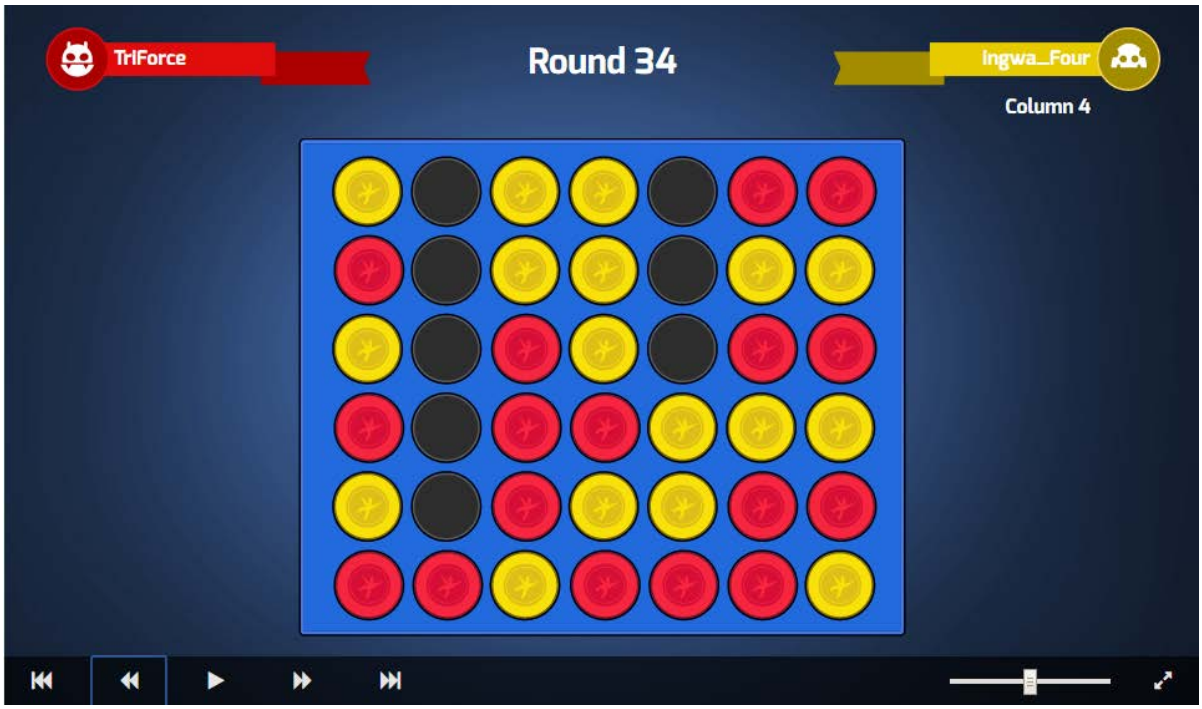
4-Gewinnt Bot

C++ Projektarbeit



Zweck	Projektarbeit
Institution	Berner Fachhochschule, Technik und Informatik
Betreuer	Prof. Dr. Jürgen Eckerle
Fach	Spieltheorie
Autoren	P. Berger, D. Zwygart
Datum	04.06.2016

Zusammenfassung

In den Modulen C++ in Embedded Systems und Spieltheorie müssen wir jeweils eine Projektarbeit abgeben. Beide Themen lassen sich sehr gut in eine einzelne Projektarbeit integrieren. Das Konkrete Wissen aus den jeweiligen Modulen lässt sich so optimal nutzen.

Durch das Modul Spieltheorie ist eine Aufgabe im Bereich der Spiele naheliegend. Wir haben uns für das Spiel 4-Gewinnt entschieden. Die Webseite von AI Games bietet eine hervorragende Plattform für Spielprogramme. Hier können die Benutzer Bots für das jeweilige Spiel hochladen und diese gegeneinander unter Wettkampfbedingungen antreten lassen. Die Interaktion zwischen den einzelnen Bots übernimmt die Game Engine von AI Games. Es müssen lediglich vordefinierte Schnittstellen Implementiert sein.

Als Resultat der Arbeit haben wir einen Bot entwickelt, der den nächsten Zug mittels Tiefensuche und Alpha-Beta-Pruning berechnet. Dieser läuft auf der Website unter dem Namen Triforce und belegt den Rang 31 in der Spielerwertung. Weiter wurde ein Bot mit iterativer Tiefensuche umgesetzt dieser wurde jedoch nur offline getestet hat jedoch eine erhebliche Performance Steigerung gezeigt. Diese Dokumentation befasst sich beim Konzept nur mit dem Bot welcher auch online gestellt wurde. In der Umsetzung wird in einem separaten Kapitel die Veränderungen für die iterative Tiefensuche kurz erläutert.

Inhaltsverzeichnis

1 Einleitung

Viele einzelne unabhängige Projektarbeiten in der Schule sind uninteressant. Wenn diese Arbeiten Modulübergreifen durchgeführt werden können, sind komplexere und interessantere Aufgaben möglich. In den Modulen C++ in Embedded Systems und Spieltheorie der Informatiker müssen wir jeweils eine Projektarbeit abgeben. Beide Themen lassen sich sehr gut in eine einzelne Projektarbeit integrieren.

Wir haben uns für das Spiel 4-Gewinnt entschieden. Die Webseite von AI Games bietet eine hervorragende Plattform für Spielprogramme unter Wettbewerbsbedingungen. Hier können die Benutzer Bots für das jeweilige Spiel hochladen und diese gegeneinander antreten lassen.

Als Vorlage diente ein Basis Projekt in Java welches von AI Games zur Verfügung gestellt wird. Dieses wurde in einem ersten Schritt in C++ nachgebildet. Da es im C++ Modul um die korrekte Planung eines Projektes und den Einsatz von UML geht wurde eine Projektplanung mittels UML umgesetzt.

2 Aufgabenstellung

Die Webseite von Ai Games bietet eine einfache Game-Engine für kleinere Spiele. Für jedes dieser Spiele läuft ein Wettbewerb, um den besten Spielbot der angemeldeten Benutzer zu finden. Diese Bots treten jeweils gegeneinander an und das Resultat des Spiels entscheidet anschliessend über die Rangierung.
Klassisch kann ein solches Spielprogramm in folgende Bereiche Unterteilt werden:

- Der **Parser** ist die Kommunikationsschnittstelle zwischen Game-Engine und dem Spielprogramm
- Eine **Suchfunktion** durchläuft alle möglichen Spielzüge und die daraus resultierenden Spielstellugnen
- Eine **Bewertungsfunktion** entscheidet schlussendlich, ob der Zug sinnvoll ist und wie gut er gegenüber anderen abschneidet.

Damit das Spielprogramm möglichst effizient ist, müssen die Suchfunktion und die Bewertungsfunktion entsprechend programmiert sein. Die Bewertung wird für jeden Zug neu gemacht, weshalb eine schnelle, aber gute Funktion gefordert ist. Bei der Suchfunktion wird normalerweise eine Baumstruktur mit möglichen Zügen und der daraus folgenden Spielsituation aufgebaut. Beim Schach können zum Beispiel nach den ersten 2 Zügen 400 Mögliche Stellungen entstehen (20 mögliche Züge für Weiss und 20 mögliche Züge für Schwarz). Somit wird ein finales Durchrechnen fast nicht möglich. Schätzungen gehen davon aus, dass im Schach etwa $2 \cdot 10^{43}$ legale Positionen existieren[1].
Beim 4-Gewinnt gibt es nicht so viele Möglichkeiten. Hier wird die Herausforderung erhöht, indem das Spielprogramm nur eine gewisse Zeit zum Rechnen zur Verfügung hat. Wird diese Zeit überschritten, so verliert der Bot automatisch das Spiel.

[1] Bonsdorf E., Fabel K., Riihimaa O.: „Schach und Zahl“, Walter Rau Verlag, 3. Auflage, Düsseldorf 1978

3 Pflichtenheft

3.1 Muss-Kriterien

- Die Kommunikation zwischen der Game Engine und dem Bot muss wie auf der Seite http://theaigames.com/competitions/four-in-a-row/getting-started beschrieben funktionieren.
- Der Bot muss einen nächsten Zug innerhalb der Zeitbeschränkung berechnen können welche von der Game-Engine vor jedem Spiel bekannt gegeben wird.
- In einem Fehlerfall muss der Bot korrekt mit der Ausgabe `"no_moves"` reagieren.

3.2 Wunsch-Kriterien

- Um das Debuging zu vereinfachen, sollte eine Ausgabe der Baumstruktur möglich sein welche dann im Programm Graphviz dargestellt wird.
- Die Debug-Ausgabe von Variablen und Resultaten oder Zwischenresultaten soll möglich sein.
- Verschiedene Suchtechniken sind zu entwerfen (Optimierung des Bots).

4 Grundlagen

4.1 Spielstrategie

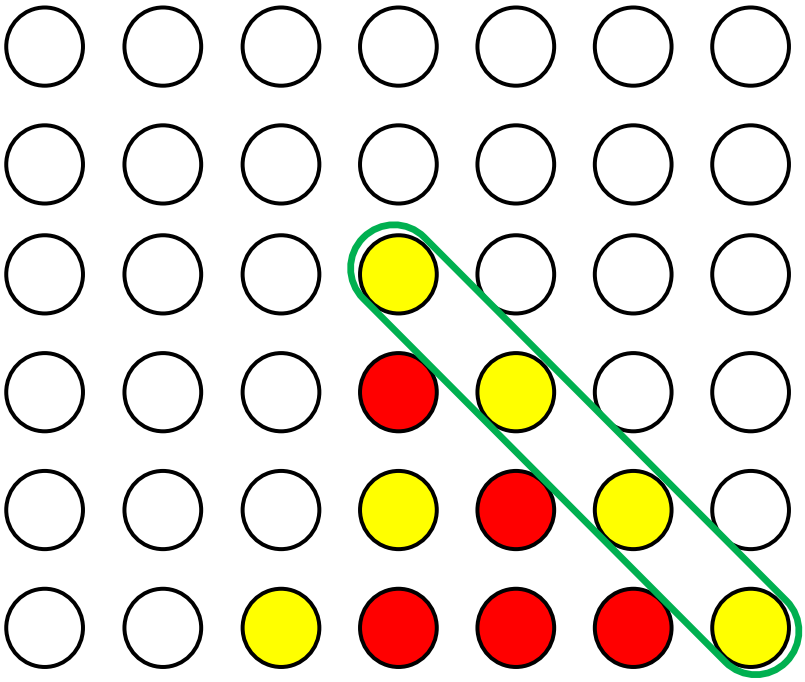


Abbildung 4-1: Grundstrategie von 4-Gewinnt.

4-Gewinnt ist ein relativ altes Spiel, dass die meisten noch aus ihrer Kindheit kennen. Dabei spielen immer 2 Personen gegeneinander. Die Spalten können nur aufgefüllt werden. Ziel von jedem Spieler ist es, dass er 4 Steine ohne Unterbrechung in einer Reihe hinbekommt. Diese können waagrecht, senkrecht oder diagonal verteilt sein. Die Abbildung 4-1 zeigt eine gewonnene Situation für den Spieler Gelb.

4.2 AI Games

AI Games ist eine Webseite, auf der in diversen Spielen Bot-Programme von Nutzern im Wettstreit gegeneinander antreten. Aktuell sind folgende Spiele möglich:

- GO
- Ultimate Tic Tac Toe
- Four In A Row
- Al Block Battle
- Texas Hold'em
- Warlight Al Challenge 2
- Heads up Ohama
- Warlight Al Challenge

Dabei werden die Wettbewerbe oft nicht auf eine einzelne Programmiersprache ausgelegt, sondern sind für alle gängigen Sprachen offen.

4.3 Kommunikation mit der Game Engine

Die komplette Kommunikation zwischen dem Bot und der Game Engine läuft über die `std::in` und `std::out` Kanäle ab. Jede einzelne Zeile der Game Engine entspricht einer Anfrage oder einer Information. Die Antwort des Bots darf auch nur eine einzelne Zeile sein.

Aktuell werden 3 Typen von Ausgaben der Game Engine unterstützt, auf die der Bot reagieren muss:

- **settings [type] [value]** wird nur am Anfang des Spiels gesendet und beinhaltet generelle Informationen.
- **action [type] [time]** löst eine Anfrage aus.
- **update [type] [value]** aktualisiert den Game Zustand.

Folgende Tabellen sind direkt von der Webseite von AI Games übernommen und zeigen alle Kommandos:

Output from engine	**Description**
settings timebank ***t***	Maximum time in milliseconds that your bot can have in its time bank
settings time_per_move ***t***	Time in milliseconds that is added to your bot's time bank each move
settings player_names [***b***,...]	A list of all player names in this match, including your bot's name
settings your_bot ***b***	The name of your bot for this match
settings your_botid ***i***	The number used in a field update as your bot's chips
settings field_columns ***i***	The number of columns of the playing field
settings field_rows ***i***	The number of rows of the playing field
update game round ***i***	The number of the current round
update game field **[[*c*,...];...]**	The complete playing field in the current game state
action move ***t***	Request for a move
Output from bot	**Description**
place_disc ***i***	The move your bot wants to make

Jedes ‘i’ kann ein beliebiger Integer Wert sein und jedes ‘b’ entspricht einem Botname. ‘t’ entspricht einer Zeit in Millisekunden und ‘c’ dem Zustand eines Feldes (0 für leer, 1 oder 2 für den jeweiligen Spieler).

Weitere Informationen können bei AI Games nachgeschlagen werden:

http://theaigames.com/competitions/four-in-a-row

4.3.1 Beispiel einer Kommunikation zwischen Engine und Bot

Als aller erstes wird das Spiel initialisiert. Dazu sendet die Game-Engine folgende Befehle:

```
settings timebank 10000
settings time_per_move 500
settings player_names player1,player2
settings your_bot player1
settings your_botid 1
settings field_columns 7
settings field_rows 6
```

Anschliessend erhält der Bot die Information über die aktuelle Runde und wie das Spielfeld aussieht. Weiter wird er aufgefordert einen eigenen Zug zu berechnen. Diese Zeilen erhält er jedes mal, wenn ein neuer Spielzug gefordert wird.

```
update game round 1
update game field 0,0,0,0,0,0,0;0,0,0,0,0,0,0;0,0,0,0,0,0,0;0,0,0,0,0,0,0;0,0,0,0,0,0,0;0,0,0,0,0,0,0;0,0,0,0,0,0,0
action move 10000
```

Anschliessend hat der Bot einen möglichen Zug berechnet und gibt die gewünschte Spalte, in welche eine Scheibe platziert werden soll, wie folgt aus:

```
place_disc 0 // places a disc in the first column from the left
```

Nachdem der Bot einen Zug bekannt gegeben hat, wird das Spielfeld aktualisiert. Folgend wird das Verhalten in Runde zwei dargestellt, in der der Gegner einen Zug macht. Bei genauer Betrachtung erkennt man, dass dieser eine Scheibe in der Spalte ganz rechts platziert hat.

```
update game field 0,0,0,0,0,0,0;0,0,0,0,0,0,0;0,0,0,0,0,0,0;0,0,0,0,0,0,0;0,0,0,0,0,0,0;0,0,0,0,0,0,0;1,0,0,0,0,0,0
update game round 2
update game field 0,0,0,0,0,0,0;0,0,0,0,0,0,0;0,0,0,0,0,0,0;0,0,0,0,0,0,0;0,0,0,0,0,0,0;0,0,0,0,0,0,0;1,0,0,0,0,0,2
```

5 Konzept

5.1 Klassendiagramm

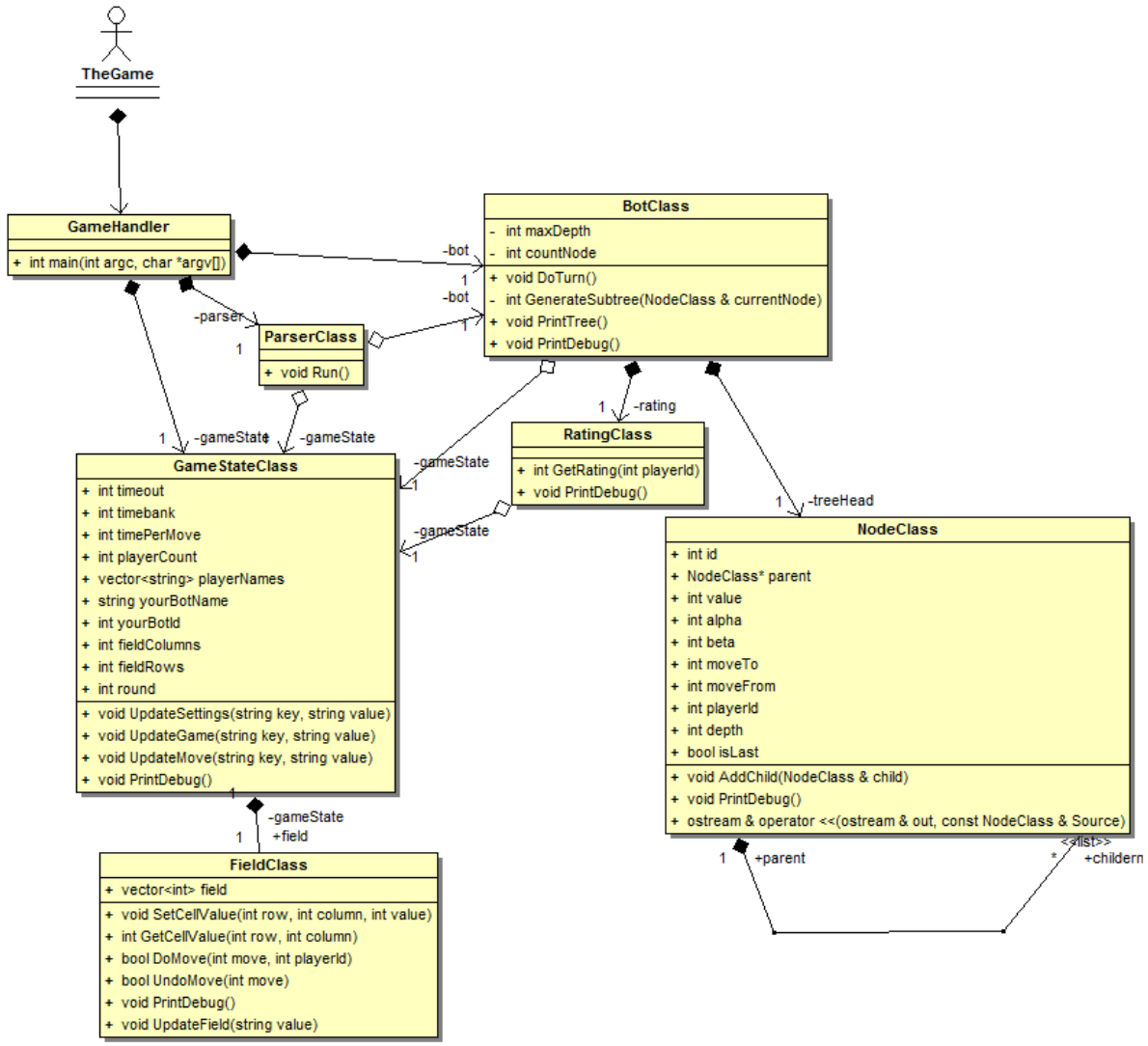


Abbildung 5-1: Klassendiagramm zum 4-Gewinnt Bot.

5.2 Klassenbeschreibung

5.2.1 GameHandler

Diese Klasse stellt das Main-Programm und baut den ganzen Bot auf. Die effektiv Endlosschlaufe befindet sich im Parser, der für die `std::in` und `std::out` Kanäle zuständig ist.

5.2.2 ParserClass

Diese Klasse bildet die Kommunikationsschnittstelle mit der Game-Engine. Sie verarbeitet sämtliche `std::in` Befehle und reicht diese entsprechend ihrer Klassierung (update, settings, action) weiter an die GameStateClass, um den Zustand des Spieles aktuell zu halten. Zusätzlich wird der Bot aufgerufen, falls der nächste Zug gefordert wird.

Weiter bietet die Parser-Klasse zusätzliche Befehle, damit Informationen über den aktuellen Spielzustand und auch den Spielbaum sowie hilfreiche Informationen der jeweiligen Klassen abgefragt werden können.

5.2.3 GameStateClass

Stellt für alle Klassen die Member für den aktuellen Spielzustand zur Verfügung. Diese können über Set-Funktionen jeweils gesetzt werden (Umwandlung der Strings in Integer-Werte). Das Abrufen kann direkt erfolgen.

5.2.4 FieldClass

Das Spielfeld im aktuellen Zustand, sowie nach jedem theoretischen Zug während den Berechnungen wird in dieser Klasse abgelegt.

5.2.5 BotClass

5.2.5.1 Beschreibung

Die eigentliche Intelligenz des Spiels wird in dieser Klasse zusammengefasst. Es könnten entsprechende Suchtechniken (MiniMax, Alpha-Beta-Puring, Deepsearch) umgesetzt werden, um den besten nächsten Zug für den jeweiligen Spieler zu generieren. Es wird eine Baumstruktur mit möglichen Zügen des Bots und den jeweiligen möglichen Gegenzügen des Gegners aufgebaut. Entsprechend der Bewertung eines Knotens wird der nächste Zug gemacht, mit dem Ziel den eigenen Sieg herbeizuführen und den des Gegners zu verhindern.

5.2.5.2 Spielbaum

Die Generierung des Spielbaums ist am einfachsten rekursiv. Aus der aktuellen Spielsituation werden alle möglichen Züge durchgeführt und bewertet. Je schneller der Aufbau des Baums ist, desto weiter kann der Bot in die Zukunft schauen und entsprechend das Spiel gestalten.

Der Bot wird über die Funktion `DoTurn()` aufgerufen und initialisiert eine neue Suche. Die Funktion `GenerateSubtree(NodeClass *currentNode)` prüft zuerst, ob der gewünschte Zug möglich ist (die Spalten werden von links nach rechts durchprobiert und dürfen noch nicht voll sein). Falls eine Platzierung möglich ist, entsteht eine neue Spielfeldsituation, die mittels der NodeClass beim Spielbaum hinzugefügt wird. Anschliessend ruft sich die Funktion selber wieder auf und übergibt den neu generierten Knoten als aktuelle Spielsituation.

Beim Zurückgehen in der Baumstruktur muss jeder einzelne Zug in der umgekehrten Reihenfolge wieder rückgängig gemacht werden, damit das Spielfeld wieder der aktuellen Situation entspricht.

5.2.6 RatingClass

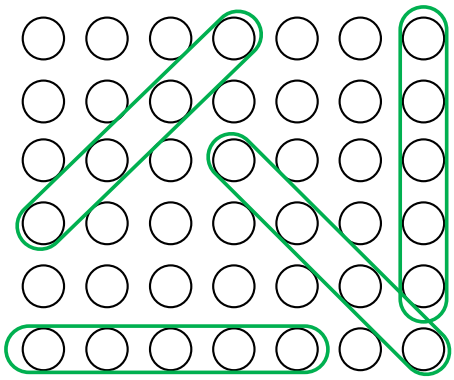


Abbildung 5-2: Gewinnsituationen für die Bewertungsfunktion.

Anhand der Bewertung eines Knotens, entscheidet der Bot ob dieser Zug interessant ist oder nicht (ob er damit gewinnt). Je höher die Bewertung ist, desto besser ist der Zug in Beziehung zu allen anderen möglichen Zügen.

Die Abbildung 5-2 zeigt alle möglichen Situationen auf, bei denen man in 4-Gewinnt das Spiel für sich entscheiden kann. Die Bewertungsfunktion muss diese 4 Typen auf dem kompletten Spielfeld schnell und sicher für beide Spieler erfassen können.

5.2.7 NodeClass

Diese Klasse stellt für die Bot-Klasse ein Knoten des Baumes zur Verfügung. Anhand der Knoten Bewertungen wird eine Gewinnstrategie gesucht. Zusätzlich kann mit den Alpha- und Beta-Schranken ein Alpha-Beta-Pruning umgesetzt werden, damit nicht die komplette Baumstruktur generiert werden muss. Die Verwendung von Hash-Schlüsseln könnte die Suche ebenfalls beschleunigen, indem gleiche Spielstellung nicht doppelt berechnet werden.

5.3 Anwendungsfälle

5.3.1 Übersicht

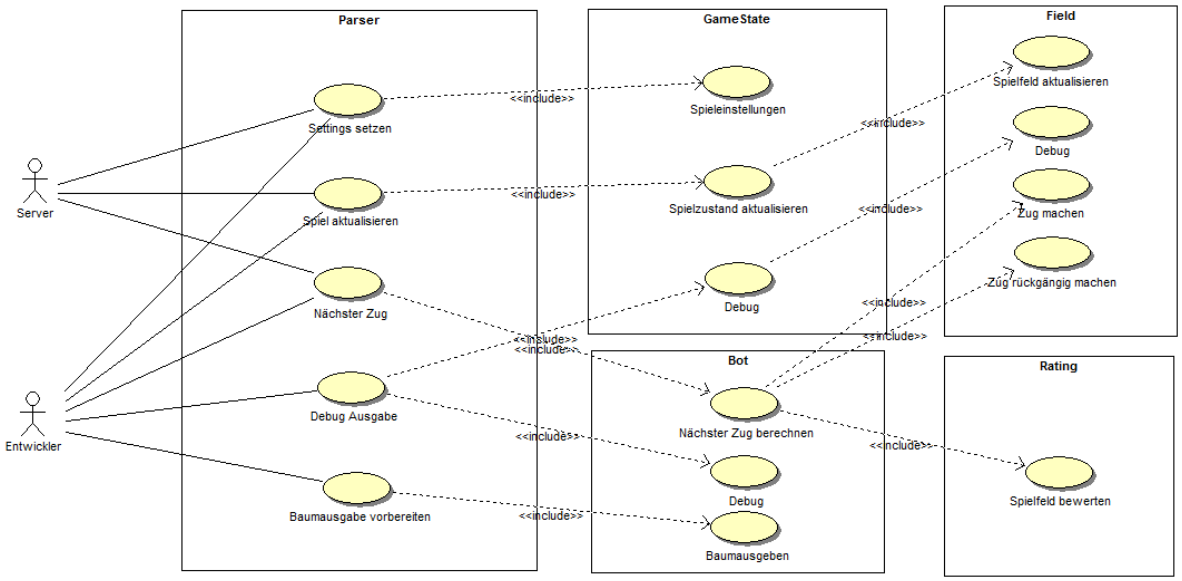


Abbildung 5-3: Use Case Diagramm des 4-Gewinnt Bots.

Weil das Use-Case Diagramm in Abbildung 5-3 nicht alle Kombinationen abdeckt, haben wir uns zusätzlich noch für eine Tabelle entschieden, die auch Falscheingaben und Zeithandling berücksichtigt. Dies vereinfacht ein mögliches Testen der kompletten Anwendung.

Tabelle 5-1: Mögliche Anwendungsfälle für den Spielbot.

	Bezeichnung	**Kurzbeschreibung**
1	Settings setzen	Einstellungen bei Spielbeginn korrekt übernehmen.
2	Spiel aktualisieren	Runde + Feld korrekt aktualisieren, nachdem ein Zug gemacht wurde.
3	Innerhalb Zeitlimit nächster Zug	Damit der Bot nicht «deaktiviert» wird und das Spiel verloren ist, muss der nächste Zug innerhalb seiner Zeitlimitte gemacht werden.
4	Idealer Startzug (bei Spieleröffnung)	Der beste Zug bei 4-Gewinnt ist als eröffnender Spieler in die mittlere Spalte zu setzen. Als zweiter Spieler, wenn diese Position besetzt ist, gleich daneben.
5	Sofortiger Sieg des Gegners verhindern	Sind 3 Steine des Gegners in einer Reihe, eine freie Position als 4. Stein und der Gegner ist nach dem Bot am Zug, so muss diese Reihe verhindert werden.
6	Sofortiger Sieg ausnutzen	Wie bei 5, nur für eigenen Spieler
7	Annahme der korrekten Timebank	Nach jedem Zug wird dem Bot eine aktualisierte Rechnungszeit für den nächsten Zug zur Verfügung gestellt. Innerhalb dieser Zeit

		muss der Bot seinen Zug machen, ansonsten geht das Spiel verloren.
8	Falscheingaben	Eingaben verwerfen
9	Debug-Anfrage	Komplette Debugausgabe aller instanziierter Objekte
10	Zeitabfrage	Zeitverbrauch der letzten Berechnung ausgeben
11	Baumstruktur ausgeben	Die aktuelle Baumstruktur im Graphviz-Format abspeichern
12	Zeitüberschreitung	Korrekter Ablauf (Verwarnung und anschliessend sofortiges verlieren)
13	Fehler im Algorithmus	„no_moves“ Ausgabe

5.3.2 Beschreibung Fall 5

5.3.2.1 Beschreibung

Wie in jedem Spiel gibt es Situationen, bei denen der Gegner als nächster Spieler (aktueller ist der Bot) das Spiel für sich entscheiden kann. Dieser Fall ist als sehr wichtig einzustufen, da das Ziel immer der eigene Sieg ist. Und «künstliche Intelligenz» sollte auch Intelligenz besitzen und solche offensichtlichen Positionen gezielt ausnutzen oder verhindern.

5.3.2.2 Vorbedingung

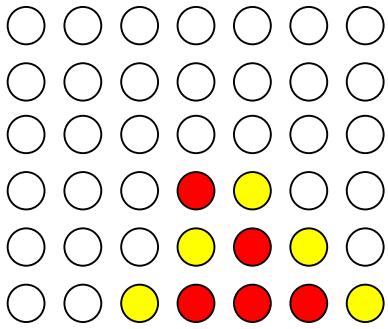


Abbildung 5-4: Ausgangsstellung für einen möglichen Sieg des Gegners.

Bei der Ausgangsstellung ist der aktuelle Spieler der Bot mit der Farbe Rot. Der Gegner (Farbe Gelb) hat in der nächsten Runde die Möglichkeit mit nur einem Stein in der 4. Spalte das Spiel für sich zu entscheiden. Diese Konstellation der Spielsteine ist stellvertretend für alle anderen Möglichkeiten, bei der der Gegner schon 3 Steine in einer möglichen 4er Reihe hat und im nächsten Zug den finalen Stein platzieren kann.

5.3.2.3 Beschreibung des Ablaufs

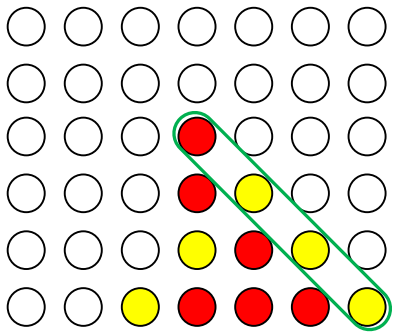


Abbildung 5-5: Identifizieren einer gegnerischen Gewinnmöglichkeit und verhindern von dieser.

Um diese Gewinnmöglichkeit des Gegners zu verhindern, muss der Bot zwingend in die 4. Spalte einen Roten Stein platzieren.

Beim Vorgehen um die Gewinnmöglichkeit zu verhindern, wird wie für jeden Zug ein neuer Spielbaum erstellt. In einem der Pfade von Zügen und gegnerischen Zügen muss der Bot den Gewinnzug des Gegners erkennen und alle anderen Optionen verwerfen (ausser der Bot kann selber einen finalen Stein platzieren).

E1) Die Game-Engine gibt die aktuelle Runde bekannt
A1) Runde in Spielzustand aktualisieren
E2) Die Game-Engine aktualisiert das Spielfeld
A2) Spielfeld in Spielzustand aktualisieren
E3) Die Game-Engine fordert den nächsten Zug
A3) Generierung der Baumstruktur mit allen möglichen Spielsituationen und den dazu führenden Zügen
A3a) Der Bot erkennt die Gewinn-Situation und verhindert diese entsprechend
A3b) Der Bot generiert einen falschen Zug

5.3.2.4 Auswirkungen

Grundsätzlich gibt es 2 Auswirkungen:

- Der Bot erkennt solche Situationen korrekt und verhindert diese. Das Spiel geht weiter und kann vielleicht gewonnen werden.
- Bei den Berechnungen für den nächsten Zug übersieht der Bot diese Gewinnchance des Gegners. Im Worst-Case Szenario war das der letzte Zug des Bots und das Spiel geht verloren. Falls der Gegner diese Gewinnchance nicht nutzt (was sehr unwahrscheinlich ist) geht das Spiel normal weiter und spätestens beim nächsten Zug des Bots muss diese Möglichkeit des Gegners verhindert werden.

5.3.2.5 Weitere Informationen

Der Unterschied zwischen Fall 5 und Fall 6 ist nur der Spieler. Beide Situationen lassen sich voneinander ableiten, indem man den aktuellen Spieler tauscht. Entsprechend sollte jeweils auch das korrekte Resultat aus den Zugberechnungen herauskommen.

Die Abbildung 5-6 zeigt eine Spielaufstellung, wie sie eigentlich nie sein sollte. Der Bot ist Spieler Rot und der Gegner Spieler Gelb. Im letzten Zug hat der Gegner den markierten Stein in Spalte 5 gesetzt, obwohl er eine Gewinnmöglichkeit in Spalte 2 gehabt hätte. Diese wurde fälschlicherweise vom Bot nicht verhindert. In dieser Situation kann der Bot irgendwo einen Stein setzten, sofern der Gegner nicht den gleichen Fehler ein 2. mal macht hat er die Partie im nächsten Zug für sich entschieden. Die beschriebene Situation entsteht nur, sofern beide Spieler einen falschen Spielzug gemacht haben. Eine etwas realistischere Betrachtung ist die Bildung einer Situation, wie sie im Sprachgebrauch „figge Mühle“ genannt wird. Eine mögliche Situation ist, wenn der gleiche Spieler ein Stein, dann eine leere Position, 2 Steine, eine leere Position und wieder ein Stein auf der gleichen Reihe hat. Hier spielt es keine Rolle mehr, welche Möglichkeit verhindert wird, weil es immer noch die andere gibt.

5.3.3 Beschreibung Fall 7

5.3.3.1 Beschreibung

Bei jedem Wettbewerb mit künstlicher Intelligenz und Computerprogrammen werden den Programmen gewisse Zeiten zur Verfügung gestellt, um ihre Berechnungen durchzuführen. Es geht darum herauszufinden, welche Programme ihre Aufgabe am effizientesten erledigen.

Bei diesem Wettbewerb erhält jeder Spieler am Anfang ein Zeitguthaben. Wird dieses Guthaben nicht komplett aufgebracht, wird es für den nächsten Zug zusätzlich zur Verfügung gestellt. Am Anfang des Spiels sind noch sehr viele Möglichkeiten für einen Sieg offen. Deswegen lohnt es sich nicht (auch zeitlich gesehen), das Spiel komplett durchzurechnen. Je weniger freie Positionen es auf dem Spielfeld hat, desto weniger Möglichkeiten müssen durchgerechnet werden, wodurch der Algorithmus weniger Zeit benötigt.

5.3.3.2 Voraussetzungen

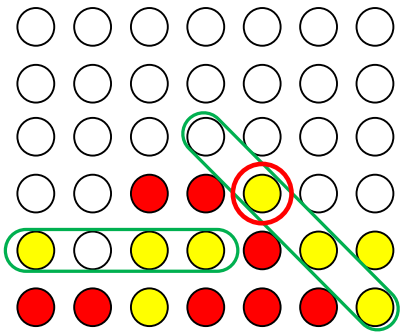


Abbildung 5-6: Zugzwang des Bots in einer verlorenen Partie.

Der Bot erhält vom Server jeweils über die `std::in` seine Zeit, die er für den nächsten Zug zur Verfügung hat.

5.3.3.3 Beschreibung des Ablaufs

E1) Bei jedem neuen Zug des Bots bekommt dieser neue Settings:

```
update game field 0,0,0,0,0,0,0;0,0,0,0,0,0,0;0,0,0,0,0,0,0;0,0,1,2,0,0,0;1,0,1,1,2,0,0;2,2,2,1,1,0,2
action move 10000
```

Obige Zeilen zeigen, dass der Bot für seinen nächsten Zug 10 Sekunden Zeit hat.

A1a) Innerhalb dieser Zeit muss nun der Bot einen Zug berechnen.
A1b) Geschieht dies nicht, bekommt er eine Verwarnung. Der Bot wird ein zweites Mal dazu aufgefordert einen Zug zu berechnen mit:

```
update game field 0,0,0,0,0,0,0;0,0,0,0,0,0,0;0,0,0,0,0,0,0;0,0,1,2,0,0,0;1,0,1,1,2,0,0;2,2,2,1,1,0,2
action move 500
```

E2) Falls danach kein Zug berechnet wurde, wird der Bot deaktiviert und das Spiel geht verloren.

Damit das Spiel erfolgreich beendet werden kann, müssen die Berechnungen innerhalb des Zeitguthabens erfolgen. Falls dies nicht geschieht, kann die Bewertung oder die Suchfunktion noch so gut sein. Im Falle eines Berechnungsfehlers kann der Bot mit `"no_moves"` reagieren um eine Deaktivierung zu verhindern.

5.3.3.4 Weitere Informationen

Genauere Informationen sind der Seite von AI Games zu entnehmen.

5.4 Sequenzdiagramm

Als einfachstes Beispiel für den Ablauf kann das Aktualisieren der Spieleinstellungen genommen werden. Die Game-Engine generiert eine Ausgabe, die vom Parser entsprechend aufgeschlüsselt wird (ob es sich um Settings, Update, Action oder um Debug-Anfragen handelt). Dementsprechend werden die korrekten Funktionen aufgerufen.

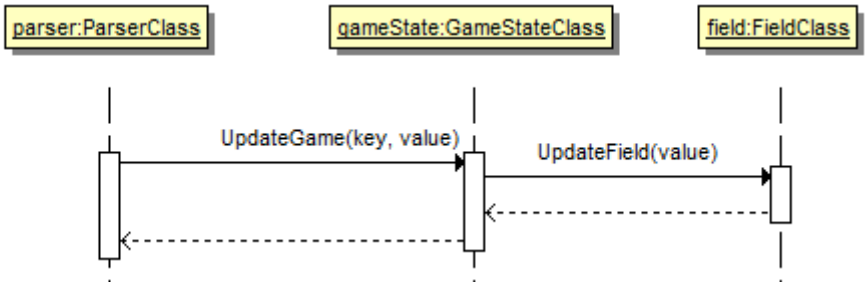


Abbildung 5-7: Sequenzdiagramm beim Aktualisieren der Spieleinstellungen.

Die Abbildung 5-7 zeigt die synchronen Aufrufe der entsprechenden Funktionen. Das komplette Design ist nicht multi-Threading fähig, weshalb die Abläufe nicht parallel geschehen. Dies wurde bewusst so umgesetzt da die Regeln von AI Games multi-Threading verbieten.

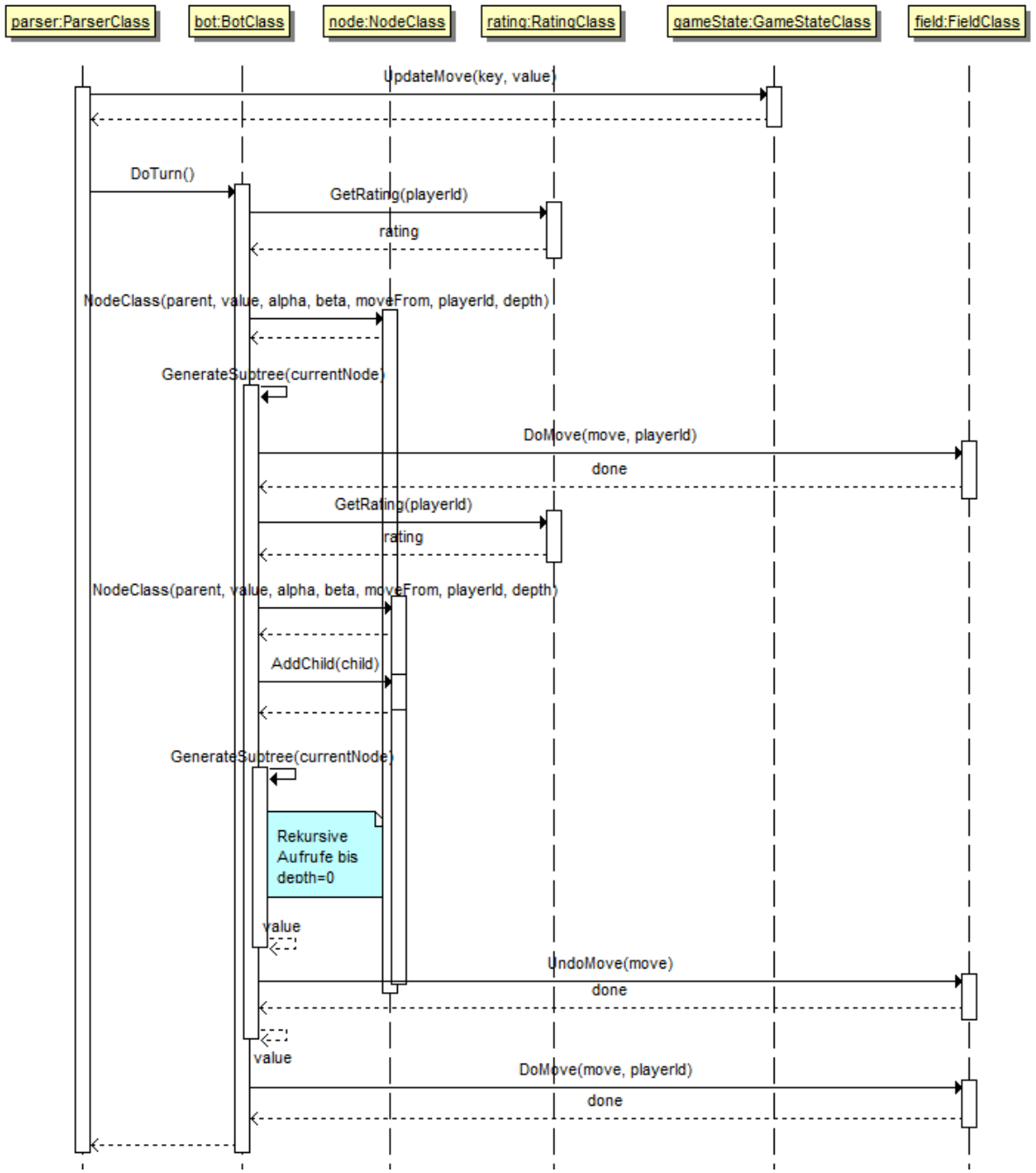


Abbildung 5-8: Sequenzdiagramm beim Berechnen eines neuen Zuges.

Das eigentliche Herz ist die Berechnung des nächsten Zuges. Die Game-Engine verlangt über die Ausgabe `action move 10000` einen neuen Spielzug des Bots innerhalb der nächsten 10 Skeunden. Anschliessend wird die Timebank in der GameState-Klasse aktualisiert und die Berechnung beginnt mit dem initialen Aufruf von `DoTurn()`. Als erstes wird überprüft, ob es sich um die ersten beiden Züge handelt. Falls dies der Fall ist, wird der Stein in optimaler Startposition (im Zentrum oder falls dies belegt ist, links daneben) platziert. Ansonsten wird bis zu einer vorgegebenen Tiefe der Spielbaum mit allen möglichen Kombinationen erstellt. Jede neue Spielsituation erhält eine entsprechende Bewertung, die im Spielbaum weiter nach oben gereicht wird. So kann der Bot seinen optimalen Spielzug ermitteln.

Damit die Übersicht im Sequenzdiagramm nicht verloren geht, zeigt die Abbildung 5-8 nur die initialen Funktionsaufrufe bis zum Beginn der Rekursion. Innerhalb der Funktion `GenerateSubtree(NodeClass *currentNode)` werden jeweils die gleichen Funktionen zur Erzeugung eines neuen Knotens sowie dessen Bewertung aufgerufen.

6 Umsetzung

6.1 Alpha Beta Pruning

Das Alpha Beta Pruning ist eine optimierte Version des MiniMax-Verfahrens. Bei beiden Suchalgorithmen werden die beiden Spieler in einen maximierenden und einen minimierenden Spieler unterteilt. Das MiniMax Verfahren rechnet den kompletten Spielbaum durch, ohne irgendwelche Optimierungen. Beim Alpha-Beta Pruning werden für den Spieler uninteressante Pfade ignoriert, weil der Algorithmus zwischen schlechter und noch schlechter nicht unterscheidet.

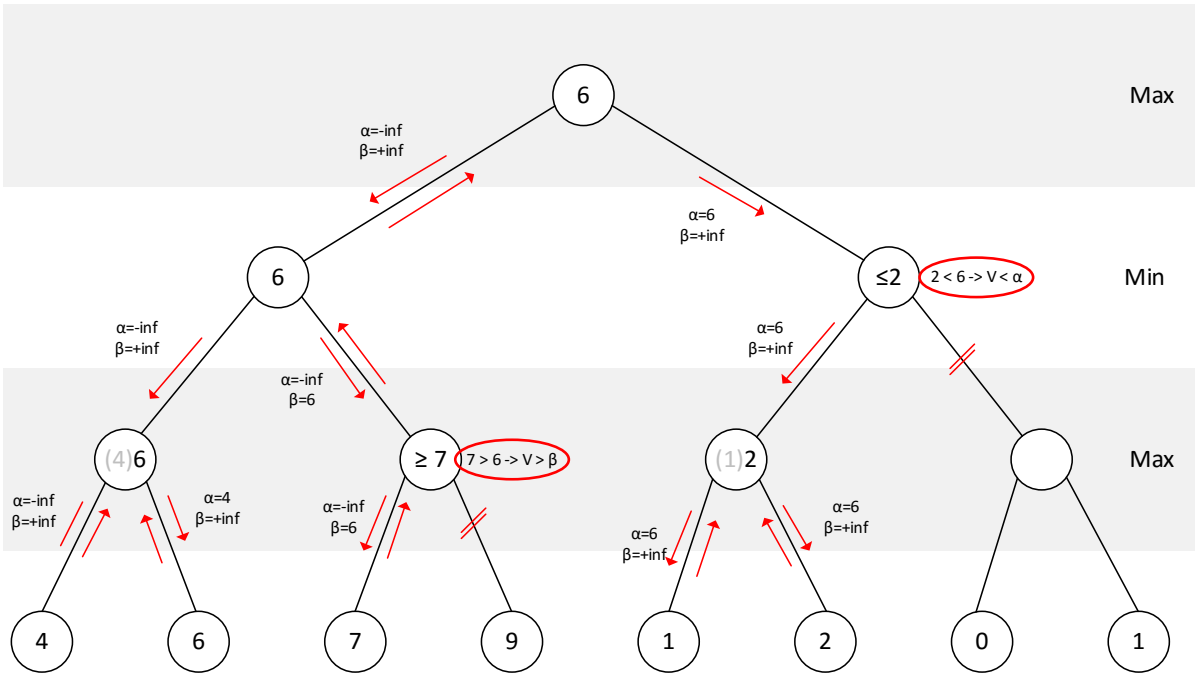


Abbildung 6-1: Suchbaum mit Alpha-Beta Pruning Verfahren.

Zum besseren Verständnis die grundlegenden Definitionen:

- **Alpha** ist die beste bekannte Option auf dem Pfad zur Wurzel für den Maximierer.
- **Beta** ist die beste bekannte Option auf dem Pfad zur Wurzel für den Minimierer.

Die Abbildung 6-1 zeigt einen möglichen Suchbaum, der von links nach rechts durchlaufen wird (die Richtung kann grossen Einfluss auf den Optimierungsgrad haben). Es sind 2 Optimierungen gezeigt, die durch den Algorithmus gar nicht durchlaufen werden müssen.

1) Bei der ersten Option hat der Maximierer den Wert 7 erhalten. Für den Minimierer wurde beim vorherigen Ast die Option 6 erarbeitet. Da wir uns nun in einem maximierenden Knoten befinden, werden keine Werte kleiner als 7 zugelassen. Der Minimierer eine Stufe weiter oben wird wiederum nie die Option ≥7 zulassen, da diese schlechter als der Wert 6 ist. Ein weiteres Durchsuchen dieses Astes ist also nicht notwendig.
2) Bei der zweiten Option befinden wir uns in einem minimierenden Knoten. Die beste bereits bekannte Option für den Maximierer ist 6. Es werden keine Werte ≥2 durch den Minimierer zugelassen. Für den Maximierer ist 2 eine Option, die er nicht wählen wird. Deshalb kann die Suche hier abgebrochen werden.

In der Umsetzung des Verfahrens wurde jedoch eine andere Darstellungsweise verwendet, die sogenannte NegaMax-Variante. Diese verwendet für beide Spieler die gleiche Bewertungsfunktion mit positiver Rückgabe, falls die Bewertung für den momentanen Spieler gut ist. Um das gleiche Verhalten trotz immer positiver Bewertung zu erhalten, wird bei der Übergabe von einem Knoten zum nächsten sowohl das Resultat invertiert sowie die Alpha- und Beta-Schranken vertauscht und invertiert.

6.2 Tiefensuche

Weiter wurde eine Tiefensuche verwendet, da diese den Vorteil mit sich bringt, dass wenn man bei jedem Knoten die Veränderung zum Vorgänger zwischenspeichert, nur eine Spielfeldversion gespeichert werden muss. Beim Durchlaufen des Baumes wird nach und nach zuerst Züge mit `DoMove(int move, int playerId)` durchgeführt und anschliessend nach der Bewertung durch die Rating Klasse wieder mit `UndoMove(int move)` rückgängig gemacht. Somit reicht ein Objekt Field aus um den gesamten Spielbaum zu erzeugen und es muss kein Speicher kopiert werden. Aus Debugging-Gründen werden jeweils alle Knoten gespeichert. Dies könnte weggelassen werden, solange keine iterierende Tiefensuche verwendet wird, da nachdem ein Knoten besucht wurde, dessen gesamten genutzten Daten (hier der evaluierte Wert) nach oben weitergegeben und somit nicht mehr verwendet wird.

Die Tiefe des Suchdurchlaufs ist entscheidend für den Spielerfolg. Wird zu weit gerechnet verliert der Algorithmus wichtige Zeit. Wird in einer speziellen Spielstellung zu wenig weit gerechnet, ist das Ergebnis verfälscht. Im Schach kann während einem Schlagabtausch nicht einfach aufgehört werden mit Rechnen, weil der Ausgang dieses Schlagabtauschs nicht bekannt ist und somit das korrekte Resultat des Spielzuges fehlt.

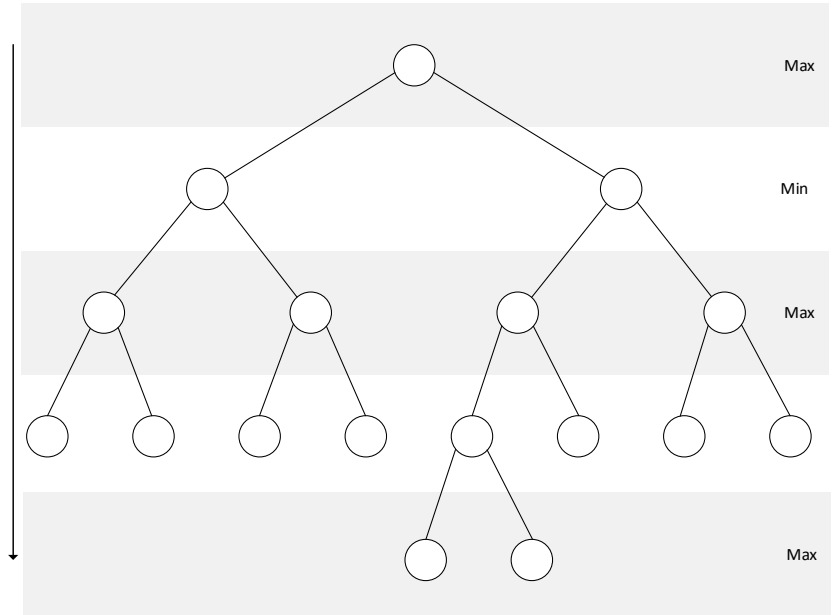


Abbildung 6-2: Adaptive Tiefensuche.

Weil im 4-Gewinnt keine Spielfiguren mit unterschiedlicher Gewichtung existieren, entstehen Spielsituationen mit Schlagabtausch nicht. Trotzdem ist die Suchtiefe ein wichtiges Kriterium. Weil das Spielfeld am Anfang mit sehr wenig Steinen belegt ist, sind sehr viele Züge möglich. Die Rechenzeit wird von der Game Engine und dem Server zur Verfügung gestellt. In dieser Zeit ist es am Spielanfang nicht

möglich das komplette Spiel bis zum Sieg durchzurechnen. Je weniger freie Felder es auf dem Spielfeld hat, desto schneller erreicht die Suchfunktion eine gegebene Tiefe, da es weniger mögliche Züge pro Zug und Spieler gibt. Dementsprechend müsste die Suchtiefe variabel sein um eine konstanten Zeitverbrach für die Berechnung zu haben. Um die Suchtiefe der verfügbaren Zeit anpassen zu können würde sich eine iterierende Tiefensuche eignen. Diese würde den weiteren Vorteil bringen, die Resultate der Berechnungen des Vorangegangenen Zuges weiter verwenden zu können. Aus zeitlichen Gründen wurde diese Variante nur in Code für den Spieltheorie Unterricht umgesetzt. Die als C++ Projekt Dokumentierte und abgegebene Variante beschränkt sich auf eine fixe Tiefe welche im `main.cpp` beim Konstruktor-Aufruf von der BotClass übergeben werden kann.

6.3 Iterative Tiefensuche

Um eine iterative Tiefensuche umsetzen zu können muss die Baumstruktur im Gegensatz zur normalen Tiefensuche abgespeichert werden. Da bei der Konzeptionierung des ersten Bots bereits an eine mögliche Erweiterung gedacht wurde, konnte die bereits umgesetzte Baumstruktur mittels den NodeClass Objekten weiterverwendet werden. In einem ersten Schritt wird auf das Löschen des Baumes bei einer neuen Zugberechnung verzichtet. Anstelle des Löschens tritt eine neue Funktion «CleanUpTreeAfterMove(int move)». Diese Funktion ermöglicht es anhand eines Zuges den Baum entsprechend zu verkleinern. So werden die nun nicht mehr gültigen Äste weggeschnitten und es wird eine neue Wurzel gesetzt.

Mit dem bereinigten Baum kann dann weiter gerechnet werden. In der Funktion «GenerateSubtree(NodeClass *currentNode, int endDepth)» wurden dazu folgende Änderungen vorgenommen. Falls der momentane Knoten bereits Kinder hat steht im Attribut «->moveTo» der beste Zug aus der vorangegangenen Iteration. Dieser Knoten wird nun als erster besucht. Dadurch wird erreicht, dass Alpha Beta Pruning zu möglichst vielen Cutoffs führt. Weiter muss für bereits besuchte Knoten keine Bewertungsfunktion mehr aufgerufen werden, da bereits bekannt ist, ob diese Situation eine Endstellung ist oder nicht (wird im Attribut «->isLast» festgehalten).

Um die Berechnung weiter zu optimieren wurde eine neue schnellere Bewertungsfunktion entwickelt, welche es ermöglicht aus dem letzten gesetzten Stein zu errechnen, ob dieser zum Spielende geführt hat. Diese Funktion kommt immer dann zum Zug, wenn ein neuer Knoten erzeugt wurde welcher sich nicht auf der Endtiefe befindet. Erst bei einem Blattknoten wird die richtige Bewertungsfunktion verwendet.

Die iterative Tiefensuche ermöglicht es neu die zur Verfügung stehende Zeit optimal zu nutzen. So wird von der Zeitreserve von 10sec zu Beginn des Spieles in jeder Runde 70% verwendet um weiter rechnen zu können als in der Zeit pro Zug von 0.5 sec. Mit diesem Wert wird die Reserve in etwa dann fertig Verbraucht, wenn gerade bis zum Ende des Spieles durchgerechnet werden kann (in etwa in Runde 23). Von da an wird nur noch Zeit gebraucht zum Rechnen, wenn der Gegner einen aus unserer Sicht unvorteilhaften Zug gemacht hat. Durch die gewählte Baumstruktur kann nicht nur nach jeder Iteration abgebrochen werden, sondern auch mitten in einer Iteration. Da der Beste Pfad aus der vorangegangenen Iteration zuerst berechnet wird, kann dieser Wert beim Erreichen der Zeitbegrenzung zur Wurzel propagiert werden, ohne ein falsches Resultat zu erhalten. So kann die Zeitplanung auf wenige ms genau eingehalten werden ohne Verlust von Daten.

Um die neue iterative Tiefensuche besser zu verstehen wurde eine neue Debugging Funktion «PrintPath()» hinzugefügt. Diese gibt nach jeder Iteration den Pfad aus, welcher gerade als bester Pfad gelten würde. Dadurch kann beobachtet werden, ab welchen Berechnungstiefe sich die Entscheidungen anfangen zu stabilisieren. Der Gewinn in der Performance durch die Erneuerungen ist beträchtlich. Aus der zuvor erreichten maximalen Tiefe von 7 Halbzügen wurde so 11 bis 13. Die Tiefe steigt mit zunehmendem Spielfortschritt, da die Möglichkeiten pro Zug immer wie mehr abnehmen. Durchschnittlich kann in Runde 23 von 42 bis zum Ende durchgerechnet werden.

7 Debugging Erweiterungen

Um während der Entwicklung der Algorithmen besser debuggen zu können wurden einige zusätzliche Terminal Befehle entworfen:

`time`	Gibt die verbrauchte Zeit für das Berechnen des letzten Zuges aus.
`debug`	Ruft alle `Debug()` Methoden aller instanziierten Objekte auf, welche alle ihre Attribute in das Terminal ausgeben.
`tree`	Erstellt eine Datei namens treeView.gv im gleichen Verzeichnis mit der momentanen Baumstruktur im Graphviz Format.

Die Ausgabe der Baumstruktur beinhaltet für jeden Knoten sowohl dessen Alpha-, Beta- und Bewertungswert, sowie die möglichen Folgezüge in Form einer dazugehörigen Zugspatenzahl auf dem Pfeil zum Nachfolgeknoten. Dabei werden Knoten des Max Spielers (also des Bots) in Grün dargestellt und jene des Gegners in Schwarz. Weiter wird der Pfad welcher als Idealer Spielverlauf evaluiert wurde mit Roten Pfeilen gekennzeichnet. Abbildung 7-1 zeigt einen Baum mit reduzierter Suchtiefe. Man beachte, dass die Wurzel den Pfad mit der Bewertung 0 bevorzugt obwohl dies der kleinste Rückgabewert der Nachfolgeknoten ist. Dies stimmt so da jeder Konten beim NegaMax-Verfahren positiv Bewertet falls die Situation für ihn vorteilhaft ist. Somit ist für den Vorangegangenen Spieler der kleinste Wert des Nachfolgers nach der Invertierung der Bewertung wiederum das beste Resultat.

8 Inbetriebnahme des Codes

Um die Bots testen zu können muss er in QT Creator zuerst erstellt werden. Danach kann die Triforce_four.exe Datei ausgeführt werden. Als nächstes können verschiedenen Beispieleingaben der Spielengine simuliert werden. Um nicht alles von Hand schreiben zu müssen bei jedem Test befinden sich einige Beispiele in «main.cpp» welche durch Kopieren und Einfügen in das Terminal übertragen werden können. Bei der iterativen Tiefensuche ist zu beachten, dass die so kopierten Eingaben natürlich nicht die Ausgaben des Bots beachtet. Da die erreichte Tiefe von der Geschwindigkeit der Hardware abhängt ist davon auszugehen, dass wenn die Berechnungen auf einem anderen System durchgeführt werden andere Ausgaben entstehen. Der so generierte Baum wird also immer vollständig verworfen da die Aufzeichnung mit einem anderen Spielverlauf arbeitet welcher sich nicht anpasst. Daher ist es normal, dass der Bot bei jeder Iteration wieder auf dem obersten Level beginnen muss. Mit dem Befehl «debug» kann zum Schluss zum Beispiel geschaut werden wie viele Knoten der letzte generierte Baum hatte. Mit dem Befehl «tree» kann dieser Baum in der Datei «treeView.gv» abgespeichert werden um ihn zu betrachten. Achtung! Dateien von über 100 kBytes können auf gewisse Systeme «gvedit.exe» bereits überlasten. Darum befindet sich auch ein Beispiel eines kleineren Baumes in dieser Dokumentation auf Abbildung 7-1.

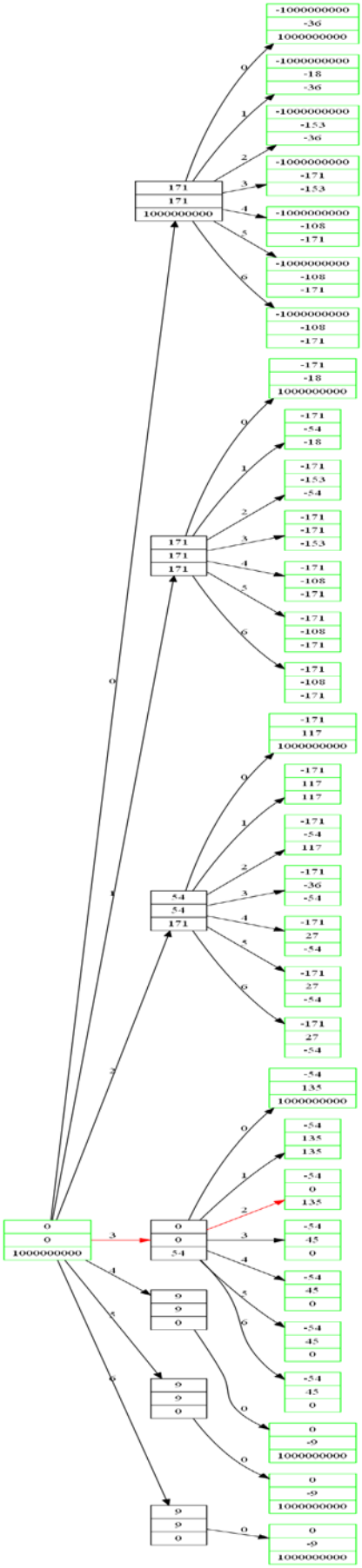


Abbildung 7-1: Baumstruktur erzeugt mittels Graphviz

9 Schlusswort

In dieser kombinierten Projektarbeit aus dem Modulen Spieltheorie und C++ in Embedded Systems haben wir erfolgreich ein Spielprogramm für 4-Gewinnt umgesetzt. Das Programm läuft auf dem Server von AI Games und belegt aktuell den Platz 31 (Stand 29. April 2016). Durch die Kombination des Wissens aus beiden Modulen und mit dem bisherigen Hintergrundwissen über Programmabläufe konnten die Algorithmen sehr effizient programmiert werden. Es konnten die Optimierungen von einer Tiefensuche, Tiefensuche mit Alpha Beta Pruning bis zu einer Iterativen Tiefensuche mit Alpha Beta Pruning systematisch vorgenommen werden. Zum lokalen Entwickeln und Debuggen des Programms wurden diverse Erweiterungen zur Ausgabe von Spielinformationen hinzugefügt, anhand welcher die Performance und das Verhalten überprüft werden konnten. Alles zusammen hat einen guten Einblick in die Welt der Spieltheorie gegeben und unser Wissen im Umgang mit C++ und Algorithmen verbessert.

Abbildungsverzeichnis