Seminar Spiele KI:

Entscheidungsarchitekturen für Spiele

Informatik-Seminar, Sommersemester 2007, Prof. Dr. Iwanowski
Felix Gebauer, winf2662@fh-wedel.de

Inhaltsverzeichnis

1. Einleitung 3
2. Entscheidungsbäume 4
3. Nutzentheorie 5
    a. Bayessches Netz 7
    b. Evidenztheorie von Dempster und Shafer 8
4. Goal-Oriented Action Planning (GOAP) 10
5. Influence Maps, Resource Allocation Trees, Dependency Graphs 14
6. Literaturverzeichnis 20

1. Einleitung

Die Entscheidungsfindung ist ein sehr wichtiges Element einer Künstlichen Intelligenz. Dies wird daran deutlich, dass eine Künstliche Intelligenz anhand ihrer Entscheidungen beurteilt wird; eine KI, die dauernd falsche Entscheidungen trifft, würde also nicht als „gute“ KI bezeichnet werden. Die Qualität der Entscheidungen hat also direkte Auswirkungen auf die Qualität der KI und dementsprechend auch auf die Qualität des Spiels, in dem diese KI implementiert ist. Eine gute KI verstärkt den Spielspaß des Spielers und verlängert somit den Produktlebenszyklus des Spiels. Anders ausgedrückt: eine gute KI ist bares Geld wert! Leider wird die KI allgemein von vielen Programmierern etwas vernachlässigt, da Game Design und Grafik im Vordergrund stehen. Im Folgenden werden einige Architekturen und Methoden vorgestellt, mit denen sich gute Entscheidungsfindungen in Spielen implementieren lassen, ohne dabei allzu großen Aufwand zu haben. Bevor man sich mit konkreten Architekturen befasst, macht es Sinn, sich erst einmal Anforderungen anzusehen, die an Entscheidungsarchitekturen gestellt werden müssen:

Die Entscheidungen, die eine KI trifft, sollten

- anpassungsfähig,
- richtig,
- realistisch,
- möglichst menschlich sein und
- Spaß machen (den Spieler herausfordern).

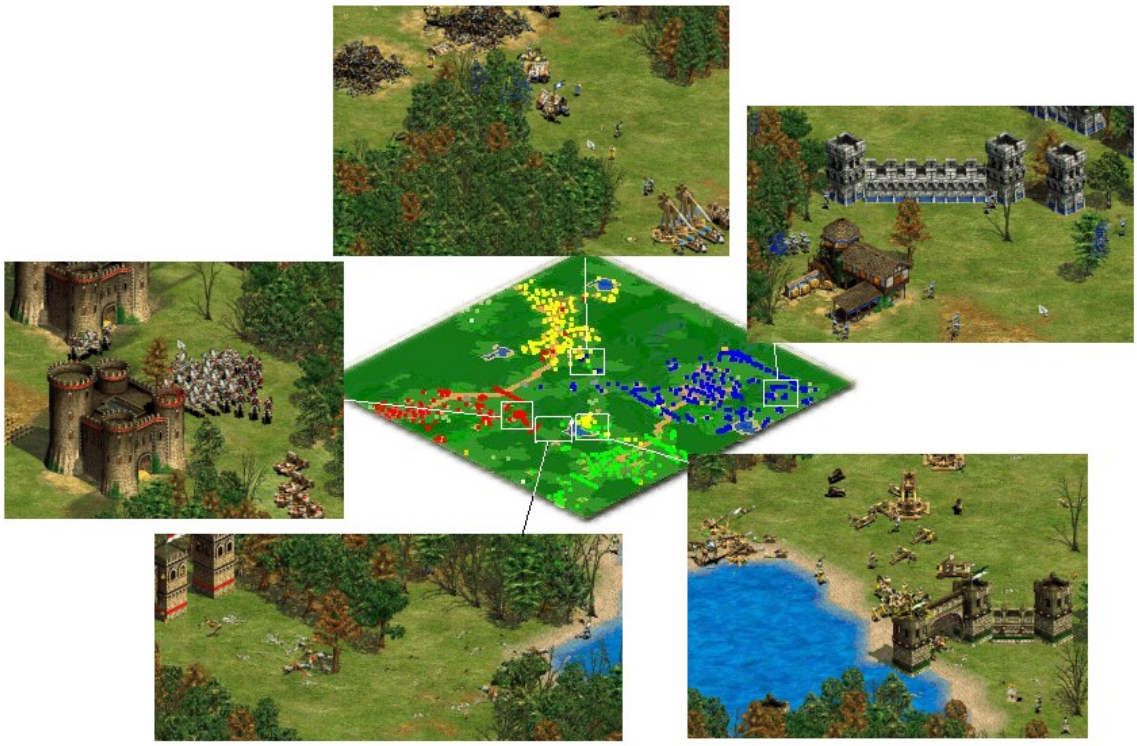
Eine KI sollte möglichst verschiedene Persönlichkeiten besitzen und ausleben können.
Eine KI sollte verschiedene Schwierigkeitsgrade besitzen.

Auf diese Anforderungen wird im Rahmen der verschiedenen Architekturen noch genauer eingegangen werden. Weiter ist es wichtig, zu wissen, welche konkreten Entscheidungen in Spielen überhaupt getroffen werden müssen, sowohl vom Spieler, als auch von der KI.

Mögliche Entscheidungen in Strategiespielen sind unter anderem:

- Welchen Gegner greife ich an?
- Wo greife ich den Gegner an und wie greife ich ihn an?
- Wo muss ich mit einem gegnerischen Angriff rechnen und mit welcher Art von Angriff muss ich rechnen?
- Welche Einheiten/Gebäude/Technologien brauche ich?
- Welche Ressourcen brauche ich dafür und wie komme ich an diese heran?
- Wie setze ich gegebene Ressourcen am besten ein?
- Wie setze ich gegebene Einheiten am besten ein?

Wenn man sich nun mit echten Entscheidungen und Anforderungen beschäftigt hat, kann man sich verschiedene Entscheidungsarchitekturen ansehen. Um die Architekturen zu visualisieren, werden sie direkt beispielhaft angewendet. Dazu dient folgendes Beispiel:



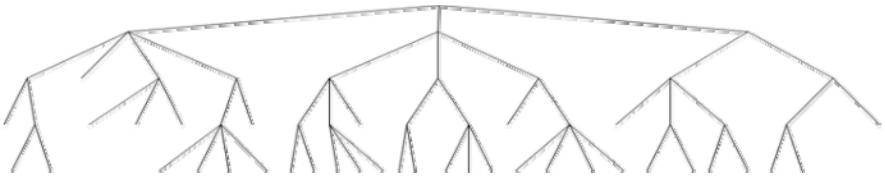
Hier sieht man eine Spielsituation aus dem Spiel „Age of Empires 2: The Age of Kings“. In diesem Beispiel sind wir der rote Spieler und wollen uns entscheiden, ob und wie wir einen Angriff durchführen. Dabei ist der gelbe Spieler unser Verbündeter und der blaue und der grüne Spieler sind unsere Gegner, ebenfalls untereinander verbündet. Man kann auf den Bildausschnitten unsere eigene Armee (umfasst einige Reiter und Belagerungseinheiten, Bild links) sehen, sowie einige andere Umstände: der grüne Spieler hat gerade versucht, uns anzugreifen, ist aber gescheitert (Bild unten links), der gelbe Spieler greift gerade den grünen Spieler an (Bild unten rechts), wird aber gleichzeitig vom blauen Spieler angegriffen (Bild oben links), welcher wiederum in seinem Lager noch einige Einheiten in Reserve hat (Bild oben rechts). Beim Spielmodus handelt es sich um Königsmord, man muss also den König des Gegners töten, um das Spiel zu gewinnen. An diesem Beispiel werden die vorgestellten Architekturen später angewandt werden.

2. Entscheidungsbäume

Die einfachste Möglichkeit, eine KI in einem Spiel Entscheidungen treffen zu lassen, sind wohl Entscheidungsbäume. Bei Entscheidungsbäumen werden verschiedene Regeln ausgewertet und bestimmte Aktionen ausgeführt oder weitere Regeln ausgeführt, sobald eine Regel zutrifft; dabei ist eine sehr tiefe Schachtelung möglich.

Ein einfacher Entscheidungsbaum für die Situation, dass ein Charakter in einem Rollenspiel auf ein Monster trifft und sich entscheiden muss, wie er nun mit diesem Monster umgeht, könnte so aussehen:

Der Charakter entscheidet sich also für eine Aktion, je nach dem, wie viele Lebenspunkte er hat. In dieser Größenordnung ist ein Entscheidungsbaum sehr leicht anwendbar, jedoch werden diese Größen in komplexeren Situationen schnell überschritten:



Ein Entscheidungsbaum hat insgesamt betrachtet sehr viele Nachteile:

- er ist unflexibel; der Programmierer muss jede mögliche Spielsituation bedenken und programmieren
- er wird sehr schnell zu komplex, ist dann kaum noch vernünftig zu lesen
- in komplexen Bäumen sind Fehler nur sehr schwer zu finden und zu beheben; eine effiziente Wartung ist kaum möglich
- für den Spieler ist er schnell zu durchschauen, da die KI immer gleich reagieren wird: keine große Herausforderung, weniger Spielspaß
- das resultierende Verhalten der KI ist nicht unbedingt menschlich, da ein Mensch in der Regel auch nicht mit Hilfe von Entscheidungsbäumen entscheidet

3. Nutzentheorie

Die Nutzentheorie (englisch: Utility Theory) ist eine sehr gute Methode, um eine KI gute Entscheidungen treffen zu lassen. Sie hat ihren Ursprung in der Ökonomie, in der Geld als Entscheidungsgröße angewandt wird. Ein Ökonom entscheidet sich also für die Option, bei der er den größten Nutzen, beziehungsweise am Ende das meiste Geld hat. Obwohl sie ihre Wurzeln in der Volkswirtschaftslehre hat, lässt sich die Nutzentheorie auch allgemeiner anwenden, zum Beispiel um eben in Spielen Entscheidungen zu treffen. Zunächst muss der Begriff der Nutzenfunktion geklärt werden: Eine Nutzenfunktion ordnet einer Situation einen Wert zu, der die Attraktivität, also den Nutzen, dieser Situation ausdrückt. Wenn man eine Nutzenfunktion hat, kann man zu einer Aktion ihren erwarteten Nutzen berechnen; das ist der Nutzen, den man erwarten kann, wenn man diese Aktion ausführt. Dazu bestimmt man für eine Aktion alle Situationen, die auf diese Aktion folgen könnten, und multipliziert für jede Situation die Eintrittswahrscheinlichkeit mit dem Nutzen. Wenn man dies über alle Situationen summiert, die eine Aktion auslösen könnte, erhält man den erwarteten Nutzen einer Aktion:

$$EU(A \mid E) = \sum P(Result_i(A) \mid E, Do(A)) * U(Result_i(A))$$

Hierbei ist *EU* der erwartete Nutzen (Expected Utility), *A* eine Aktion, *E* das verfügbare Wissen über die Umwelt (Evidence), *U(S)* unsere Nutzenfunktion und *Result i* eine mögliche Situation, die auf Aktion *A* folgt. Das verfügbare Wissen über die Umwelt muss hier mit einbezogen werden, da sich bei anderem verfügbaren Wissen die Wahrscheinlichkeiten für resultierende Situationen ändern. Ein Agent, der mit Hilfe der Nutzentheorie entscheidet, berechnet nun für jede mögliche Aktion den erwarteten Nutzen und entscheidet sich nach dem so genannten MEU-Prinzip für eine Aktion. Das MEU-Prinzip (MEU = Maximum Expected Utility) sagt aus, dass man diejenige Aktion wählen sollte, die den höchsten erwarteten Nutzen hat.
Das macht man sich am Besten an einem Beispiel etwas verständlicher:

Angenommen, man ist der Gewinner einer Gameshow und hat soeben 1.000.000€ gewonnen. Nun bietet einem der Showmaster an, um den Gewinn zu spielen, und zwar wirft er eine Münze; bei Kopf gewinnt man 3.000.000€, bei Zahl verliert man alles. Die Frage ist nun, ob man dieses Angebot annehmen sollte.
Betrachten wir zunächst, bei welcher Option man den höheren Gewinn erwarten kann (EMV = Expected Monetary Value):

EMV(Annehmen) = ½ * 0€ + ½ * 3.000.000€ = 1.500.000€
EMV(Ablehnen) = 1.000.000€

Der erwarte Gewinn ist also höher, wenn man das Angebot annimmt. Ist das aber wirklich die richtige Entscheidung? Um den erwarteten Nutzen einer Situation zu berechnen, muss man wissen, wie viel Geld man sowieso schon hat, beispielsweise auf seinem Bankkonto, da auch dieses Geld einen Nutzen hat. Wir kommen also zu folgenden Gleichungen:

EU(Annehmen) = ½ * U(k €) + ½ * U(k + 3.000.000€)
EU(Ablehnen) = U(k + 1.000.000€)

k ist hierbei der Betrag, über den man bereits verfügt. Jetzt brauchen wir eine Nutzenfunktion, die einem Betrag einen Nutzen zuordnet. Zum Beispiel könnte man bereits 100.000€ auf der Bank haben und dies mit einem Nutzen von 5 bewerten. Der Nutzen, den die erste Million bringt, ist vergleichsweise hoch, also vielleicht 8. Der Nutzen für die dritte Million steigt wieder relativ gesehen geringer an, die die erste Million mehr wert ist, er könnte also 10 sein. So erhalten wir folgende Werte:

EU(Annehmen) = ½ * 5 + ½ * 10 = 7.5
EU(Ablehnen) = 8

In dem Fall sollte man das Angebot also ablehnen und sich der ersten eigenen Million erfreuen. Nehmen wir nun aber an, man hat bereits 500.000.000€ auf der Bank und nimmt nur zum Spaß an Gameshows teil, dann könnte man davon ausgehen, dass der Nutzen der 500. Million und der der 501. Million etwa gleich sind; wenn man also mehr Geld weniger Geld vorzieht, würde man das Angebot annehmen.

Nun möchte ich noch auf zwei Methoden aus der Wahrscheinlichkeitslehre eingehen, die sich besonders dazu eignen, in Spielen Wahrscheinlichkeiten zu ermitteln und diese für die Berechnung des erwarteten Nutzens zu verwenden.

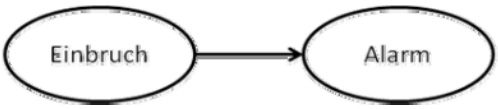
3.a. Bayessches Netz

Das Bayessche Netz (zurückzuführen auf Thomas Bayes, englischer Mathematiker und Pfarrer des 18. Jahrhunderts) dient der Repräsentation von unsicherem Wissen und daraus möglichen Schlussfolgerungen und ist im Prinzip ein gerichteter Graph aus Aussagen (Knoten) und Abhängigkeiten (Kanten). Die Grundlage für das Arbeiten mit Bayesschen Netzen ist das Bayestheorem, welches das Umkehren von Schlussfolgerungen erlaubt:

$$P(A \mid B) = \frac{P(B \mid A) * P(A)}{P(B)}$$

mit P (B | A): Wahrscheinlichkeit, dass Ereignis B auftritt, unter der Bedingung, dass Ereignis A auftritt. So kann man eine bedingte Abhängigkeit, wenn man diese bereits ermittelt hat, umkehren.

Ein einfaches Bayessches Netz könnte jetzt so aussehen:



Nehmen wir an, wir haben zuhause eine Alarmanlage, die Alarm schlagen soll, wenn bei uns eingebrochen wird, und unser Nachbar ruft uns in der Arbeit an und teilt uns mit, dass bei uns der Alarm an ist. Die klassische Wahrscheinlichkeitstheorie würde jetzt die Wahrscheinlichkeit berechnen, dass der Alarm losgeht, wenn eingebrochen wird, wir wollen aber die andere Richtung haben, nämlich: Wie hoch ist die Wahrscheinlichkeit, dass eingebrochen wurde, wenn der Alarm losgeht?

Dazu brauchen wir zunächst die unabhängige Wahrscheinlichkeit dafür, dass an irgendeinem Tag bei uns eingebrochen wird, also P(Einbruch). Angenommen wir wohnen in einem recht sicheren Stadtteil, dann könnte P(Einbruch) = 0.001 sein und P(kein Einbruch) demnach 0.999. Nun brauchen wir die Wahrscheinlichkeit, dass der Alarm losgeht, diese könnte man etwa den Herstellerinformationen entnehmen. Im Beispiel sei die Wahrscheinlichkeit, dass der Alarm losgeht, wenn eingebrochen wurde (P(Alarm | Einbruch)), 0.95, und dass er losgeht, wenn nicht eingebrochen wurde (P(Alarm | kein Einbruch)), 0.002. Die Wahrscheinlichkeit, dass der Alarm an irgendeinem Tag losgeht (P(Alarm)), ist nun

P(Alarm | E.) * P(E.) + P(Alarm | k.E.) * P(k.E.) = 0.95 * 0.001 + 0.002 * 0.999 = 0.0029

Diese Werte lassen sich nun in die Formel des Bayestheorems einsetzen:

$$P(\text{Einbruch} \mid \text{Alarm}) = \frac{P(Alarm \mid Einbruch) * P(Einbruch)}{P(Alarm)} = \frac{0.95 * 0.001}{0.0029} = 0.33$$

Mit einer 33%igen Wahrscheinlichkeit wurde also bei uns eingebrochen, mit einer Wahrscheinlichkeit von 67% handelt es sich um einen Fehlalarm. Die Chance auf einen Fehlalarm scheint sehr hoch, was aber daran liegt, dass die Wahrscheinlichkeit für einen Einbruch einfach sehr gering ist.

Die Vorteile eines Bayesschen Netzes sind die einfach Handhabung, wenn man erst einmal einen Graphen bestimmt hat, und die schnelle Anpassung der Wahrscheinlichkeiten an das Auftreten neuer Informationen; die Informationen müssen nur einmal das Bayessche Netz durchlaufen und die Wahrscheinlichkeiten müssen neu berechnet werden. Der große Nachteil dieser Struktur ist allerdings, dass

in komplexen Systemen, beziehungsweise Spielen, das vollständige Ermitteln der Abhängigkeiten sehr schwer bis unmöglich ist. Oft kommt man also gar nicht erst in die Situation, ein Bayessches Netz anwenden zu können. Des Weiteren ist das Updaten des Bayesschen Netzes mit sehr komplizierten und aufwendigen Algorithmen verbunden. Dennoch ist das Bayessche Netz ein mächtiges Werkzeug, um Wahrscheinlichkeiten zu berechnen und direkt Entscheidungen zu treffen.

3.b. Evidenztheorie von Dempster und Shafer

Die Evidenztheorie, entwickelt vom amerikanischen Physiker Arthur Dempster im 20. Jahrhundert und weiterentwickelt von Glenn Shafer, wird ebenfalls bei unsicherem Wissen angewandt und ist eine Alternative zu Bayesschen Netzen, wenn deren Anforderungen nicht eingehalten werden können. Dabei ist die Evidenztheorie besonders geeignet, um die Informationen unterschiedlicher Quellen zu einer Gesamtaussage zusammenzufassen. Die Wahrscheinlichkeit eines Ereignisses wird hierbei durch zwei Werte ausgedrückt: Die Glaubwürdigkeit (Grad des Vertrauens in eine Quelle) und die Plausibilität (Wahrscheinlichkeitsbereich mit einer unteren und einer oberen Grenze). Ein kleines Beispiel zur Veranschaulichung:

Angenommen, der Wetterbericht kündigt für morgen schönes Wetter an und wir wissen, dass die Vorhersage in acht von zehn Fällen zutrifft. Die Glaubwürdigkeit des Wetterberichts ist also 80%. Also ist auch die Glaubwürdigkeit für schönes Wetter 80%. Im Gegensatz zur klassischen Wahrscheinlichkeitsrechnung werden die übrigen 20% nun nicht dem Fall „schlechtes Wetter“ zugeordnet, sondern dem Fall „irgendein Wetter“, da keine Aussage darauf hindeutet, dass morgen schlechtes Wetter sein könnte.

bel(schönes Wetter) = 0.8
bel(irgendein Wetter) = 0.2

bel = Glaubwürdigkeit (belief mass)

Die Plausibilität für schönes Wetter setzt sich nun aus der Glaubwürdigkeit für schönes Wetter (untere Grenze) und 1 - der Glaubwürdigkeit für das Gegenteil (obere Grenze) zusammen. Da die Glaubwürdigkeit für schlechtes Wetter 0 ist, gilt:

Plausibilität(schönes Wetter) = 0.8 – 1.0

Wie bereits angesprochen, lassen sich Aussagen verschiedener Quellen unter Berücksichtigung ihrer Glaubwürdigkeiten zu einer Gesamtaussage zusammenfassen. Dies geschieht mit Dempsters Regel:

Kombination (N) = $\frac{\sum_{X \cap Y = N} S1(X) * S2(Y)}{1 - \sum_{X \cap Y = \emptyset} S1(X) * S2(Y)}$

Hierbei ist N die Aussage, die wir evaluieren wollen (zum Beispiel, dass morgen schönes Wetter ist), X und Y sind Hinweise zweier verschiedener Quellen und Si (A) ist die Glaubwürdigkeit von Quelle i für ein Ereignis i.

Man berechnet also für alle Hinweispaare, bei denen sich die Aussagen überschneiden, so dass N ein Teil der Aussage ist, das Produkt der Glaubwürdigkeiten und summiert diese auf. Im Anschluss dividiert man durch 1 – die Summe aller Hinweispaare, die unmöglich sind, da sie jeweils das Gegenteil behaupten. Auch diese Formel muss man an einem Beispiel anwenden, um sie zu verstehen. Angenommen wir haben zwei Wettervorhersagen für morgen, die unterschiedliche Prognosen machen, und wollen nun die Wahrscheinlichkeit für gutes Wetter bestimmen:

Quelle 1: bel(gutes Wetter) = 0.5, bel(schlechtes Wetter) = 0.3, bel(gutes Wetter oder bewölkt) = 0.2
Quelle 2: bel(gutes Wetter) = 0.6, bel(irgendein Wetter) = 0.4

Jetzt müssen wir für jedes Hinweispaar die Schnittmenge bestimmen:

	gutes Wetter 0.5	schlechtes Wetter 0.3	gutes Wetter oder bewölkt 0.2
gutes Wetter 0.6	gutes Wetter: 0.5 * 0.6 = 0.3	unmöglich: 0.3 * 0.6 = 0.18	gutes Wetter: 0.2 * 0.6 = 0.12
irgendein Wetter 0.4	gutes Wetter: 0.5 * 0.4 = 0.2	schlechtes Wetter: 0.3 * 0.4 = 0.12	gutes Wetter oder bewölkt: 0.2 * 0.4 = 0.08

Die Glaubwürdigkeit für gutes Wetter ist also nach Dempsters Regel:

bel(gutes Wetter) = $\frac{0.3 + 0.12 + 0.2}{1 - 0.18}$ = 0.76

Plausibilität(gutes Wetter) = 0.76 – 0.85

Die obere Grenze der Plausibilität berechnet sich durch:
1 – bel(kein gutes Wetter) = 1 – bel(schlechtes Wetter) = 1 – $\frac{0.12}{1-0.18}$ = 1 – 0.15 = 0.85

Nun wollen wir die Nutzentheorie an unserem Beispiel einmal anwenden:

Der rote Spieler muss sich entscheiden, ob und wie er einen Angriff durchführen soll. Dabei könnte er vier mögliche Aktionen ausgemacht haben:

1. den grünen Spieler unten angreifen
2. den grünen Spieler oben angreifen
3. den blauen Spieler angreifen
4. abwarten und die eigene Armee erst noch weiter aufbauen

Auf die 1. Aktion könnte folgen, dass er durch die Mauer bricht und der grüne Spieler von zwei Seiten angegriffen wird. Diese Situation könnte mit einer Wahrscheinlichkeit von 90% eintreten und einen Nutzen von 1 haben. Andererseits könnte es auch passieren, dass er es nicht durch die Mauer schafft, von hinten angegriffen wird und

in der Falle sitzt; die Wahrscheinlichkeit hierfür könnte 10% sein und der Nutzen wäre 0. Der erwartete Nutzen dieser Aktion ist also

EU (1) = 0.9 * 1 + 0.1 * 0 = 0.9

Analog könnte man für die anderen Aktionen verfahren:

EU (2) = 0,2 * 0,75 + 0,2 * 0,75 + 0,1 * 1 + 0,5 * 0,5 = 0,65
EU (3) = 0,5 * 0,2 + 0,5 * 0,4 = 0,3
EU (4) = 0,3 * 0,5 + 0,2 * 1 + 0,5 * 0 = 0,35

Aktion 3 zum Beispiel könnte mit einer 50:50 Chance entweder knapp zu Gunsten des roten Spielers (Nutzen 0.4) oder knapp zu Gunsten des blauen Spielers (Nutzen 0.2) ausfallen.

MEU = max (0,9 ; 0,65 ; 0,3 ; 0,35) = 0,9

Die KI, die in diesem Spiel mit der Nutzentheorie entscheidet, würde also den grünen Spieler unten angreifen.

Die Wahrscheinlichkeiten in diesem Beispiel sind geschätzt, da sich hier weder die Evidenztheorie noch ein Bayessches Netz sinnvoll anwenden lassen.

Mit Blick auf die bereits erwähnten Anforderungen an die Entscheidungsfindung kann man sagen, dass die Entscheidungen sowohl anpassungsfähig (der Nutzen wird einfach neu berechnet, sobald sich an dem verfügbaren Wissen über die Umwelt etwas ändert), als auch menschlich sind (auch Menschen denken unbewusst mit Nutzen, die Nutzentheorie wurde ja auch von Menschen entwickelt). Bei maximiertem Nutzen sind die Entscheidungen auch so richtig, wie sie nur sein können. Verschiedene Persönlichkeiten und Schwierigkeitsgrade können sich implementieren lassen, indem man in den Nutzenfunktionen einige Faktoren speichert, die das Bewerten von Situationen beeinflussen. Zum Beispiel ließe sich so eine KI erschaffen, die Situationen, die offensiv ausgerichtet sind, besonders positiv bewertet, oder die nur Aktionen ausführt, die einen mittleren Nutzen haben, um den Spieler nicht so sehr zu fordern.

4. Goal-Oriented Action Planning (GOAP)

GOAP ist eine einfach zu implementieren Methode, um eine KI Planung durchführen zu lassen. Ein Agent, der Planung durchführt, kann selbstständig seine Aktionen manipulieren, langfristig auf ein Ziel hinarbeiten und sich schnell an neue Situationen anpassen. Bevor man sich mit dem Planen befasst, muss man einige Begriffe kennen:

Ein Plan

- ist eine Abfolge von Aktionen, die ein Ziel erfüllt (Aktionen haben eine Reihenfolge),
- bringt einen Charakter von einem Anfangs- zu einem Endzustand,
- kann neben einem Hauptziel auch Teilziele haben,
- hat Effekte auf die Umwelt und
- hat eine Priorität.

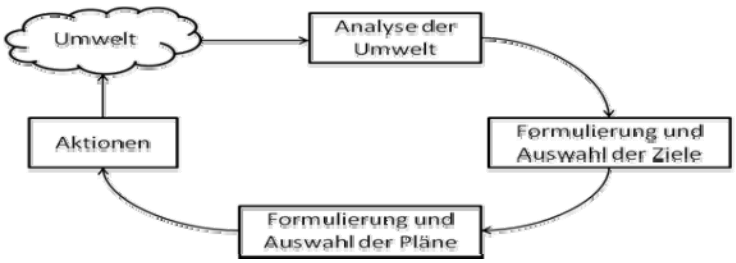
Ein Ziel

- ist eine Bedingung, die ein Charakter erfüllen will (Waffe bekommen, Tür öffnen, Gegner eliminieren),
- kann aus Teilzielen bestehen,
- weiß, wann es erfüllt ist, bzw. zu wie viel Prozent es erfüllt ist und
- hat eine Priorität.

Eine Aktion

- ist eine nicht weiter teilbare Handlung (zu Punkt [x,y] bewegen, Einheit X erschaffen, Waffe nachladen),
- hat eine Dauer,
- hat Vorbedingungen,
- hat Effekte auf die Umwelt und
- benötigt Ressourcen.
- Die Aktionen, die Spieler und KI ausführen können, sollten identisch sein.
- Ein Spiel hat eine festgelegte Menge von Aktionen (Aktionsraum)

Ein Plan hat folgende Phasen:



Was geschieht nun genau in den einzelnen Phasen?

Die Analyse der Umwelt ist ein sehr wichtiger Bestandteil des Planungsprozesses, da der Plan, der am Ende steht, immer nur so gut sein kann, wie die Basis, auf der er gemacht wurde. Im Prinzip wird hier die Gesamtheit der verfügbaren Daten in Werte zusammengefasst, die zur Entscheidungsfindung benutzt werden können. Dabei wird für jede relevante Kategorie (Kategorien können sein: eigene Kampfkraft, Kampfkraft des Gegners, eigene Verwundbarkeit, etc) die verfügbare Information als Zahl auf einer Skala ausgedrückt; für jede Kategorie gibt es also eine Funktion. Man sollte darauf achten, dass alle Funktionen dieselbe Skala benutzen und dass die Skalen auch ausgenutzt werden.
Wenn die Umwelt jetzt analysiert wurde, weiß man, was man gerade für Möglichkeiten und Bedürfnisse hat. Daraus gilt es jetzt, Ziele zu formulieren. Eine hohe

eigene Kampfkraft und eine hohe Verwundbarkeit beim Gegner könnten zu dem Ziel führen, diesen Gegner anzugreifen. Außerdem muss man bestehende Ziele evaluieren, also prüfen, ob sie erfüllt ist. Sinnvoll ist es auch, einige Ziele die ganze Zeit über zu verfolgen, zum Beispiel das Ziel, das Spiel gewinnen zu wollen. So macht die KI auch Fortschritte, wenn sie gerade nicht unter Druck steht. Für jedes Ziel wird eine Priorität gespeichert, also ein Wert, der alle einfließenden Größen zusammenfasst. Nach diesen Werten können wir nun die Ziele, die wir formuliert haben, sortieren und diejenigen auswählen, die die höchste Priorität haben. An dieser Stelle kann man zusätzlich mit Faktoren sehr leicht verschiedene Persönlichkeiten implementieren. Ein Agent könnte etwa alle Ziele, die defensive Zwecke verfolgen, höher bewerten und würde so sehr defensiv spielen. So können sich zwei Agenten, die beide Planung durchführen, völlig unterschiedlich verhalten, obwohl sie sich nur in einem einzigen Faktor unterscheiden.
Jetzt wurden also die Ziele, die man verfolgen will, ausgewählt und es geht darum, Pläne zu finden, mit denen man diese Ziele erreicht. Die einfachste, aber auch am wenigsten flexible Methode ist es, zu Zielen vorgefertigte Pläne abzuspeichern, die die KI zur Laufzeit lädt und an die momentane Situation anpasst. Die flexiblere Möglichkeit besteht darin, die Pläne direkt zur Laufzeit zu formulieren. Da ein Plan nichts anderes ist als eine Abfolge von Aktionen, muss man den Aktionsraum nach einer Aktionsfolge durchsuchen, die den Charakter vom Anfangs- zum Endzustand bringt. Diese Suche ist im Prinzip ein Navigationsproblem und ist in der Tat auch mit dem A*-Algorithmus durchführbar. Dabei sind eine Vorwärts- und eine Rückwärtssuche möglich. Bei der Rückwärtssuche startet der Suchalgorithmus beim Ziel, das erfüllt werden soll, und sucht nach Aktionen, die dieses Ziel erfüllen. Dann läuft er über die Vorbedingungen dieser Aktionen, bis er schließlich bei einer Aktion ankommt, die ohne weitere Vorbedingung ausgeführt werden kann. In der Regel wird man für ein Ziel mehrere Pläne finden, die es erfüllen, also muss man sich noch für einen Plan entscheiden. Am einfachsten ist es, den Plan zu nehmen, der zuerst gefunden wurde. So wird jedoch möglicherweise ein Plan ausgewählt, der weniger effizient ist, als ein anderer. Also kann man auch für Pläne einen Wert abspeichern, der die Relevanz angibt. Einflussfaktoren für diesen Wert können die benötigten Ressourcen eines Plans und die bereits vorhandenen Ressourcen sein. Jetzt kann man den Plan auswählen, der die höchste Relevanz hat.
Endlich hat man also einen Plan. Dieser muss jetzt nur noch umgesetzt werden. Im einfachsten Fall kann man nacheinander alle Aktionen ausführen; dies setzt jedoch eine statische Umgebung und eine lineare Reihenfolge voraus, was in den meisten Spielen nicht gegeben ist. Bei dynamischen Umgebungen gibt es Einflüsse auf unseren Plan, zum Beispiel durch andere Spieler, und man muss während der Planausführung den Plan auf Fehler testen. Am Besten eignet sich dazu das Plan Monitoring.
Das Plan Monitoring testet den Plan auf globaler Ebene und führt dabei folgende Tests durch:

- Gibt es im Plan verletzte Vorbedingungen, ob die Aktionen, die diese Bedingungen erfüllen sollten, bereits ausgeführt wurden?
- Gibt es Aktionen, die übersprungen werden können, weil die von ihnen erzielten Änderungen schon eingetreten sind?
- Ist die nächste Aktion überhaupt ausführbar?

Das würde als Pseudo-Code ungefähr so aussehen:

```
repeat
   if there is a precondition for an unexecuted action whose value is false and whose causal
action has already
   been executed
      abort the plan
   from the final goal and working backward
      if the goal‘s or action‘s changes are already true
         for each precondition of the goal or action
            mark the action that causes the precondition as skipable
   get the next action in the plan with a change not marked as skipable
   do
      perform the action
   until the action is finished or time runs out
   if the action‘s changes are false
      abort the plan
until the plan is empty
```

Plan Monitoring ist ein sehr rechenzeitaufwendiger Prozess und ob man es einsetzt ist sehr abhängig vom konkreten Spiel. Im Allgemeinen ist es jedoch sinnvoll, da es viele Gründe geben kann, dass Pläne scheitern: eine dynamische Umgebung, Aktionen anderer Spieler und Sabotage.

Wie reagiert man nun, wenn man merkt, dass ein Plan gescheitert ist?
Man kann entweder neu planen, was die einfachste Möglichkeit ist, aber oft auch bedeutet, dass man einen Großteil der Arbeit doppelt macht. Wenn man das nicht möchte, kann man versuchen, das Problem zu beheben, also die verletzte Vorbedingung zu erfüllen. Das ist leider nicht immer möglich und so hat man als weitere Möglichkeiten das Ausprobieren eines alternativen Plans und das Aufgeben des Ziels, in der Hoffnung, dass es später wieder erreichbar wird.

Der Hauptvorteil dieser Architektur liegt eindeutig in der großen Flexibilität. Eine KI, die Planung durchführt, findet sogar Pläne für Situationen, an die der Programmierer bei der Entwicklung des Spiels vielleicht gar nicht gedacht hat. Der Programmierer muss also nicht zwingend an alles denken, da die KI sich so leicht anpasst. Ein weiterer Vorteil ist das sehr einfache Implementieren von verschiedenen Persönlichkeiten.

Auch diese Architektur soll einmal an unserem Beispiel angewandt werden:

Bei der Analyse der Umwelt könnte die KI zu folgendem Ergebnis kommen: Die eigene Kampfkraft ist sehr hoch, die Kampfkraft des grünen Spielers ist gering, die Kampfkraft des blauen Spielers ist ebenfalls sehr hoch, die eigene Verteidigung ist relativ schwach und unser Verbündeter ist gerade in Kämpfe verwickelt.
Da Königsmord gespielt wird, sollte die KI als ständige aktive Ziele das Beschützen des eigenen und das Töten der feindlichen Könige haben. Zusätzlich könnte sie aus den Möglichkeiten und Bedürfnissen, die sie ermittelt hat, die Ziele formulieren, den Verbündeten zu unterstützen und die eigene Verteidigung aufzubauen. Eine offensive KI würde jetzt die Gelegenheit nutzen und den grünen Spieler angreifen. Das zu erfüllende Ziel ist jetzt also, den grünen König zu töten. Dazu könnte der Suchalgorithmus per Rückwärtssuche folgenden Plan formulieren:

Bei aktivem Plan Monitoring würde die KI auch Fehler erkennen. Wenn beispielsweise die Mauer bereits zerstört wäre, würde die Aktion „Mauer angreifen“ übersprungen werden; würde man feststellen, dass der König nicht mehr in seiner Burg weilt, wäre die Vorbedingung für „König angreifen“ verletzt und der Plan würde abgebrochen werden.

Erfüllung der Anforderungen an die Entscheidungen:

GOAP ist besonders anpassungsfähig, da es diverse Möglichkeiten gibt, auf gescheiterte Pläne zu reagieren. Die getroffenen Entscheidungen sind richtig, solange bei der Analyse der Umwelt alles richtig gemacht wurde, und menschlich sind sie auch, da auch ein Mensch für seine Entscheidungen kurz- und langfristige Pläne schmiedet. Verschiedene Persönlichkeiten lassen sich denkbar einfach realisieren, und auch verschiedene Schwierigkeitsgrade lassen sich implementieren, indem man bei geringem Schwierigkeitsgrad zum Beispiel nur kurzfristige Pläne ausführt.

5 .Influence Maps, Resource Allocation Trees, Dependency Graphs

Bei diesen drei Begriffen handelt es sich um Methoden, die vorhandene Daten so darstellen sollen, dass die KI daraus optimal Rückschlüsse ziehen kann. Da diese Techniken miteinander kombiniert werden können, bietet es sich an, sie in einem Kapitel zusammenzufassen. Techniken zur Aufbereitung der Daten sind sehr wichtig, da, wie bereits vorher erwähnt, die Entscheidungen einer KI nur so gut sein können wie die Daten; die rohen Daten an sich sind nutzlos. Alle diese Techniken werden hauptsächlich in Strategiespielen angewandt, jedoch sind verschiedene Einsatzgebiete auch in anderen Genres denkbar.

Was ist eine Influence Map?

Eine Influence Map basiert auf der geographischen Karte des Spiels, welche in mehrere Zellen unterteilt wird. Jede Zelle dient dabei als kleine Datenbank, das heißt, es werden für jede Zelle verschiedene Werte gespeichert. Gespeicherte Werte können sind: Anzahl Einheiten, HitPoints und Kampfkraft dieser Einheiten, Anzahl Ressourcen, Passierbarkeit, Anzahl und Art der Gebäude, etc.. Im Normalfall speichert jeder Computerspieler eine Influence Map für sich selbst und je eine für jeden Gegner. Dies kann bei mehreren Spielern aber schnell die Performance

drücken, die Alternative wäre, eine Influence Map zu haben, auf die jeder Computerspieler zugreift, dies würde allerdings bedeuten, dass die Computerspieler cheaten, also schummeln, da sie auf Daten zugreifen können, auf die sie nicht zugreifen sollten. Die Werte der Einheiten einer Zelle würden nun an benachbarte Zellen weitergegeben werden, um zu berücksichtigen, dass die Einheiten sich bewegen und benachbarte Zellen beeinflussen können. Durch eine Influence Map wird erkennbar, wo Einheiten in Zukunft sein könnten, wo die Front verläuft, wo Engpässe und Schwachpunkte in der Verteidigung sind und vieles mehr.

Ein kleines Beispiel:

			X
	X	X	
	X X		X
X		X	

0	0,5	-1	-1
0,5	1,5	-0,5	-1,5
1,5	2,5	-0,5	-1
1	1	-1	-1

Ein schwarzes X bezeichnet eine eigene, ein rotes X eine feindliche Einheit. Jede eigene Einheit wird mit 1 bewertet, jede feindliche mit -1. Diese Werte werden je eine Zelle weitergegeben, und zwar um die Hälfte reduziert, wodurch wir die rechte Karte erhalten.

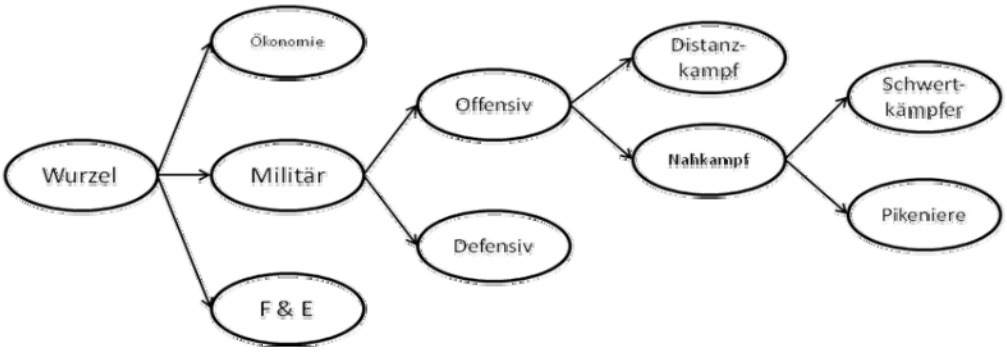
Um mit einer Influence Map jetzt Entscheidungen zu treffen, berechnet man für jedes Entscheidungsproblem einen Attraktivitätswert für jede Zelle. Dies geschieht am einfachsten, indem man alle relevanten Faktoren für das konkrete Problem durch eine gewichtete Summe addiert. Relevante Faktoren für das Entscheidungsproblem „Wo greife ich an?“ könnten eigenes Offensivpotential und feindliches Defensivpotential sein. Die Zelle mit dem höchsten Attraktivitätswert wird für die Entscheidung ausgewählt.

Das Weitergeben der Einflüsse ist im Prinzip sehr einfach: Man legt fest, über wie viele Zellen der Einfluss der weitergegeben werden soll und bestimmt eine Dekrementierungsregel, zum Beispiel halbieren. Was man hier jedoch beachten muss, ist, dass verschiedene Einheiten gegeneinander abgegrenzt werden müssen. Eine Wassereinheit kann nicht über Land und eine Landeinheit nicht über Wasser oder Gebirge gehen; eine Einheit mit großer Reichweite kann eine benachbarte Zelle vielleicht direkt angreifen; eine schnelle Einheit gibt ihren Einfluss stärker weiter als eine langsame Einheit. Mit diesen Problemen sollte man sich als Programmierer intensiv befassen, um ein wenig Herumprobieren kommt man dabei wahrscheinlich nicht herum.

Was ist ein Resource Allocation Tree?

Ein Resource Allocation Tree ist eine Baumstruktur, die den Zweck aller Einheiten und Gebäude repräsentiert, die man gerade besitzt. Dazu werden alle Einheiten und Gebäude in funktionale Kategorien eingeteilt, so dass man sehen kann, welchem Zweck sie gerade zugeordnet sind. Die Knoten des Baumes sind Kategorien, die Blätter Einheiten und Gebäude, die momentan der übergeordneten Funktion dienen. Auch hier sollte jeder Computerspieler einen eigenen Resource Allocation Tree für sich und jeden Gegner speichern, um cheating zu vermeiden.

Ein kleines Beispiel:



Dies ist natürlich nur ein Ausschnitt eines Resource Allocation Trees, da auch die anderen Kategorien sich jeweils weiter aufteilen würden. Es könnte zum Beispiel auch Pikeniere geben, die momentan Verteidigungszwecken zugeordnet sind, diese würden dann als Kinder des Defensiv-Knotens auftauchen.

Wenn man mit Hilfe des Resource Allocation Trees Entscheidungen treffen möchte, kann man die gewünschte und die momentane Allokation berechnen und diese Werte vergleichen. Zur Berechnung der gewünschten Allokation initialisiert man den Wurzel-Knoten mit dem Wert 1 und verteilt diesen Wert prozentual nach vorher definierten Regeln an die Kinder. Die Verteilung ist dabei abhängig vom Design des Spiels, jeder Knoten sollte aber einen eigenen Code hierfür besitzen. Der Grad der Verteilung sollte sich hierbei im Laufe des Spiels auch ändern können. Um die momentane Allokation zu berechnen, startet man bei den Blättern und berechnet für jedes Blatt einen Wert, der die aktuelle Allokation wiedergibt. Dies könnte etwa (Anzahl Einheiten * Kampfkraft der Einheiten * HitPoints der Einheiten) sein. Diese Werte werden jeweils eine Stufe nach oben weitergegeben und jeder Knoten summiert die Werte, die er von seinen Kindern bekommt und gibt die Summe wiederum nach oben weiter. Am Ende steht beim Wurzelknoten ein ziemlich großer Wert, durch den man alle anderen Werte dividieren kann, um wieder Werte zwischen 0 und 1 zu erhalten. Durch den Vergleich dieser beiden Werte (gewünschte und momentane Allokation) lassen sich jetzt unterallokierte Knoten finden. Die KI kann diese Unterallokation ausgleichen, indem sie neue Einheiten erschafft oder exisitierende Einheiten anderen Funktionen zuordnet.
Auch hier lassen sich durch das Anpassen der Koeffizienten bei der Verteilung der Werte an die Kinder bei der Berechnung der gewünschten Allokation leicht verschiedene Persönlichkeiten einer KI realisieren. Für Kämpfe kann es durchaus

sinnvoll sein, eine Tabelle zu haben, in der zu jeder Einheit angegeben ist, gegen welche Einheiten sie besonders effektiv ist. So kann die KI, wenn sie weiß, auf welche Einheiten der Gegner setzt, entsprechend reagieren. Des Weiteren kann man speichern, welche Typen von Einheiten im bisherigen Verlauf des Spiels besonders effizient waren, um in Zukunft vermehrt diese Einheiten zu produzieren.

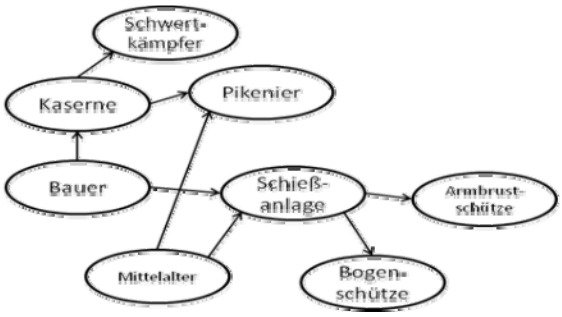
Was ist ein Dependency Graph?

Ein Dependency Graph modelliert Abhängigkeiten zwischen Einheiten und Gebäuden; die traditionelle Anwendung ist der Technologie-Baum in einem Strategiespiel, der anzeigt, welche Einheiten und Gebäude produziert werden und was dazu nötig ist. Ein Dependency Graph ist dabei im Prinzip ein Bayessches Netz, und die Methoden des Bayesschen Netzes können auch auf diesen angewandt werden. Es gibt folgende Typen von Abhängigkeiten zwischen Einheiten und Gebäuden (Abhängigkeiten können zwischen Einheiten, zwischen Gebäuden oder zwischen Einheiten und Gebäuden exisitieren):

- erschaffende Abhängigkeiten: Was muss ich besitzen, bevor ich eine bestimmte Einheit erschaffen oder ein bestimmtes Gebäude errichten kann?
- unterstützende Abhängigkeiten: Was brauche ich, damit ich eine bestimmte Einheit oder ein bestimmtes Gebäude sinnvoll einsetzen kann?

Eine unterstützende Abhängigkeit könnte so aussehen, dass ein Magier zum Zaubern Mana und daher einen Manaschrein, welcher Mana produziert, braucht, um zu funktionieren.
Die Knoten im Dependency Graph können durchaus mit den Blättern im Resource Allocation Tree übereinstimmen, im Dependency Graph werden auch Einheiten aufgeführt, die gerade produziert werden.

Ein kleines Beispiel:



In diesem Beispiel handelt es sich ausschließlich um erschaffende Abhängigkeiten. Man braucht zum Beispiel einen Bauern, der eine Kaserne errichten kann, die man wiederum braucht, um Schwertkämpfer zu erschaffen. Um Pikeniere erschaffen zu können, muss man zusätzlich noch das Mittelalter erreicht haben.

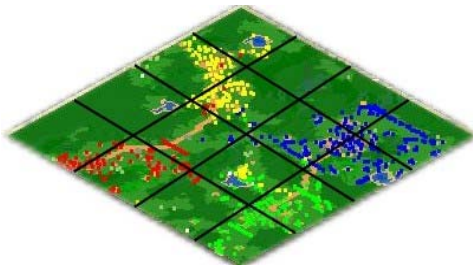
Mit einem Dependency Graph kann man nun auf ein Ziel hinarbeiten; wenn man weiß, dass man im späteren Spielverlauf eine bestimmte Einheit erschaffen können möchte, dann kann man mit dem Dependency Graph ermitteln, was man dazu benötigt. Außerdem kann man verwundbare Knoten finden; ein Knoten ist zum Beispiel verwundbar, wenn er starke Kinder produzieren kann, da diese Kinder nicht mehr produziert werden können, wenn dieser Knoten nicht mehr existiert. Diese Information ist gut, um eigene Schwachpunkte zu finden und diese besser zu schützen, und um Schwachpunkte beim Gegner zu finden, die man angreifen kann. Weiter kann man mit einem Dependency Graph Rückschlüsse ziehen: Wenn wir beim Gegner einen Bogenschützen sehen, wissen wir, dass er eine Schiessanlage hat, und somit steigt die Wahrscheinlichkeit, dass er bereits Armbrustschützen hat. Auf

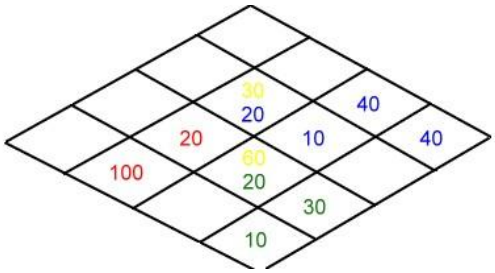
ähnliche Weise kann man auch das Vorhandensein von Einheiten weniger wahrscheinlich machen. Wenn man weiß, dass in bestimmter Zeit nur gewisse Fortschritte erzielbar sind und ein Spieler von zwei Einheitentypen nur einen besitzen kann, sinkt die Wahrscheinlichkeit, dass er auch den zweiten Einheitentyp besitzt, wenn man den ersten bei ihm sieht.

Alle drei Strukturen zusammen ergeben ein sehr mächtiges Werkzeug, wenn sie dieselben Daten speichern und untereinander austauschen. Die Kombination ist mächtiger als die bloße Summe der drei Architekturen:

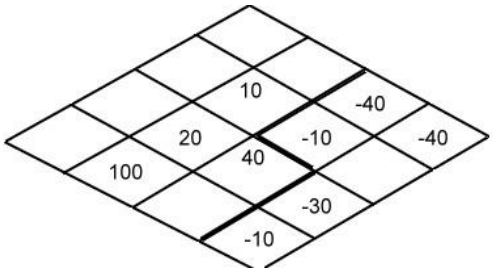
Die Influence Map sagt mir, wo der Gegner ist.
Der Resource Allocation Tree sagt mir, was ich brauche, um ihn zu schlagen.
Der Dependency Graph sagt mir, wie ich das bekomme.

Zu unserem Beispiel könnte man folgende Influence Map erstellen:

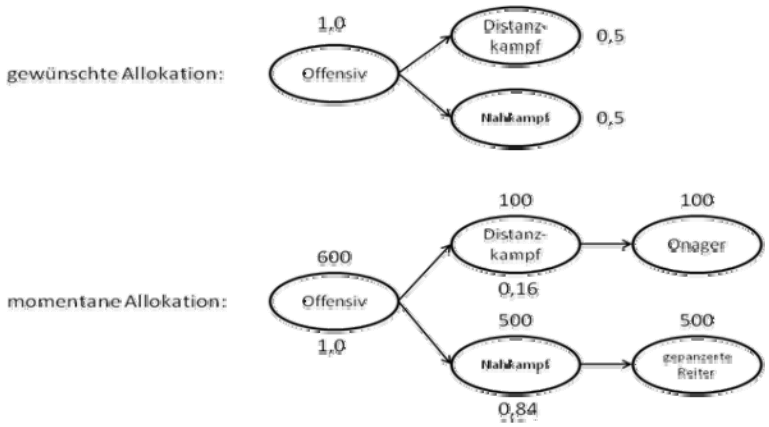




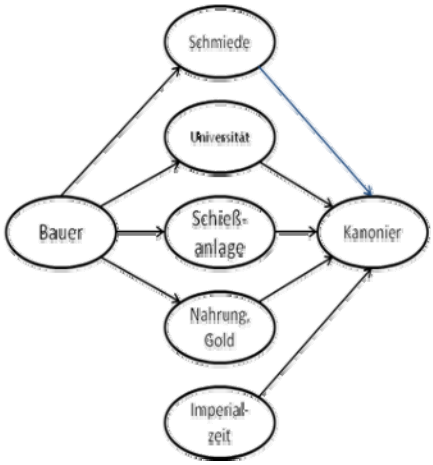
Die Zahlen sind hier ein geschätzter Wert, der die Kampfkraft der Einheiten der einzelnen Spieler ausdrückt. Für den roten und gelben Spieler könnte man diese in positive Werte und für den grünen und blauen Spieler in negative Werte verwandeln und diese gegen-einander aufsummieren:



Die Zelle unten links wäre in diesem Fall diejenige, die man am ehesten angreifen sollte, da der eigene Einfluss dort am größten und der feindliche Einfluss dort am geringsten ist.
Um zu ermitteln, mit welchen Einheiten man angreifen sollte, werfen wir einen Blick auf den Ausschnitt eines Resource Allocation Trees:



Da der Distanzkampf-Knoten in diesem Fall unterallokiert ist, müsste man versuchen, dies auszugleichen. Dazu muss man nicht zwingend neue Onager erschaffen, man könnte zum Beispiel abgespeichert haben, dass Kanoniere gegen den grünen Spieler schon besonders erfolgreich waren, und demnach neue Kanoniere erschaffen. Was man dazu braucht verrät einem der Dependency Graph:



Der blaue Pfeil beschreibt hier eine unterstützende Abhängigkeit, da eine Schmiede nicht erforderlich ist, um einen Kanonier zu erschaffen, aber wesentliche Updates liefert, die seine Kampfkraft erhöhen. Bei allen anderen Abhängigkeiten handelt es sich um erschaffende.

Jetzt wissen wir, wo wir unseren Gegner angreifen können, womit wir ihn am besten angreifen und wie wir dies bekommen können.

Mit Blick auf unsere bereits formulierten Anforderungen lässt sich sagen, dass alle drei Architekturen sehr anpassungsfähig sind, da sie sich ständig aktualisieren und immer aktuelle Werte speichern. Sie liefern dazu optimale Voraussetzungen, um richtige Entscheidungen zu treffen, die zudem menschlich sind, da auch ein erfahrener Spieler instinktiv auf Basis der Karte des Spiels ähnliche Entscheidungen trifft und im Hinterkopf hat, welche Einheiten besonders effektiv sind und waren. Verschiedene Persönlichkeiten lassen sich ebenfalls, wie wir gesehen haben, realisieren, bei Schwierigkeitsgraden geht dies auf ähnliche Weise. Aufpassen muss man bei dem Aspekt des cheatens, da ein Computerspieler durch Zugriff auf die Strukturen der anderen Spieler leicht an Daten kommt, die er nicht sehen dürfte. Entscheidungen, die dadurch zustande kämen, wären unrealistisch. Ob dies erwünscht ist oder nicht, hängt vom Spiel ab.

6. Literaturverzeichnis

Zum Thema Nutzentheorie und Künstliche Intelligenz:

- Kapitel 14 - 17 aus *Stuart J. Russell - Artificial Intelligence: A Modern Approach, Prentice Hall International, ISBN-13: 978-0131038059*

Zum Thema Bayessches Netz:

- *Paul Tozour - Introduction to Bayesian Networks and Reasoning Under Uncertainty*, erschienen in *AI Game Programming Wisdom, Thomson Learning, ISBN-13: 978-1584500773*
- Wikipedia: http://de.wikipedia.org/wiki/Bayessches_Netz

Zum Thema Evidenztheorie:

- *François Dominic Laramée - A Rule-Based Architecture Using the Dempster-Shafer-Theory*, erschienen in *AI Game Programming Wisdom, Thomson Learning, ISBN-13: 978-1584500773*
- Wikipedia: http://de.wikipedia.org/wiki/Evidenztheorie

Zum Thema Goal-Oriented Action Planning:

- *John O'Brien - A Flexible Goal-Based Planning Architecture*, erschienen in *AI Game Programming Wisdom, Thomson Learning, ISBN-13: 978-1584500773*
- *Bryan Stout - Adding Planning Capabilities to Your Game AI*, erschienen im *Game Developer Magazine, Ausgabe Januar 1998*
- *Jeff Orkin - Applying Goal-Oriented Action Planning to Games*, erschienen in *AI Game Programming Wisdom 2, B&T, ISBN-13: 978-1584502890*

Zum Thema Influence Maps, Resource Allocation Trees, Dependency Graphs:

- *Paul Tozour - Influence Mapping*, erschienen in *Game Programming Gems 2, B&T, ISBN-13: 978-1584500544*
- *Paul Tozour - Strategic Assessment Techniques*, erschienen in *Game Programming Gems 2, B&T, ISBN-13: 978-1584500544*
- *Penny Sweetser - Strategic Decision-Making with Neural Networks and Influence Maps*, erschienen in *AI Game Programming Wisdom 2, B&T, ISBN-13: 978-1584502890*