Modul: Künstliche Intelligenz
Universität Rostock

Projektaufgabe: Wumpuswelt

Lösung und Dokumentation

Sven Berger, 209204572
Martin Müller, 209203328
Nils Richter, 6200839
Stephan Saß, 209204373
John Trimpop, 209206264

Inhaltsverzeichnis

Inhaltsverzeichnis....2
Allgemeines....3
Suche....4
2.1 Einrichten des Agenten....4
2.2 Implementieren der Suchalgorithmen....4
2.3 Heuristik für A*....4
2.4 Testfälle der Suchalgorithmen....5
2.5 Gegenbeispiel....6
Genetische Algorithmen....7
3.1 Reaktiver Agent....7
3.2 Regeln 7
3.3 Genetischer Algorithmus....7
3.4 Einfluss von Rekombination, Mutation und Größe der Ausgangspopulation....7
3.5 Einbindung stochastischer Wahrscheinlichkeiten....7
Fragen....8

Allgemeines

Im Rahmen des Moduls Künstliche Intelligenz der Universität Rostock für den Bachelor Studiengang Informatik ist ein Gruppenprojekt in Form von einer Programmier- und Analyseaufgabe zu erstellen.
Die Hauptaufgabe besteht darin in der sogenannten, vorgefertigten „Wumpuswelt“ einen intelligenten Agenten zu implementieren, welcher in der Lage ist, die in der Aufgabenstellung erwähnten Ziele selbständig zu lösen.
Die Wumpuswelt ist ein zweidimensionales, Schachbrett ähnliches Feld, das durch Wände begrenzt ist, Gruben- und Goldfelder enthält und in dem sich der Agent und der Wumpus, ein übel riechendes, gefährliches Monster, frei bewegen können.
Der Agent und der Wumpus verbrauchen jeweils einen Zug, um sich drehen, fortbewegen oder sonstige Aktionen durchführen zu können; Wände sind dabei nicht passierbar, ebenfalls kann nicht diagonal gegangen werden. Der Wumpus hat um sich herum eine acht Felder große Geruchsausdehnung, welche der Agent in der Lage ist zu erkennen, sobald er sich in einem direkt daneben befindlichen Feld aufhält.

Das Projekt ist in drei verschiedene Grundaufgaben unterteilt, welche verschiedene Modifikationen der Wumpuswelt voraussetzen.
Die erste Aufgabe beschränkt sich auf Suchverfahren und ihre Effizienz. Der Wumpus befindet sich noch nicht in der Höhle. Aufgabe zwei beschäftigt sich mit genetischen Algorithmen. Der Agent „kämpft“ nun gegen den Wumpus und sollte dies möglichst effizient vollführen. Die letzte Aufgabe bezieht sich auf etwaige Verbesserungsvorschläge. Insgesamt können 100 Punkte erzielt werden.

Suche

2.1 Einrichten des Agenten

Sehen Sie sich für diese Aufgabe bitte die entsprechenden Java-Quelltexte und Java–Klassen.

2.2 Implementieren der Suchalgorithmen

Neben den drei geforderten Algorithmen, haben wir noch eine Erweiterung der normalen A*-Suche implementiert (A*-Spezial). A*-Spezial versucht neben der Minimierung der besuchten Felder auch die Anzahl der benötigten Aktionen zu minimieren. Sehen Sie sich für diese Aufgabe bitte die entsprechenden Java-Quelltexte und Java–Klassen:
GoldsammelAgent.java, TestGoldsammelAgent.java, WegMitLaenge.java, Wegpunkt.java, WegpunktQualitaet.java

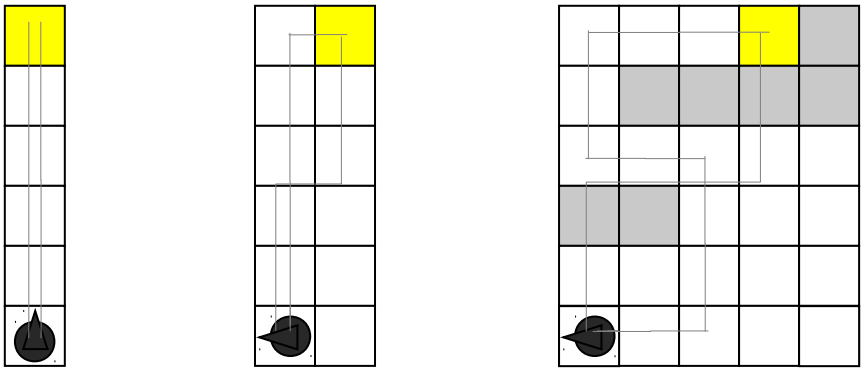
2.3 Heuristik für A*

Als Heuristik für A* wird der Luftlinienabstand zwischen dem Agenten und den Goldfeldern verwendet.

Beweis für die Gültigkeit der Heuristik:
Sei nun *n* die Länge eines solchen Luftlinienabstandes. Diese wird mittels Anwendung des Satzes des Pythagoras wie folgt berechnet:

$$n = \sqrt{(GoldX - AgentX)^2 + (GoldY - AgentY)^2}$$

Der Agent muss nun im besten Falle, wenn die Luftlinie genau senkrecht oder waagerecht verläuft und der Agent ihr folgen kann, genau *n* Schritte tätigen. Weil er nicht diagonal gehen kann und/oder sich Gruben innerhalb dieser Luftlinie befinden könnten, ist der tatsächliche Weg in jedem anderen Fall länger als die Luftlinie.
Illustration:



Best Case: n Schritte

Zusätzliche Schritte notwendig

Zusätzliche Schritte notwendig

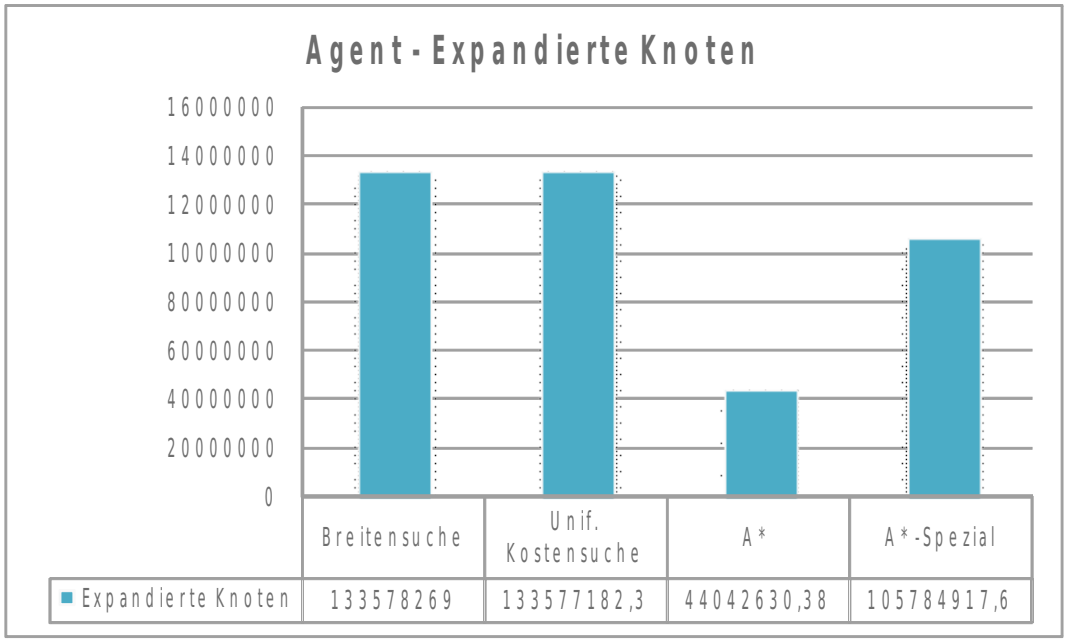
Dies ist eine gültige, unterschätzende Heuristik (*n* ≤ tatsächlicher Weg).

2.4 Testfälle der Suchalgorithmen

Die Suchverfahren wurden auf 7701 Höhlen getestet, wobei die 7701 Höhlen zwar zufällig, aber für alle Suchverfahren die gleichen 7701 Höhlen waren.

Suchver-fahren	Anzahl der Simulationen	Durchschnitt aller besuchten Felder	Durchschnitt der Gesamtpunktzahl	Durchschnitt der expandierten Knoten
BS	7701	151260	1636500	133578269
UK	7701	151260	1636500	133577182
A*	7701	151260	1636500	44042630
A*-Spez.	7701	151260	1636500	105784918

Es ist aus der Tabelle schnell ersichtlich, dass A*, das effizienteste Suchverfahren in Bezug auf die zu untersuchenden Parameter, Anzahl der besuchten Felder und Anzahl der expandierten Knoten, ist. Die grafische Darstellung der expandierten Knoten zeigt dies ebenfalls sehr deutlich:
TODO (John bis Donnerstag 20.00 Uhr): die nachkommastellen im diagramm rausnehmen



Nur weil dies in der Aufgabenstellung explizit gefordert wurde, fügen wir hier das Diagramm der besuchten Felder ein:

TODO (John bis Donnerstag 20.00 Uhr): diagramm einfügen

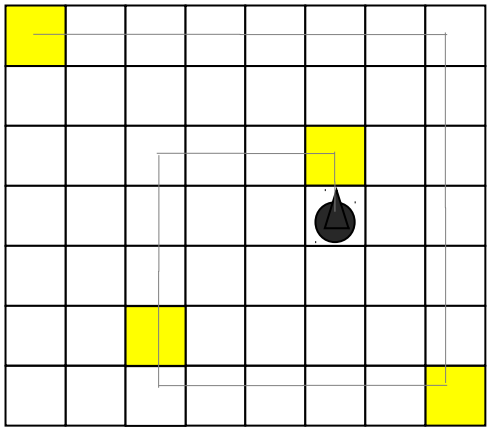
Zwar berechnet A*-Spezial meist einen effizienteren Weg, in Bezug auf die Anzahl der benötigten Aktionen, als A*, weil die jedem Agenten zur Verfügung stehenden 500 Aktionen aber in allen getesteten Fällen mit allen Verfahren

zum Einsammeln aller Goldstücke reichte, ist A* mit den wenigsten expandierten Knoten das in dieser Situation effizienteste. Breitensuche und uniforme Kostensuche sind aufgrund der Rahmenbedingung, „konstante Kosten“, das gleiche Suchverfahren und die leicht unterschiedlichen Ergebnisse ergeben sich nur aus der „zufälligen“ Art der zu verwendenden SimpleEventQueue (siehe Kommentar im Quelltext).

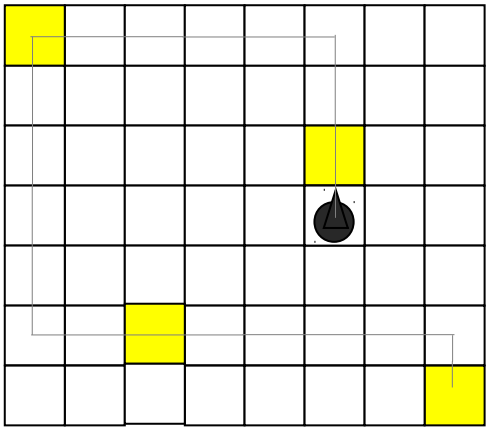
2.5 Gegenbeispiel

Nein, der Weg, den der Agent wählt, ist global gesehen *nicht* optimal. Der tatsächlich kürzeste Weg zwischen mehreren Goldfeldern kann geringer ausfallen, als ein Weg, der von unseren Suchverfahren, welche nur lokal kürzeste Wege zum Gold finden, ermittelt wird. Dies gilt für alle Suchverfahren gleichermaßen.
Wie im unten aufgeführten Beispiel leicht zu erkennen ist, wäre der linke Weg eine möglicher Weg bei Breitensuche, uniformer Kostensuche, A* und A*-Spezial. Der beste Weg (rechts im Beispiel) ist mit fünf besuchten Feldern weniger aber noch effizienter.

Gegenbeispiel:



Weg durch A*-Spezial (26 besuchte Felder)



Ein besserer Weg (21 besuchte Felder)

Genetische Algorithmen

3.1 Reaktiver Agent

Sehen Sie für diese Aufgabe bitte die entsprechenden Java-Quelltexte und Java–Klassen:
AgentenAktion.java, AgentenVorgehen.java, AgentenZustand.java, Bewegen2Aktion.java, BewegenAktion.java, FliehenAktion.java, Gefangen.java, GenetischerAlgo.java, GoldaufhebenAktion.java, GoldGesehen.java, JagenAktion.java, NichtsFestgestellt.java, PfeilabschiessenAktion.java, RandGesehen.java, RegelAgent.java, RegelAktionID.java, RegelAktion.java, Regel.java, SituationsStatusID.java, SituationsStatus.java, TestRegelAgent.java, WartenAktion.java, WumpusGerochen.java, WumpusGesehen.java, WumpusVoraus.java
Die wichtigsten Klassen dabei sind

1. RegelAgent – der Agent
2. Regel – Kapselung der Regel mit der Situation in der sie angewendet werden soll und den dann auszuführenden Aktionen
3. SituationsStatus – Abstrakte Basisklasse aller Situationsbeschreibungen
4. RegelAktion – Abstrakte Basisklasse für alle Aktionen die der Agent ausführen kann

3.2 Regeln

Unsere Regeln lauten:

1. Nichts wurde festgestellt => wir bewegen uns in der Höhle
2. Es wurde Gold gesehen => wir wollen dieses aufheben
3. Sind wir an einem Spielfeldende => bewegen wir uns trotzdem weiter
4. Wenn der Wumpus gerochen wurde => jagen wir ihn
5. Wenn der Wumpus gerochen wurde => fliehen wir
6. Wenn der Wumpus gesehen wurde, fliehen wir
7. Wenn der Wumpus gesehen wurde und wir gefangen sind => jagen wir
8. Wenn der Wumpus gesehen wurde => schiessen wir einen Pfeil ab
9. Wenn wir gefangen sind (umgeben von Gruben/Spielfeldende) => gehen wir auch über Gruben

Für die Implementierung dieser Regeln wird jeweils eine neue Instanz der Regel Klasse angelegt und mit den entsprechenden SituationsStatus Klassen und der jeweiligen Aktion gefüttert. Die Möglichkeit einer Regel mehrere Aktionen zuzuweisen nutzen wir mit diesem Regelsatz nicht.

3.3 Genetischer Algorithmus

Durch das Testen der Fitness auf verschiedenen Höhlen, kann es sein, das verschiedene Agenten verschieden „gute“ Höhlen bekommen und die Fitness

der Agenten eigentlich nicht mehr direkt mit einander verglichen werden kann. Man könnte viel mehr Höhlen pro Agenten testen, wodurch sich die Vor-/Nachteile ausgleichen würde oder man übergibt unserem Programm mit dem Kommandozeilenargument RXXXL, wobei XXX für einen Long steht ein Seed, welches es vor jedem Ausführen von ExerciseTwo setzt und somit alle Agenten an der gleichen Höhle testen lässt.

3.4 Einfluss von Rekombination, Mutation und Größe der Ausgangspopulation

/////////////// Neuer Durchgang //
Ausgangs Populationsgroesse: 50
Mutationswahrscheinlichkeit: 5%
Rekombinationsstelle (-1==Zufall): 3
Fortpflanzungsteilnehmer (die besten x%): 30%
Fitness der 1. Generation: 514.0
Fitness der 2. Generation: 210.0
Fitness der 3. Generation: 572.0
Fitness der 4. Generation: 301.0
Fitness der 5. Generation: 600.0
Fitness der 6. Generation: 24.0
Fitness der 7. Generation: 447.0
Fitness der 8. Generation: 696.0
Fitness der 9. Generation: 401.0
Fitness der 10. Generation: -135.0
Fitness der 11. Generation: 419.0
Fitness der 12. Generation: 660.0
Fitness der 13. Generation: 349.0
Fitness der 14. Generation: 559.0
Fitness der 15. Generation: 596.0
Fitness der 16. Generation: 683.0
Fitness der 17. Generation: 228.0
Fitness der 18. Generation: 515.0
Fitness der 19. Generation: -213.0
Fitness der 20. Generation: 817.0

/////////////// Neuer Durchgang //
Ausgangs Populationsgroesse: 100
Mutationswahrscheinlichkeit: 5%
Rekombinationsstelle (-1==Zufall): 3
Fortpflanzungsteilnehmer (die besten x%): 30%
Fitness der 1. Generation: 592.0
Fitness der 2. Generation: 295.0
Fitness der 3. Generation: 258.0
Fitness der 4. Generation: 283.0
Fitness der 5. Generation: 262.0

Fitness der 6. Generation: 352.0
Fitness der 7. Generation: 481.0
Fitness der 8. Generation: 445.0
Fitness der 9. Generation: 397.0
Fitness der 10. Generation: 569.0
Fitness der 11. Generation: 611.0
Fitness der 12. Generation: 350.0
Fitness der 13. Generation: 216.0
Fitness der 14. Generation: 665.0
Fitness der 15. Generation: 652.0
Fitness der 16. Generation: 381.0
Fitness der 17. Generation: 317.0
Fitness der 18. Generation: 129.0
Fitness der 19. Generation: 447.0
Fitness der 20. Generation: 611.0

////////////// Neuer Durchgang //
Ausgangs Populationsgroesse: 50
Mutationswahrscheinlichkeit: 30%
Rekombinationsstelle (-1==Zufall): 3
Fortpflanzungsteilnehmer (die besten x%): 30%
Fitness der 1. Generation: 563.0
Fitness der 2. Generation: 722.0
Fitness der 3. Generation: 794.0
Fitness der 4. Generation: 213.0
Fitness der 5. Generation: 68.0
Fitness der 6. Generation: 565.0
Fitness der 7. Generation: 438.0
Fitness der 8. Generation: 510.0
Fitness der 9. Generation: 244.0
Fitness der 10. Generation: 587.0
Fitness der 11. Generation: 277.0
Fitness der 12. Generation: 363.0
Fitness der 13. Generation: 673.0
Fitness der 14. Generation: 411.0
Fitness der 15. Generation: 340.0
Fitness der 16. Generation: 626.0
Fitness der 17. Generation: 149.0
Fitness der 18. Generation: 613.0
Fitness der 19. Generation: 239.0
Fitness der 20. Generation: 494.0

////////////// Neuer Durchgang //
Ausgangs Populationsgroesse: 50
Mutationswahrscheinlichkeit: 5%
Rekombinationsstelle (-1==Zufall): -1
Fortpflanzungsteilnehmer (die besten x%): 30%
Fitness der 1. Generation: 697.0

Fitness der 2. Generation: 498.0
Fitness der 3. Generation: 513.0
Fitness der 4. Generation: 399.0
Fitness der 5. Generation: 237.0
Fitness der 6. Generation: 555.0
Fitness der 7. Generation: 765.0
Fitness der 8. Generation: 395.0
Fitness der 9. Generation: 679.0
Fitness der 10. Generation: 653.0
Fitness der 11. Generation: 362.0
Fitness der 12. Generation: 576.0
Fitness der 13. Generation: 653.0
Fitness der 14. Generation: 747.0
Fitness der 15. Generation: 691.0
Fitness der 16. Generation: 680.0
Fitness der 17. Generation: 616.0
Fitness der 18. Generation: 203.0
Fitness der 19. Generation: 222.0
Fitness der 20. Generation: 644.0

TODO (nicht ich):
auswerten, schlußfolgerung

3.5 Einbindung stochastischer Wahrscheinlichkeiten

Um stochastische Wahrscheinlichkeiten zu ermöglichen, würde es bei uns reichen in der Klasse RegelAktion einfach ein Attribut Wahrscheinlichkeit einführen, und bei Berechnung der Aktion innerhalb der Klasse testen, ob wir die Aktion jetzt wirklich ausführen sollen, ansonsten liefert man direkt null zurück und es wird die nächste/alternative Aktion erledigt.
TODO (Sven/Stefan): Wie verändert dies die Erfolgsaussichten des Algorithmus?

Fragen

TODO (alle):
Ideen sammeln - hier rein schreiben

Martin, Stefan und Nils gucken dann, wie technisch Umsetzbar