ApoSkunkmanKI von Kilian Gärtner

# Einleitung

Ich habe zwei KIs geschrieben. Die eine KI names Meldanor (wird auch von der Klasse „Meldanor“ gestartet) werde ich zuerst erläutern. Diese erfüllt die Klausurvorraussetzungen.

Die zweite KI ist eine CheatKI. Ich greife über Reflections auf private Variablen zu und verändere das Spiel je nach Situation. Diese zweite KI ist meine Haupteinreichung. Die Erläuterungen dazu finden im zweiten Teil statt.

Das komplette Projekt ist auf meinem GitHub Repo unter <https://github.com/Meldanor/MyApoShunkman> zu finden.

# Meldanor – Die normale KI

## Übersicht

Diese KI benutzt ein zielorientiertes System, um das Hauptziel zu erreichen. Die Wegfindung wird mit Astern realisiert.

Folgende Klassen gehören zu dieser KI:

**AIGoalX, AILeftRightBot, AIManager, AStar, AStarLevel, Goal, GoodiCosts, LevelCosts, Node, Meldanor, MeldanorPlayer, PlantbombGoal, TakeCoverGoal, Tickable und WalkGoal**

Die Klasse **Meldanor** macht nichts weiter, als den **AIManager** zu instanzieren und bei jedem Tick der Think Methode aufzurufen. Der AIManager entscheidet beim allersten Tick, an welche KI er den Tick weiterleiten soll. Dies basiert auf der Spielsituation und den Aufgaben, die die KI erfüllen muss.

Für GoalX Level wird die Klasse **AIGoalX** benutzt, für den LeftRightBot wird **AILeftRightbot** benutzt.

## Das Goal-System

Ich hab eine abstrakte Klasse **Goal** erstellt. Sie repräsentiert alle Aufgaben des Bots. So lange ein Goal noch nicht gefinished oder gecancelled wurde, wird dieses bearbeitet. Es war auch angedacht, dass die Goals nach Prioritäten in einer PriorityQueue sortiert werden und immer das „kritischte“ Goal dort bearbeitet wird.

Implementierte Goals sind **WalkGoal**,welches die Aufgabe „Gehe zu einem Ziel“ repräsentiert. Alle Bewegungen, wo kein Busch im Weg ist, werden darüber realisiert. Solange ein Knoten auf dem Pfad existiert, geht der Spieler bei jedem Tick einen Schritt weiter.

Das **TakeCoverGoal** wird aufgerufen, wenn eine Bombe gefunden wurde, die den Spieler bedroht. Dieses Ziel sucht sich erst eine Deckung und dann einen Weg mit AStern dahin. Das Ziel ist erreicht, wenn die Bombe explodiert ist.

Das **PlantBombGoal** wird benutzt, wenn der Spieler eine Bombe irgendwo platzieren will. Es erbt von WalkGoal und der einzige Unterschied ist, dass nach Erreichen des letzten Knoten des Pfades das Ziel noch nicht erreicht wurde. Er wird dann so lange versuchen eine Bombe zu legen, bis dies geht. Erst dann ist das Ziel erreicht.

Alle Goals werden einer Queue gehalten und ein Goal noch dem anderem abgehandelt. Ich habe das System anders angedacht, deswegen ist die jetzige Implementation ziemlich umständlich und doppelt. Es könnten höhere Goals erstellt werden, die aus mehreren SubGoals bestehen.

## Die Wegfindung mit A\* (A-Stern)

Ich benutze für die Wegfindung A\* in einer leicht optimierten Variante. A\* selber wird in der Klasse **AStar** gehandhabt. Der Graph und das Handeling wird in der Klasse **AStarLevel** gekapselt. Der Graph selber besteht aus einzelnen **Nodes** , den Knoten des Graphen. Diese sind im 2D-Raum angesiedelt und erben deswegen von der Klasse **Point**. Die Gewichtungen für die **H-Werte** der Knoten sind in den beiden Enums **GoodieCosts** und **LevelCosts** gespeichert.

A\* benutzt für die *openList* eine PriorityQueue, da ich so das Suchen nach dem geringsten Knotenwert in O(1) realisieren kann. Das Einfügen in die *openList* dauert jedoch O(log n). Die PriorityQueue Klasse von Java basiert auf einen binärem Heap, eine mögliche Verbesserung wäre die eigene Implementierung eines FibonacciHeaps.

Die *closedList* ist bei mir ein HashSet, so dass ich in O(1) überprüfen kann, ob ein Knoten bereits in der *closedList* ist. Das Problem ist jedoch, dass für die Pfadgenerierung durch den Graphen der zuletzt in die *closedList* eingefügte Knoten benötigt wird. Durch ein HashSet ist dies nicht realisierbar, deswegen speichere ich jeweils den letzt hinzugefügten Knoten zwischen.  
Um die Generierung des Hashes für alle Knoten zu beschleunigen, speichere ich diese zwischen, wenn die Knoten erstellt werden.Für jeden Aufruf von A\* existiert jeder Knoten für sich selber, zwei verschiedene ASterns haben zwei verschiedene Graphen. Da die Knoten nicht translatiert werden, bleibt der Hash für die komplette Instanz von A\* gleich und so ist das Cachen des Hashes möglich.

Sobald A\* benötigt wird, erstellt er einen Graphen auf Basis des 2D Array , welches das Level repräsentiert. Knoten, die einen Stein abbilden, sind null pointer, um Speicherplatz zu sparen. Dieses Level existiert nur für diesen A\* und für jeden A\* wird ein neues Level erzeugt. Da der Graph in diesem Spielfeld gerade einmal 15x15 , also 225, Knoten beinhalten kann, ist dies zu vernachlässigen.

## Heuristik

Der H Wert jedes Knotens setzt sich aus der Luftlinie zum Ziel multipliziert mit der Gewichtung seiner Position zusammen. Die Gewichtungen sind in den Enums GoodieCosts und LevelCosts gespeichert. LevelCosts speichert die Gewichtungen für ein freies Feld , einen Bush und eine Bombe. GoodieCost speichert die Gewichtungen für die Goodies ab.

Verallgemeinert lässt sich sagen, dass alle guten Goodies einen geringeren H Wert als ein freies Feld bevorzugen. So steigt die Chance, dass bei der Wegfindung ein Goodie mit aufgenommen wird.

Da der Spieler für eine Bewegung von einem Tile zu einem anderen Tile immer die selbe Zeit benötigt, ist auch der G Wert immer der G Wert des Vorgängers plus eins.

## MeldanorPlayer

Diese Klasse erweitert den normalen **ApoSkunkmanAIPlayer** um eine wenige Funktionen, wie die Wegfindung oder das Bewegen im 2D-Raum.

## Deckungssuche

Der Deckungssuchalgorithmus sucht entweder eine Nische auf der Achse oder aber den Punkt auf der Achse mit dem minimalsten Abstand zur Bombe außerhalb des Radius.

Er schaut sich alle Achsenrichtungen um die Bombe an und überprüft, ob dort ein Hindernis steht.

In Abbildung 1 ist B die Bombe(Radius = 1) und die Pfeile simbolisieren die aktuell geprüfte Richtung. Oben und Links ist direkt ein Hinderniss und werden bei der weiteren Ausdehnung des Suchradius nicht mehr betrachtet. Da keine Nischen exisiteren auf der rechten Seite oder unten, wird der Suchradius ausgeweitet.

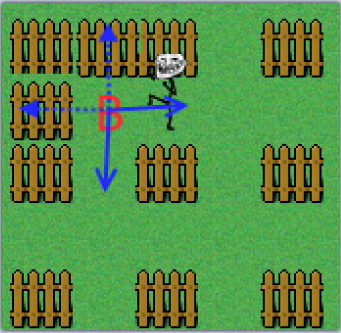
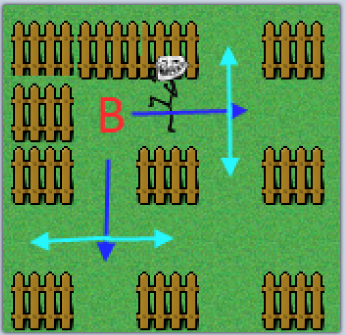


Abbildung 1

Abbildung 2

In Abbildung 2 wurde der Suchradius ausgeweitet und es werden nicht mehr oben und links betrachtet. Die türkisen Pfeile simbolisieren mögliche Nischen. Alle möglichen Deckungspunkte werden in eine Liste getan. Diese Liste wird abgearbeitet und zu jeder Deckung ein Pfad berechnet. Der Pfad mit den wenigstens Knoten ist dann der Pfad in die Deckung.

## GoalX Level

Sobald eine **AIGoalX** Instanz erzeugt wird, sucht der einen globalen Pfad durch das Level vom Start- zum Zielpunkt. Dazu wird A\* benutzt, jedoch ignoriert A\* hier Büsche und betrachtet diese als betretbar.So gibt es einen optimalen Weg durch das Level, jedoch müssen die Büsche weggesprengt werden. Der Weg durch das Level wird nun einmal traversiert und jedes mal, wenn der aktuelle Knoten ein Busch ist, wird am Punkt davor ein **PlantBombGoal** erstellt und in eine Queue gepackt. Ist der große Pfad abgearbeitet, besteht die Queue aus mehreren **PlantBombGoals** und zum Schluss einem einzigen **WalkGoal**(dies ist der Weg vom letzten Busch zum Ziel).

Nun greift das oben genannt Prinzip des **Goal**systemes. Das nächste Goal wird aus der Queue entfertn(also ein **PlantBombGoal**) und nun wird mit A\* der Ziel von der aktuellen Position zum Zielpunkt berechnet. Danach bewegt sich der Spieler mit jedem Tick genau einen Knoten auf das Ziel zu und beim Erreichen des Zieles, legt er die Bombe vor dem Busch.

Bevor jedoch auch nur ein Ziel abgehandelt wird, wird in jedem Tick überprüft, ob eine Bombe im Level liegt.Wenn eine liegt, wird überprüft , ob diese den Spieler treffen könnte. Dies geschieht, wenn die Bombe auf einer Achse mit dem Spieler liegt und die Bombe in Reichweite ist(hier werden noch keine Barrieren betrachtet, spätere Implementationen dieses Algorithmus haben die Verbesserung). Wenn eine Bombe den Spieler bedroht, wird ein primäres **TakeCoverGoal** erstellt und vor das aktuelle Ziel gestellt. Ist die Bombe explodiert, ist das CoverGoal erfüllt, für das letzte Ziel wird der Weg neu berechnet und die Queue wird weiter abgearbeitet.  
Mögliche Verbesserung wäre es bei diesem LevelTyp unter idealen Bedingungen(keine Gegner) sich die Bombe zu merken, statt 15x15 Überprüfungen zu machen, ob im kompletten Level eine Bombe liegt.

Dieser Algorithmus wird so lange ausgeführt, bis das Ziel erreicht ist.

## Left-Right-Bot

Der Algorithmus hierfür ist 1:1 der Algorithmus aus GoalX. Das Ziel hier ist jedoch nicht der Schatz, sondern der Bot. Ist die Queue mit den Zwischenbefehlen komplett abgearbeitet, steht die KI beim Bot und legt eine Bombe an der Position des Spielers. Dies tut er solange, bis die KI tot ist.

Dies sind die Algorithmen für die normale Einreichung. Im folgendem Abschnitt wird die CheatAI erläutert.

# MeldanorTroll – Reflectionsbasierende KI

## Übersicht

Diese „künstliche Intelligenz“ verändert das Spiel je nach Spielsituation. Ich habe mich vor einigen Wochen mit Reflections in Java auseinander gesetzt und aus Interesse angefangen, herauszufinden, was man mit dieser API machen kann. Jedes Attribut und jede Methode einer Klasse werden in Metadaten in den .class Dateien gespeichert und Reflection kann auf diese zugreifen, obwohl diese zur Compilezeit noch nicht bekannt sind.

Die Einstiegsklasse für diese KI ist **MeldanorTroll**

Mir ist bewusst, dass diese KI keinesfalls Wettbewerbsfähig ist, denn wie kann man eine KI schlagen, die sofort gewinnen kann? Die einzige Möglichkeit wäre es, wenn eine zweite KI das selbe tut und einen Tick vorher zum Zuge kommt. Da es langweilig wäre, sofort zu gewinnen, habe ich vier Verhaltensmuster implementiert, die im Laufe der Dokumentation erläutert werden.

Folgende Klassen gehören zu dieser CheatKI: **CheatAIManager, CheatAIBots, CheatAIEpicBattle, CheatAIGoalX, CheatAIHuman, Initionable, MeldanorTroll, Tickable, TrollAtomicBombEntity, TrollBackgroundEntity, TrollBombEntity, TrollMessageEntity, TrollPortalEntity, TrollStoneEntity.**Alle .png Datei gehören ebenfalls zu dieser KI. Alle Bildrechte sind im Quellcode explizit angegeben.

## Die Vier Strategien

Der **CheatAIManager** nutzt je nach Spielsituation einer der vier Strategien. Zusätzlich stellt es eine Möglichkeit da, Nachrichten auf dem Spielfeld auszugeben. Dazu greife ich auf die fireList zu aus ApoSkunkmanLevel, um eine FakeFire Entity auszugeben. Statt Feuer jedoch zeichnet sie nur eine Nachricht.

Die erste Strategie ist das Bezwingen eines GoalX Levels. Die zweite ist das Kampf gegen Bots. Die dritte ist das Kampf gegen andere KIs. Dies wird erkannt, indem ich prüfe, ob eine der KIs im Spiel eine andere Klasse als ApoAILeftRight, ApoAIRunner oder StinkerMain implementiert. Der letzte Kampf ist der Kampf der CheatKI gegen sich selber.

Jede der Strategien habe ich als eine Art Bossskript geschrieben. Jede Strategie wird initiert und die ganzen nötigen Attribute und Felder angelegt. Bilder werden geladen und das Spiel soweit vorbereitet. Nach diesem ersten Tick wird immer **handleLevel(long delta)** aufgerufen, wo die eigentliche Spielmechanik abgehandelt wird. Das Delta gibt die Differenz von diesem Spielzug zum letzen an und dient dazu, um timerbasierende Events auszulösen. Dieses Konstrukt ist in allen vier Strategien zu finden.

## GoalXLevel – Die weglaufende Truhe

Zum Start wird das komplett Level gelöscht und alles neugezeichnet. Danach wird eine Art Pfad wie im Spiel „Snake“ berechnet, wo die Truhe langrennen kann. Mein Spieler rennt dann dieser Truhe hinterher, bis der Pfad endet. Der Pfad ist nach maximal 500 Schritten vorbei.  
Sollten andere Spieler mit auf dem Feld sein, so werden diese in die linke, obere Ecke gefesselt und können nicht zufällig die Truhe aufnehmen.

## Kampf gegen Bots – Armageddon

Zum Start teleportiert sich mein Spieler in die Mitte des Spielfeldes und legt einen unzerstörbaren Zaun um sich. Danach fängt er an, Bomben vom Himmel regnen zu lassen. In den ersten 15 Sekunden sind diese jedoch nicht in der Nähe der Bots zu finden, danach allerdings kann es die Bots auch direkt treffen. Er lässt eine Bombe alle 0.5 bis 1.5 Sekunden fallen.  
Die BotSkins werden durch RageFaces ausgetauscht.  
In Abständen von ca. 10 Sekunden ändert sich der Bombenradius im Intervall von 1 bis 15. Dies wird auch jeweils auf dem Monitor angezeigt.  
Häufig reicht dies aus, um die Bots nach wenigen Sekunden aus dem Spiel zu entfernen. Sollten jedoch Bots überlebt haben , geht mein Spieler in den Enrage Modus und verändert das komplette Spielfeld. Wo vorher eine friedliche Landschaft war, ist es nun ein gefährliche. Die Bomben werden nun alle 0.25 Sekunden fallen gelasssen, alle Goodies sind BadGodGoodies und alle guten Goodies verschwinden. Wenn er im Enrage Modus ist, sind die restlichen Bots meißt in wenigen Sekunden K.O.

## Kampf gegen Spieler – Unterstützende Strategie

Hier habe ich am längsten gebraucht und viel ausprobiert. Wie kann man Spielern helfen, ohne deren KIs zu zerschiessen, wenn diese auf ein statisches Level ausgelegt sind?

Meine KI verwandelt sich zuerst in GoodGuyGreg und hilft allen anderen Spielern. Alle 1,75 bis 2 Sekunden überprüfe ich alle Bots, ob sie von Bomben getroffen werden können. Wenn ja, lösche ich die Bombe.

Sollte ich jedoch getroffen werden können, teleportierte ich mich immer wieder weg, bis nur noch eine KI existiert. Dann gebe ich auf und teleportiere mich nicht mehr weg und lasse so die KI gewinnen. Wenn diese es schafft, eine Bombe bei mir zu platzieren.

Ansonsten verteile ich noch zufällig Goodies an die KIs.

## Kampf gegen mich selbst – Der geplante epische Kampf

In dieser Strategie wollte ich einen epischen Kampf inszenieren, wo meine KI gegen sich selber kämpft. Wegen Zeitproblemen schaffte ich es jedoch nicht mehr und deswegen ist nur ein relativ langweiliger Kampf zu sehen.

Beide KIs teleportieren sich in die Mitte und lassen Atombomben vom Himmel regnen. Nach 30 Sekunden ist das komplett Spielfeld voll davon und es endet.

Angedacht war für diese Strategie noch viel mehr.