithmen CTA, STA und SVM mit unterschiedlichen Szenarien und Topologien getestet. Bei der Splitting-Vektor-Methode werden die Ergebnisse der Testdurchläufe sowohl mit $\alpha = 1$ und $\alpha = 4$ vorgestellt, um eine allgemeinere Aussage über SVM zu ermöglichen. Die Abbildungen zu den Messergebnissen befinden sich im Anhang. (Abbildung 26 bis Abbildung 30)

Bei der Nachrichtenkomplexität lässt sich beobachten, dass unabhängig von der Topologie und der Agentenzahl SVM eine vergleichsweise sehr niedrige Nachrichtenkomplexität aufweist, STA in etwa so viele Kontrollnachrichten wie Basisnachrichten benötigt und CTA die höchste Nachrichtenzahl aufweist. Die Beobachtungen, die in den vorherigen Abschnitten für die DTD-Algorithmen bei einer Smallworld-Topologie mit $\phi = 0{,}5$ gemacht worden, treffen also auch für Smallworld-Topologien mit einem anderen ϕ und der Gitter-Topologie zu. Insbesondere ist die Nachrichtenkomplexität von STA im Durchschnitt in allen Szenarien deutlich niedriger als die theoretische Worst-Case-Abschätzung. (siehe Abschnitt 4.6) Die Unterschiede zwischen den DTD-Algorithmen wachsen mit der Agentenzahl. Denn während die Nachrichtenmenge von SVM kaum Wachstum bei steigender Agentenzahl aufweist, wachsen CTA und STA mit der Agentenzahl deutlich. Anders als bei CTA und STA

ist die Nachrichtenzahl von SVM unabhängig von der Nachrichtenzahl von COHDA. Die niedrige Nachrichtenkomplexität von SVM zeigt auch, dass mittels der dynamischen Wartezeit eine geringe Anzahl an Terminierungstests erreicht werden kann.

Die Nachrichtenzahl von CTA wächst mit der Erhöhung der Agenten zusätzlich, weil im Fall von CTA die Anzahl der Kontrollnachrichten auch stark von der Anzahl der Kanten abhängt. Denn die Finished-Nachrichten müssen bei einer höheren Kantenzahl mehr Kanten durchlaufen. Aus diesem Grund steigt die Anzahl der Kontrollnachrichten auch bei einer Erhöhung von ϕ. So benötigt CTA beispielsweise bei einer Smallworld-Topologie mit $\phi = 4$ und 25 Agenten bereits doppelt so viele Kontrollnachrichten wie Basisnachrichten.

Bei der Verzögerung der Terminierungserkennung zeigt sich ein anderes Bild bezüglich der Qualität der DTD-Algorithmen. Mit Ausnahme von den Ergebnissen für fünf Agenten weist CTA die niedrigste Verzögerung auf. Die Messergebnisse überraschen, weil CTA nach Worst-Case-Abschätzungen in Abschnitt 4.6 eine schlechtere Verzögerung als STA hat. Die Messergebnisse zeigen jedoch, dass bei den getesteten Szenarien die Durchschnittsverzögerung von CTA besser als die von STA ist. Dies lässt darauf schließen, dass bei CTA in den meisten Fällen wenig Antwort-Kontrollnachrichten nach der Terminierung verschickt werden müssen. Dahingegen besitzt SVM die größte Verzögerung. Denn SVM hat in allen Fällen den Nachteil, dass die Wurzel nach der Terminierung wartet, bevor ein Terminierungstest gestartet wird. Die Unterschiede zwischen den Algorithmen bei der Verzögerung der Teminierungserkennung sind jedoch weniger groß als bei der Nachrichtenzahl. Auch das Wachstumsverhalten ähnelt sich bei den drei Algorithmen, insofern als die Abstände zwischen den Messergebnissen der Algorithmen nicht auffällig zunehmen. Eine weitere Gemeinsamkeit ist, dass die Algorithmen die Terminierung bei einem größeren ϕ schneller feststellen. Bei STA und SVM lässt sich dies mit den kürzeren Pfaden im Spannbaum erklären.

Ingesamt zeigen die Ergebnisse, dass kein DTD-Algorithmus eindeutig zu bevorzugen ist. So ist CTA beispielsweise für eine niedrige Verzögerungszeit am besten geeignet, weist jedoch eine sehr hohe Nachrichtenmenge auf. Alternativ kann der CTA-Algorithmus für einen Initiator verwendet werden, bei dem nicht jeder Agent das System mit Finished-Nachrichten flutet. (siehe Abschnitt 4.3) Hierfür müsste COHDA jedoch so angepasst werden, dass die Heuristik zu Beginn nur einen Agenten startet. Der Static-Tree-Algorithmus liegt bei beiden gemessenen Kriterien zwischen den anderen DTD-Algorithmus. Wohingegen bei SVM die Verzögerungszeit höher als bei CTA und STA ist, dafür jedoch eine vergleichsweise sehr niedrige Nachrichtenkomplexität aufweist. Bei der Nachrichtenkomplexität muss dabei berücksichtigt werden, dass SVM die größte Nachrichtengröße besitzt. (vgl. Abschnitt 4.6) Das Verhältnis zwischen der für die Nachrichten benötigte Gesamtdatenmenge der einzelnen

DTD-Algorithmen kann sich also vom Verhältnis der Nachrichtenzahlen unterscheiden. Die Gesamtdatenmenge der Nachrichten hängt dabei jedoch auch von zusätzlichen Daten, wie Adressinformationen, ab. Wie groß eine Nachricht ist, hängt somit von der Realisierung der Nachricht ab.

7. Fazit

Ziel der vorliegenden Arbeit war es, Termination Detection und Agent Discovery Algorithmen zu finden und zu untersuchen, die das verteilte Einsatzplanungsverfahren COHDA erweitern können. Dazu wurden verschiedene Algorithmen in einem theoretischen Teil erläutert und analysiert und anschließend als Erweiterung von COHDA implementiert und evaluiert. Es wurde gezeigt, dass sowohl für Agent Discovery als auch für Termination Detection unterschiedliche verteilte Algorithmen existieren, die einerseits COHDA erweitern können und andererseits auch in anderen verteilten Systemen mit einer zusammenhängenden Topologie einsetzbar sind. Die Algorithmen sind dabei sehr flexibel, insofern sie bei unterschiedlichen Topologien, beliebiger Nachrichtenverzögerung und einer asynchronen Kommunikation funktionieren können. Ein weiterer Vorteil der Algorithmen ist, dass sie keine zentrale Instanz oder Verbindung zur Umgebung benötigen, sondern kooperativ und verteilt von den Agenten ausgeführt werden können.

Für Agent Discovery wurde ein Echo-Algorithmus und ein Join-Ansatz untersucht und implementiert. Beide Algorithmen lösen das Problem auf unterschiedliche Weise. Während der Echo-Algorithmus mittels Explorer und Echo-Nachrichten das verteilte System durchläuft und Informationen über die Agenten sammelt, werden beim Join-Ansatz solange Agenten zusammengefügt bis nur noch ein Agent übrig bleibt, der die Agentenzahl aus der Anzahl der Zusammenfügeoperationen ableiten kann. Im Evaluationsteil konnte anhand der Messergebnisse festgestellt werden, dass im Fall von COHDA der Echo-Algorithmus weniger Zeit und deutlich weniger Nachrichten als der Join-Ansatz benötigt. Für COHDA ist der Echo-Algorithmus somit als Agent Discovery Erweiterung zu empfehlen.

Im Weiteren wurden die Termination Detection Verfahren Computation-Tree-Algorithmus, Static-Tree-Algorithmus und Splitting-Vektor-Methode analysiert und implementiert. Alle drei Verfahren ermöglichen eine Erweiterung von COHDA um Terminierungserkennung, bieten jedoch ebenfalls verschiedene Vorgehensweisen mit unterschiedlichen Eigenschaften bezüglich Nachrichtenkomplexität, Verzögerung der Terminierungserkennung, Nachrichtengröße und Speicheraufwand. Da kein Algorithmus in allen Kriterien überlegen ist, kann im Fall von Termination Detection nicht ohne weiteres eine Empfehlung für ein Verfahren ausgesprochen werden. So konnte beispielsweise anhand der Messergebnisse beobachtet werden, dass CTA im Kon-

text von COHDA zwar eine gute Verzögerung der Terminierungserkennung aufweist, jedoch im Gegenzug eine sehr hohe Nachrichtenzahl benötigt. Wohingegen die Splitting-Vektor-Methode eine auffällig niedrige Nachrichtenzahl hat, aber einen hohen Speicheraufwand, eine große Nachrichtengröße und eine stärker verzögerte Terminierungserkennung besitzt. STA hingegen bietet den niedrigsten Speicheraufwand, weist jedoch in den anderen drei Kriterien eine Qualität zwischen CTA und SVM auf. Aufgrund der unterschiedlichen Vor- und Nachteile der Algorithmen hängt die Wahl des DTD-Algorithmus von den gewünschten Eigenschaften und dem System ab, in dem COHDA und der DTD-Algorithmus eingesetzt werden. Die Ergebnisse der praktischen Untersuchung in Abschnitt 6.2 und der theoretische Vergleich in Abschnitt 4.6 können dazu als Richtlinie dienen, wann welches Verfahren sinnvoll ist.

Aus den Ergebnissen der Bachelorarbeit ergeben sich weitere Forschungsfragen. So stellen die untersuchten Algorithmen zwar sehr wenige Anforderungen an das verteilte System, sind jedoch nicht fehlertolerant. Somit kann eine Untersuchung sinnvoll sein, ob oder mit welchen Einschränkungen fehlertolerantes Vorgehen von zuverlässigen, aber in ihrem Einsatzgebiet eingeschränkten Algorithmen auf die untersuchten Algorithmen übertragbar ist. Eine andere Optimierungsmöglichkeit bietet die Verwendung des Echo-Algorithmus und Computation-Tree-Algorithmus für einen Initiator. Denn wie im theoretischen Teil gezeigt wurde, sind bei beiden Algorithmen die Varianten für einen Initiator bzgl. Nachrichtenkomplexität deutlich effizienter. Daher kann es empfehlenswert sein, COHDA so anzupassen, dass das Verfahren mit einem Initiator funktioniert. Wenn dies ohne große Nachteile möglich ist, hat man mit CTA einen DTD-Algorithmus der nicht nur eine gute Nachrichtengröße und Verzögerungszeit hat, sondern auch einen guten Speicheraufwand aufweist. Zusätzlich wäre die Nachrichtenkomplexität bei großen Agentenzahlen deutlich niedriger. Auch wäre es denkbar, die Messungen der DTD-Algorithmen mit anderen Basisalgorithmen vorzunehmen, um zu überprüfen, wie stark die Ergebnisse vom gewählten Basisalgorithmus abhängen und ob eine allgemeingültige Aussage über die Qualität der DTD-Algorithmen möglich ist. Insofern bietet die Bachelorarbeit zusätzlich eine Basis für eine weitere Untersuchung von Agent Discovery und Termination Detection Verfahren.

Literatur

[1] Bovenkamp, R.; Kuipers, F.; Mieghem, P.: *Gossip-Based Counting in Dynamic Networks*. NETWORKING 2012, Band 7290, 404–417, Springer Berlin Heidelberg, 2012

[2] Chang, E. J.: *Echo Algorithms: Depth Parallel Operations on General Graphs*. IEEE Transactions on Software Engineering, 8, 391–401, 1982

[3] Dijkstra, E. W.; Scholten, C.: *Termination detection for diffusing computations.* Information Processing Letters, 11(1), 1–4, 1980

[4] Eriksen, O.; Skyum, S.: Symmetric Distributed Termination. DAIMI, Computer Science Department, Aarhus University, 1985

[5] Hinrichs, C.: *Selbstorganisierte Koordinationsverfahren für ein dezentrales Supply-Demand-Matching im elektrischen Verteilnetz.* Energieinformatik 2011 : Tagungsband, 2011

[6] Hinrichs, C.; Sonnenschein, M.; Lehnhoff, S.: *Evaluation of a Self-Organizing Heuristic for Interdependent Distributed Search Spaces.* International Conference on Agents and Artificial Intelligence (ICAART 2013), 2013 `http://www-ui.informatik.uni-oldenburg.de/download/Publikationen/HSL13.pdf`

[7] Huang, S.-T.: *Detecting termination of distributed computations by external agents.* Distributed Computing Systems, 1989., 9th International Conference on, 79–84, 1989

[8] Jelasity, M.; Montresor, A.; Babaoglu, O.: *Gossip-based Aggregation in Large Dynamic Networks.* ACM Transactions on Computer Systems, 23, 219–252, 2005

[9] Mahapatra, N. R.; Dutt, S.: *An efficient delay-optimal distributed termination detection algorithm.* Journal of Parallel and Distributed Computing, 67(10), 1047–1066, 2007

[10] Matocha, J.; Camp, T.: *A taxonomy of distributed termination detection algorithms.* The Journal of Systems and Software, 43, 207–221, 1998

[11] Mattern, F.: *Global quiescence detection based on credit distribution and recovery.* Information Processing Letters, 30(4), 195–200, 1980

[12] Mattern, F.: *Experience with a new distributed termination detection algorithm.* Distributed Algorithms, Band 312, 127–143, Springer Berlin Heidelberg, 1988

[13] Mattern, F.: Verteilte Basisalgorithmen, *Informatik-Fachberichte*, Band 226. Springer, 1989

[14] Serugendo, G. D. M.; Gleizes, M.-P.; Karageorgos, A.: *Self-Organisation and Emergence in MAS: An Overview.* Informatica, 30, 45–54, 2006

[15] Shah, D.: Gossip Algorithms. Foundations and trends in networking, Lightning Source Incorporated, 2009

[16] Shavit, N.; Francez, N.: *A new approach to detection of locally indicative stability.* Automata, Languages and Programming, Band 226, 344–358, Springer Berlin Heidelberg, 1986

[17] Vallejo, D.; Garcia-Munoz, L.; Albusac, J.; et al.: *Developing Intelligent Surveillance Systems with an Agent Platform.* Agent and Multi-Agent Systems. Technologies and Applications, Band 7327, 199–208, Springer Berlin Heidelberg, 2012

[18] Venkatesan, S.: *Reliable protocols for distributed termination detection.* Reliability, IEEE Transactions on, 38(1), 103–110, 1989

[19] Zhang, L.; Wong, T.; Fung, R.: *A Multi-Agent System for Dynamic Integrated Process Planning and Scheduling Using Heuristics.* Agent and Multi-Agent Systems. Technologies and Applications, Band 7327, 309–318, Springer Berlin Heidelberg, 2012

A. Anhang

A.1. Nachrichtenkomplexität des Join-Ansatzes

Will man bestimmen, wie viele Nachrichten der Join-Ansatz benötigt, müssen die nötigen Nachrichten für die drei Teile *Partnersuche*, *Zusammenfügen* und *Anzahl verschicken* berechnet werden.

Für die *Partnersuche* kann angenommen werden, dass ein Agent für eine erfolgreiche Partnersuche k_1 Nachrichten und eine erfolglose Partnersuche k_2 Nachrichten benötigt. Die Konstanten k_1 und k_2 hängen dabei von der Implementierung des Algorithmus ab. Im besten Fall ist jede Partnersuche sofort erfolgreich. Dann werden $k_1 \cdot N$ Nachrichten von den N Agenten benötigt. Im schlimmsten Fall fragen immer alle Agenten gleichzeitig an, nur der Agent mit der höchsten ID findet einen Partner und folglich wird nur ein Agent entfernt. Dann ist die Anzahl erfolgloser Partnersuchen beim ersten mal N, beim zweiten mal $N-1$, beim dritten mal $N-2$ usw. Im schlechtesten Fall werden folglich zusätzlich $k_2 \cdot 0{,}5 \cdot (N^2 - N)$ Nachrichten benötigt. (siehe Gleichung 4)

$$
\begin{aligned}
&(N-1)+(N-2)+(N-3)+\cdots+2+1 \\
=&\sum_{k=1}^{N} N-k \\
=&\sum_{k=1}^{N} N-\sum_{k=1}^{N} k \\
=&N^2-\frac{N(N+1)}{2} \\
=&\frac{N^2-N}{2}
\end{aligned} \tag{4}
$$

Beim zweiten Part, dem *Zusammenfügen*, muss jeder Agent jedem Nachbar Informationen über seine Kanten mitteilen. Nehmen wir an, dass abhängig von der Implementierung k_3 Nachrichten benötigt werden, um einen Nachbarn zu informieren. Dann benötigt ein Agent maximal $k_3 \cdot G$ Nachrichten, wenn G der maximale Grad im Graphen ist. Bis auf einen Agenten werden alle Agenten zusammengefügt. Es werden also $k_3 \cdot G \cdot (N-1)$ Nachrichten für das Zusammenfügen benötigt.

Da jeder Agent zu allen Nachbarn die Anzahl der Agenten weiterleitet, wird jede Kante zweimal besucht und folglich benötigt der letzte Part $2 \cdot E$ Nachrichten. Insgesamt ergeben sich aus den zusammengerechneten Nachrichtenzahlen der drei Teile des Algorithmus eine Best-Case Nachrichtenkomplexität von $O(N \cdot G + E)$ und eine Worst-Case Nachrichtenkomplexität von $O(N^2 + N \cdot G + E)$.

A.2. Messergebnisse - Agent Discovery

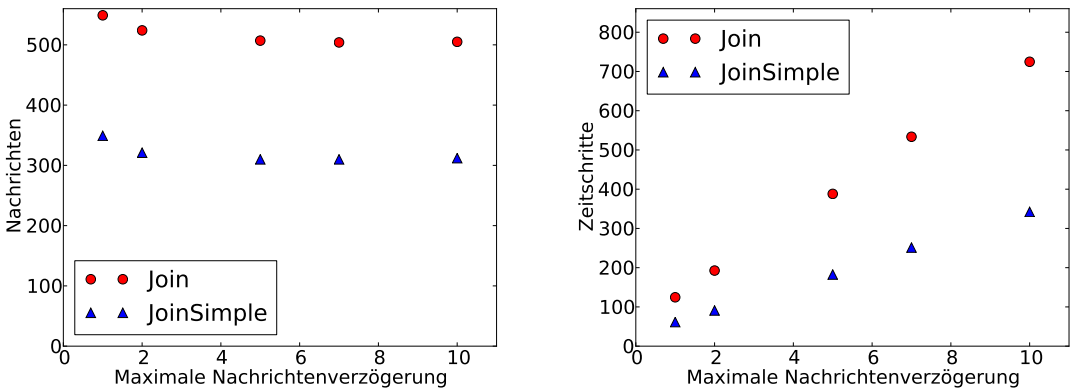


Abbildung 18: Einfluss von Nachrichtenverzögerung auf Nachrichtenzahl und Simulationsdauer des Join-Ansatzes(Messwerte gemittelt über 100 Durchläufe)

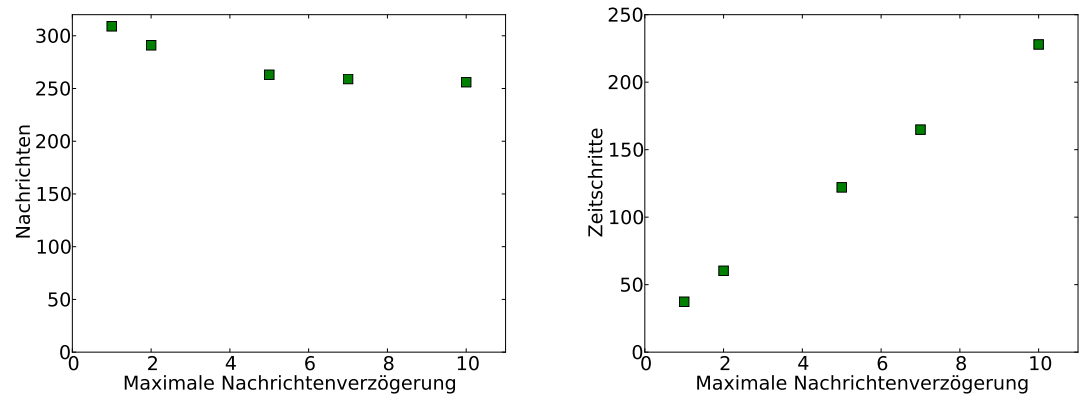


Abbildung 19: Einfluss von Nachrichtenverzögerung auf Nachrichtenzahl und Simulationsdauer des Echo-Algorithmus(Messwerte gemittelt über 100 Durchläufe)

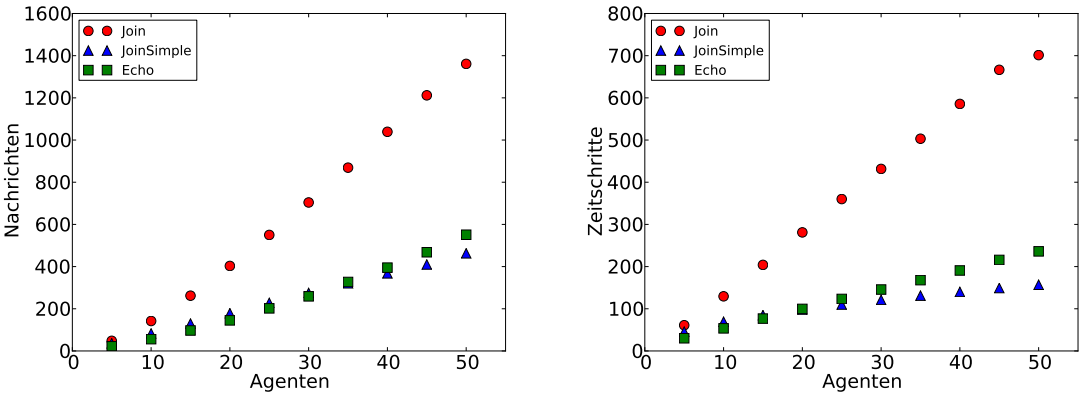


Abbildung 20: Ring/Smallworld $\phi = 0$ (Messwerte gemittelt über 100 Durchläufe)

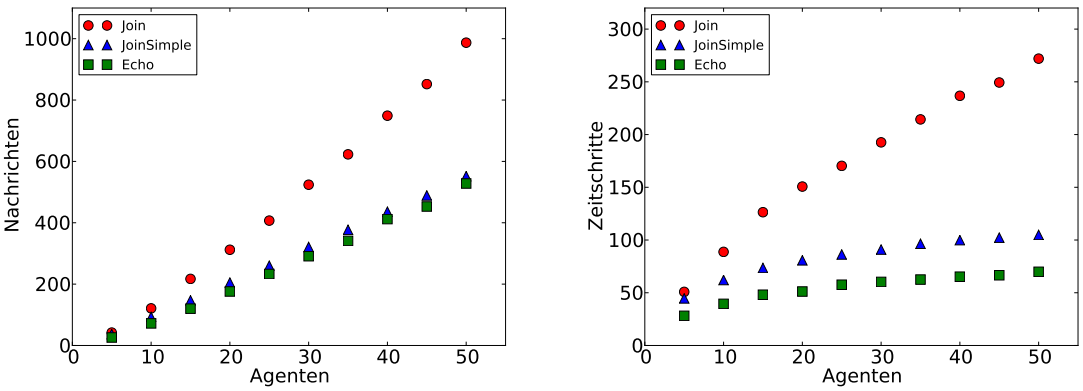


Abbildung 21: Smallworld $\phi = 0,5$ (Messwerte gemittelt über 100 Durchläufe)

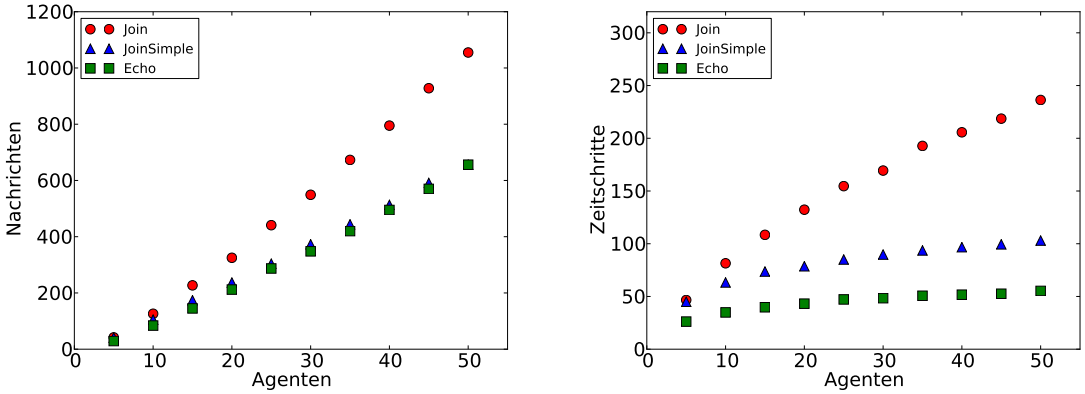


Abbildung 22: Smallworld $\phi = 1$ (Messwerte gemittelt über 100 Durchläufe)

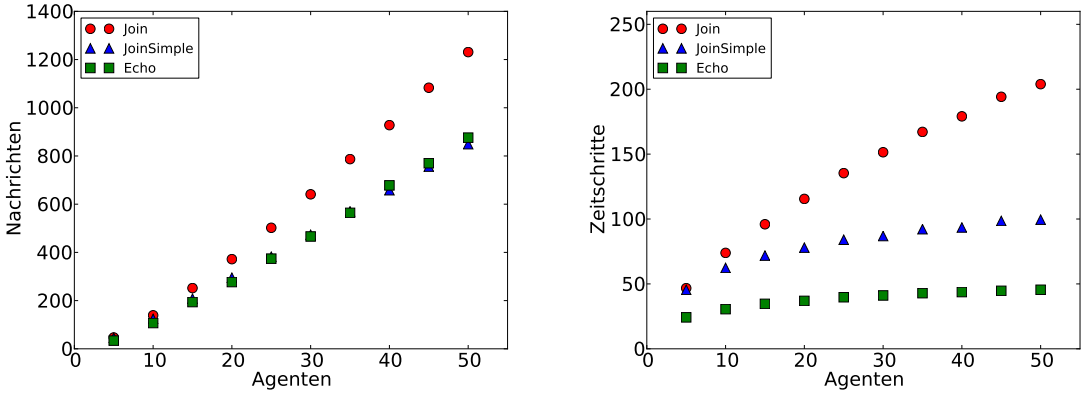


Abbildung 23: Smallworld $\phi = 2$ (Messwerte gemittelt über 100 Durchläufe)

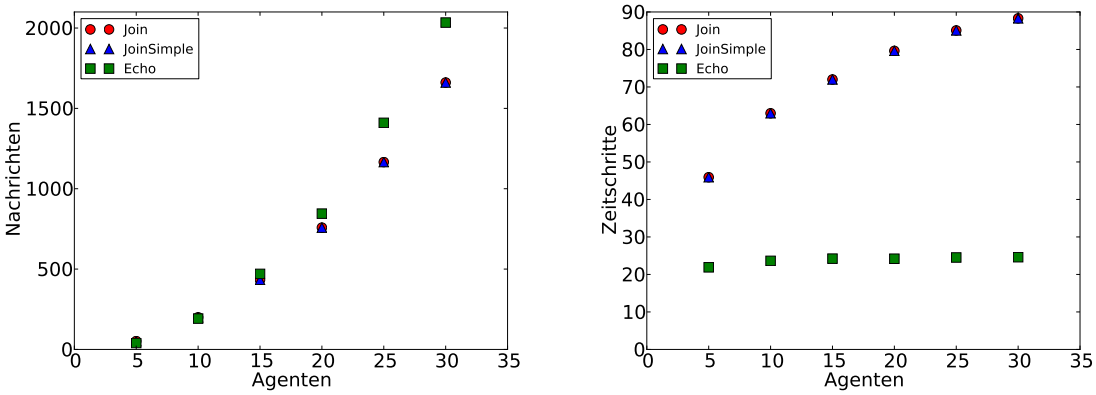


Abbildung 24: Vollvermascht (Messwerte gemittelt über 100 Durchläufe)

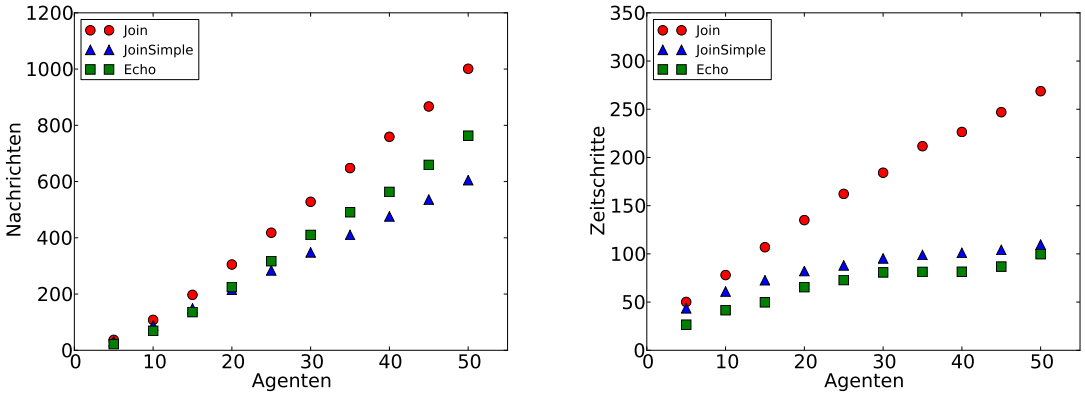


Abbildung 25: Gitter (Messwerte gemittelt über 100 Durchläufe)

A.3. Messergebnisse - Termination Detection

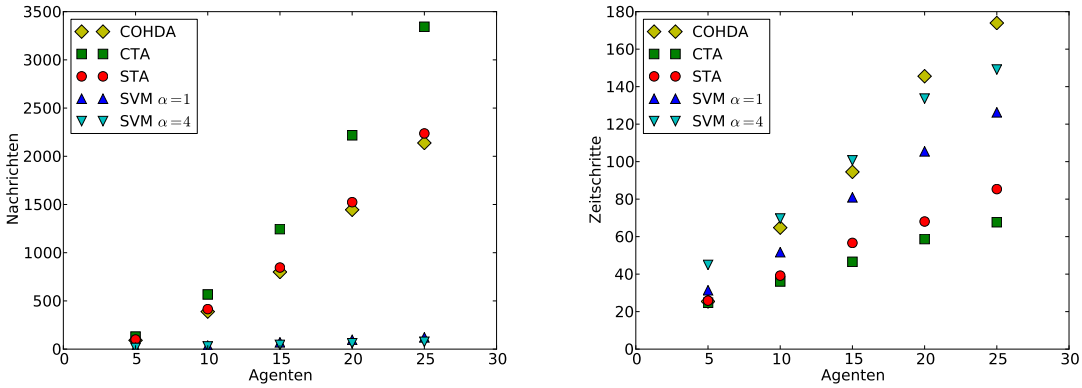


Abbildung 26: Ring/Smallworld $\phi = 0$ (Messwerte gemittelt über 50 Durchläufe)

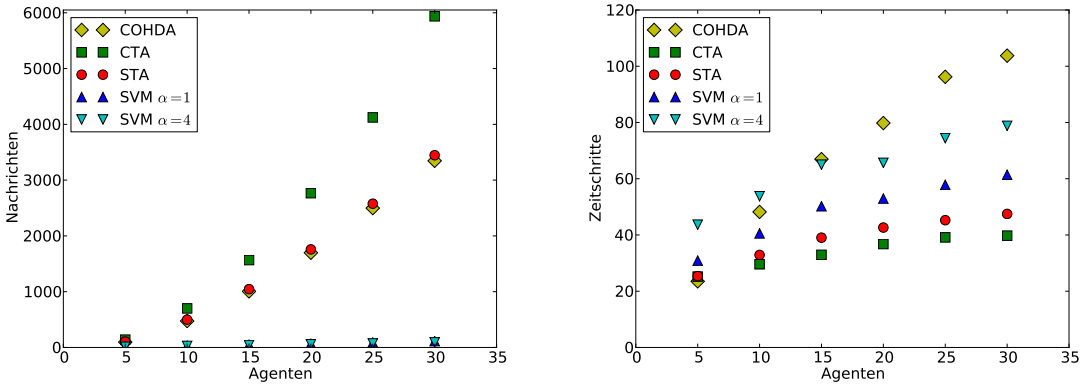


Abbildung 27: Smallworld $\phi = 0,5$ (Messwerte gemittelt über 50 Durchläufe)

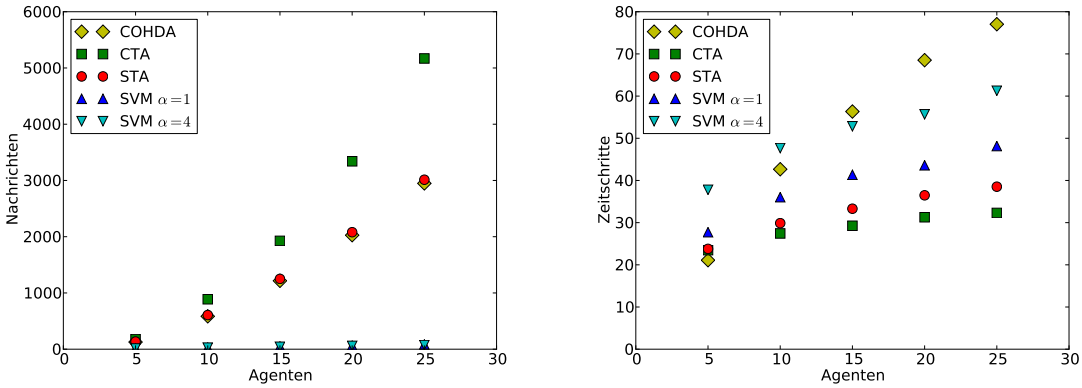


Abbildung 28: Smallworld $\phi = 1$ (Messwerte gemittelt über 50 Durchläufe)

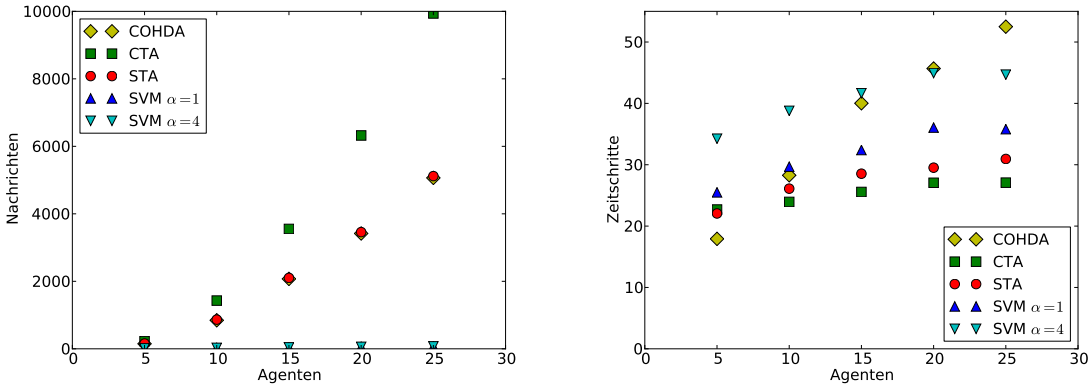


Abbildung 29: Smallworld $\phi = 4$ (Messwerte gemittelt über 50 Durchläufe)

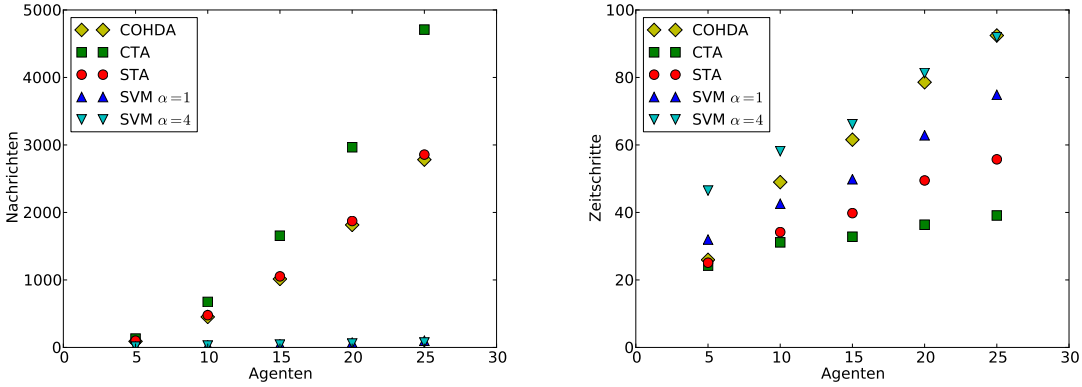


Abbildung 30: Gitter (Messwerte gemittelt über 50 Durchläufe)

Erklärung

Hiermit versichere ich, dass ich diese Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe. Außerdem versichere ich, dass ich die allgemeinen Prinzipien wissenschaftlicher Arbeit und Veröffentlichung, wie sie in den Leitlinien guter wissenschaftlicher Praxis der Carl von Ossietzky Universität Oldenburg festgelegt sind, befolgt habe.

Oldenburg, 31. Juli 2013 *(Sönke Martens)*