

Methoden der KI

Portfolioprüfung

Studienrichtung

Technische Informatik

Muhammad Aman Bin Ahmad Tifli

Matrikelnummer: 2042550

Prüfer: Prof. Dr. Thomas Rist

Abgabedatum: xx.xx.2021



**Hochschule
Augsburg** University of
Applied Sciences

**Fakultät für
Informatik**

Hochschule für angewandte
Wissenschaften Augsburg

An der Hochschule 1
D-86161 Augsburg

Telefon +49 821 55 86-0
Fax +49 821 55 86-3222
www.hs-augsburg.de
[info\(at\)hs-augsburg.de](mailto:info(at)hs-augsburg.de)

Fakultät für Informatik
Telefon +49 821 55 86-3450
Fax +49 821 55 86-3499

Verfasser der Diplomarbeit
Max Mustermann
Beispielstraße 31
86150 Augsburg
Telefon +49 821 55 86-3450
max@hs-augsburg.de

Inhaltsverzeichnis

1	Introduction	3
2	Formulierung von Problemen und Lösungen in der Symbolischen Informationsverarbeitung	5
2.1	Typische KI-Problemstellungen	5
2.2	Problemlösung mit KI	5
2.2.1	Schritte um Probleme zu lösen	5
2.2.2	Performanzmaß berechnen	6
2.3	Beispielformulierungen von Zielen und Problemen	6
2.3.1	8er Puzzle (Sliding block puzzle)	6
2.3.2	Staubsauger-Roboter	7
2.4	Klassifikation von Problemen	8
2.5	Rationaler autonomer Agent als Problemlöser	9
2.5.1	Einfaches Beispiel für einen rationalen autonomen Agenten	10
2.5.2	Arten von rationalen Agenten	11
3	Problemlösung als Suchaufgabe	15
3.1	Wegsuche ohne Karte	15
3.1.1	Bug Algorithmen Beispiel	15
3.1.2	Problem mit dem Bug-Algorithmus	16
3.2	Repräsentation von Suchräumen	16
3.2.1	Suchraum als Karte	16
3.2.2	Charakterisierung von Suchproblemen in Graphen	17
3.3	Wegsuche als systematisches Ablaufen von Graphen	17
3.3.1	Expliziter Graphen und sukzessive Expansion	17
3.3.2	Generelle Wegfindungsstrategie	17
3.3.3	Generelle Bewertung von Suchverfahren	17
3.4	Arten von Suchverfahren	18
3.4.1	Breitensuche in expliziten Graphen	18

3.4.2	Uniforme Kostensuche	18
3.4.3	Modifizierte Uniforme Kostensuche / Dijkstra	19
3.4.4	Ablauf eines Graphen mit Tiefensuche	19
3.4.5	Tiefensuche und Backtracking-Algorithmen	20
3.4.6	Limitierte Tiefensuche	20
3.4.7	Iterative Tiefensuche	20
3.4.8	Bidirektionale Suche	20
Literaturverzeichnis		21

1. Introduction

2. Formulierung von Problemen und Lösungen in der Symbolischen Informationsverarbeitung

Um Probleme mit Hilfe von KI zu lösen, müssen sie zunächst in einer Weise dargestellt werden, die von Computern verarbeitet werden kann. Dies kann mit herkömmlichen Programmiersprachen über symbolische Informationsverarbeitung geschehen

2.1 Typische KI-Problemstellungen

Viele Probleme können mit Hilfe von KI gelöst werden. Die häufigsten sind:

- **Navigation** z.B: Labyrinth/Navigationsspiele, autonomer Staubsauger, Wegplanung
- **Strategiespiele** z.B: Brettspiele, Puzzle
- **Komplexe Aufgaben** z.B: Robocup (Navigation + Strategie)

2.2 Problemlösung mit KI

2.2.1 Schritte um Probleme zu lösen

1. Zielformulierung:

- Soweit möglich, Plausibilitäts-Check dabei durchführen: Ist das Ziel machbar?
- **Beispiel:** Hans will von A nach B, kennt aber den Weg nicht.

2. Problemformulierung

- Ausgangssituation formulieren.
- feststellen welche Operationen möglich sind (z.B Spielregeln).
- **Beispiel:** Durch ausführen von Fahr-Aktionen von A über verbundene Nachbarorte nach B kommen. Mögliche Operationen wären: in die benachbarten Städte zu fahren.

3. Konstruktion einer Lösung

- bewerte Güte einer Lösung
- wähle effektiven Lösungsweg
- **Beispiel:** Ein möglicher Weg zur Lösung des Problems wäre die Erstellung eines Suchbaums.

4. Ausführung

- Läuft alles wie geplant?

2.2.2 Performanzmaß berechnen

- Oft gibt es mehrere zulässige Lösungswege zu einem Probleme
- Wie findet man die optimalste Lösung?
- Zur Bewertung der **Güte** einer Lösung berechnet man die Gesamtkosten

$$\text{Gesamtkosten} = \text{Suchkosten} + \text{Pfadkosten}$$

- Es ist oft schwierig, die Güte einer Lösung zu verrechnen, da es oft viele mögliche Aspekte gibt, die beobachtet und gemessen werden können.
- Manchmal ist es besser, die weniger optimale Lösung zu wählen, die schneller berechnet werden kann: Genauere Planung kann mehr Zeit kosten als sie erspart!

2.3 Beispielformulierungen von Zielen und Problemen

2.3.1 8er Puzzle (Sliding block puzzle)

- Hochgradig kombinatorisches NP-vollständiges Problem. Oft genutzt als Standardtest für neue Suchalgorithmen.

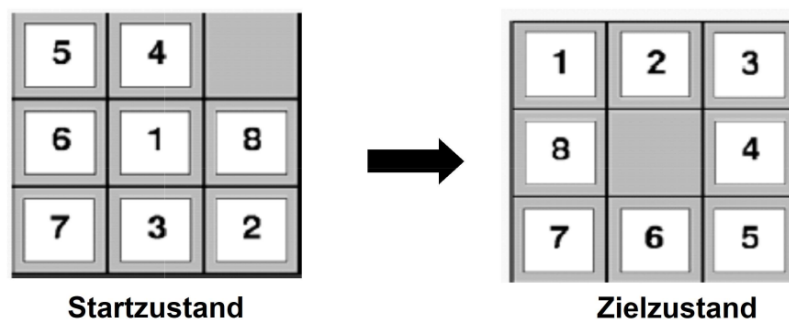


Abbildung 2.1: 8er puzzle Start- und Zielzustand

- Zustände: Lokalität der 8 Fliesen in eine der 9 Flächen plus eine freie Kachel
- Operatoren: Blank nach Links, Rechts, Auf, Ab
- Ziel-test: Blank-Kachel in der Mitte
- Pfadkosten: jeder Schritt kostet eine Einheit

2.3.2 Staubsauger-Roboter

Vieles an der Implementierung dieses Roboters muss abstrahiert werden:

- World States: Umfassen alle Aspekte der realen Welt
- Problem States: Nur Aspekte der relevant für das Problem sind. Die Modellierung von diesen Aspekten erfolgt meist in Form **symbolischer** Beschreibungen.
- Zunächst müssen die möglichen World States als Problem States dargestellt werden:

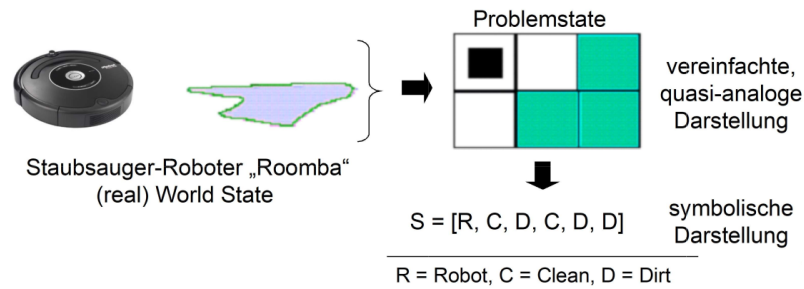


Abbildung 2.2: Abbildung World States auf Problem States

2.3.2.1 Stark vereinfachte Staubsaugerwelt

Eine sehr einfache Darstellung von einer Staubsaugerwelt hat zwei Orte. Jeder Ort kann entweder Staub enthalten oder nicht. Es gibt also 8 mögliche Zustände:

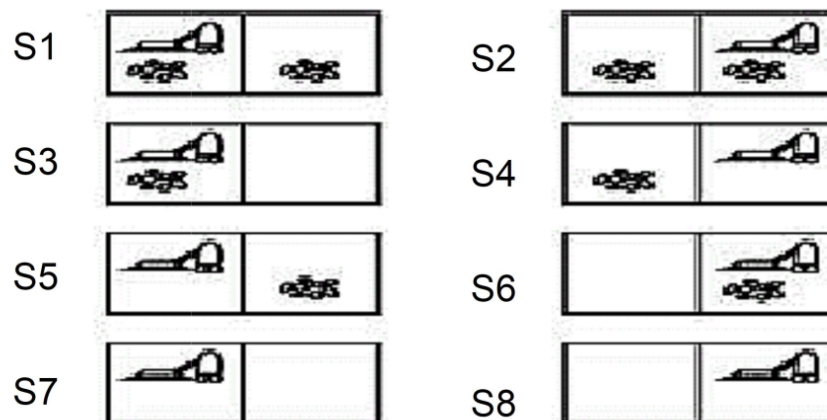


Abbildung 2.3: Staubsauger Zustände

Der Staubsauger kennt in diesem Fall 3 Operationen: Links, Rechts, Saugen und hat das Ziel, die Zustände S7 oder S8 zu erreichen, wo es keinen Staub mehr gibt.

In diesem einfachen Fall können die Lösung mit Hilfe eines endlichen Automaten gefunden werden:

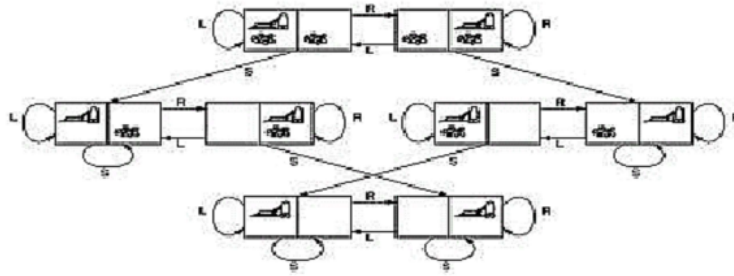


Abbildung 2.4: Staubsauger Pfadsucheautomat

2.4 Klassifikation von Problemen

Es gibt viele Arten von Problemen und sie werden durch die Informationen des Löser bestimmt.

1. Probleme mit Einfach Zuständen

Für den Problemlöser ist klar, in welchem Zustand er sich befindet und was seine möglichen Aktionen bewirken werden. Wie bereits im Roomba-Beispiel (Kap. 2.3.2) erwähnt, kann dies als endlicher Zustandsautomat modelliert werden.

2. Probleme mit Mehrfach-Zuständen

Der genaue Zustand, in dem sich der Problemlöser befindet, ist nicht bekannt, und der Problemlöser weiß nicht, was seine Aktionen bewirken werden.

Anhand des Roomba-Beispiels: In dem Extremfall dass der Roomba keine Sensoren hat, kann er als mehrfaches Problem modelliert werden. In einem solchen Fall kann der Startzustand einer der Zustände S1 bis S8 sein (siehe Abb. 2.3).

Eine mögliche Lösung besteht darin, die Menge der aktuell möglichen Zustände zu verwenden, um die Aktionen des Löser zu bestimmen:

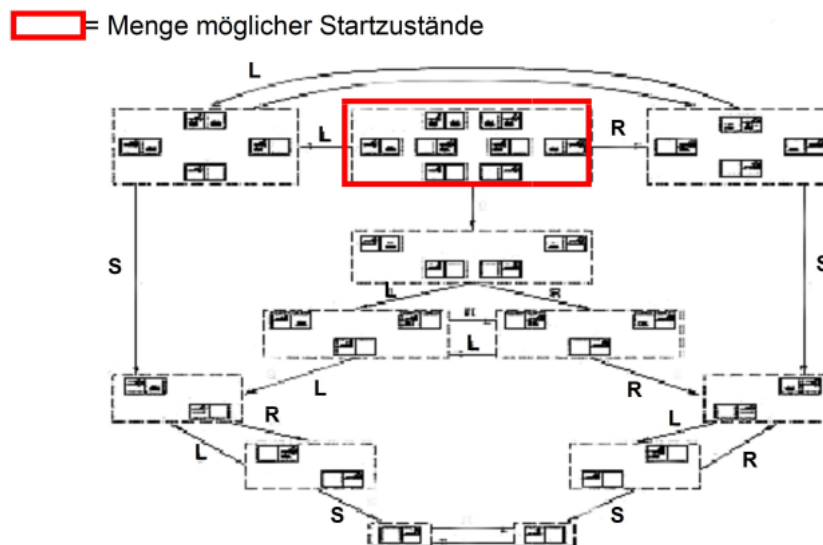


Abbildung 2.5: Staubsaugerwelt als Mehrfach-Problem

3. Zufall-Probleme

Der L?ser hat keine vollst?ndige Kenntnis einer sich st?ndig ver?ndernden Welt. Er kann nur die lokale Umgebung wahrnehmen.

Auch hier wieder das Roomba-Beispiel: Der Roboter befindet sich im Zustand S1 oder S3 und kann die Aktionen: Saugen, nach rechts fahren, Saugen ausf?hren.

Der Roboter saugt und bewegt sich dann nach rechts. Wenn er aber sich vor der Bewegung im Zustand S3 befand, geht er zum Zustand S8 ?ber. Er versucht dann zu saugen aber die Aktion schlagt fehl, da in dem Zustand kein Staub vorhanden ist.

Um dieses Problem zu l?sen, ben?tigt der Roboter einen Sensor, der das Vorhandensein von Staub erkennt.



Abbildung 2.6: Roomba Zufallproblem

4. Explorations-Probleme

Der L?ser hat keine Kenntnis von der Welt und muss seine Umgebung erkunden, um die m?glichen Zust?nde und die Auswirkungen seiner Aktionen zu erfahren.

Ein gutes Beispiel daf?r ist der Marsrover. Der Rover muss zun?chst Daten sammeln, um seine Umgebung kennenzulernen und eine Karte zu erstellen. Mit dieser Karte kann es dann erfolgreich die Pfadfindung durchf?hren.

2.5 Rationaler autonomer Agent als Probleml?ser

Ein rationaler autonomer Agent ist ein Wesen, das seine Welt permanent wahrnehmen und unabh?ngig reagieren kann.

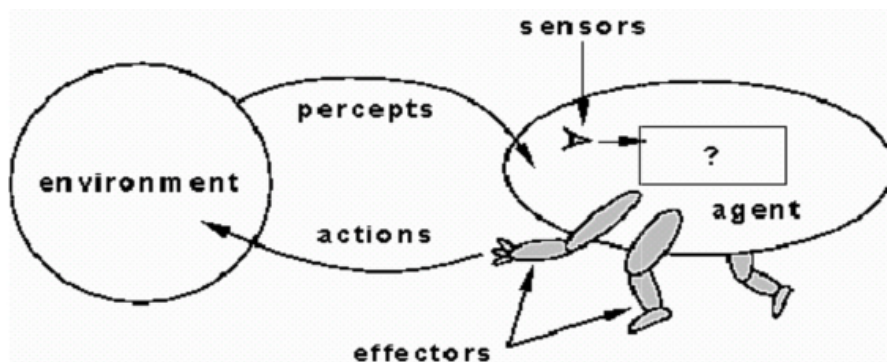


Abbildung 2.7: Rationaler autonomer Agent

Die Abbildung 2.7 veranschaulicht die Interaktion zwischen einem Agenten und seiner Umgebung unter Verwendung seiner Perzeptoren (PAGE = Percepts, Actions, Goals, Environment).

2.5.1 Einfaches Beispiel für einen rationalen autonomen Agenten

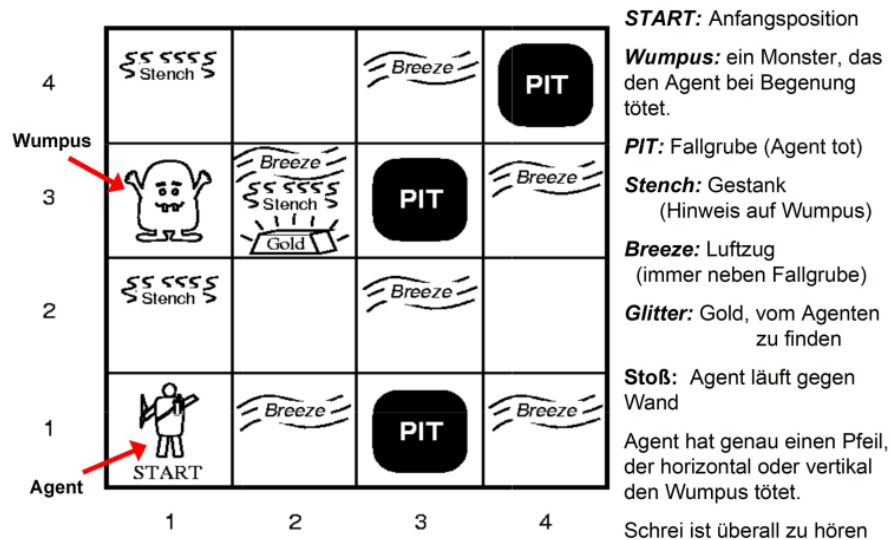


Abbildung 2.8: Wumpus World

Wumpus World ist eine Höhle mit 4/4 Räumen, die durch Gängen verbunden sind. Es sind also insgesamt 16 Räume miteinander verbunden. Es gibt einen wissensbasierten Agenten, der durch diese Welt gehen wird. Die Höhle hat einen Raum mit einem Monster namens Wumpus, das jeden frisst, der den Raum betritt. Wumpus kann vom Agenten erschossen werden, aber der Agent hat einen einzigen Pfeil. In der Welt von Wumpus gibt es mehrere endlose Lochräume und wenn ein Agent in ein Loch fällt, wird er dort für immer gefangen sein. In einem der Räume dieser Höhle befindet sich ein Haufen Gold. Das Ziel des Agenten ist es also, das Gold zu finden und aus der Höhle zu kommen, ohne in das Loch zu fallen oder von Wumpus gefressen zu werden. Der Agent wird belohnt, wenn er mit Gold herauskommt, und er bekommt eine Strafe, wenn er von Wumpus gefressen wird oder in ein Loch fällt.

Umgebung von Wumpus World

- Ein 4x4-Raster von Räumen.
- Der Agent beginnt im Feld [1,1] und zeigt nach rechts.
- Standort von Wumpus und Gold werden zufällig ausgewählt, außer Feld [1,1].
- Jedes Quadrat der Höhle kann mit Wahrscheinlichkeit 0,2 eine Grube sein, außer dem ersten Quadrat

Eigenschaften von Wumpus World

- Die Welt ist teilweise beobachtbar, da der Agent nur die nahe Umgebung wahrnehmen kann, in der er sich befindet.
- Die Welt ist deterministisch, da die Auswirkungen aller Handlungen bekannt sind.
- Die Reihenfolge ist wichtig, also ist die Welt sequentiell.
- Die Welt ist statisch wie Wumpus und die Gruben bewegen sich nicht.

- Die Umgebung ist diskret.
- Einzelagentenumgebung (Wumpus wird nicht als Agent betrachtet, da er statisch ist).

2.5.2 Arten von rationalen Agenten

Reflex Agenten

Ein Reflexagent ist eine Entit?t, die reaktiv ist und mit Sensoren arbeitet. Dieser Agent setzt sich keine Ziele und k?mmert sich nicht um die Auswirkungen seiner Handlungen.

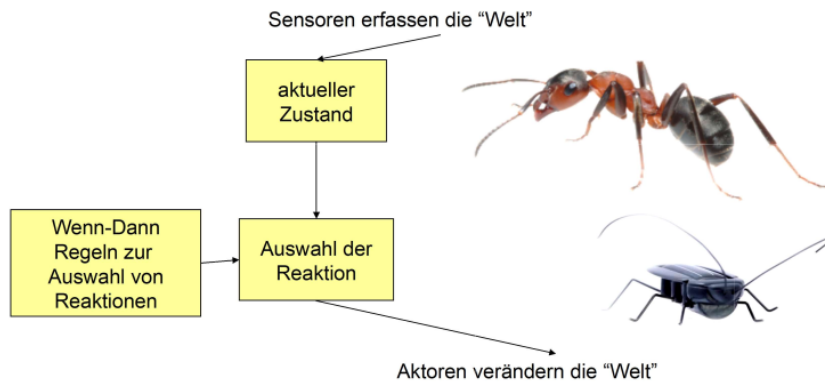


Abbildung 2.9: Reflex-Agenten

Ziel orientierter Agent

Zielorientierte Agenten planen ihre Handlungen und antizipieren die Auswirkungen dieser Handlungen, um ihre Ziele zu erreichen.

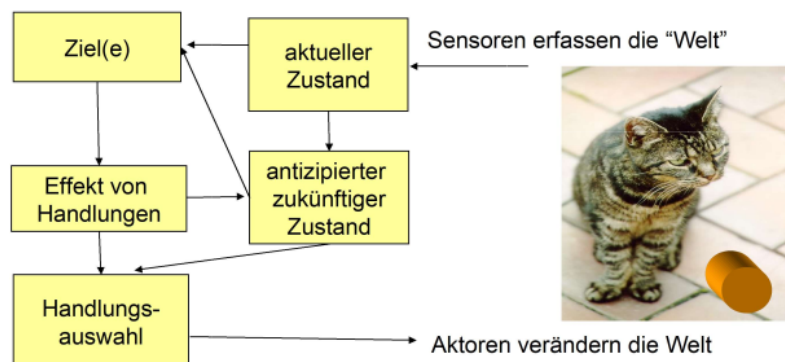


Abbildung 2.10: Ziel-orientierter Agenten

Nutzen-orientierter Agent

Ein nutzen-orientierter Agent wägt die Kosten und Gewinne von Handlungen ab, um die Effektivität seiner Handlungen beim Erreichen eines Ziels zu maximieren.

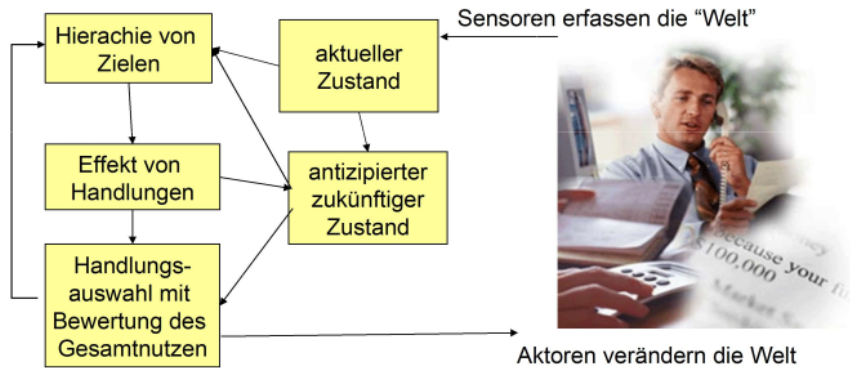


Abbildung 2.11: Nutzen-orientierter Agenten

Lern-fähiger Agent

Diese Art von Agent ist in der Lage, die Effekte seiner Handlungen zu lernen, um seine Ziele besser zu erreichen, indem er seine Handlungen effektiver auswählt.

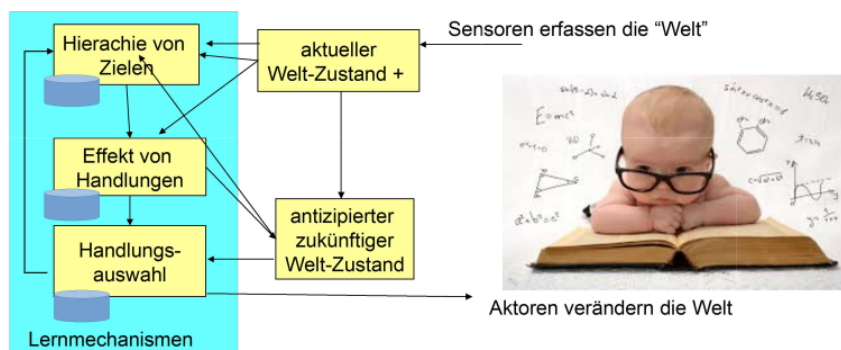


Abbildung 2.12: Lern-fähiger Agenten

Emotionaler Agent

Dieser Agent verwendet neben rationalem Denken eine emotionale Bewertung, um reale menschliche Entscheidungsprozesse besser zu simulieren.

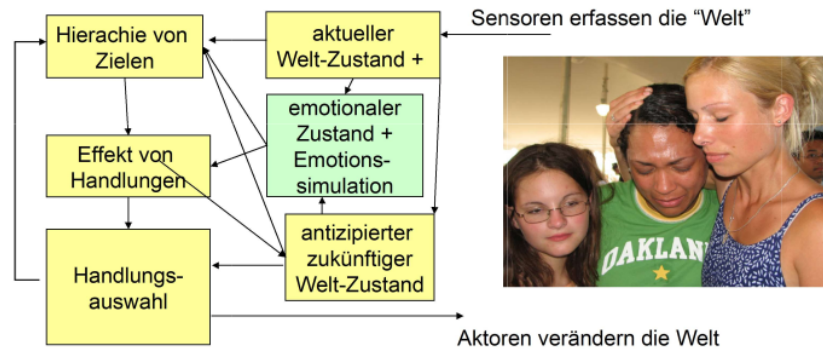


Abbildung 2.13: Emotionaler Agent

Kreativer Agent

Dieser Agent hat die Fähigkeit, neue Aktionen zu entwickeln, um neue Wege zur Lösung von Problemen zu schaffen, die nicht vorprogrammiert waren.

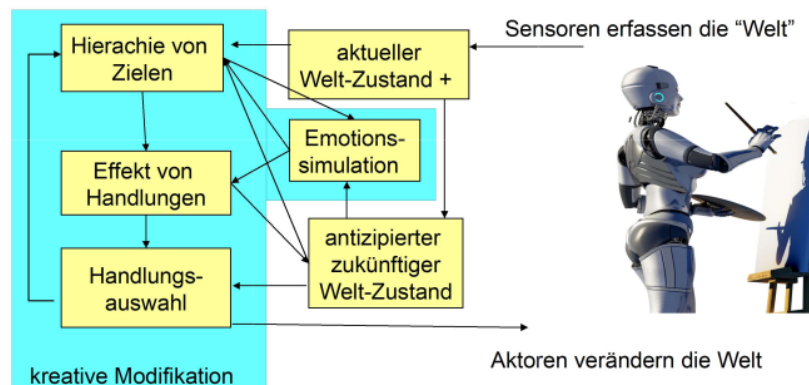


Abbildung 2.14: Kreativer Agent

Kooperative Agenten

Eine Situation, in der mehrere Agenten zusammenarbeiten, um komplexe Aufgaben zu lösen. Dies erfordert die Modellierung kooperativer Verhaltensstrategien und das Verständnis der Absichten anderer Agenten.



Abbildung 2.15: Robocup competition: Cooperative agents

3. Problemlösung als Suchaufgabe

3.1 Wegsuche ohne Karte

“Wegsuche ohne Karte” bedeutet Wegfindung in einer unbekannten Umgebung ohne eine Karte, die den Agenten leitet.

Beispiel: **Roboter R** befindet sich in einem unbekannten Gebiet und muss sich zu einem Zielobjekt bewegen. Dies kann durch Anwendung eines **Bug-Algorithmus** gelöst werden, der voraussetzt, dass der Roboter mit Sensoren ausgestattet ist, um Hindernisse und das **Zielobjekt S** zu erkennen.

3.1.1 Bug Algorithmen Beispiel

Ein Beispiel für einen Bug-Algorithmus ist wie folgt:

1. Wenn das **Zielobjekt S** in Sichtweite ist, fährt **Roboter R** direkt darauf zu
2. Wenn **S** nicht in Sicht ist, aber stattdessen ein Hindernis vorhanden ist, bewegt sich **R** gemäß einer bestimmten Regel um das Hindernis herum (z. B. im Uhrzeigersinn).
3. **R** scannt erneut nach dem Objekt **S** und wiederholt die Schritte 1 und 2, bis das Ziel erreicht ist.

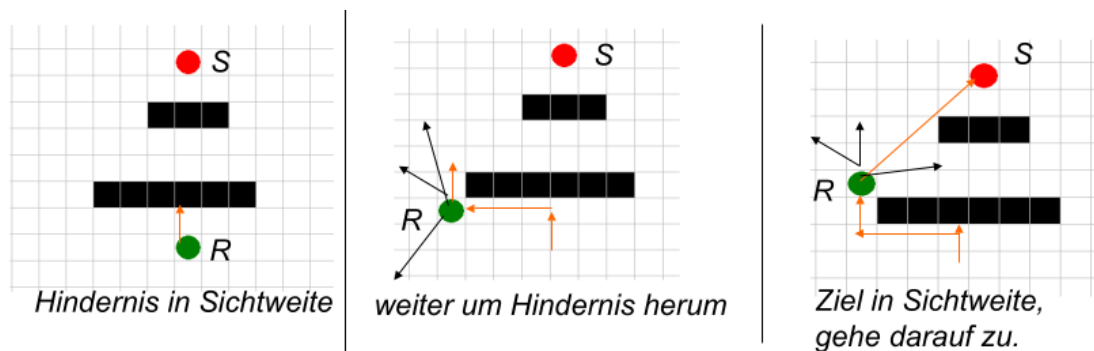


Abbildung 3.1: Beispiel von Bug-Algorithmus Verfahren

3.1.2 Problem mit dem Bug-Algorithmus

In bestimmten Situationen (z.B. siehe Abb. 3.2) ist der Roboter mit dem Ansatz in Abschnitt 3.1.1 nicht in der Lage, das Ziel zu finden.

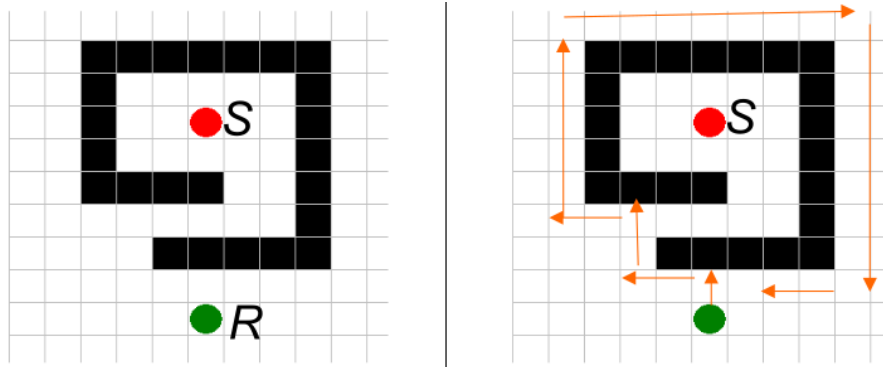


Abbildung 3.2: Der Roboter kann das Ziel nicht sehen

Der Algorithmus muss verbessert werden, z. B. durch Bewegen gegen den Uhrzeigersinn, um eine Bewegung in einem kontinuierlichen Kreis zu vermeiden.

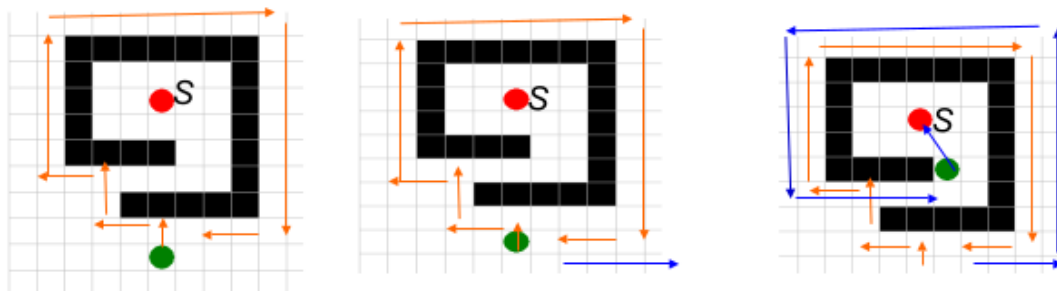


Abbildung 3.3: Gegen den Uhrzeigersinn bewegen, um das Ziel zu erreichen

3.2 Repräsentation von Suchräumen

3.2.1 Suchraum als Karte

Ein Suchraum wird normalerweise als grafische Karte dargestellt. Diese Karten können als Wegenetz oder als Gitter mit benachbarten Zellen dargestellt werden, wie in Abbildung 3.4 gezeigt.

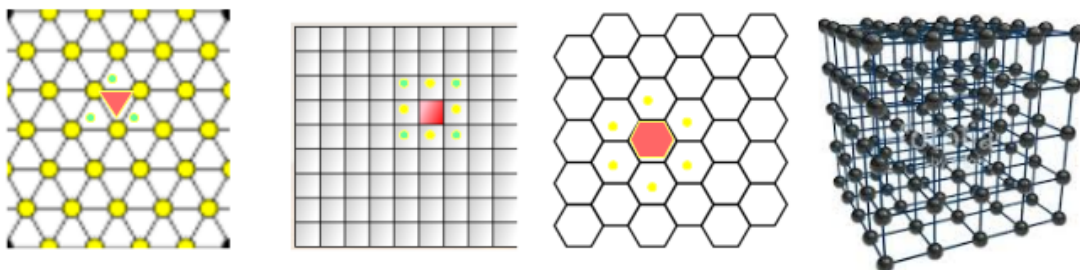


Abbildung 3.4: Beispiele von Gittermustern

Je nach verwendetem Gittermuster werden unterschiedliche Suchgraphen basierend auf der Anzahl der Nachbarn jeder Zelle im Gitter gebildet. Zum Beispiel: Eine dreieckige Zelle hat sechs Nachbarn, eine quadratische Zelle hat 4 (oder 8, wenn Diagonalen erlaubt sind) und eine sechseckige Zelle hat 6.

3.2.2 Charakterisierung von Suchproblemen in Graphen

Um ein Suchproblem in einem Graphen darzustellen, ist es wichtig, einen **Startzustand S**, einen gewünschten **Endzustand Z** und die **möglichen Aktionen zu definieren, um von einem Zustand zum anderen zu gelangen**.

Um diese Suchproblemlogik anzuwenden, in einem gerichteten Graphen G :-

- Zustände: Knoten im Graph G
- Anfangs- und Zielzustand: ausgezeichnete Knoten in G
- Zustandsübergänge: Entspricht dem Passieren von Kanten in G

3.3 Wegsuche als systemstisches Ablaufen von Graphen

Bei der Wegfindung mit Hilfe eines Graphen müssen ein **Startknoten** und eine **Funktion zum Testen**, ob der Zielknoten erreicht wurde, definiert werden. Mit Hilfe des Startknotens und dieser Funktion kann eine Folge von Knoten gefunden werden, die die Testfunktion erfüllen können. Wichtig ist, dass die für die Sequenz ausgewählten Knoten **benachbarte Knoten sind, die durch Kanten verbunden** sind.

3.3.1 Expliziter Graphen und sukzessive Expansion

Es ist wichtig, zwischen einem expliziten Graphen und einem “gedachten” Graphen als Repräsentation eines Suchraums für Lösungen zu unterscheiden.

Expliziter Graphen

Eine explizite Darstellung der Knoten und Kanten ist verfügbar (z.B Adjazentmatrix). Dies ist nur für kleinere Knotenmengen praktikabel und die meisten Probleme erfordern keine vollständige Repräsentation des Suchgraphen.

Sukzessive Expansion

Diagramm ist verfügbar, aber nicht vollständig explizit. Der Suchprozess erfolgt durch Knotenerweiterung, indem Knoten für Knoten vorgegangen wird.

3.3.2 Generelle Wegfindungsstrategie

Zyklen und Mehrfachbesuche sind zu vermeiden, da sie den Suchbaum exponentiell wachsen lassen können. Generell sollte man keinen bereits benutzten Weg nochmal gehen, keine Wege mit Zyklen kreieren, einen bereits besuchten oder ausgebauten Zustand nicht nochmal besuchen oder erzeugen.

3.3.3 Generelle Bewertung von Suchverfahren

Bewertungskriterien eines Pfadfindungsprozesses sind wie folgt:

Korrektheit

Es ist wichtig, dass die Wegfindungslösung tatsächlich eine Lösung des Problems ist.

Vollständigkeit

Existiert eine Lösung, terminiert der Algorithmus nach endlicher Zeit und generiert eine Lösung.

Optimalität

Die optimalste Lösung wird gefunden, wenn mehrere möglich sind.

Zeitkomplexität

Die Zeit, die im worst-case/average-case benötigt wird, um eine optimale Lösung zu finden.

Speicherkomplexität

Wie viel Speicher, die im worst-case/average-case benötigt wird.

3.4 Arten von Suchverfahren

Es gibt viele Möglichkeiten, eine Wegfindungssuche durchzuführen. Diese nächsten Abschnitte werden darauf eingehen.

3.4.1 Breitensuche in expliziten Graphen

Breitensuche ist auf Englisch “breadth first search”. Für diesen Algorithmus werden für jeden Knoten zusätzliche Daten benötigt: **noch nicht gesucht** (weiß dargestellt), **entdeckt, aber noch nicht verarbeitet** (grau dargestellt), **verarbeitet** (schwarz dargestellt).

Überblick über den Ablauf:

1. Wähle einen Startknoten aus dem Graphen und legen Sie diesen in die Warteschlange, Q. Alle Knoten in Q sind grau markiert.
2. Solange es Elemente in Q gibt, markiere alle Nachbarn der Elemente grau und füge diese der Warteschlange Q hinzu.
3. Nachdem alle Nachfolger eines Knotens grau markiert wurden, markiere den Knoten schwarz und entferne ihn aus der Warteschlange. Prüfen Sie, während Sie die Nachfolgerknoten grau markieren, ob der zu markierende Knoten der Zielknoten ist.
4. Wiederhole Schritt 2 und Schritt 3, bis die Warteschlange Q leer ist.

Komplexitätsbetrachtung

Speicherbedarf und Zeitbedarf sind exponentiell. Dies führt dazu, dass große Graphen unlösbar sind und sogar relativ kleine Graphen zu lange brauchen, um praktikabel zu sein (z.B Graphen Tiefe 12 brauch 35 Jahre Rechenzeit).

3.4.2 Uniforme Kostensuche

Bei diesem Suchalgorithmus wird die Breitensuche so verändert, dass auch die **Kosten der Nachbarknoten** berücksichtigt werden. Da die Kosten zweier benachbarter Knoten normalerweise bereits bekannt sind, kann **die Warteschlange in einen Heap umstrukturiert werden**, der in **aufsteigender Reihenfolge der Pfadkosten** sortiert ist. Auf diese Weise wird gehofft, dass der kürzeste Weg vom Startknoten zum Zielknoten gefunden wird.

3.4.3 Modifizierte Uniforme Kostensuche / Dijkstra

Um eine optimale Lösung für ein Wegsucheproblem zu finden, ist es notwendig, alternative Pfade parallel zu konstruieren. Wenn sich diese Pfade am selben Punkt treffen, kann der **kürzeste Pfad beibehalten werden**, während der Rest aus dem Suchbaum entfernt wird, um die kürzeste Gesamtroute zu finden.

Wenn ein Pfad zum Zielknoten gefunden wird, werden alle **anderen parallelen Pfade fortgesetzt**, es sei denn, ihre Kosten übersteigen den bereits gefundenen Pfad. Dies führt zu einer besseren Effizienz bei der Suche nach dem kürzesten Weg.

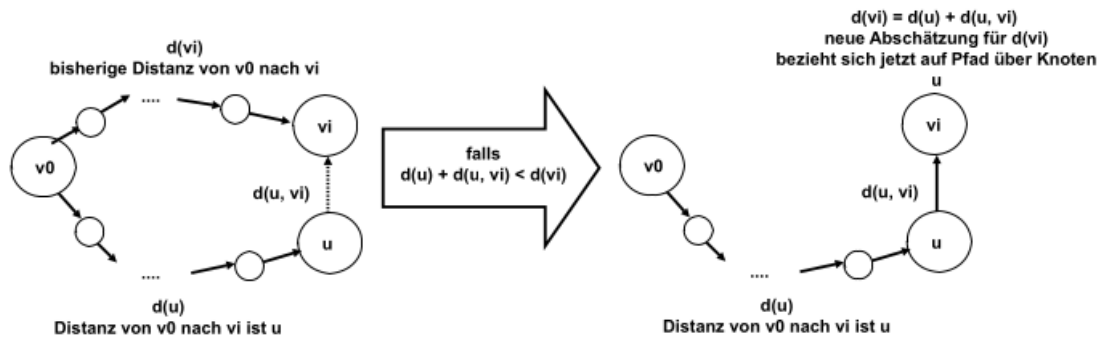


Abbildung 3.5: Dijkstra-Algorithmus Pfadsuche

Dieser Algorithmus ist in Abbildung 3.5 demonstriert. Anhand der Abbildung kann man sehen, wie die beiden Routen von v_0 nach v_i verglichen werden und die Kosten verglichen werden, um nur die kürzere Route beizubehalten.

3.4.4 Ablauf eines Graphen mit Tiefensuche

Im Gegensatz zur Traversierung mit Breitensuche verläuft die Traversierung mit Tiefensuche auf möglichst langen Wegen und führt die Weiß-Grau-Schwarz-Markierung (siehe Abschnitt 3.4.1) durch. Dieser Algorithmus wird normalerweise rekursiv implementiert.

Ablauf von Tiefensuche

1. Wähle einen Startknoten v_s . Markiere den Knoten als grau ein.
2. Für jeden Nachbarknoten v_k von v_s , der weiß markiert ist, führe folgendes durch:
 - (a) Füge v_k zum Stack hinzu
 - (b) Markiere v_k als grau
 - (c) Falls v_k Nachbarn hat, führe Schritt (2) rekursiv aus
 - (d) Wenn v_k keine Nachbarn hat oder seine Nachbarn schwarz markiert sind, markiere v_k schwarz und entferne v_k vom Stack
3. Der Algorithmus endet, sobald der Stack leer ist.

Komplexitätsbetrachtung

Nur benachbarte Knoten des aktuellen Suchpfads werden auf dem Stack gespeichert. Dies bedeutet viel weniger Knotenerweiterungen im Vergleich zur Breitensuche, was zu weniger Speicherverbrauch führt. Die Zeitkomplexität ist jedoch ebenso wie die Breitensuche exponentiell.

3.4.5 Tiefensuche und Backtracking-Algorithmen

In realen Anwendungen beinhaltet eine Lösung die Verwendung vieler verschiedener Komponenten. Die endgültige Lösung verwendet daher normalerweise viele Teillösungen, die möglicherweise nicht richtig erweitert werden können (z.B: erreicht die Teillösung eine Sackgasse). In diesen Fällen muss die Teillösung durch eine weniger komplexe Lösung ersetzt werden.

Genereller Ablauf von einem Backtracking-Algorithmus

Gegeben sind ein Array “SolutionComponents”, das alle Werte der endgültigen Lösung enthält, und die Funktionen “FirstTrialValue()”, “NextTrialValue()” und “CheckValid()”.

1. Der ersten Komponente wird der erste Versuchswert gegeben:

```
SolutionComponents[0] = FirstTrialValue(0)
```

2. Die Gültigkeit der Lösung wird überprüft:

```
CheckValid(SolutionComponents)
```

3. Wenn die bisherige Lösung gültig ist, fahren Sie mit dem nächsten Wert fort ($i = 1, 2, 3, \dots$) und überprüfe den Wert erneut

```
SolutionComponents[i] = FirstTrialValue(i)  
CheckValid(SolutionComponents)
```

4. Wenn der CheckValid-Funktion an einem bestimmten Punkt false zurückgibt, versuche es mit den nächsten Werten. Wenn alle Werte ebenfalls falsch zurückgeben, führe ein Backtracking durch, indem Sie i um i reduzieren und den nächsten Wert für dieses i versuchen. Erhöhen Sie dann i wieder um 1 und probieren Sie alle Werte aus.
5. Schritt 4 wird so oft wie nötig wiederholt. Für den Fall, dass i null erreicht und es keine Versuchswerte mehr für $i = 0$ gibt, gibt es keine mögliche Lösung.

3.4.6 Limitierte Tiefensuche

Es ist möglich, die Nachteile der Tiefensuche zu vermeiden, indem man eine maximale Tiefe des Pfades einstellt. Das macht die Suche vollständig aber nicht immer optimal. Um diese Idee zu erweitern, kann eine iterative Tiefensuche versucht werden.

3.4.7 Iterative Tiefensuche

Bei dieser Methode wird die Tiefe mit jeder Iteration erhöht, um sicherzugehen, dass eine Lösung gefunden wird.

3.4.8 Bidirektionale Suche

Anstatt nur vom Startknoten aus zu beginnen, führen Sie eine Suche vom Start- und vom Zielknoten aus durch. Wenn sich die beiden Verfahren in der Mitte treffen, ist eine Lösung gefunden.

Literaturverzeichnis