Berner Fachhochschule
Technik und Informatik

Fachbereich Technik und Informatik

Bachelor Thesis - AI Bot für Computerspiele

Ants AI Challenge

Studierende: Lukas Kuster
Stefan Käser

Betreuung: Dr. Jürgen Eckerle

Experte: Dr. Federico Flückiger

Datum: 3. Januar 2013
Version: V01.00

Management Summary

Ants AI Challenge ist ein Programmierwettbewerb, bei welchem ein Bot programmiert wird, der ein Ameisenvolk steuert. Das Ameisenvolk soll auf einer Karte Futter suchen sowie gegnerische Völker angreifen und vernichten. Dabei müssen Problem wie die Pfadsuche, das Verteilen von Aufgaben sowie das Schwarmverhalten gelöst werden. Als Einstieg in die AI-Programmierung haben wir vor der Bachelorarbeit, ein Bot geschrieben der bereits die Basisfunktionen beherscht. Dieser wurde nun während der Bachelorarbeit mit strategischen und taktischen Modulen erweitert. Dazu gehören unter anderem die Verwendung einer Influence Map, Ressourcenverteilung, die Parametrisierbarkeit des Bots. Im Weiteren wurden die Fähigkeiten des Bots verfeinert. Es wurde ein Suchframework erstellt, dass für ähnliche Computerspiele wiederverwendet werden kann. Dieses bietet die Pfadsuchalgorithmen A* und HPA* sowie eine Breitensuche an.

Datum	3. Januar 2013
Name Vorname	Lukas Kuster
Unterschrift	.....
Name Vorname	Stefan Käser
Unterschrift	.....

Inhaltsverzeichnis

Abbildungsverzeichnis

1 Einleitung

Im Rahmen des Moduls "Projekt 2" (7302) haben wir uns mit der Implementierung eines Bots für den Online-Wettbewerb AI-Challenge (Ants) beschäftigt. Die AI-Challenge ist ein Wettbewerb, der im Herbst 2011 zum 3. Mal stattfand und jedes Jahr mit einem anderen Spiel durchgeführt wird. Ziel ist es jeweils, einen Bot zu programmieren, der durch geschickten Einsatz von KI-Technologien das Spiel möglichst erfolgreich bestreiten kann. In dieser Durchführung ging es darum, ein Ameisenvolk durch Sammeln von Ressourcen und Erobern von gegnerischen Hügeln zum Sieg über die gegnerischen Ameisen zu führen.

Wir hatten uns zum Ziel gesetzt, einen Bot zu implementieren, der möglichst alle Bereiche des Spiels beherrscht, also Nahrung sammeln, die Gegend entdecken, Hügel erobern und gegen feindliche Ameisen kämpfen. Im Gegenzug legten wir kein besonderes Gewicht darauf, dass der Bot eines dieser Verhalten besonders gut beherrschen muss. Das primäre Ziel war es, Erfahrungen zu sammeln im Hinblick auf die Bachelor-Arbeit.

Den grössten Aufwand bei der Implementierung steckten wir in die Pfadsuche, da diese eine Voraussetzung für nahezu alle Teil-Aufgaben des Bots ist. Nachdem wir mit dem bekannten A*-Algorithmus zwar kleine Erfolge erzielten, aber auch schnell Performance-Probleme bekamen, entschlossen wir uns, auf Basis eines Clustering des Spielfeldes den HPA*-Algorithmus umzusetzen. Den Algorithmus konnten wir erfolgreich implementieren, aber leider fehlte uns gegen Ende des Projekts dann die Zeit, ihn noch zu optimieren. Die Performance ist aktuell vergleichbar mit der A*-Pfadsuche.

Ein weiterer Punkt, auf den wir viel Wert legten, war die Programmstruktur. Unser Bot ist Objektorientiert aufgebaut; die zentrale Einheit sind die verschiedenen Tasks, die jeweils für eine Aufgabe der Ameisen zuständig sind.

1.1 Projektverlauf

Die Projektarbeit richtete sich nach folgendem Zeitplan:

1.2 Projektorganisation

1.2.1 Beteiligte Personen

Studierende:
Lukas Kuster *kustl1@bfh.ch*
Stefan Käser *kases1@bfh.ch*

Betreuung:
Dr. Jürgen Eckerle *juergen.eckerle@bfh.ch*

Experte:
Dr. Federico Flückiger *federico.flueckiger@bluewin.ch*

1.2.2 Projektmeetings

- Es fand jeweils ein Treffen mit dem Betreuer alle 1-2 Wochen statt.
- Ein Treffen mit dem Experten fand am Anfang der Arbeit statt. Ein zweites Meeting wurde von beiden Seiten nicht für notwendig erachtet.

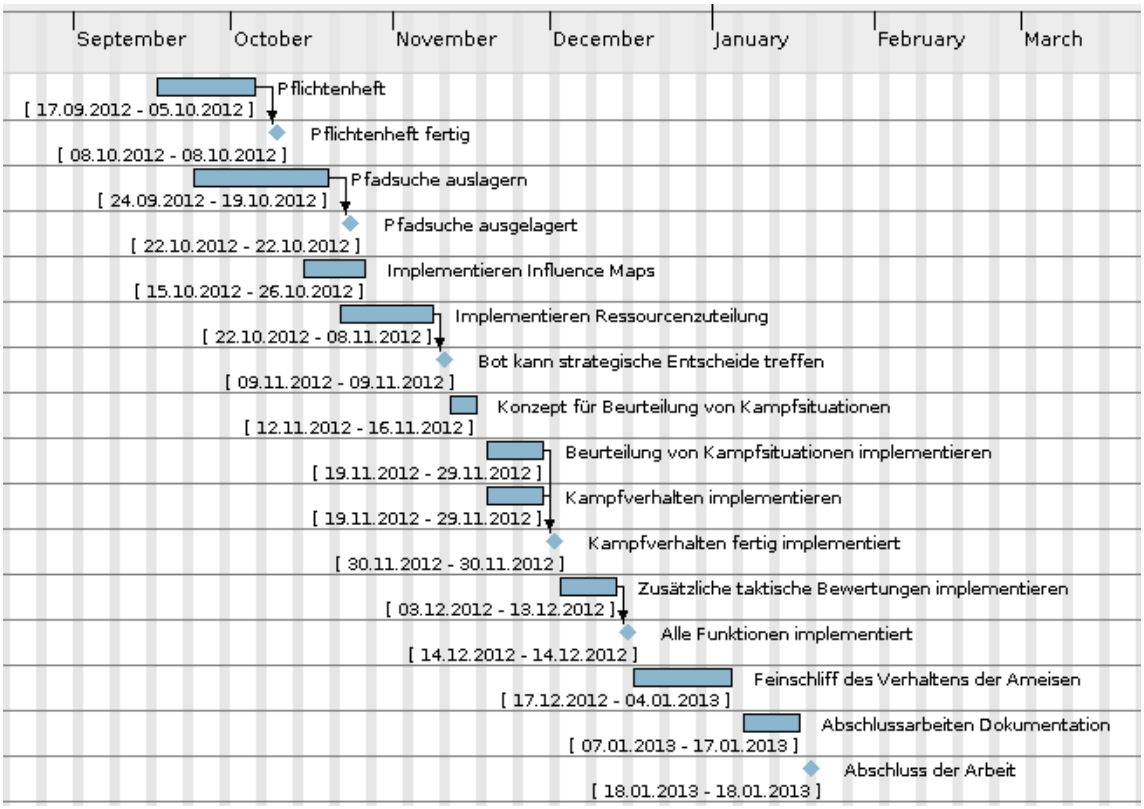

Abbildung 1.1: Projektablauf

1.2.3 Dokumentation

Die Dokumentation soll sich am Aufbau und Inhalt des Berichts aus dem Projekt 2 anlehnen.

- Das Dokument beschränkt sich auf das Wesentliche.
- Verwendete AI-Techniken werden erläutert
- Entscheidungen und deren Grundlagen sind dokumentiert.
- Testberichte dokumentieren die durchgeführten Modultests.
- Klassendiagramme sollen einen oberflächlichen Detailierungsgrad haben, so dass das Wichtigste auf den ersten Blick sichtbar ist.
- Anleitung zum Ausführen eines Spiels

1.2.4 Abgabe

Folgende Lieferobjekte werden am Ende der Arbeit abgegeben.

- Dokumentation
- Sourcecode

1.3 Ziele

Nachfolgend sind die Ziele aufgelistet welche wir uns vor der Arbeit gestellt und im Pflichtenheft niedergeschrieben haben. Die farbigen Pfeile zeigt den Erfüllungsgrad an. Ist ein Ziel nicht vollständig erreicht wird in *kursiver* Schrift ein Grund angegeben.

↗ Vollständig Erfüllt
→ Teilweise Erfüllt
↘ Nicht Erfüllt

Der im Rahmen von Projekt 2 entwickelte Bot soll um Logik für taktische und strategische Entscheidungen und koordinierte Bewegung erweitert werden.

1.3.1 Funktionale Anforderungen

1.3.1.1 Musskriterien

↗ Der Bot unterscheidet zwischen diversen Aufgaben:

- Nahrungsbeschaffung
- Angriff
- Verteidigung
- Erkundung

Der Bot identifiziert zur Erfüllung dieser Aufgaben konkrete Ziele, wie z.B.:

- ↗ Gegnerische Hügel angreifen, was bei Erfolg den Score erhöht und das eigentliche Ziel des Spiels ist.
- ↗ Isolierte gegnerische Ameisen angreifen.
  Grund: Keine Zeit für blabal...
- → Schwachstellen in der gegnerischen Verteidigung ausnutzen.
- ↘ Engpässe im Terrain sichern bzw. versperren.
- Konfliktzonen, d.h. viele Ameisen auf einem engen Raum, erkennen und entsprechend reagieren.

Die Auswahl von Taktik und Strategie basiert auf der Bewertung der Situation auf dem Spielfeld, z.B. anhand folgender Kriterien:

- Dominante/unterlegene Position
- Sicherheit verschiedener Gebiete des Spielfelds (eigener/gegnerischer Einfluss)
- Konfliktpotenzial in verschiedenen Gebieten des Spielfelds

Anhand der Situationsbeurteilung werden die unterschiedlichen Aufgaben entsprechend gewichtet. Stark gewichtete Aufgaben erhalten mehr Ressourcen (Ameisen) zur Durchführung.

Die Situationsbeurteilung fliesst auch in die taktische Logik ein, wie folgende Beispiele illustrieren:

- Bei der Pfadsuche wird die Sicherheit der zu durchquerenden Gebiete berücksichtigt
- In Kampfsituationen kann der Bot die Ameisen in Formationen gliedern, die geeignet sind, eine lokale Überzahl eigener gegenüber gegnerischen Ameisen zu erzeugen
- Beim Aufeinandertreffen mit gegnerischen Ameisen wird entschieden, ob angegriffen, die Stellung gehalten oder geflüchtet wird.

1.3.1.2 Kannkriterien

Das Verhalten des Bots ist konfigurierbar, so dass zum Beispiel ein "agressiver" Bot gegen einen defensiven Bot antreten kann.

1.3.2 Nicht funktionale Anforderungen

1.3.2.1 Musskriterien

Modularer Aufbau für eine gute Testbarkeit der Komponenten.

Wichtige Funktionen wie die Pfadsuche und die Berechnung von Influence Maps sollen in separaten Modulen implementiert werden, damit sie auch von anderen Projekten verwendet werden könnten.

Die Codedokumentation ist vollständig und dient der Verständlichkeit.

1.3.2.2 Kannkriterien

Für die wiederverwendbaren Module wird jeweils ein kleines Tutorial geschrieben, wie die Module verwendbar sind.

1.4 Herausforderungen

1.4.1 Module testen

Ein neuer Algorithmus oder eine neue Idee ist schnell mal in den Bot integriert, doch bringen die geschriebenen Zeilen den gewünschten Erfolg? Was wenn der neue Codeabschnitt äusserst selten durchlaufen wird und dann noch fehlschlägt? Wie wissen wir welche Ameise genau diesen nächsten Schritt macht?

Um diese Probleme zu bewälitgen haben wir ein ausgeklügeltes Logging auf die Beine gestellt, in welchem wir schnell an die gewünschten Informationen gelangen. (siehe 7) Zudem können wir dank der Erweiterung des HTML-Viewer sofort sehen, welches die akutelle Aufgabe jeder einzelen Ameise ist. (siehe 7.2) Weitergeholfen haben uns auch etliche Unit- und Funktionstests, mit welchen wir neu geschriebenen Code testen und auf dessen Richtigkeit prüfen konnten. (siehe 9.1)

1.4.2 TODO

lorem ipsum mehr herausfoderungen ??

1.4.3 Vergleich mit Bots aus dem Wettbewerb

Nach Ablauf des Wettbewerbs im Januar 2012, haben einige der Teilnehmer ihren Bot zugänglich gemacht. Dadurch war es uns möglich unseren Bot gegen Bots antreten zu lassen die tatsächlich am Wettbewerb teilgenommen haben. So konnten wir auch eine wage Einschätzung machen wir stark unser Bot ist. Mehr dazu unter siehe 9.3.

1.5 Fazit

1.6 Spielbeschrieb

1.6.1 Der Wettbewerb

Die AI Challenge[1] ist ein internationaler Wettbwerb des University of Waterloo Computer Science Club der im Zeitraum Herbst 2011 bis Januar 2012 zum 3. Mal stattgefunden hat. Das Spiel ist ein zugbasiertes Multiplayerspiel in welchem sich Ameisenvölker gegenseitig bekämpfen. Ziel einer AI-Challenge ist es, einen Bot zu schreiben, der die gegebenen Aufgaben mit möglichst intelligenten Algorithmen löst. Die zu lösenden Aufgaben der Ants AI Challenge sind die Futtersuche, das Explorieren der Karten, das Angreifen von gegnerischen Völkern und deren Ameisenhaufen sowie dem Schützen des eigenen Ameisenhaufen.

1.6.2 Spielregeln

Nachfolgend sind die wichtigsten Regeln, die während dem Spiel berücksichtigt werden müssen, aufgelistet.

- Pro Zug können alle Ameisen um ein Feld (vertikal oder horizontal) verschoben werden.
- Pro Zug steht insgesamt eine Rechenzeit von einer Sekunde zur Verfügung. Es dürfen keine Threads erstellt werden.
- Bewegt sich eine Ameise in die 4er Nachbarschaft eines Futterpixel, wird dieses eingesammelt. Beim nächsten Zug entsteht bei dem Ameisenhügel eine neu Ameise.
- Die Landkarte besteht aus passierbaren Landpixel sowie unpassierbaren Wasserstellen.
- Ein Gegener wird geschlagen, wenn im Kampfradius der eigenen Ameise mehr eigene Ameise stehen als gegnerische Ameisen im Kampfradius der Ameise die angegriffen wird.
- Ein Gegner ist ausgeschieden wenn alle seine eigenen Ameisenhügel vom Gegner vernichtet wurden. Pro verlorenem Hügel gib es einen Punkteabzug. Pro feindlichen Hügel, der zerstört wird gibt es zwei Bonuspunkte.
- Steht nach einer definierbaren Zeit (Anzahl Züge) kein Sieger fest, wird der Sieger anhand der Punkte ermittelt.

Die ausführlichen Regeln können auf der Webseite nachgelesen werden: `http://aichallenge.org/specification.php`

1.6.3 Schnittstelle

Die Spielschnittstelle ist simpel gehalten. Nach jeder Spielrunde erhält der Bot das neue Spielfeld mittels String-InputStream, die Spielzüge gibt der Bot dem Spielcontroller mittels String-OutputStream bekannt. Unser MyBot leitet von der Basis-Klasse Bot[2] ab. Ein Spielzug wird im folgendem Format in den Output-Stream gelegt:

o <Zeile> <Spalte> <Richtung>

Beispiel:

```
o 4 7 W
```

Die Ameise wird von der Position Zeile 4 und Spalte 7 nach Westen bewegt.
Der Spielcontroller ist in Python realisiert, der Bot kann aber in allen gängigen Programmiersprachen wie Java, Python, C#, C++ etc. geschrieben werden.

[1] `http://www.aichallenge.org`
[2] Die Klasse ist im Code unter ants.bot.Bot.Java auffindbar

1.7 Abgrenzungen

Da unsere Arbeit auf dem vorgängigen Modul 'Projekt 2' aufbaut wurden nicht alle Module während der Bachelorarbeit erstellt. Da wir im Modul 'Projekt 2' auf einen sauberen Aufbau geachtet haben, war es uns möglich die meisten Komponeten zu übernehmen. Es folgt eine Auflistung was bereits bestand bzw. was wir noch erweitert haben.

Erstellt in Modul 'Projekt 2'	Erweiterung in während der Bachelorarbeit
Grundfunktionalitäten des Bots	
Pfadsuche Simple, A*, HPA*	Auslagerung in ein eigenes Framework, Performanceverbesserungen, Erweiterung des HPA* Clustering, Pfadsuche mittels InfluenceMap
Logging	Loggen in verschiedene Logfiles
Tasks	Ressourcenmanagement, welcher Task wieviel Ameisen zugeteilt bekommnt
Missionen	Die Missionen wurden verfeinert und mit strategischen un taktischen Entschiedungen erweitert
MinMax-Algorithmus	Bei den Kampfsituationen wurde versucht den Min-Max Alogrithmus einzubauen, welcher wir bereits im Modul Spieltheorie erarbeitet haben. (Mehr dazu siehe Kapitel Strategie)

TODO EINHEITLICHE TABELLEN FORMATIERUNG

2 Architektur

2.1 Modulabhängigkeiten

2.2 Sequenzdiagramme

3 API

Unsere AITools API dient als Grundbaustein des gesamten Projekt. Sie entkapselt die Interfaces zu unseren Search- und Strategieimpelementierungen. Sie beinhaltet zudem Basisklassen, welche überall verwendet werden. Der Inhalt ist wiefolgt:

3.1 Entities

- **Aim**: Richtungsangabe zum Beschreiben einer Bewegung der Ameise
- **Tile**: Repräsentiert eine Zelle auf dem Spielfeld, welche mit Row (Zeile) und Column (Spalte) beschrieben ist.
- **Move**: Diese Klasse beschreibt einen Spielzug mit den Eigenschaften Tile, von wo aus der Zug statt findet, und Aim, in welche Richtung der Zug ausgeführt wird.
- **Unit**: Unit besteht aus Tile und Spieler und definiert eine Einheit eines Spielers auf der Karte.

3.2 Map

- **TileMap**: Dieses Interface definiert, welche Methoden eine auf Tiles aufbauende Spielkarte anbieten muss. Dazu gehören die Masse der Karte, ob ein Tile auf der Karte sichtbar und passierbar ist, sowie Distanz Messfunktion wie ManhattanDistanz, Luftlinie, quadrierte Distanz.
- **UnitMap**: UnitMap ist auch ein Interface welches auf TileMap aufbaut und zusätzlich die Methoden definiert die Einheiten und Spieler zurückzugeben.
- **AbstractWraparoundMap**: Implementiert das Interface SearchableUnitMap (siehe: Abschnitt Search). Hier sind alle Methoden implementiert, welche die Karte anbieten muss, damit sie mit den Suchalgorihmen verwendet werden kann. Zudem sind die Methoden aus TileMap implementiert. Diese geben über die Geländebeschaffenheit und Distanzen Auskunft.
- **WorldType**: WorldType ist ein Enum und definiert die Art der Karte. Der Typ Globus hat keine Kartenränder, ist also ringsum begehbar. Von diesem Typ ist auch die Ants Challenge somit auch die Klasse AbstractWraparoundMap. Der zweite Enumtyp ist Pizza und definiert eine Welt so wie unsere Erde vor 500 Jahren noch definiert wurde, eine Scheibe mit Rändern, welche die Welt begrenzen. Dieser zweite Typ wurde provisorisch erstellt. Falls diese API eine Weiterverwendung findet, kann dieser Typ zusätzlich implementiert werden.

TODO Grafik vererbung UnitMap SearchableMap WarparaoundMAP

3.3 Search

- **PathPiece**: Das PathPiece ist ein Interface für Strukturen, die als Suchknoten in der Pfadsuche verwendet werden können. Es definiert die für die Suche nötigen Methoden, wie getSuccessors(), getCost(), oder getPath(). Implementierende Klassen sind Edge (repäsentiert eine Kante in einem Cluster) und Tile. Erweiterungen dieser Klassen sind DirectedEdge (eine gerichtete Kante) und Vertex (eine Zelle mit zugehörigen Kanten).
- **SearchableMap**: Das Interface SearchableMap erweitert das Interface TileMap. Es beschreibt die Methoden zur Pfadsuche.

- **SearchableUnitMap**: Dieses Interface dient ausschliesslich zur Zusammenführung der beiden Interfaces UnitMap und SearchableMap.

3.3.0.1 SearchTarget

4 Suchalgorithmen

4.1 Pfadsuche

Wir haben drei mögliche Pfadalgorithmen in unserem Code eingebaut. Via PathFinder-Klasse kann für die Pfadsuche der Algorithmus ausgewählt werden.

4.1.1 Simple Algorithmus

Der Simple Algorithmus versucht das Ziel zu erreichen indem er zuerst die eine, dann die andere Achse abläuft. Sobald ein Hindernis in den Weg kommt, bricht der Algorithmus ab. In der Abbildung 4.1 sucht der Algorithmus zuerst den Vertikal-Horizontal Pfad. Da dieser Pfad wegen dem Wasserhindernis (blau) nicht ans Ziel führt, wird via Horizontal-Vertikal Pfad gesucht. Hier wird ein Pfad gefunden. Dieser Algorithmus ist, wie der Name bereits aussagt, sehr einfach aufgebaut und kostet wenig Rechenzeit. Dafür kann er keinen Hindernissen ausweichen.

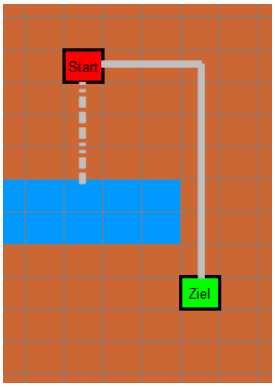

Abbildung 4.1: Simple-Path Algorithmus

Folgendes Codesnippet zeigt auf wie ein Pfad mittels Pfadsuche Simple gefunden wird. Ein SimplePathFinder wird mit der Karte initialisiert. Danach kann die Suche mit pf.search(...) gestartet werden. Als Parameter wird der Suchalgormithmus Strategy.Simple, der Startpunkt (Position der Ameise) und Endpunkt (Position des Futters), sowie die maximalen Pfadkosten (hier: 16) mitgegeben.

```
SimplePathFinder pf = new SimplePathFinder(map);
List<Tile> path = pf.search(PathFinder.Strategy.Simple, ant.getTile(), foodTile,16);
```

4.1.2 A* Algorithmus

Beim A* Algorithmus werden für jeden expandierten Knoten die geschätzten Kosten f(x) für die gesamte Pfadlänge berechnet. f(x) besteht aus einem Teil g(x) welches die effektiven Kosten vom Startknoten zum aktuellen Knoten berechnet. Der andere Teil h(x) ist ein heuristischer Wert, der die Pfadkosten bis zum Zielknoten approximiert. Dieser Wert muss die effektiven Kosten zum Ziel immer unterschätzen. Dies ist in unserem Spiel dadurch gegeben, dass sich die Ameisen nicht diagonal bewegen können, wir aber für den heuristischen Wert die Luftlinie zum Ziel verwenden. Die Pfadsuche wird immer bei dem Knoten fortgesetzt welcher die kleinsten Kosten f(x) hat.

Die Abbildung 4.2 zeigt den effektiven Pfad (grau) vom zu expandierenden roten Knoten mit den minimalen Kosten von 10 Pixel. Die Luftlinie (blau) als heuristischer Wert hat aber nur eine Länge von 7.6 Pixel. Damit erfüllt unsere Heuristik die Anforderungen des Algorithmus.

Eine Pfadsuche mit A* wird gleich ausgelöst wie die Suche mit dem Simple-Alogirthmus, ausser dass als Parameter die Strategy AStar gewählt wird.

```
SimplePathFinder pf = new SimplePathFinder(map);
List<Tile> path = pf.search(PathFinder.Strategy.AStar, ant.getTile(), foodTile,16);
```

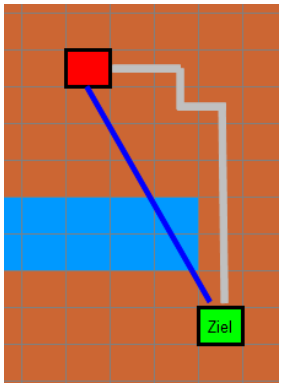

Abbildung 4.2: Heuristische Kosten (blau), Effektive Kosten (grau)

Dieser A*-Algorithmus wird in unserem Code für eine Pfadsuche über alle Pixel (jedes Pixel ist ein Node) verwendet. Der gleiche Code wir aber auch für die Pfadsuche mit dem Pfadnetz des HPA* verwendet.

4.1.3 HPA* Algorthmus

Eine Pfadsuche A* über alle Pixel ist sehr teuer, da es viel Pfade gibt, die zum Teil nur ein Pixel nebeneinander liegen. Es werden bis zum Schluss verschiedenen Pfaden nachgegangen. Abhilfe zu dieser sehr feinmaschigen Pfadsuche bietet der Hierarchical Pathfinding A* bei welchem im sogenanten Clustering über mehrere Pixel verlaufende Kanten und Knoten berechnet werden.

4.1.3.1 Clustering

Das Clustering wird während dem ClusteringTask ausgeführt, Dabei wird die Landkarte in sogenannte Clusters unterteilt. Auf dem Bild 4.3 wurde die Karte in 16 Clusters aufgeteilt.

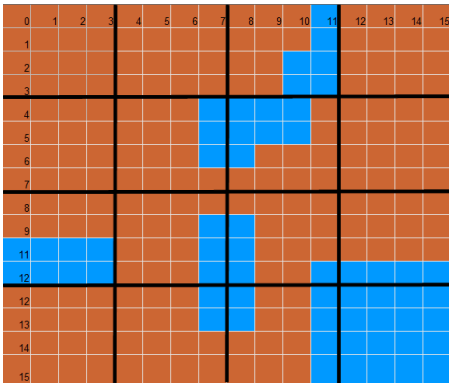

Abbildung 4.3: Clustereinteilung auf der Landkarte. Clustergrösse 4x4, Landkarte 16x16

Danach werden für jeden Cluster und einen Nachbar-Cluster aus der Vierer-Nachbarschaft die Verbindungskanten berechnet. Dies kann natürlich nur für Clusters gemacht werden die auf einem sichtbaren Teil der Landkarte liegen, was zu Begin des Spiel nicht gegeben ist. Deshalb wird der ClusteringTask in jedem Spielzug aufgerufen, in der Hoffnung ein Cluster komplett verbinden zu können. Sobald eine beliebige Seite eines Clusters berechnet ist, wird diese Aussenkante im Cluster und dem anliegenden Nachbar gespeichert und nicht mehr neu berechnet.

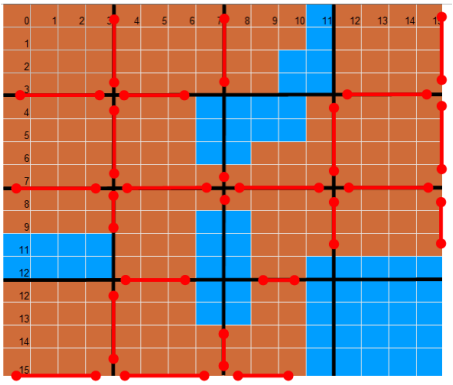

Abbildung 4.4: Die Kanten jedes Clusters wurden berechnet

Sobald ein Cluster zwei oder mehrere Aussenkanten kennt berechnet er die Innenkanten mit A* welche die Knoten der Aussenkanten verbinden. Dies ergibt nun ein Pfadnetz über die Gesamtkarte. Im nachfolgenden Bild sind die Innenkanten (gelb) ersichtlich, die bei den ersten 8 Cluster berechnet wurden.

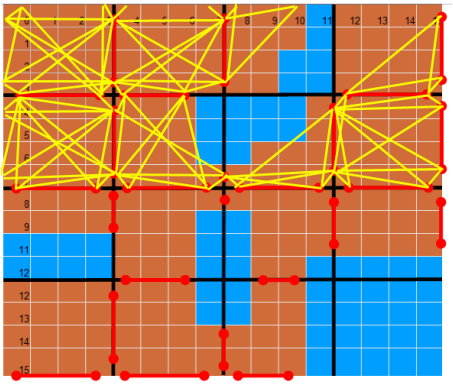

Abbildung 4.5: Darstellung der Innenkanten

In der Abbildung 4.6 wird ein Pfad vom Pixel (3,9) nach (13,9) mittels HPA* gesucht (grüne Punkte). Zuerst wird eruiert in welchem Cluster sich das Start- bzw Zielpixel befindet. Danach wird in dem gefundenen Cluster ein Weg zu einem beliebigen Knoten auf der Clusterseite gesucht. Sind diese Knoten erreicht (blaue Pfade), wird nun das vorberechnete Pfadnetz mittels bereits beschrieben A* Algorithmus verwendet um die beiden Knoten auf dem kürzesten möglichen Pfad (gelb) zu verbinden.[1]

Um eine Pfadsuche mit HPA* durchzuführen muss ein ClusteringPathFinder instanziert werden. Als Parameter erwartet der Konstruktor die Karte auf welcher das Clustering und die Pfadsuche gemacht wird, sowie die Clustergrösse (hier: 10) und den Clustertyp. Das Clustering wird mit pf.update() durchgeführt. Danach kann die Pfadsuche durchgeführt werden. Falls das Clustering auf dem benötigten Kartenausschnitt nicht komplett durchgeführt werden konnte, weil nicht alle Tiles in einem Cluster sichtbar waren, wird versucht mit A* einen Pfad zu suchen.

```
ClusteringPathFinder pf = new ClusteringPathFinder(map, 10, type);
pf.update();
List<Tile> path = pf.search(PathFinder.Strategy.HpaStar, start, end, -1);
```

[1] Der resultierende Pfad könnte mittels Pathsmoothing verkürzt werden. Dies wurde aber in unserer Arbeit nicht implementiert.

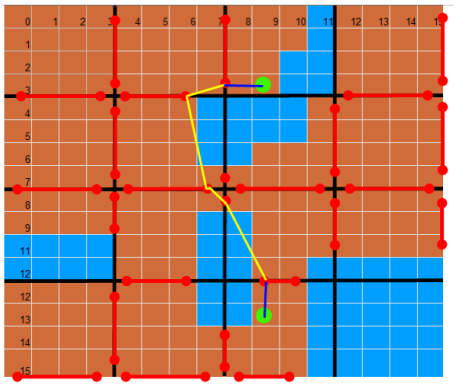

Abbildung 4.6: Errechneter Weg mittels HPA*

4.1.4 Pfadsuche mittels Influence Map

Die Influence Map, welche wir während der Bachelorarbeit neu implementiert haben, kann auch für die Pfadsuche verwendet werden. Dabei sind die Pfadkosten für Gebiete in die vom Gegner kontrolliert sind höher als für neutrale Gebiete und tiefer für solche Gebiete die von unseren Ameisen kontrolliert werden. (Details zur Implementierung der Influence Map sieh Kapitel TODO) Die Methode getActualCost(...) in der Klasse SearchStrategy wurde erweitert. Falls die Suche mit einer InfluenceMap initialisiert wurde, sind die Kosten nicht eine Einheit pro Pfadtile, sondern können zwischen 1 (sicheres Gebiet) - 4 (gefährliches Gebiet) Einheiten varieren. (Die Pfadkosten dürfen nicht negativ sein, sonst würde der A* Algorithmus nicht mehr korrekt funktionieren.) Die Kosten für jeden Pfadabschnitt werden durch die Methode getPathCosts(...) der InfluenceMap berechnet.

```
protected final int getActualCost(Node current, PathPiece piece) {
    int costOfPiece = 0;
    if (useInflunceMap)
        costOfPiece = pathFinder.getInfluenceMap().getPathCosts(piece);
    else
        costOfPiece = piece.getCost();
    return current.getActualCost() + costOfPiece;
}
```

Dadurch resultiert ein Pfad der durch sicheres Gebiet führt. Folgende Ausgabe, welche durch einen UnitTest generiert wurde, bezeugt die korrekte Funktionalität. Der rote Punkt soll mit dem schwarzen Punkt durch einen Pfad verbunden werden. Auf der Karte sind zudem die eigenen, orangen Einheiten sowie die gengerischen Einheiten (blau) abgebildet. Jede Einheit trägt zur Berechnung der Influence Map bei. Pro Tile wird die Sicherheit ausgegeben, negativ für Gebiete die vom Gegner kontrolliert werden und positiv in unserem Hoheitsgebiet.

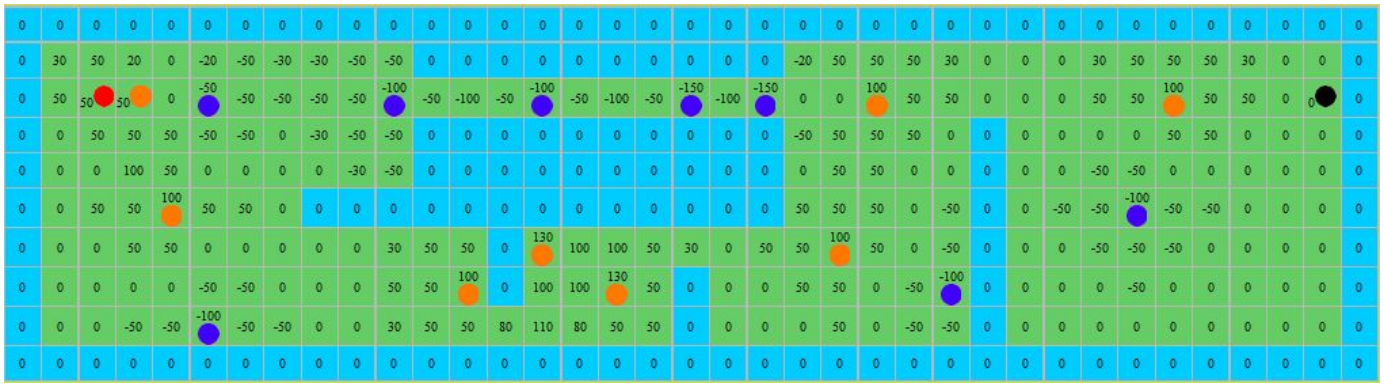

Abbildung 4.7: Ausgangslage Pfadsuche mit A* und InfluenceMap

Ohne Berücksichtigung der InfluenceMap würde der A* Algorithmus einen Pfad finden der auf direktem Weg waagrecht zum Zielpunkt führt. Sobald aber die InfluenceMap berücksichtigt wird, führt der Pfad nicht mehr auf dem direktestem Weg zum Ziel, sondern nimmt einen Umweg über sicheres Gebiet. Unten abgebildet ist der kürzeste Pfad mit Berücksichtigung der Influence Map (blau) und ohne Influence Map-Berücksichtigung (orange).

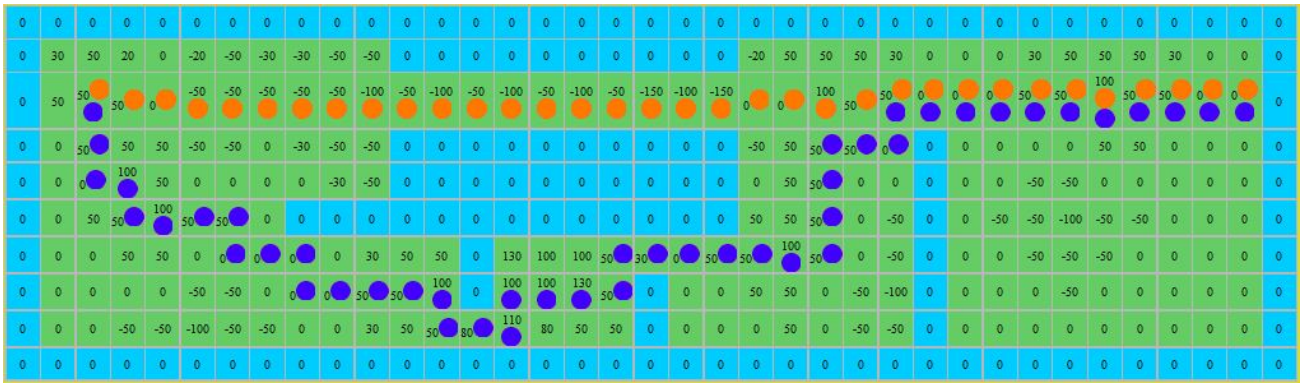

Abbildung 4.8: Resultierende Pfade mit und ohne Berücksichtigung der Influence Map

Die Pfadkosten für beide Pfade verglichen, legt offen, dass je nach Berücksichtigung der Influence Map nicht der gleiche Pfad als der 'Kürzeste' von A* gefunden wird.

	Kosten ohne Influence Map	Kosten mit Influence Map
Oranger Pfad	**34**	110
BlauerPfad	46	**106**

4.2 Breitensuche

Die Breitensuche (engl. breadth-first search (BFS)) war eine der Neuimplementierungen während der Bachelorarbeit. Man könnte die BFS auch für die Pfadsuche verwenden, dies wäre aber sehr ineffizient. Wir verwenden diese Suche vielmehr für die Umgebung einer Ameise oder eines Hügels zu analysieren. Sie wurde generisch implementierte, so dass sie vielseitig einsetzbar ist. So können zum Beispiel mittels 'GoalTest' je nach Anwendungsfall die Tiles beschrieben werden welche gesucht sind. Folgende Breitensuche findet die Ameise welche am nächsten bei einem Food-Tile <r:20,c:16> ist. Sie wird initialisiert indem im Konstruktor die Spielkarte mitgegeben wird, welche durchforscht wird. Zusätzlich gilt die Einschränkung das die Breitensuche nur 40 Tiles durchsuchen darf, was einem Radius von zirka 7 entspricht. Falls keien Ameise gefunden wird gibt der Algorithmus NULL zurück.

```
AntsBreadthFirstSearch bfs = new AntsBreadthFirstSearch(Ants.getWorld());
Tile food = new Tile(20,16);
Tile antClosestToFood = bfs.findSingleClosestTile(food, 40, new GoalTest() {
      @Override
      public boolean isGoal(Tile tile) {
          return isAntOnTile(tile);
      }
  });
```

Es ist auch möglich mehrere Tiles zurück zu bekommen. Dazu wird die Methode findClosestTiles(...) aufgerufen.

Der gleiche Alogrithmus kann aber auch alle passierbaren Tiles in einem gewissen Umkreis zurückgeben. Dies haben wir unteranderem beim Initialisieren der DefendHillMission verwendet. Wir berechnen beim Erstellen der Mission die passierbaren Tiles rundum den Hügel. Runde für Runde prüfen wir diese Tiles auf gegnerische Ameisen um die entsprechenden Verteidigungsmassnahmen zu ergreifen. Der Parameter controlAreaRadius2 definiert den Radius des 'Radars' und kann je nach Profile unterschiedlich eingestellt werden.

```
public DefendHillMission(Tile myhill) {
    this.hill = myhill;
    BreadthFirstSearch bfs = new BreadthFirstSearch(Ants.getWorld());
    tilesAroundHill = bfs.floodFill(myhill, controlAreaRadius2);
}
```

5 Strategie und Taktik

5.1 Influence Map

Die Influence Map haben wir nach den Beschreibungen des Buches Artificial Intelligence for Games implementiert. Jede bekannte Spieleinheit auf der Spielkarte 'strahlt' eine gewissen Einfluss aus. In unserer Implementation unterscheiden wir zwischen drei Einflussradien, der Angriffsradius, der erweiterte Angriffsradius und der Sichtradius. Den Radien haben wir folgende Werte zugewiesen.

Radius	Wert	Radius in Tiles*
Angriffsradius	50	2.2
Erweiterter Angriffsradius	30	5
Sichtradius	10	8.8

* Die Radiuslänge kann je nach Spieleinstellung ändern. Angegeben sind die Defaultwerte.

Wir verenden die Influence Map vorallem für die Bestimmung der Sicherheit. Abgebildet ist eine Sicherheitskarte (Desirability Map) für den orangen Spieler, wobei die Einflusswerte des Gegners von den Einflusswerten des eigenen Spielers je Tile subtrahiert werden. Positive Werte bedeuten sicheres Terrain und negative Werte unsicheres, vom Gegner kontrolliertes Gebiet.

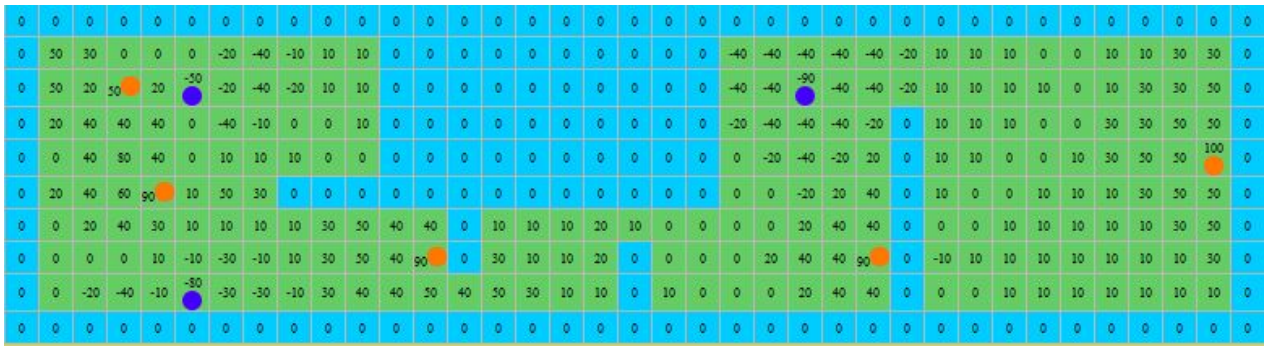

Abbildung 5.1: Infleunce Map, dargestellt ist die Sicherheit je Tile.

5.1.1 Update

Die Influence Map wird zu Beginn des Spiels initialisiert, danach wird vor jeder Spielrunde ein Update gemacht. Dabei definert ein Decay-Wert zwischen 0 und 1, wieviel von den alten Werten beibehalten wird. Folgende Formel bestimmt den neuen Wert für jede Zelle:

$$val_{(x,y)} = val_{(x,y)} * decay + newval_{(x,y)} * (1 - decay)$$

5.1.2 Andwendungsfälle

In folgenden Modulen berücksichtigen wir Werte aus Influence Map um Entscheide zu fällen.

- **Pfadsuche mit Influence Map Berücksichtigung**: Siehe Kapitel 4.1.4
- **CombatSituation: Flucht**: Müssen wir die Flucht ergreifen, bewegen wir unsere Ameise auf das nächste sicherste Tile.

- **Abbruch Mission**: Falls eine Ameise auf einer andere Mission als die GatherFoodMission ist und ein Food Tile in seiner Nähe antrifft, wird abgewogen ob die Mission zu Gunsten von Futter sammeln abgebrochen werden soll. Dabei ist ein Entscheidungsfaktor auch die 'Sicherheit' des Futters. Falls das Futter nicht auf einem sicheren Weg geholt werden kann, wird die Mission nicht abgebrochen.

Natürlich könnte man die Influence Map auch für weitere Entscheidungen verwenden. Auch der Einsatz von Spannungskarte (Tension Map), welche auch auf der InfluenceMap aufbaut, wäre denkbar. Dies wurde während dieser Arbeit nicht angeschaut bzw. implementiert.

5.2 Combat Situations

Kampfsiuationen werden immer dann erstellt wenn gegnerische Ameisen auf unsere Ameisen treffen. Dies ist vorallem der Fall wenn ein gegnerischer Hügel angegriffen wird, oder unserer Hügel verteidigt werden muss. Eine Kampfsituation kann sich aber auch sonst wo auf der Karte ereignen.

5.2.1 DefaultCombatPositioning

DefaultCombatPositioning implementiert das Interface CombatPositioning und führt die Postionierung für die drei Verhalten FLEE, DEFEND, ATTACK an. Das Verhalten wird in der Methode determineMode(...) wie folgt bestimmt, wobei das 'DEFAULT' Verhalten dem ATTACK-Verhalten entspricht.

```
protected Mode determineMode() {
    final boolean enemyIsSuperior = enemyUnits.size() > myUnits.size();
    if (enemyIsSuperior)
        return Mode.FLEE;
    return Mode.DEFAULT;
}
```

Wird nicht ein DefaultCombatPositioning initialisiert, sondern ein AttackingCombatPositioning (in der AttackHillMission) oder ein DefendingCombatPositioning (in der DefendHillMission) so wird das Verhalten anders bestimmt, indem die Methode determineMode() überschrieben ist.

TODO DIAGRAMM

DefendingCombatPositioning

TODO

AttackingCombatPositioning

TODO

Die bereits erwähnten Verhalten, nehmen folgende Positionierung der Ameisen vor.

5.2.1.1 FLEE

Für jede Unit wird das sicherste Nachbartile mittels Influence Map bestimmt. Die Unit verschieb sich auf das sicherste Nachbartile.

```
for (Tile myUnit : myUnits) {
    nextMoves.put(myUnit, map.getSafestNeighbour(myUnit, influenceMap));
}
```

5.2.1.2 DEFEND

TODO

5.2.1.3 ATTACK

TODO

6 Ants

6.1 State-Klassen

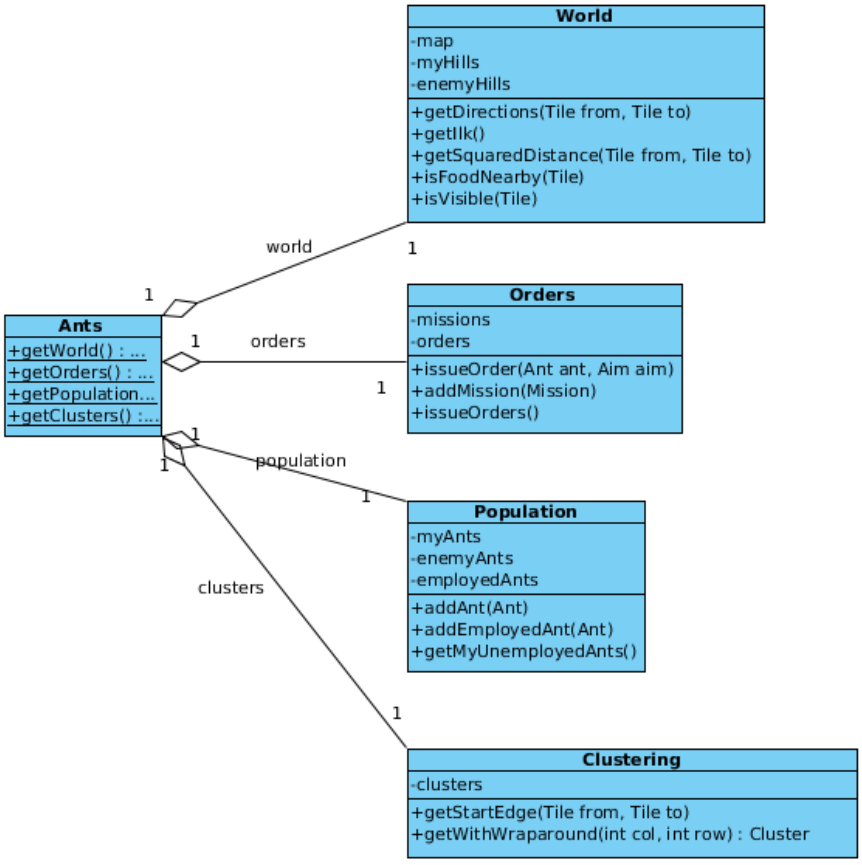

Abbildung 6.1: State-Klassen (vereinfacht)

Abbildung 6.1 zeigt eine Übersicht über die Zustands-Klassen. Für das Diagramm wurden lediglich die wichtigsten Methoden und Attribute berücksichtigt. Die State-Klassen implementieren alle das Singleton-Pattern.

6.1.1 Ants

Die Ants Klasse ist die zentrale State-Klasse. Sie bietet auch einfachen Zugriff auf die anderen State-Klassen. Ursprünglich hatten wir alle Methoden, die mit dem Zugriff auf den Spielzustand zu tun hatten, direkt in der Ants Klasse implementiert, haben aber schnell gemerkt, dass das unhandlich wird. Die Ants Klasse dient jetzt vor allem als Container für die anderen State-Klassen und implementiert nur noch einige Methoden, die Zustandsänderungen in verschiedenen Bereichen vornehmen.

6.1.2 World

Die World Klasse enthält Informationen zur Spielwelt. Hier wird die Karte abgespeichert, in der für jede Zelle die aktuell bekannten Informationen festgehalten werden. Das beinhaltet die Sichtbarkeit der Zelle und was die Zelle aktuell enthält (Ameise, Nahrung, Wasser, ...). Ausserdem werden Listen geführt, wo sich die eigenen und die bekannten gegnerischen Hügel befinden. Die Klasse bietet Methoden zur Distanzberechnung, gibt Auskunft über einzelne Zellen und darüber, ob sich Nahrung in der Umgebung einer bestimmten Zelle befindet.

6.1.3 Orders

In der Orders Klasse wird über Befehle und Missionen der einzelnen Ameisen Buch geführt. Die Liste der Befehle wird dabei in jedem Zug geleert und neu befüllt, während die Liste der Missionen zugübergreifend geführt wird. Das zentrale Verwalten der Befehle und Missionen dient dazu, sicherzustellen, dass keine widersprüchlichen Befehle ausgegeben werden wie: mehrere Befehle für eine Ameise, gleiche Ziel-Koordinaten für mehrere Ameisen, eine Ameise in mehreren Missionen etc.

6.1.4 Population

Die Population Klasse dient der Verwaltung der eigenen und der gegnerischen Ameisen-Völker. Hier werden die Ameisen mit ihren aktuellen Aufenthaltsorten festgehalten. Wenn für eine Ameise ein Befehl ausgegeben wird, wird die Ameise als beschäftigt markiert. Über die Methode `getMyUnemployedAnts()` kann jederzeit eine Liste der Ameisen abgefragt werden, die für den aktuellen Zug noch keine Befehle erhalten haben.

6.1.5 Clustering

Die Clustering Klasse dient dem Aufteilen des Spielfeldes in Clusters für die HPA*-Suche (s. Abschnitt 4.1.3). Hier werden die berechneten Clusters abgelegt, damit diese nicht bei jeder Verwendung neu berechnet werden müssen. Der Zugriff auf sie erfolgt ebenfalls über die Clustering Klasse.

6.2 Spiel-Elemente (Welt)

Abbildung 6.2 zeigt die wichtigsten Klassen, die die Elemente des Spiels repräsentieren. Der Übersichtlichkeit wegen wurden nur die wichtigsten Attribute und Operationen in das Diagramm aufgenommen.

6.2.0.1 Ant

Eine Ant gehört immer zu einem Spieler; über die Methode isMine() können unsere eigenen Ameisen identifiziert werden. Eine Ameise weiss jeweils in welcher Zelle sie steht. Das Feld nextTile dient der Verfolgung einer Ameise über mehrere Züge – das Feld wird jeweils aktualisiert, wenn der Ameise ein Befehl ausgegeben wird. Im nächsten Zug können wir dann prüfen ob die Ameise den Befehl korrekt ausführen konnte. Eine Ameise kennt auch die anderen Ameisen in ihrer Umgebung: Über die Methoden getEnemies()/FriendsInRadius() können alle bekannten Freunde und Feinde in einem bestimmten Radius ermittelt werden.

6.2.0.2 Tile

Das Tile repräsentiert eine Zelle des Spielfelds. Es implementiert das SearchTarget Interface (s. 3.3.0.1). Es bietet zudem Methoden für die einfache Distanzberechnung, sowie für das Bestimmen der Richtungen, in der ein anderes Tile liegt.

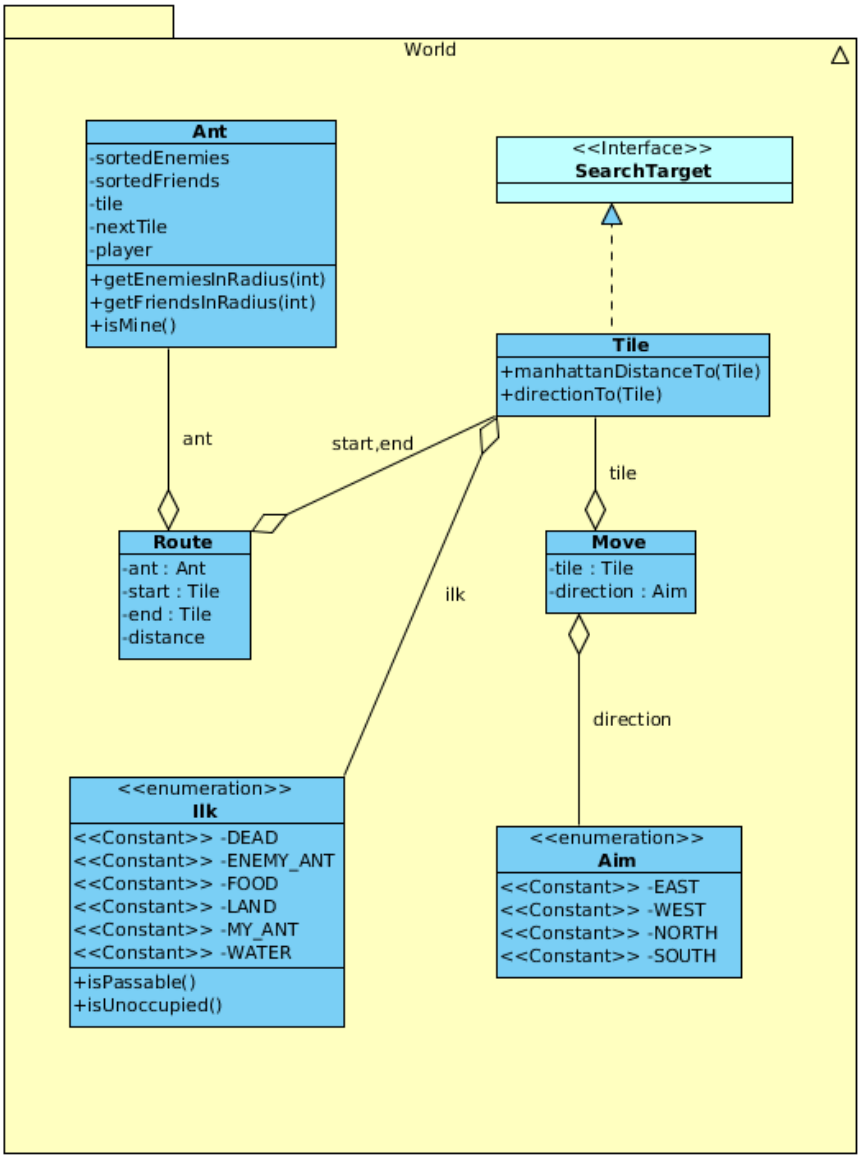

Abbildung 6.2: Spiel-Elemente der Spielwelt (vereinfacht)

6.2.0.3 Route

Eine Route repräsentiert eine einfache Start-Ziel Verbindung. Sie hält für eine Ameise die Luftliniendistanz zu einem bestimmten Zielfeld fest.

6.2.0.4 Move

Ein Move entspricht einem Zug einer Ameise. Für ein bestimmtes Tile wird angegeben, in welche Richtung sich die Ameise bewegen soll.

6.2.0.5 Ilk

Ilk ist der Typ einer Zelle. Der Ilk einer Tile-Instanz gibt an, was sich gerade in der Zelle befindet. Dies kann ein Gelände-Typ sein, wenn die Zelle ansonsten leer ist, oder es kann eine Ameise, Nahrung, oder ein Hügel sein. Die Ilk-Enumeration bietet Hilfsmethoden, um festzustellen, ob eine Zelle passierbar oder besetzt ist.

6.2.0.6 Aim

Aim ist einfach eine Repräsentation einer Himmelsrichtung

6.2.1 Spiel-Elemente (Suche)

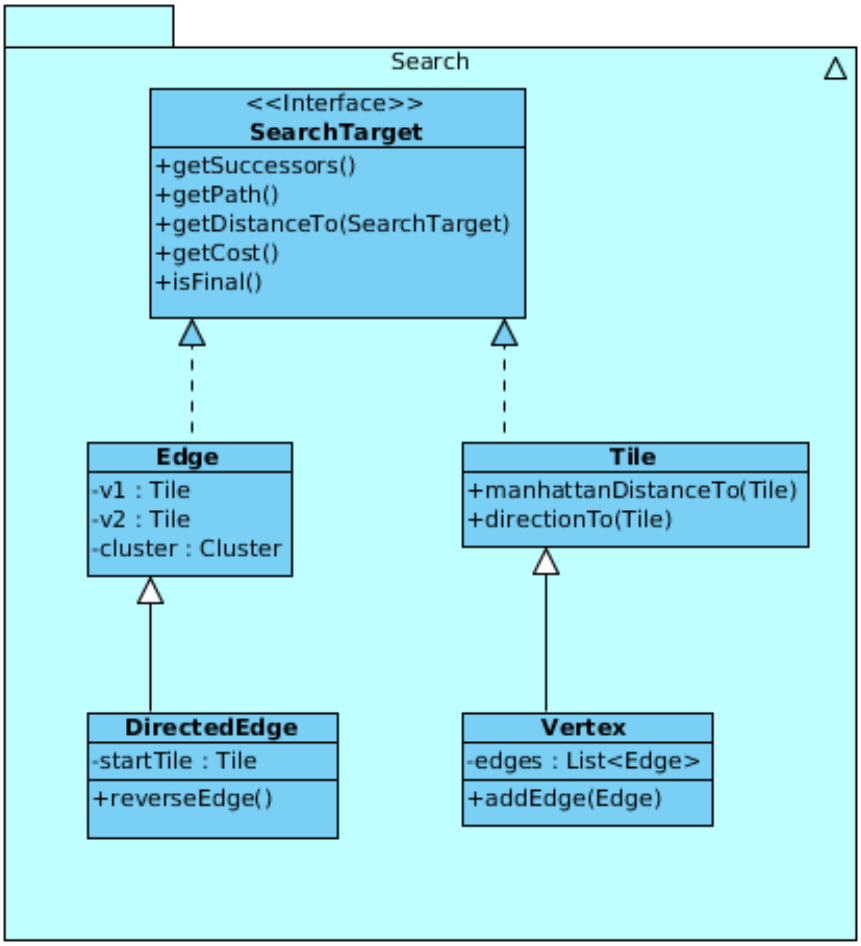

Abbildung 6.3: Spiel-Elemente für die Suche (vereinfacht)

Abbildung 6.3 zeigt die wichtigsten Klassen, die für die Pfadsuche verwendet werden. Der Übersichtlichkeit wegen wurden nur die wichtigsten Attribute und Operationen in das Diagramm aufgenommen.

6.3 Bot

Als Basis für unsere Bot Implementation haben wir den Beispiel-Bot verwendet, der im Java-Starter-Package enthalten ist, das von der AI-Challenge-Website heruntergeladen werden kann. Dieser erbt von den Klassen AbstractSystemInputReader und AbstractSystemInputParser, die die Interaktion mit der Spiele-Engine über die System-Input/Output Streams kapseln. Für eine optimierte Lösung könnte der Bot auch angepasst werden, indem er selber auf die Streams zugreift. Im Rahmen dieser Arbeit erschien uns das aber noch nicht nötig.

6.3.1 Ablauf eines Zugs

Abbildung 6.4 zeigt den Ablauf des ersten Zugs, während Abbildung 6.5 den Ablauf aller weiteren Züge zeigt.

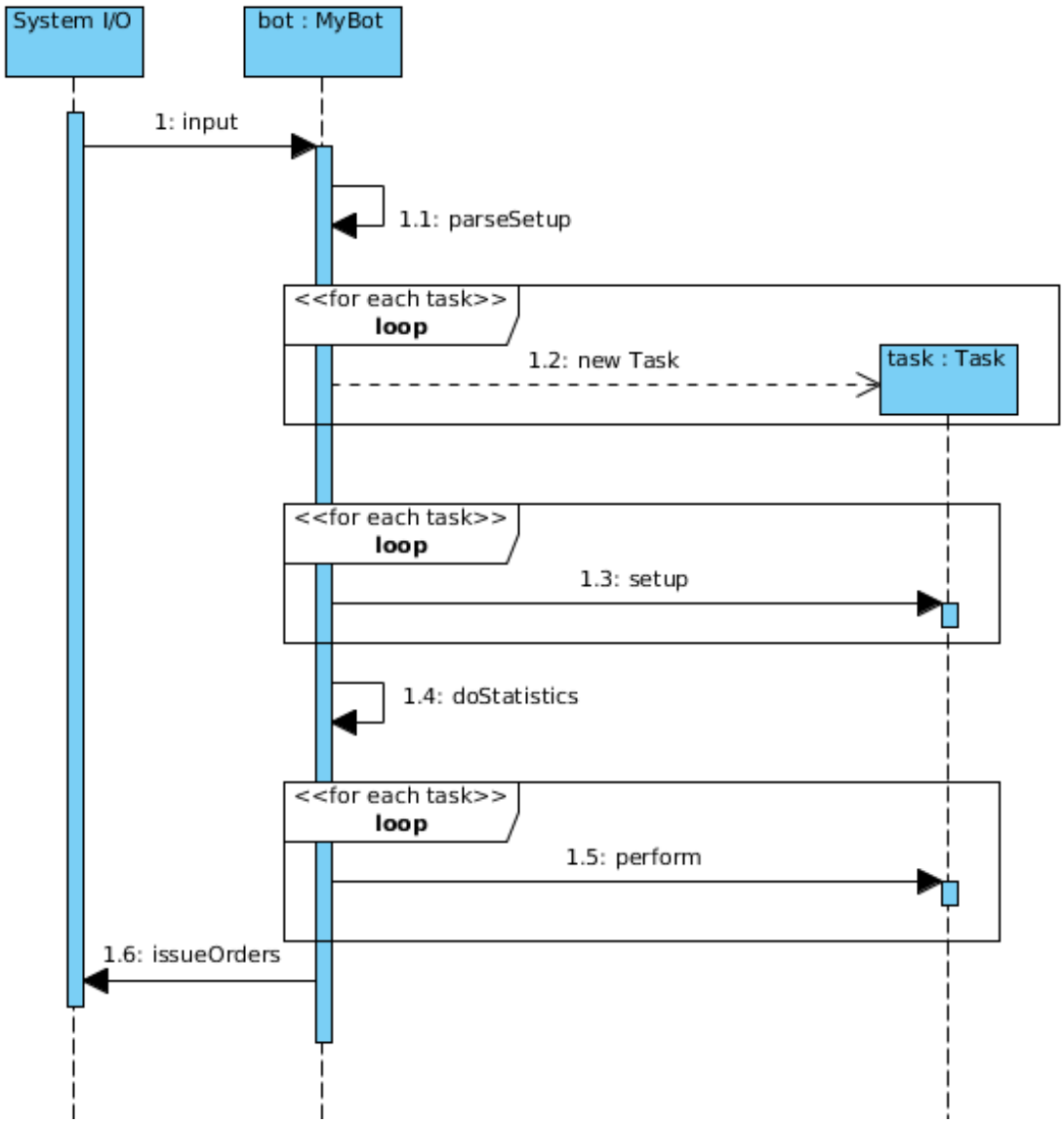

Abbildung 6.4: Ablauf des ersten Zugs des Spiels

Jeder Zug beginnt mit dem Einlesen des Inputs vom SystemInputStream. Wenn der Bot das Signal "READY" (1. Zug) oder "GO" (alle weiteren Züge) erhält, kann er den gesammelten Input verarbeiten (Methode parseSetup() resp. parseUpdate()). Danach wird die eigentliche Logik des Bots ausgeführt.

Im 1. Zug werden dabei Instanzen der Tasks erstellt. Abgesehen davon unterscheidet sich der 1. Zug von diesem Punkt an nicht mehr von allen nachfolgenden Zügen. Die Tasks werden vorbereitet (Aufruf der jeweiligen setup() Methode; danach werden einige statistische Werte aktualisiert und in jedem 10. Zug auch geloggt. Dann werden die Tasks in der definierten Reihenfolge aufgerufen. Hier wird der Löwenanteil der Zeit verbracht, denn die Tasks enthalten die eigentliche Logik unserer Ameisen.

Zum Schluss werden dann mit issueOrders() die Züge der Ameisen über den SystemOutputStream an die Spielengine übergeben.

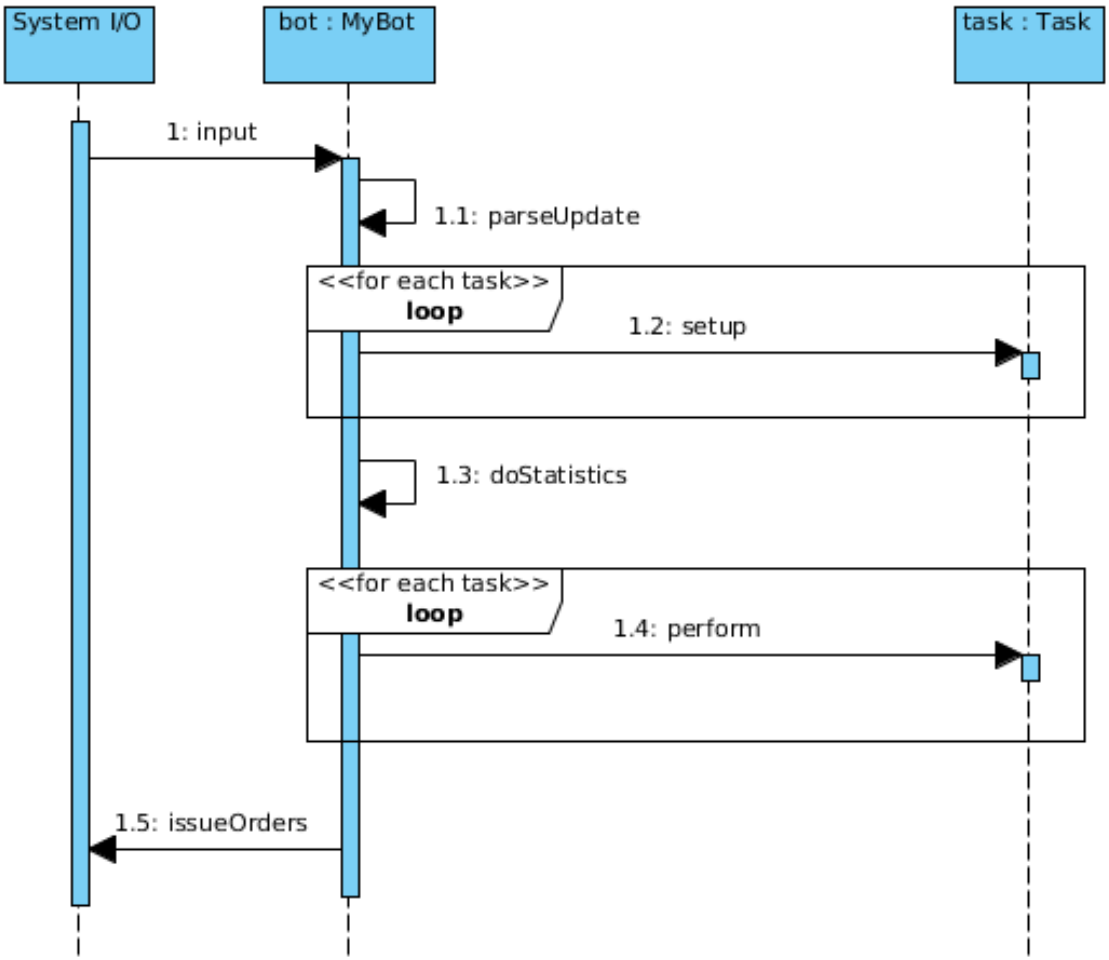

Abbildung 6.5: Ablauf der weiteren Züge des Spiels

6.4 Tasks

Zu Beginn des Projekts haben wir die wichtigsten Aufgaben einer Ameise identifiziert. Diese Aufgaben wurden als Tasks in eigenen Klassen implementiert. Das Interface Task[1] definiert eine setup()-Methode welche den Task initiiert, sowie eine perform()-Methode welche den Task ausführt. Im Programm werden die Tasks nach deren Wichtigkeit ausgeführt, was auch der nachfolgenden Reihenfolge entspricht. Jedem Task stehen nur die unbeschäftigten Ameisen zur Verfügung, d.h. jene welchen noch keine Aufgabe zugeteilt wurde.

6.4.1 MissionTask

Dieser Task prüft alle aktuellen Missionen auf deren Gültigkeit, beispielsweise ob die Ameise der Mission den letzten Zug überlebt hat und die Mission weiterführen kann. Falls gültig, wird der nächste Schritt der Mission ausgeführt.

6.4.2 GatherFoodTask

Für jedes Food-Tile werden in einem definierbaren Radius r die nächsten Ameisen bestimmt. Danach wird nach aufsteigender Luftliniendistanz mit dem Pfadsuchalgorithmus SIMPLE (s. Abschnitt **??**) oder – falls dieser keinen Pfad gefunden hat – mit A* eine passierbare Route gesucht. Wenn ein Pfad existiert, kann mit der Ameise und dem

[1] Das Interface ist im Code unter ants.tasks.Bot.Java auffindbar.

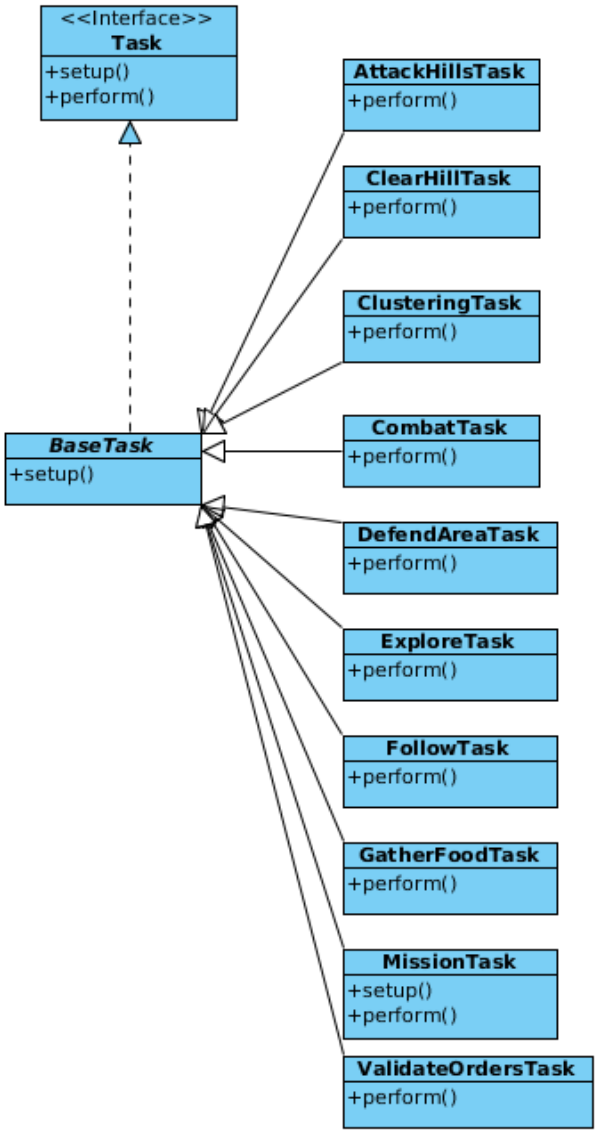

Abbildung 6.6: Tasks

Food-Tile eine GatherFoodMission erstellt werden, welche die Ameise zum Food-Tile führt. Zu jedem Food-Tile wird immer nur eine Ameise geschickt.

6.4.3 AttackHillsTask

Sobald gegnerische Ameisenhaufen sichtbar sind, sollen diese angegriffen werden. Das Zerstören eines gegnerischen Haufens ist wie erwähnt 2 Punkte wert. Die Kriterien, nach denen eine Pfad zum gegnerischen Haufen gesucht wird, sind die selben wie beim GatherFoodTask, ausser dass mehrere Ameisen das Ziel angreifen können. Es wird eine AttackHillMission erstellt.

6.4.4 CombatTask

Beim Angriffstask wird berechnet ob wir in einem Kampfgebiet (definiert über den Sichtradius einer Ameise) die Überhand, d.h. mehr Ameisen platziert haben. Falls ja, wird die gegnerische Ameise angegriffen.

6.4.5 ExploreTask

Für alle noch unbeschäftigten Ameisen wird mittels ManhattanDistance der nächste Ort gesucht, der noch nicht sichtbar, also unerforscht ist. Falls ein Pfad mittels Pfadsuchalgorithmus gefunden wird, wird eine ExploreMission (s. Abschnitt **??**) erstellt. Die Ameise wird den gefundenen Pfad in den nächsten Spielzügen ablaufen.

6.4.6 FollowTask

Der FollowTask ist für Ameisen angedacht welche aktuell keine Aufgabe haben. Diese Ameisen sollen einer nahe gelegenen, beschäftigten Ameise folgen, damit diese nicht alleine unterwegs ist.

6.4.7 ClearHillTask

Dieser Task bewegt alle Ameisen, welche neu aus unserem Hügel "schlüpfen", vom Hügel weg. So werden nachfolgende Ameisen nicht durch diese blockiert.

6.4.8 ClusteringTask

Der ClusteringTask wird als Vorbereitung für den HPA* Algorithmus verwendet. Hier wird für alle sichtbaren Kartenregionen ein Clustering vorgenommen. Das Clustering wird im Kapitel **??** im Detail beschreiben.

6.5 Missionen

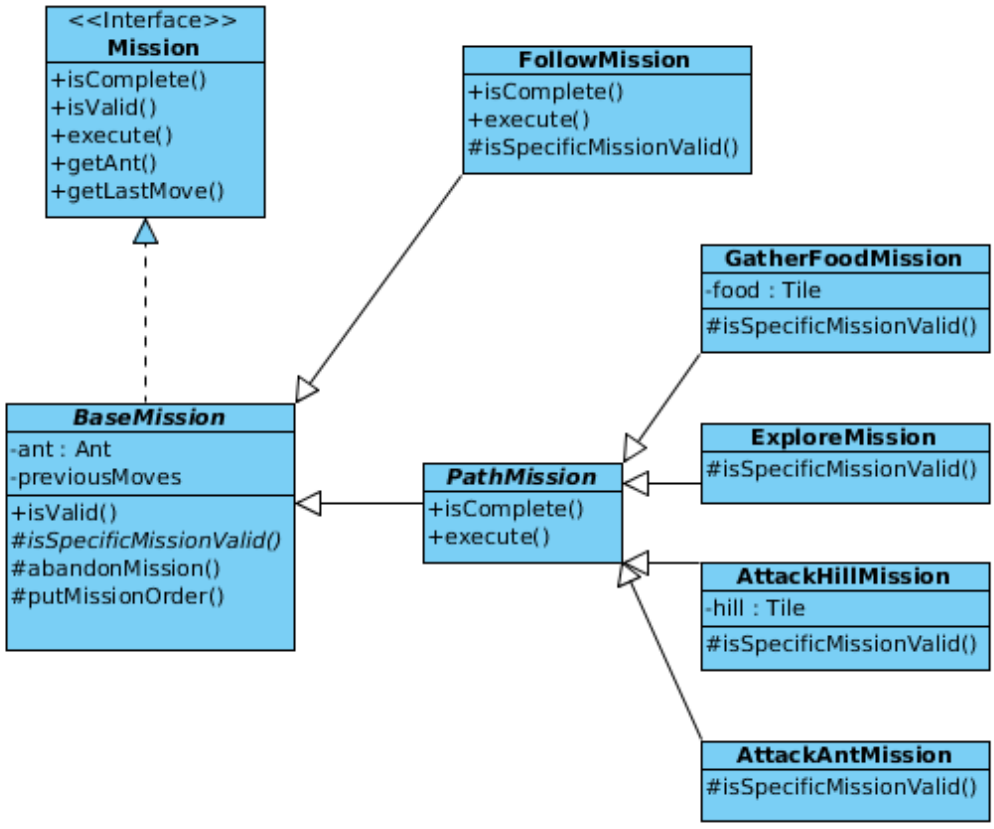

Abbildung 6.7: Missionen

Eine Mission dauert über mehrere Spielzüge. Die meisten Missionen (GatherFoodMission, ExploreMission, AttackHillMission, AttackAntMission) sind Pfadmissionen[2], bei welchen die Ameise einem vorgegebenen Pfad, der bereits beim Erstellen der Mission berechnet wurde, folgt. Die FollowMission ist eine spezielle Mission, mit der eine Ameise einfach einer anderen Ameise hinterherläuft.

Eine Mission kann auch abgebrochen werden, wenn es keinen Sinn mehr macht, sie weiter zu verfolgen. Je nach spezifischer Mission sind aber die Abbruchbedingungen anders. Zum Beispiel die GatherFoodMission ist nur solange gültig wie das Futter noch nicht von einer anderen Ameise eingesammelt wurde. Abbildung 6.7 zeigt einen Überblick über die wichtigsten Missionen und ihre Hierarchie.

6.6 Profile

[2]Die abstrakte Klasse PathMission ist im Code unter ants.missions.PathMission.java auffindbar.

7 Logging

Nach einem absolvierten Spiel analysierten wir jeweils die Spielsituationen, welche sich ergeben haben. Dazu gehörte das Analysieren des geschriebenen Logs. Dabei bedienten wir uns den nachfolgenden Mechanismen.

7.1 Logkategorien und Loglevel

Jeder Logeintrag gehört einer Logkategorie an. Je Logkategorie kann der Loglevel defniert werden. Die Loglevel lauten TRACE, DEBUG, INFO und ERROR. Wenn also zum Beispiel bei der Logkategorie ATTACKHILLMISSION der Loglevel auf INFO gestellt ist, werden nur die Fehler auf Stufe INFO und ERROR in das Logfile geschieben. Zudem kann, falls erwünscht, jede Logkategorie in ein eigenes Logfile geschrieben werden. Die meisten Module habe ihre eigene Logkategorie, so kann durch korrekte Logeinstellung erzwungen werden dass nur die Logs, welche für das Analysieren eines bestimmten Spielmoduls von Bedeutung sind, ins Logfile geschrieben werden. Dadurch müssen nicht riesige Mengen an Logs druchwälzt werden um an die Informationen heran zu kommen.

7.2 JavaScript Addon für HMTL-Gameviewer

Das Codepaket welches von den Challenge-Organisatoren mitgeliefert wird, bietet bereits eine hilfreiche 2D-Visualisierung des Spiels, mit welchem das Spielgeschehen mitverfolgt werden kann. Die Visualisierung wurde mit HMTL und Javascript implementiert. Leider ist es nicht möglich zusätzliche Informationen auf die Seite zu projizieren. Deshalb haben wir den Viewer bereits im Projekt 2 mit einer solchen Funktion erweitert. Mit der Codezeile Logger.liveInfo(...) kann eine Zusatzinformation geschrieben werden, welche auf dem Viewer später sichtbar ist. Es muss definiert werden mit welchem Zug und wo auf dem Spielfeld die Infomation angezeigt werden soll. Im Beispiel wird an der Position der Ameise (ant.getTile()) ausgegeben welchen Task die Ameise hat.

```
Logger.liveInfo(Ants.getAnts().getTurn(), ant.getTile(),
                "Task: %s ant: %s", issuer, ant.getTile());
```

Auf der Karte wird ein einfaches aber praktisches Popup mit den geschriebenen Informationen angezeigt. Dank solcher Zusatzinformationen muss nicht mühsam im Log nachgeschaut werden, welcher Ameise wann und wo welcher Task zugeordnet ist.

Das angezeigte Popup zeigt welchen Task (GatherFoodTask) die Ameise hat, wo sie sich befindet <r:28 c:14>, welches Futterpixel angesteuert wird <r:35 c:13> und welchen Pfad dazu berechnet wurde. Im Rahmen der Bachelorarbeit wurde dieses Addon erweitert. Nun werden alle Pixel welche in dem Popup ausgegeben werden auf der Karte markiert. Siehe (Abb. **??**)

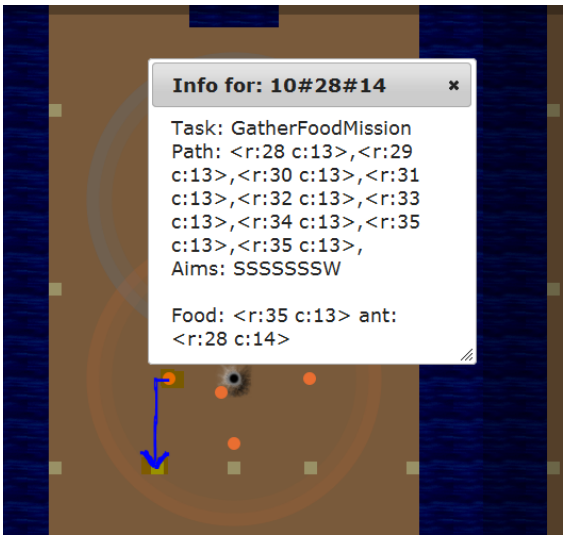

Abbildung 7.1: Im Popupfenster steht die Aufgabe der Ameise sowie die Pixel des Pfades (falls vorhanden), welcher die Ameise ablaufen wird.

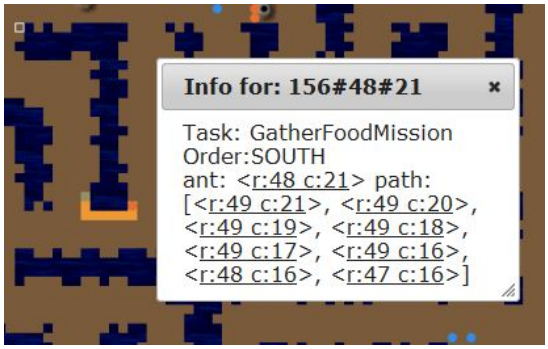

Abbildung 7.2: Mit der erweiterten Version wird der Pfad (orange) der Ameise von <r:48 c:21> nach <r:47 c:16> auf der Karte abgebildet.

8 Testreader

9 TestCenter

todo besserer name

9.1 Unit- und Funktionstests

9.2 Verschiedene Bots

9.3 Testreport Profile

10 Spielanleitung

Dieser Anhang beschreibt kurz, wie ein Spiel mit unserem Bot ausgeführt werden kann.

Das File `Ants.zip` enthält das Eclipse-Projekt mit dem gesamten Source-Code unserer Implementation, und der offiziellen Spiel-Engine. Zum einfachen Ausführen eines Spiels haben wir ein ANT-Buildfile (build.xml) erstellt. Dieses definiert 3 Targets, mit denen ein Spiel mit jeweils unterschiedlichen Parametern gestartet werden kann.

1. Das Target `testBot` ist lediglich zum einfachen Testen eines Bots sinnvoll und entspricht dem Spiel, das verwendet wird, um Bots, die auf der Website hochgeladen werden, zu testen.
2. Das Target `runTutorial` führt ein Spiel mit den Parametern aus, die im Tutorial auf der Website zur Erklärung der Spielmechanik verwendet werden.
3. Das Target `maze` führt ein Spiel auf einer komplexeren und grösseren, labyrinthartigen Karte aus und ist das interessanteste von den 3.

Im Unterordner `tools` befindet sich die in Python implementierte Spiel-Engine. Unter `tools/maps` liegen noch weitere vordefinierte Umgebungen, und unter `tools/mapgen` liegen verschieden Map-Generatoren, die zur Erzeugung beliebiger weiterer Karten verwendet werden können.

Im Unterordner `tools/sample_bots` befinden sich einige einfache Beispiel-Bots, gegen die man spielen kann. Viele der Teilnehmer haben zudem ihren Quellcode auf dem Internet publiziert, an möglichen Gegner besteht also auch kein Mangel.