Welfenlab Competition 2006

Teilnehmer:	Jan Gosmann
Anschriftt:	Am Bokemahle 13c
	30171 Hannover
E-Mail:	blubb@hyper-world.de
Homepage:	http://www.hyper-world.de
Schule:	Bismarckschule Hannover
	An der Bismarckschule 5
	30173 Hannover
Klassenstufe:	13
Programmiersprache:	C++

Einleitung

Bei der Welfenlab Competition 2006 ging es um die Entwicklung einer künstlichen Intelligenz für das Brettspiel Backgammon sowie der Möglichkeit ein Backgammon-Spiel über eine Netzwerkverbindung austragen zu lassen.

Diese Ausarbeitung geht auf die Organisation des Projektes, die Bedienung des Programmes und die verwendeten Algorithmen ein. Dabei kann ich leider nicht auf jede Funktion ausführlich eingehen, da dies den Rahmen der Ausarbeitung sprengen würde. Behandeln werde ich nur die Algorithmen, die direkt mit der Aufgabenstellung zusammen hängen oder von besonderer Wichtigkeit für das Programm sind. Nicht ganz so komplexe Funktionen, die keiner Erläuterung benötigen sollten, werden teilweise auch nur erwähnt.

Zusätzlich zu dieser Ausarbeitung kann mit Doxygen eine Referenz-Dokumentation aus den Kommentaren im Sourcecode generiert werden (siehe Kapitel 1.3). Diese enthält dann zu den meisten Klassen, Member-Variablen und Funktionen eine kurze Beschreibung.

Inhaltsverzeichnis

Teil I.

Allgemeines

1. Projektorganisation

1.1. Ordnerstruktur

Das Wurzelverzeichnis enthält diverse Dateien mit Projekteinstellungen. Darunter befindet sich mit welfenlab_comp06.kdevelop auch eine Projetdatei für KDevelop (verwendete Version 3.3.1) und ein configure-Skript.

Die Unterordner ai-evolver und welfenlab_comp06 enthalten den Sourcecode für die gleichnamigen, zum Projekt zugehörigen Programme. In diesen beiden Ordnern befinden sich jeweils die Unterordner moc mit den von Qt generierten Meta-Object-Code sowie obj mit den kompilierten Objektdateien. Der Ordner welfenlab_comp06/gui enthält den Sourcecode für die grafische Benutzeroberfläche und die mit dem Qt Designer erstellten ui-Dateien. Die vom GUI verwendeten Grafiken befinden sich im Ordner welfenlab_comp06/gui/data.

Die von mir kompilierten Programmen befinden sich zusammen mit benötigten DLL- bzw. Shared-Object-Files im Ordner bin. Werden diese wie in Kapitel 1.2 beschrieben erneut kompiliert, so werden die Programmdateien in den Ordnern mit dem Sourcecode erstellt. Unter Windows liegen diese dort in einem weiteren Unterordner release und/oder debug.

Diese Ausarbeitung befindet sich in verschiedenen Formaten (LaTeX-Source, DVI, PS, PDF) im Ordner doc/ausarbeitung. Eine mit Doxygen (siehe Kapitel 1.3) erstellte Referenzdokumentation ist in den restlichen Unterverzeichnissen von doc nach Formaten sortiert abgelegt.

Im Ordner templates liegen Vorlagen für Sourcecode-Dateien.

1.2. Kompilieren des Programmes

1.2.1. Voraussetzungen

Zum Kompilieren des Projektes muss ein C++-Compiler sowie eine Installation von Qt[1] vorhanden sein. Die minimale Qt-Version ist 4.2. Ich selbst habe das Programm unter openSuse 10.1 mit g++ 4.1.0 kompiliert und getestet sowie unter Windows XP SP2 mit g++ 3.4.2 (mingw-special). Bei beiden Systemen habe ich Qt 4.2.2 verwendet.

1.2.2. Konfigurieren

Vor dem eigentlichen Kompilieren müssen die Umgebung konfiguriert und die Makefiles erstellt werden. Dies geschieht durch einen Aufruf von `qmake` oder unter Linux alternativ durch einen Aufruf des `configure`-Skriptes. Bei Verwendung des configure-Skriptes kann es nötig sein das Qt-Installationsverzeichnis mit dem Argument `--qtdir=[PATH]` anzugeben.

[1] Bezugsquelle: http://www.trolltech.com

1.2.3. Kompilieren

Durch einen Aufruf von `make` bzw. dem entsprechenden Kommando auf dem jeweiligen System (`gmake`, `mingw32-make`, ...) im Wurzelverzeichnis des Projekts werden sowohl ai-evolver als auch welfenlab_comp06 kompiliert. Um nur eines der beiden Programme zu kompilieren, muss `make` im entsprechenden Unterverzeichnis ausgeführt werden.

1.3. Erstellen der Doxygen-Dokumentation

Die Doxygen-Referenzdokumentation wird durch den Aufruf von `doxygen` im Wurzelverzeichnis des Projektes erstellt. Die Ausgabe erfolgt im Ordner `doc` in nach den Formaten benannten Unterordner. Doxygen kann unter http://www.doxygen.org bezogen werden. Die Einsendung zur Welfenlab Competition 2006 enthält bereits eine aktuelle und fertig generierte Doxygen-Dokumentation in den Formaten HTML, LaTeX, PDF (im Ordner `latex`) und XML.

1.4. Weitere Hinweise

Neben den Doxygen-Kommentaren, finden sich im Sourcecode Zeilen ähnlich zu dieser:

```
/*< \label{...} >*/
```

Diese dienen dazu die entsprechenden Zeilen in dieser Ausarbeitung zu referenzieren. In der Regel habe ich diese Kommentare nur an den nötigen Stellen eingefügt.

2. Bedienung

Dieses Kapitel geht lediglich auf die Bedienung von welfenlab_comp06 ein. Auf die Verwendung von ai-evolver wird im Kapitel 6.4.1 eingegangen.

2.1. Allgemein

Das Hauptfenster des Programmes ist mit sogenannten Dock-Widgets in mehrere Bereiche aufgeteilt, welches eine Anpassung an die eigenen Bedürfnisse und Vorlieben ermöglicht. Per Drag & Drop können die Dock-Widgets neu angeordnet und auch aus dem Hauptfenster ausgeklinkt werden. Letzteres kann alternativ auch über das Fenster-Symbol jeweils oben rechts geschehen. Über entsprechende Menüeinträge, Toolbar-Buttons und jeweils das X oben rechts können die Dock-Widgets ein- und ausgeblendet werden.

Spielfeld: Der zentrale Bereich des Programmfensters wird immer von dem Backgammon-Spielfeld eingenommen. Mit dem Menüeintrag *Ansicht/Spielbrett drehen* oder dem entsprechenden Button auf der Toolbar lässt sich das Spielfeld um 180° drehen. Züge werden per Drag & Drop eingegeben. Um einen Spielstein auszuwürfeln, ist dieser auf den linken oder rechten Bereich ohne „Zungen“ zu ziehen. Die Farben des Spielfeldes können über das Menü *Einstellungen* festgelegt werden.

Würfel: In diesem Dock-Widget wird das jeweils letzte Würfelergebnis angezeigt. Sofern kein automatisches Würfeln aktiviert ist, wird mit einem Klick auf die Würfel, den entsprechenden Button in der Toolbar oder durch Drücken von Ctrl + W gewürfelt. Auf die gleiche Weise wird ein lokales Spiel gestartet. Netzwerkspiele werden automatisch gestartet.

Zugliste: Die Zugliste listet alle Züge eines Spiels auf. Wird ein Spielstein geschlagen, so wird dies mit einem Sternchen (*) gekennzeichnet. Wird ein Zug mehrmals ausgeführt, so werden alle diese Züge zusammengefasst und die Anzahl in Klammern dahinter angezeigt.

Statistiken: In den Statistiken wird angezeigt, die wievielte Runde gerade gespielt wird und wie viele Punkte der weiße und schwarze Spieler jeweils erhalten haben.

Chat-Fenster: Wurde eine Verbindung zu einem Backgammon-Server aufgebaut, kann über das Chat-Fenster mit anderen angemeldeten Benutzer gechattet werden. Dazu wird der Benutzer, an den die Nachricht gehen soll, in der Combo-Box ausgewählt und die Nachricht in das Eingabefeld eingegeben. Mit einem Klick auf *Senden* oder durch Drücken von Enter wird die Nachricht gesendet. (Im Chat-Bereich des Dialogfensters für ein neues Spiel ist es nicht möglich die Enter-Taste zum Senden zu verwenden, da mit dieser das Fenster geschlossen und ein neues Spiel gestartet wird.)

Ein- und ausgehende Chat-Nachrichten werden im Chat-Fenster schwarz angezeigt. Servermeldungen sind blau, Fehlermeldungen rot und Änderungen des Verbindungsstatus werden grau dargestellt.

Statuszeile: In der Statuszeile werden verschiedene Informationen angezeigt. Im linken Teil erscheint der Grund, warum ein Zug nicht gültig ist, wenn versucht wird einen ungültigen Zug zu setzen. Weiterhin wird hier auch eine kurze Hilfe zu den Menüpunkten angezeigt oder dass ein Spieler zugunfähig ist, wenn blockierende Meldungen für diesen Spieler deaktiviert sind.

Der rechte Teil der Statusleiste zeigt den aktuellen Spieler und ob die KI gerade nach einem Zug sucht.

2.2. Neues Spiel oder neue Runde starten

Mit *Datei/Neues Spiel...* oder dem entsprechenden Toolbar-Button kann ein neues Spiel gestartet werden. Wenn gerade ein Netzwerkspiel läuft, wird dieses (nach Bestätigung einer entsprechenden Frage) abgebrochen. Somit kann dieses im Gegensatz zu einem lokalen Spiel nach einem Klick auf *Abbrechen* nicht fortgesetzt werden.

In dem Dialogfenster für ein neues Spiel kann festgelegt werden, welche Farbe von wem kontrolliert wird (*menschlicher Spieler*, *Computer*, *Netzwerkspieler*). *Netzwerkspieler* kann dabei nur ausgewählt werden, wenn einem Netzwerkspiel beigetreten wurde und dieser Wert kann nur für genau einen Spieler ausgewählt werden. Die KI kann mit einem Klick auf den Schraubenschlüssel konfiguriert werden. Der Timeoutwert der KI sollte bei Netzwerkspielen etwas niedriger als der Timeoutwert des entsprechenden Netzwerkspiels liegen, da das Verarbeiten und Senden eines Zuges der KI noch einen kurzen Moment nach Ablauf des Timeouts benötigen kann. Auf die Bedeutung der einzelnen Parameter für die KI wird in Kapitel 6.2 eingegangen.

Ist die Option *Blockierende Meldung anzeigen, wenn dieser Spieler zugunfähig ist* aktiv, so wird das Spiel pausiert, wenn der entsprechende Spieler zugunfähig ist, und eine Dialogbox angezeigt, die erst bestätigt werden muss. Andernfalls wird dies nur in der Statusleiste eingeblendet und das Spiel läuft sofort weiter. Normalerweise muss jeder Spieler erst durch einen Klick auf die Würfel oder den entsprechenden Toolbar-Button würfeln. Mit der Option *Automatisch Würfeln* wird für den entsprechenden Spieler automatisch gewürfelt. Für einen Netzwerkspieler wird immer automatisch gewürfelt.

Wird ein neues Spiel gestartet, werden die Statistiken zurückgesetzt. Stattdessen kann mit *Datei/Nächste Runde starten* oder durch den entsprechenden Toolbar-Button ein Spiel mit den gleichen Einstellungen gestartet werden, wobei die Statistiken nicht gelöscht werden. Mit dem Menüpunkt *Datei/Runden automatisch starten* wird nach Beendigung einer Runde direkt die nächste gestartet, ohne dass der Gewinner angezeigt wird. Bei einem Offline-Spiel wird dies so lange fortgesetzt, bis diese Option wieder deaktiviert wird. Bei einem Netzwerkspiel werden dagegen nur so lange neue Runden gestartet, bis die bei der Erstellung des Spiels angegebene Anzahl an Runden gespielt wurde.

2.3. Verbindung zu Netzwerkspiel herstellen

Um die Verbindung zu einem Netzwerkspiel herzustellen, muss zuerst die Verbindung zu einem Backgammon-Server hergestellt werden. Dies geschieht im Dialogfenster für ein neues Spiel unter dem Reiter *Netzwerkverbindung*. Für den Server kann entweder ein Hostname oder eine IP angegeben werden. Ist der Server nicht unter dem Standard-Port 30167 erreichbar, so kann ein alternativer Port mit einem Doppelpunkt (:) getrennt hinter dem Servernamen bzw. der IP angegeben werden. Die letzten zehn Server und Benutzernamen, die verwendet wurden, werden gespeichert und lassen sich direkt in den Combo-Boxes auswählen.

Nach dem Herstellen der Verbindung zu einem Backgammon-Server kann ein neues Spiel mit dem einem Klick auf den Button *Spiel erstellen...* angelegt werden. Um einem bestehenden Spiel beizutreten, muss dies in der Liste ausgewählt und dann auf *Spiel beitreten* geklickt werden. Tritt man einem Spiel bei, so ändert sich der Button *Spiel beitreten* in *Spiel verlassen*. Nach einem Klick auf *OK* startet ein Netzwerkspiel sofort im Gegensatz zu einem lokalen Spiel, welches erst durch einen Klick auf die Würfel gestartet werden muss.

Im unteren Teil des Reiters *Netzwerkverbindungen* befindet sich auch ein Chat-Fenster, welches sich von dem des Hauptfensters nicht unterscheidet. Allerdings sollte hier nicht die Enter-Taste zum Senden von Nachrichten verwendet werden, da diese das Dialogfenster bestätigen und schließen würde.

Es ist übrigens auch möglich eine Netzwerkverbindung herzustellen ohne anschließend einem Netzwerkspiel beizutreten. So kann man z. B. auch während eines lokalen Spiels mit anderen angemeldeten Benutzern auf dem Backgammon-Server chatten.

Hinweis: Leider hat der Backgammon-Server auf werefkin.gdv.uni-hannover.de zur Zeit (Stand: 24. Februar 2007) noch einige Bugs. Daher sollten bei Verwendung dieses Servers der Welfenlab Competition vorher die Informationen im Anhang B gelesen werden.

2.4. Programmargumente

Da das Programm Qt basiert ist, kennt es auch die durch Qt bereitgestellten Argumente (z. B. `-style`). Nähere Informationen dazu sind der Qt-Dokumentation zu entnehmen.

Das einzige weitere Programmargument, dass das Programm kennt ist `--netdbg`. Es sorgt dafür, dass sämtliche ein- und ausgehenden Netzwerknachrichten in die Standardfehler-Ausgabe geschrieben werden. Eingehende Nachrichten beginnen mit einem ‚>', ausgehende mit einem ‚<'.

Dem Programm nicht bekannte Argumente werden ignoriert.

Teil II.

Arbeitsweise und Implementation der Algorithmen

3. Klasse Backgammon

Die Klasse `Backgammon`[1] implementiert den Programmkern. Sie ist in der Lage den aktuellen Spielstand eines Backgammon-Spiels zu speichern und stellt Funktionen bereit, um Züge auszuführen. Die Verteilung der Spielsteine wird mit den Variablen `m_points`[2], `m_on_bar`[3] und `m_beared_off`[4] gespeichert. Dabei gibt `m_points` die Verteilung der Spielsteine an: Positive Werte sind Spielsteine des weißen, negative Werte Spielsteine des schwarzen Spielers. Weiß zieht in Richtung der höheren Indizes des Array, Schwarz dementsprechend entgegengesetzt. Weiterhin wird mit `m_on_bar` die Zahl der Steine auf der Bar und mit `m_beared_off` die Zahl der ausgewürfelten Spielsteine gespeichert. Die Werte dieser beiden Arrays sind immer positiv und das Element mit dem Index 0 ist jeweils die Angabe für den weißen Spieler, das Element mit dem Index 1 für den schwarzen.

Für eine bessere Lesbarkeit des Sourcecodes habe ich die Enumeration `Player`[5] definiert, welche die Elemente `WHITE`[6] und `BLACK`[7] bereitstellt. Weiterhin stellt die Enumeration `Position`[8] die Werte `BAR`[9] und `OUT_OF_GAME`[10] bereit, welche dazu dienen die Bar und die „Position“ für ausgewürfelte Spielsteine zu bezeichnen.

Weiterhin speichert die Klasse `Backgammon` in `m_act_player`[11] den aktuellen Spieler und im Array `m_dice`[12] die Augenzahlen, die maximal gesetzt werden können (unter Umständen kann nicht für alle gesetzt werden, da der Spieler vorher zugunfähig wird). Wenn nur noch für weniger als vier Augenzahlen gesetzt werden muss, wird den restlichen Array-Elementen der Wert 0 zugewiesen.

Für das Übergeben von Zügen an Funktionen wird die Klasse `BackgammonMove`[13] verwendet. Sie speichert die Startposition und Zugweite eines Zuges. Mehrere solche Züge können mit der Klasse `BackgammonTurn`[14] mit noch ein paar zusätzlichen Informationen zusammengefasst werden. Diese Klasse kommt bei der Speicherung der Zugliste zum Einsatz, auf die ich hier aber nicht näher eingehen werde.

Auch wenn sich eine genauere Erklärung erübrigen dürfte, sollten die Funktionen

[1] `class BG::Backgammon : public QObject;` welfenlab_comp06/backgammon.h:194
[2] `short int BG::Backgammon::m_points[ 24 ];` welfenlab_comp06/backgammon.h:337
[3] `short int BG::Backgammon::m_on_bar[ 2 ];` welfenlab_comp06/backgammon.h:349
[4] `short int BG::Backgammon::m_beared_off[ 2 ];` welfenlab_comp06/backgammon.h:346
[5] `enum BG::Player;` welfenlab_comp06/backgammon.h:43
[6] `BG::WHITE = 0;` welfenlab_comp06/backgammon.h:51
[7] `BG::BLACK = 1;` welfenlab_comp06/backgammon.h:52
[8] `enum BG::Position;` welfenlab_comp06/backgammon.h:87
[9] `BG::BAR = 24;` welfenlab_comp06/backgammon.h:95
[10] `BG::OUT_OF_GAME = 25;` welfenlab_comp06/backgammon.h:96
[11] `BG::Player BG::Backgammon::m_act_player;` welfenlab_comp06/backgammon.h:354
[12] `short int BG::Backgammon::m_dice[ 4 ];` welfenlab_comp06/backgammon.h:357
[13] `class BG::BackgammonMove;` welfenlab_comp06/backgammon.h:128
[14] `class BG::BackgammonTurn;` welfenlab_comp06/backgammon.h:157

`reset()`[15] und `calc_win_height()`[16] erwähnt werden. Erstere setzt die Klasse zurück und startet so ein neues Backgammon-Spiel, letztere berechnet nach Beendigung eines Spiels die Gewinnhöhe.

Im Folgenden werde ich auf die Funktionen eingehen, welche feststellen, ob ein Spieler zugunfähig ist, ein Zug gültig ist, oder einen Zug ausführen. In vielen dieser Funktionen und auch einigen Funktionen, auf die ich hier nicht näher eingehe, wird die Konstante `PLAYER_IND` definiert. Wenn der aktuelle Spieler weiß ist, ist der Wert 1, ansonsten -1. Dies ermöglicht es z. B. if-Abfragen kompakter zu formulieren, da durch eine Multiplikation mit dieser Konstante derselbe Vergleich von Werten für beide Spieler verwendet werden kann.

Oft wird in diesen Funktionen zudem die Funktion `is_able_to_move()`[17] aufgerufen. Sie prüft für einen übergebenen Zug, ob ein Spielstein vorhanden ist, der gezogen werden kann und das Zielfeld frei ist. Weiterhin kann der Funktion ein zweiter Zug übergeben werden, wobei die Funktion dann davon ausgeht, dass zuerst dieser Zug ausgeführt wird und dann prüft, ob der andere anschließend möglich ist. Ansonsten führt die Funktion aber keine weiteren Überprüfungen durch, ob der Zug gültig ist (beispielsweise wird nicht geprüft, ob von der Bar gezogen werden müsste).

3.1. Feststellen, ob ein Spieler zugunfähig ist

Die Funktion `is_valid_move_possible()`[18] prüft, ob der aktuelle Spieler einen gültigen Zug ausführen kann. Wenn dies der Fall ist gibt sie **`true`**, andernfalls **`false`**, zurück. Zudem ist sie in der Lage gegebenenfalls einen gültigen Zug zurückzugeben. Diese spezielle Funktionalität wird von der KI verwendet, um sicherzustellen, dass sie in jedem Fall einen gültigen Zug ausführt.

Das Setzen des Arrays `m_dice_result_has_to_be_used`[19] ist eine weitere wichtige Aufgabe der Funktion. Die beiden Elemente des Arrays liefern Informationen darüber, welche Augenzahlen gesetzt werden können. Ist `m_dice_result_has_to_be_used[ 0 ]` gleich **`true`** muss bzw. kann die höhere Augenzahl der beiden gewürfelten gesetzt werden. Wenn `m_dice_result_has_to_be_used[ 1 ]` wahr ist, muss die kleinere Augenzahl gesetzt werden. Ist einer dieser Werte **`false`** kann bzw. darf für die entsprechende Augenzahl nicht gesetzt werden. Bei einem Pasch sind beide Elemente wahr oder, sollte nur für eine Augenzahl gesetzt werden können, nur `m_dice_result_has_to_be_used[ 0 ]`. Die Werte dieses Arrays werden von allen Funktionen verwendet, die prüfen, ob ein Zug gültig ist.

Die Arbeitsweise der Funktion `is_valid_move_possible()` unterscheidet sich, je nachdem für wieviele Augenzahlen evtl. noch gesetzt werden kann. Wenn für keine Au-

[15] `void BG::Backgammon::reset( void );` welfenlab_comp06/backgammon.cpp:933

[16] `BG::WinHeight BG::Backgammon::calc_win_height( void );` welfenlab_comp06/backgammon.cpp:1322

[17] `bool BG::Backgammon::is_able_to_move( const BG::BackgammonMove *move, const BackgammonMove *before ) const;` welfenlab_comp06/backgammon.cpp:1210

[18] `bool BG::Backgammon::is_valid_move_possible( BG::BackgammonMove *move = NULL );` welfenlab_comp06/backgammon.cpp:499

[19] `bool BG::Backgammon::m_dice_result_has_to_be_used[ 2 ];` welfenlab_comp06/backgammon.h:367

genzahl mehr gesetzt werden kann, so kehrt die Funktion sofort mit **`false`** zurück. Wenn maximal nur noch für eine Augenzahl gesetzt werden kann, wird zuerst geprüft, ob der Spieler noch Steine auf der Bar hat und ob es möglich ist davon einen zurück ins Spiel zu bringen. Wenn der Spieler keine Steine mehr auf der Bar hat, werden sämtliche „Zungen“ durchgegangen und geprüft, ob von einer dieser „Zungen“ die Augenzahl gezogen werden kann.

Hat der Spieler für zwei oder mehr Augenzahlen noch nicht gesetzt, wird es etwas komplizierter. Sollten noch mindestens zwei Steine auf der Bar sein, reicht es für beide Augenzahlen zu prüfen, ob es möglich ist, damit einen Spielstein wieder in das Spiel zu bringen.

Ist höchstens ein Spielstein auf der Bar, wird zuerst für jede der beiden Augenzahlen eine „Zunge“ gesucht, von der diese Augenzahl gezogen werden kann. Dabei ist zu beachten, dass möglichst für beide Augenzahlen eine „Zunge“ gefunden werden sollte. Falls dies nicht möglich ist, sollte möglichst für die größere der beiden Augenzahlen eine „Zunge“ gefunden werden und, wenn das auch nicht geht, für die kleinere der beiden Augenzahlen. Dies habe ich so implementiert, dass alle Kombinationen von zwei „Zungen“ (a, b) durchgegangen werden (wobei $(a, b) \neq (b, a)$) und für jede dieser Kombinationen geprüft wird:

1. Wenn die höhere Augenzahl von Position a aus gesetzt werden kann...
   1.1. Wenn vorher die höhere Augenzahl von Position a gesetzt wurde und dann noch die niedrigere Augenzahl von Position b gesetzt werden kann, wurde eine Zugkombination gefunden, die die Bedingungen bestmöglich erfüllt.
   1.2. Wenn vorher die höhere Augenzahl von Position a gesetzt wurde, die niedrigere Augenzahl nicht mehr von Position b gesetzt werden kann und noch keine anderere Position a gefunden wurde, von der die höhere Augenzahl gesetzt werden kann, wurde eine Zugkombination gefunden, die die Bedingungen besser als die vorhergehenden Kombinationen erfüllt, aber noch nicht optimal ist.
2. Wenn die höhere Augenzahl von Position a aus nicht gesetzt werden kann, aber die niedrigere Augenzahl von Position b...
   2.1. Wenn die höhere Augenzahl von Position a gesetzt werden kann unter der Bedingung, dass die niedrigere Augenzahl von b vorher gesetzt wurde, so wurde eine Zugkombination gefunden, die die Bedingungen bestmöglich erfüllt.
   2.2. Wenn die höhere Augenzahl auch dann nicht von Position a gesetzt werden kann, wenn vorher die niedrigere Augenzahl von b gesetzt wurde, und noch keine Position gefunden wurde, von der irgendeine der Augenzahlen gesetzt werden kann, dann wurde eine Zugkombination gefunden, die die obigen Bedingungen minimal erfüllt.

Ist genau ein eigener Spielstein auf der Bar, wird erst geprüft, ob sich dieser mit der höheren Augenzahl ins Spiel zurückbringen lässt. Wenn sich dann noch die niedrigere Augenzahl von b oder der wieder ins Spiel gebrachter Spielstein anschließend um diese Augenzahl setzen lässt, müssen beiden Augenzahlen verwendet werden und die Funktion kehrt zurück. Kann die niedrigere Augenzahl anschließend nicht gesetzt werden, wird mithilfe der Variable `is_move_set` vermerkt, dass mit der höheren Augenzahl der Spielstein ins Spiel gebracht werden kann. Wenn dieser nun mit der niedrigeren Augenzahl zurück gebracht werden kann

und anschließend die höhere Augenzahl von b oder mit dem zurückgebrachten Spielstein gesetzt werden kann, müssen beide Augenzahlen verwendet werden und die Funktion kann zurückkehren. Ist dies nicht der Fall und ist `is_move_set` wahr, muss für die größere Augenzahl gesetzt werden und für die kleinere kann nicht gesetzt werden. Ist `is_move_set` dagegen falsch (aber der Spielstein von der Bar kann mit der kleineren Augenzahl wieder ins Spiel gebracht werden), muss für die kleinere Augenzahl gesetzt werden und die größere kann nicht verwendet werden. Kann der Spielstein weder mit der höheren noch mit der niedrigeren Augenzahl ins Spiel zurückgebracht werden, ist der Spieler zugunfähig.

Sind keine Steine mehr auf der Bar, wird geprüft, ob eine Position a gefunden wurde, von der die höhere Augenzahl gezogen werden kann. Wenn dann noch eine Position b für die niedrigere Augenzahl gefunden wurde, muss für beide Augenzahlen gesetzt werden, andernfalls nur für die höhere. Wurde keine Position a entsprechend der Bedingungen gefunden, aber eine Position b, muss für die kleinere Augenzahl gesetzt werden und für die größere ist dies nicht möglich.

Wenn jetzt noch kein gültiger Zug gefunden wurde, gibt die Funktion **`false`** zurück, da der Spieler zugunfähig ist.

3.2. Gültigkeit eines Zuges feststellen

Um zu prüfen, ob ein Zug gültig ist, wird die Funktion `is_valid_move()`[20] verwendet. Bei dem übergebenen Zug dürfen mehrere Züge zusammengefasst werden, wenn z. B. eine 1 sowie eine 4 gewürfelt wurden und von Position 3 über Position 4 zu Position 8 gezogen werden soll, kann ein Zug übergeben werden, dessen Startposition 3 und dessen Zugweite 5 ist. Die Funktion kann auch Informationen darüber zurückgeben, warum ein Zug nicht möglich ist oder welche Augenzahlen verwendet werden müssten, um den Zug auszuführen. In folgender Reihenfolge prüft die Funktion, ob der Zug gültig ist:

1. Ist die Zugrichtung korrekt (bzw. ist die Zugweite positiv?)
2. Sind noch Steine auf der Bar und wird in diesem Fall von der Bar gezogen?
3. Ist auf der „Zunge“, von der gezogen werden soll, überhaupt ein eigener Spielstein?
4. Wird versucht den Spielstein auszuwürfeln?
   - 4.1. Sind alle eigenen Spielsteine bereits im Homeboard?
   - 4.2. Wenn sich der Spielstein mit den gewürfelten Augenzahlen nicht passend auswürfeln lässt: Hat ein anderer Spielstein Vorrang?
5. Ist das Zielfeld noch frei (bzw. höchstens ein gegnerischer Spielstein auf diesem)?
6. Kann die Zugweite mit den gewürfelten Augenzahlen gesetzt werden?

Der letzte Punkt ist etwas komplexer. Als erstes wird geprüft, ob die erste gewürfelte Augenzahl der Zugweite entspricht. Wenn dies der Fall ist könnte der Zug mit dieser

[20] `bool BG::Backgammon::is_valid_move( const BG::BackgammonMove &move, BG::IllegalMove *reason = NULL, bool *dice_used = NULL, bool check_bar = true );`
welfenlab_comp06/backgammon.cpp:98

Augenzahl ausgeführt werden. Allerdings muss noch geprüft werden, ob diese Augenzahl überhaupt verwendet werden darf (siehe dazu auch Kapitel 3.1). Muss auch noch für die andere Augenzahl gesetzt werden, wird noch überprüft, ob dies nach dem Zug noch immer möglich ist. Falls die erste gewürfelte Augenzahl nicht der Zugweite entspricht, wird das gleiche Schema für die zweite Augenzahl durchlaufen, sofern diese der Zugweite entspricht.

Sollte die Funktion noch nicht zurückgekehrt sein, muss für den Zug auf jeden Fall eine Kombination mindestens zweier Augenzahlen verwendet werden. Sollte mehr als ein Stein auf der Bar sein, ist der Zug ungültig, da erst alle Spielsteine zurück ins Spiel gebracht werden müssen bevor von woanders als der Bar gezogen werden darf.

Danach werden die jeweiligen „Zwischenpositionen“ berechnet, als wenn für jeweils nur eine Augenzahl gesetzt werden würde. Diese dürfen nicht beide so groß sein, dass alleine mit einer Augenzahl der Spielstein, der gesetzt werden soll, ausgewürfelt würde. Denn dann würde bereits ein Würfel für den Zug reichen, aber die Funktion hat zu diesem Zeitpunkt bereits festgestellt, dass ein einzelner Würfel nicht reicht. Daher ist in diesem Fall der Zug ungültig.

Wenn nun die Summe der beiden Augenzahlen der Zugweite entspricht und eine der „Zwischenpositionen“, wenn nur für eine Augenzahl gesetzt würde, frei ist, so ist der Zug gültig.

Wenn die Funktion zu diesem Zeitpunkt noch nicht zurückgekehrt ist, müssen in jedem Fall mindestens drei Augenzahlen verwendet werden. Dies ist nur bei einem Pasch möglich, woraus folgt, dass alle Augenzahlen gleich sind. Somit werden im nächsten Schritt die Augenzahlen durchgegangen und jeweils berechnet, welche „Zunge“ zusammen mit den vorhergehenden Augenzahlen erreicht wird. Ist diese „Zunge“ frei (oder auch wenn der Spielstein ausgewürfelt wird) kann die verbleibende Zugweite entsprechend verringert werden. Ist die restliche Zugweite danach null, so ist der Zug gültig. Wenn der Spielstein bereits ausgewürfelt sein sollte, bevor die restliche Zugweite null ist oder die Zugweite auch nach Durchgehen aller Augenzahlen nicht null ist, so ist der Zug ungültig.

Gültigkeit mehrer Züge gleichzeitig feststellen: Mit `are_valid_moves()`[21] steht eine Funktion bereit, die für mehrere Züge (die als `std::vector` übergeben werden) auf einmal prüft, ob diese gültig sind. Dazu wird eine Kopie des aktuellen Backgammon-Spiels erstellt und in einer Schleife wird so lange versucht die noch nicht gesetzten Züge zu setzen bis bei einem Schleifendurchlauf kein weiterer Zug mehr gesetzt werden konnte. Wurden zu dem Zeitpunkt noch nicht alle Züge gesetzt, so ist die übergebene Kombination der Züge nicht gültig, andernfalls ist die Kombination der Züge gültig. Die Funktion ist zudem in der Lage aufgeschlüsselt nach den einzelnen Zügen zurückzugeben, welche Würfel verwendet wurden und stellt dabei auch sicher, dass es dort keine Überschneidungen gibt.

[21] `bool BG::Backgammon::are_valid_moves( const std::vector< BG::BackgammonMove > &move, BG::IllegalMove *reason = NULL, bool **dice_used = NULL );` welfenlab_comp06/backgammon.cpp:398

3.3. Zug ausführen

Zum Ausführen eines oder mehrerer Züge dient die überladene Funktion `move()`[22]. Ein einzelner Zug kann mit der Form `move( const BG::BackgammonMove &move, ... )` ausgeführt werden. Zum Ausführen einer beliebigen Anzahl Züge gleichzeitig dient die Form `move( const std::vector< BG::BackgammonMove > &moves, ... )`. Ist ein Spieler zugunfähig, so kann er die zweite Form mit einem leeren Vektor oder die Form `move( void )` aufrufen. Sollte der übergebene Zug bzw. die Züge nicht gültig sein, so werden diese nicht ausgeführt und die Funktion gibt `false` zurück. Bei `move( void )` ist dies der Fall, wenn der Spieler gar nicht zugunfähig ist. Wurde der Zug ausgeführt wird `true` zurückgegeben. Weiterhin sind diese Funktionen (außer `move( void )` aus offensichtlichen Gründen) wie auch `is_valid_move()`[23] und `are_valid_moves()`[24] in der Lage die benutzten Augenzahlen oder den Grund, warum der Zug/die Züge ungültig sind, zurückzugeben.

Die Form `move( void )` prüft, ob der aktuelle Spieler zugunfähig ist und ruft in diesem Fall die Funktionen `refresh()`[25] und `end_turn()`[26] auf. Auf diese beiden Funktionen werde ich unten etwas näher eingehen.

Die anderem beiden Funktionen der `move()`-Funktion prüfen zuerst, ob der Zug/die Züge gültig sind und wenden sie dann einzeln mit `apply_move()`[27] an. Anschließend werden alle noch nicht benutzten Augenzahlen in `m_dice`[28] an den Anfang des Arrays verschoben. Zum Schluss folgen noch ein Aufruf der Funktion `refresh()` und, sofern der Spieler keinen weiteren gültigen Zug ausführen kann, ein Aufruf der Funktion `end_turn()`.

Die Funktion `refresh()` aktualisiert zum einen Informationen zum Spielstatus, wie z. B. welcher Spieler bereits alle Spielsteine im Homeboard hat. Zum anderen wird die Funktion `is_valid_move_possible()`[29] aufgerufen, damit sichergestellt ist, dass die Werte von `m_dice_result_has_to_be_used`[30] korrekt gesetzt sind (siehe Kapitel 3.1).

Mit einem Aufruf der Funktion `end_turn()` kommt der nächste Spieler an die Reihe.

---

[22] `bool BG::Backgammon::move( void );` welfenlab_comp06/backgammon.cpp:742
`bool BG::Backgammon::move( const BG::BackgammonMove &move, BG::IllegalMove *reason = NULL, bool *dice_used = NULL );`
welfenlab_comp06/backgammon.cpp:759
`bool BG::Backgammon::move( const std::vector< BG::BackgammonMove > &moves, BG::IllegalMove *reason = NULL, bool *dice_used = NULL );`
welfenlab_comp06/backgammon.cpp:810

[23] `bool BG::Backgammon::is_valid_move( const BG::BackgammonMove &move, BG::IllegalMove *reason = NULL, bool *dice_used = NULL, bool check_bar = true );`
welfenlab_comp06/backgammon.cpp:98

[24] `bool BG::Backgammon::are_valid_moves( const std::vector< BG::BackgammonMove > &move, BG::IllegalMove *reason = NULL, bool **dice_used = NULL );` welfenlab_comp06/backgammon.cpp:398

[25] `void BG::Backgammon::refresh( void );` welfenlab_comp06/backgammon.cpp:1431

[26] `void BG::Backgammon::end_turn( void );` welfenlab_comp06/backgammon.cpp:1371

[27] `void BG::Backgammon::apply_move( BG::BackgammonMove move, bool dice_used[ 4 ] );`
welfenlab_comp06/backgammon.cpp:1114

[28] `short int BG::m_dice[ 4 ];` welfenlab_comp06/backgammon.h:357

[29] `bool BG::Backgammon::is_valid_move_possible( BG::BackgammonMove *move = NULL );`
welfenlab_comp06/backgammon.cpp:499

[30] `bool BG::Backgammon::m_dice_result_has_to_be_used[ 2 ];`
welfenlab_comp06/backgammon.h:367

4. Netzwerkfunktionalität

Funktionen zum Aufbau der Verbindung und Kommunikation mit einem Backgammon-Server werden von der Klasse `NetBackgammonConnection`[1] bereitgestellt. Die Klasse führt Buch über den Verbindungsstatus, ob der User eingeloggt ist und er einem Spiel beigetreten ist. Zudem sendet sie für jede eingehende Nachricht vom Server das Qt-Signal `received_msg()`[2]. Dabei wird mit einer Instanz der Klasse `NetBackgammonMsg`[3] die Nachricht gespeichert. Diese Klasse bietet einen einfacheren Zugriff auf die Nachrichtenteile, so dass der Nachrichten-String nicht jedesmal neu aufgetrennt werden muss. Ein Empfänger des Signals ist unter anderen die Klasse `ChatWidget`[4], welche eine Vielzahl der Nachrichten verarbeitet, um sie im Chat-Fenster auszugeben.

Für das eigentliche Backgammon-Spiel wird die Klasse `NetBackgammon`[5] verwendet. Diese Klasse benötigt einen Zeiger auf eine Instanz der Klasse `NetBackgammonConnection` damit sie mit dem Backgammon-Server kommunizieren kann. Zudem muss (sofern es sich nicht um ein lokales Spiel handelt) `m_net_player`[6] auf den Spieler gesetzt werden, der über das Netzwerk gesteuert werden soll.

Jeweils wenn der nächste Spieler an die Reihe kommt, wird `transmit_last_turn()`[7] aufgerufen. Diese Funktion prüft, ob der letzte Spieler der lokal gesteuerte ist, und sendet gegebenenfalls seine Züge mit der Funktion `NetBackgammonConnection::turn()`[8], welche die Züge vom internen Format in das des Servers konvertiert und diese so anordnet, dass sie vom Server akzeptiert werden, denn für die Klasse `Backgammon`[9] ist die Reihenfolge der Züge nicht von Bedeutung, so lange diese zusammen gültig sind.

Weiterhin verarbeitet die Klasse `NetBackgammon` eingehende Servernachrichten in der Funktion `process_srv_msg()`[10]. Insbesondere folgende Nachrichten werden verarbeitet:

- `BOARD`: Übermittelt der Server das aktuelle Spielbrett, so wird dies in das klasseninterne Format konvertiert und gespeichert.
- `DICE` und `INFO:DICE`: Wenn die Würfel übermittelt werden, wird `m_dice`[11] ent-

[1] `class NetBackgammonConnection : public QTcpSocket;` welfenlab_comp06/netbackgammon.h:75
[2] `void NetBackgammonConnection::received_msg( NetBackgammonMsg msg );` welfenlab_comp06/netbackgammon.h:123
[3] `class NetBackgammonMsg;` welfenlab_comp06/netbackgammon.h:36
[4] `class ChatWidget : public QWidget, public Ui::ChatWidget;` welfenlab_comp06/gui/chatwidget.h:38
[5] `class NetBackgammon : public BG::Backgammon;` welfenlab_comp06/netbackgammon.h:155
[6] `BG::Player NetBackgammon::m_net_player;` welfenlab_comp06/netbackgammon.h:191
[7] `void NetBackgammon::transmit_last_turn( void );` welfenlab_comp06/netbackgammon.cpp:473
[8] `void NetBackgammonConnection::turn( const BG::BackgammonTurn &turn );` welfenlab_comp06/netbackgammon.cpp:244
[9] `class BG::Backgammon : public QObject;` welfenlab_comp06/backgammon.h:194
[10] `void NetBackgammon::process_srv_msg( NetBackgammonMsg msg );` welfenlab_comp06/netbackgammon.cpp:490
[11] `short int BG::Backgammon::m_dice[ 4 ];` welfenlab_comp06/backgammon.h:357

sprechend gesetzt.

- `INFO:TURN`: Wird ein Zug übermittelt, so wird dieser in das interne Format konvertiert, ausgeführt und in der Zugliste gespeichert. Zudem kommt der nächste Spieler an die Reihe.
- `ENDGAME`: Sollte das Backgammon-Spiel vorzeitig beendet werden, so werden Variablen der Klasse so gesetzt, dass das aktuelle Spiel auch programmintern beendet wird.

5. Die grafische Benutzeroberfläche

Sämtliche Klassen, die die verschiedenen Elemente des grafischen Userinterface implementieren, liegen im Ordner welfenlab_comp06/gui. Die Dateien welfenlab_comp06/gui/flowlayout.h und welfenlab_comp06/gui/flowlayout.cpp wurden nicht von mir geschrieben, sondern aus der Qt-Dokumentation kopiert. Sie sind allerdings auch nur von untergeordneter Bedeutung. Die Grundstruktur des Hauptfensters und der Dialogfenster wurde mit dem Qt-Designer erstellt. Sofern dies nötig war, wurden die so generierten Klassen nochmals abgeleitet. Auf den Großteil der Klassen für das GUI dürfte nicht näher eingegangen werden müssen, denn diese verwenden keine komplexen Algorithmen. Trotzdem folgt hier eine kurze Aufzählung der Klassen mit einer kurzen Beschreibung. Die Implementation findet sich jeweils in den Dateien mit gleichen Namen, allerdings komplett in Kleinschreibung, und den Endungen .h und .cpp.

- `AISettingsDialog`: Stellt ein Dialogfenster zum Konfigurieren der KI bereit.
- `BackgammonWidget`: Stellt ein Backgammon-Spielfeld dar und bietet dem User die Möglichkeit per Drag & Drop Züge einzugeben.
- `ChatWidget`: Stellt ein Widget bereit, welches Chat- und Servernachrichten anzeigt, sowie in der Lage ist Chat-Nachrichten zu versenden.
- `DiceWidget`: Hierbei handelt es sich um ein Widget, das einen Würfel darstellt und auf diesem verschiedene Augenzahlen anzeigen kann.
- `MainWindow`: Dies ist die Klasse für das Hauptfenster des Programmes.
- `NewGameDialog`: Stellt den Dialog bereit, in dem die Einstellungen für ein neues Spiel gesetzt werden können.

Weiterhin kommen folgende vom Qt-Designer generierte Klassen zum Einsatz, die nicht weiter abgeleitet werden:

- `Ui::InfoDialog`: Stellt die Elemente für ein Dialogfenster mit Informationen zum Programm bereit. Bestimmte Strings des Textes müssen vor der Anzeige allerdings noch ersetzt werden.
- `Ui::LicenseDialog`: Stellt die Elemente für ein Dialogfenster bereit, welches die Lizenz des Programmes – die GNU General Public License – anzeigt.
- `Ui::NewNetGameDialog`: Stellt die Elemente für ein Dialogfenster bereit, in dem die Einstellungen für ein neues Netzwerkspiel vorgenommen werden können.

5.1. Klasse BackgammonWidget

Wie oben bereits erwähnt, dient die Klasse `BackgammonWidget`[1] dazu ein Backgammon-Spielfeld darzustellen. Dieses wird von der Funktion `paintEvent()`[2] gezeichnet. Nachdem Füllen mit der Hintergrundfarbe wird die Zeichenebene gegebenenfalls verschoben und rotiert. Anschließend werden die Abgrezungen des Spielfeldes und der Bar gezeichnet. Danach folgen die obere und untere Hälfte des Spielbrettes, wobei zuerst die jeweilige „Zunge“ und dann die Spielsteine auf dieser gezeichnet werden. Für letzteres wird die Funktion `draw_checkers()`[3] verwendet, welche eine Grafik mit einer bestimmten Anzahl an Spielsteinen erstellt, welche dann auf der entsprechenden „Zunge“ ausgegeben wird. Zum Schluss wird die Drehung und Verschiebung der Zeichenfläche wieder rückgängig gemacht, um die Nummerierung der „Zungen“ auszugeben.

Beim Drücken einer Maustaste wird die Funktion `mousePressEvent()`[4] aufgerufen. Diese wiederrum startet gegebenenfalls eine Drag & Drop-Aktion. Sobald der Drag & Drop-Vorgang abgeschlossen ist wird `dropEvent()`[5] aufgerufen. Diese Funktion muss darauf achten, dass, wenn ein Spielstein ausgewürfelt wird, für den Zug die kleinere Augenzahl verwendet wird, falls der Spieler anschließend die höhere Augenzahl noch setzen will. Beide Funktionen verwenden `conv_mouse_pos()`[6] zur Konvertierung der Mauskoordinaten in eine programminterne Positionsangabe.

5.2. Klasse MainWindow

Ich möchte weitgehend die wichtigsten Funktionen dieser Klasse einfach nur kurz erwähnen, denn auch sie verwenden keine komplexeren Algorithmen, die einer genaueren Erläuterung bedürften:

- `init_new_game()`[7]: Zeigt den Dialog zum Starten eines neuen Spiels an und startet dieses anschließend.
- `start_next_round()`[8]: Startet die nächste Runde eines Matches.
- `next_player()`[9]: Wird aufgerufen, wenn der nächste Spieler an die Reihe kommt. Setzt die Einstellungen des Interface entsprechend.

---

[1] `class BackgammonWidget : public QWidget;` welfenlab_comp06/gui/backgammonwidget.h:37

[2] `void BackgammonWidget::paintEvent( QPaintEvent *event );` welfenlab_comp06/gui/backgammonwidget.cpp:400

[3] `QPicture BackgammonWidget::draw_checkers( unsigned short int n, double size, QColor fg, QColor bg, bool invert );` welfenlab_comp06/gui/backgammonwidget.cpp:185

[4] `void BackgammonWidget::mousePressEvent( QMouseEvent *event );` welfenlab_comp06/gui/backgammonwidget.cpp:322

[5] `void BackgammonWidget::dropEvent( QDropEvent *event );` welfenlab_comp06/gui/backgammonwidget.cpp:236

[6] `BG::Position BackgammonWidget::conv_mouse_pos( int x, int y );` welfenlab_comp06/gui/backgammonwidget.cpp:82

[7] `void MainWindow::init_new_game( void );` welfenlab_comp06/gui/mainwindow.cpp:185

[8] `void MainWindow::start_next_round( void );` welfenlab_comp06/gui/mainwindow.cpp:268

[9] `void MainWindow::next_player( void );` welfenlab_comp06/gui/mainwindow.cpp:419

- `refresh_turn_list()`[10]: Aktualisiert die Anzeige der Zugliste im Hauptfenster (s. u.).
- `show_winner()`[11]: Zeigt den Gewinner des Backgammon-Spiels an.

Nur zu der Funktion `refresh_turn_list()` möchte ich noch ein paar Worte sagen. Sie wird nach jeder Änderung an der Zugliste aufgerufen. Für die jeweils letzten beiden Elemente der Zugliste (`MainWindow::m_game::m_turn_list`[12]) wird ein String erzeugt, der in die Anzeige der Zugliste eingefügt wird oder gegebenenfalls einen älteren ersetzt. Dieser String wird folgendermaßen erzeugt:

1. Kopiere die Liste mit den zu konvertierenden Zügen nach `move_list`.
2. Gehe `move_list` mit dem Iterator `move_list_iter[ 0 ]` durch ...
   - 2.1. Den Zug `move_list_iter[ 0 ]` in das Ausgabeformat konvertieren und dem String `str` zuweisen.
   - 2.2. Züge in der Liste `move_list`, die `move_list_iter[ 0 ]` in der Liste folgen, mit dem Iterator `move_list_iter[ 1 ]` durchgehen...
     - 2.2.1. Schließt der Zug des Iterators `move_list_iter[ 1 ]` direkt an den Zug `move_list_iter[ 0 ]` an, konvertiere `move_list_iter[ 1 ]` in das Ausgabeformat, hänge den konvertierten String an `str` an und lösche `move_list_iter[ 1 ]` aus der Liste.
   - 2.3. Hänge `str` als neues Element an die Liste `moves` an.
3. Initialisieren den Ausgabestring für die Ausgabe mit dem Würfelergebnis.
4. Setze den Zähler `j` auf 1.
5. Gehe die Elemente der Liste `moves` mit dem Iterator `moves_iter[ 0 ]` durch...
   - 5.1.1. Gehe die `moves_iter[ 0 ]` nachfolgenden Elemente in der Liste `moves` mit dme Iterator `moves_iter[1]` durch...
     - 5.1.1.1. Sind die Strings `moves_iter[ 0 ]` und `moves_iter[ 1 ]` gleich, dann erhöhe `j` um eins und lösche `moves_iter[ 1 ]` aus der Liste `moves`.
   - 5.2.2. Füge den String `moves_iter[ 0 ]` mit Angabe der Häufigkeit `j` an den Ausgabestring an.

Zur Konvertierung einer programminternen Positionsangabe zu der auszugebenden Positionsangabe wird die Funktion `conv_pos()`[13] verwendet.

[10] `void MainWindow::refresh_turn_list( void );` welfenlab_comp06/gui/mainwindow.cpp:543
[11] `void MainWindow::show_winner( void );` welfenlab_comp06/gui/mainwindow.cpp:742
[12] `std::vector< BG::BackgammonTurn > MainWindow::m_game::m_turn_list;` welfenlab_comp06/backgammon.h:390
[13] `QString MainWindow::conv_pos( short int pos, BG::Player player );` welfenlab_comp06/gui/mainwindow.cpp:856

6. Künstliche Intelligenz

Die grundlegende Arbeitsweise der künstlichen Intelligenz ist folgende: Es werden alle möglichen Zugmöglichkeiten bis zum Ablauf des Timeouts durchgegangen und die Spielsituationen, die nach Setzen der Züge entstehen, mit einer Bewertungsfunktion unter Berücksichtigung verschiedener Faktoren bewertet.

Implementiert wird die KI durch die Klassen `AI`[1] und `AIThread`[2]. Mittels letzterer wird dabei ein Thread gestartet, der die Berechnungen für die KI vornimmt, so dass die grafische Oberfläche während dieser Zeit bedienbar bleibt. Die Klasse `AI` dient dazu, den Thread einfacher zu kontrollieren und realisiert den Timeout, damit die KI spätestens nach einer bestimmten Zeit einen Zug ausführt.

Mit der Funktion `AI::move()`[3] wird die KI dazu aufgefordert, zu ziehen. Dies läuft so ab, dass der Timer für den Timeout gestartet wird, anschließend wird darauf gewartet, dass der entsprechende KI-Thread `AI::m_ai_thread`[4] wieder im Ruhezustand ist. Das heißt, er hat alle Berechnung vom letzten Zug beendet oder abgebrochen. Dies ist wichtig, falls die gleiche Instanz der KI zweimal schnell hintereinander an die Reihe kommt, damit nicht die Berechnungen des alten Zuges auch für den neuen weiterverwendet werden. Nun wird dem KI-Thread über die Funktion `AIThread::request()`[5] mitgeteilt, dass sie einen Zug suchen soll. Sobald die KI ihre Berechnungen komplett beendet hat oder das Timeout abgelaufen ist, wird die Funktion `AI::do_move()`[6] aufgerufen, welche den besten von der KI bis zu dem Zeitpunkt gefundenen Zug ausführt.

6.1. Suchen nach dem besten Zug

Wenn der KI-Thread dazu aufgefordert wird einen möglichst günstigen Zug zu suchen, wird die Funktion `AIThread::find_move()`[7] aufgerufen, sofern der aktuelle Spieler nicht zugunfähig ist. Diese Funktion geht systematisch alle möglichen Zugkombinationen (mithilfe rekursiver Funktionsaufrufe) durch und bewertet diese. Die jeweils beste wird gespeichert, so dass nachdem alle Zugkombinationen durchgegangen wurden oder der Timeout abgelaufen ist, diese gesetzt werden kann.

Vier Variablen, die die Funktion `find_move()` verwendent, sind als private Membervariablen der Klasse `AIThread`[8] deklariert. Für eine Variable – nämlich `m_game_copy`[9]

[1] `class AI : public QObject;` welfenlab_comp06/ai.h:39
[2] `class AIThread : public QThread;` welfenlab_comp06/aithread.h:39
[3] `void AI::move( void );` welfenlab_comp06/ai.cpp:75
[4] `AIThread AI::m_ai_thread;` welfenlab_comp06/ai.h:72
[5] `inline void AIThread::request( RequestFlag request_flag );` welfenlab_comp06/aithread.h:108
[6] `void AI::do_move( void );` welfenlab_comp06/ai.cpp:113
[7] `void AIThread::find_move( short int rec_depth );` welfenlab_comp06/aithread.cpp:100
[8] `class AIThread : public QThread;` welfenlab_comp06/aithread.h:39
[9] `BG::Backgammon AIThread::m_game_copy;` welfenlab_comp06/aithread.h:216

– war der Grund, dass so nicht jedesmal neu Speicher für die Klasse belegt werden und diese initialisiert werden muss. Diese Variable dient dazu, wenn nötig, eine Kopie der aktuellen Spielsituation des Backgammon-Spiels aufzunehmen. Für die anderen drei Variablen `m_dice`[10], `m_changes`[11] und `m_moves`[12] war der Grund, dass diese so nicht bei jedem rekursiven Funktionsaufruf als Argument übergeben werden müssen. Die erste dient dazu die noch zu setzenden Augenzahlen aufzunehmen, auf die zweite werde ich etwas weiter unten eingehen und die dritte speichert die aktuell untersuchte Zugkombination beim systematischen Durchgehen der Züge.

Beim ersten noch nicht rekursiven Aufruf der Funktion werden die Variablen `m_dice` und `m_changes` zuerst initialisiert.

Als nächstens sorgt die Funktion, wenn noch kein rekursiver Aufruf stattgefunden hat, dafür, dass in `m_best_move`[13] auf jeden Fall ein kompletter, gültiger Zug gespeichert ist. Dazu wird die Funktion `BG::Backgammon::is_valid_move_possible()`[14] verwendet, da sie in der Lage ist gültige Züge zurückzugeben (siehe auch Kapitel 3.1).

Anschließend wird geprüft, ob eine der folgenden Bedingungen erfüllt ist:

1. Für die Züge in `m_moves` müssten alle verfügbaren Augenzahlen gesetzt werden.
2. In `m_moves` sind so viele Züge, dass es eventuell mit diesen möglich ist alle Spielsteine auszuwürfeln.
3. Die Rekursionstiefe ist größer oder gleich zwei, woraus folgt, dass in `m_moves` mindestens zwei Züge gespeichert sind. Dies ist nötig zu prüfen, denn z. B. werden bei einem Pasch auch dann alle vier Augenzahlen nach `m_dice` kopiert, wenn nur zwei oder drei dieser Augenzahlen gesetzt werden können. Ist dies der Fall, so kann dies nicht über Bedingung 1 festgestellt werden.

Wenn eine dieser Bedingungen erfüllt ist, so ist es *möglich*, dass die Zugkombination in `m_moves` ein gültiger Zug ist. Daher wird in diesem Fall eine Kopie der aktuellen Spielsituation in `m_game_copy` erstellt. Nun wird versucht die Züge in `m_moves` anhand der Kopie auszuführen. Wenn dies möglich ist – die Zugkombination also eine gültige ist – und die Zahl der Züge ausreicht, dass der nächste Spieler an die Reihe kommt, wird die Spielsituation nach Ziehen der Züge mithilfe der Bewertungsfunktion `rate_game_situation()`[15] bewertet. Sollte die Bewertung höher als die der Zugkombination in `m_best_move` sein, so wird diese Zugkombination ersetzt. Nähere Informationen zur Bewertungsfunktion finden sich in Kapitel 6.2.

Der letzte Abschnitt der Funktion dient dazu alle Zugkombinationen systematisch durchzugehen. In einer ersten Schleife werden sämtliche „Zungen“ und die Bar durchgegangen. Sofern auf einer Zunge eigene Spielsteine sind, werden in einer weiteren Schleife die noch nicht verwendeten Augenzahlen durchgegangen. Für jede dieser Augenzahlen wird ein Zug von

[10] `short int AIThread::m_dice[ 4 ];` welfenlab_comp06/aithread.h:219
[11] `short int AIThread::m_changes[ 25 ];` welfenlab_comp06/aithread.h:223
[12] `std::vector< BG::BackgammonMove > AIThread::m_moves;` welfenlab_comp06/aithread.h:231
[13] `std::vector< BG::BackgammonMove > AIThread::m_best_move;` welfenlab_comp06/aithread.h:190
[14] `bool BG::Backgammon::is_valid_move_possible( BG::BackgammonMove *move = NULL );` welfenlab_comp06/backgammon.cpp:499
[15] `double AIThread::rate_game_situation( const BG::Backgammon *game, BG::Player player ) const;` welfenlab_comp06/aithread.cpp:282

der entsprechenden Position und Zugweite an `m_moves` angehängt. Die durch diesen Zug auf dem Spielbrett entstehenden Änderungen werden mittels des Arrays `m_changes` gespeichert. Weiterhin wird die entsprechende Augenzahl in `m_dice` auf null gesetzt. Schließlich ruft sich die Funktion rekursiv auf. Anschließend wird der Zug wieder aus `m_moves` entfernt und auch die Änderungen an `m_dice` und `m_changes` rückgängig gemacht.

6.2. Bewertungsfunktion

Eine Spielsituation wird von der KI mit der Funktion `rate_game_situation()`[16] bewertet. Die Bewertung setzt sich aus mehreren Summanden zusammen, die mit einer Ausnahme auf einen Wert zwischen 0 und 1 normiert sind. Die Gewichtung der einzelnen Summanden lässt sich über die Werte im Array `m_rating_factors`[17] ändern. Dies ist auch wichtig, da sowohl Summanden, die positive Sachverhalte bewerten, als auch Summanden, die negative Sachverhalte bewerten, positiv sind. Daher müssen bestimmte Summanden mit einem negativen Faktor für die Gewichtung multipliziert werden. Zur besseren Verständlichkeit können die Elemente des Arrays `m_rating_factors` mit den Werten der Enumeration `RatingFactors`[18] angesprochen werden. Die Elemente dieser Enumeration benutze ich auch zur Benennung der einzelnen Summanden der Bewertungsfunktion nutzen, welche ich im Folgenden aufliste:

- `PROB_CHECKER_IS_HIT`: Summand für die Wahrscheinlichkeit, dass der gegnerische Spieler einen eigenen Spielstein schlagen kann. Dabei werden die Wahrscheinlichkeiten aller einzelnen eigenen Spielsteine – also der Spielsteine, die geschlagen werden können – addiert. Diese Wahrscheinlichkeiten werden jeweils noch mit zwei weiteren Faktoren multipliziert: Der erste Faktor gibt die Entfernung von der ersten „Zunge“ mit einem Wert zwischen 0 (erste „Zunge“) und 1 (letzte „Zunge“) an. Dies ist sinnvoll, denn je näher ein Spielstein an der letzten „Zunge“ ist, desto nachteiliger wäre es, wenn dieser geschlagen würde. Beim zweiten Faktor handelt es sich um die Wahrscheinlichkeit den Spielstein nicht wieder ins Spiel bringen zu können, wenn dieser geschlagen wird. Diese wird über die Gegenwahrscheinlichkeit bestimmt, nämlich eins minus die Wahrscheinlichkeit den Spielstein zurück ins Spiel bringen zu können. Zu diesem Faktor wird zudem noch 1 hinzuaddiert, denn ansonsten würde dieser ganze Summand in der Bewertungsfunktion null, wenn die Wahrscheinlichkeit den Spielstein ins Spiel zurückzubringen 1 beträgt.

- `PROB_CANNOT_MOVE`: Bei diesem Summanden der Bewertungsfunktion wird für jeden eigenen Spielstein die Wahrscheinlichkeit aufaddiert, dass sich dieser in der nächsten Runde nicht setzen lässt (wobei nicht beachtet wird, dass der Gegner vorher noch zieht). Zudem werden diese Wahrschienlichkeiten jeweils mit einem Faktor für die Entfernung zur letzten „Zunge“ zwischen 0 (letzte „Zunge“) und 1 (erste „Zunge“) multipliziert, denn wenn weit vom Homeboard entfernte Spielsteine blockiert werden

[16] `double AIThread::rate_game_situation( const BG::Backgammon *game, BG::Player player ) const;` welfenlab_comp06/aithread.cpp:282

[17] `double AIThread::m_rating_factors[ AIThread::NUM_RATING_FACTORS ];` welfenlab_comp06/aithread.h:202

[18] `enum AIThread::RatingFactors;` welfenlab_comp06/aithread.h:73

ist dies wesentlich ungünstiger als wenn diese bereits nahe am Homeboard sind. In zuletzt geannter Situation muss der Gegner nämlich zwangsweise die Blockade aufgeben, um mit seinen Spielsteinen zum eigenen Homeboard zu gelangen. Bei diesem Summanden wird die Bar nicht mit einbezogen, da der Spieler sowieso gezwungen ist von dieser zu ziehen und daher keine Möglichkeit hat die Wahrscheinlichkeit, dass er mit den Spielsteinen auf der Bar nicht setzen kann, zu beeinflussen.

- `PROB_OP_CANNOT_MOVE` und `PROB_OP_CANNOT_MOVE_BAR`: Bei diesen Summanden wird im Grunde genau das gleiche wie bei `PROB_CANNOT_MOVE` für den gegnerischen Spieler berechnet, nur dass diesmal mit `PROB_OP_CANNOT_MOVE_BAR` auch die Bar miteinbezogen wird, denn die KI kann evtl. gegnerische Spielsteine schlagen. Für diese gibt es einen eigenen Faktor für die Gewichtung, da der Gegner nicht ziehen kann, wenn Spielsteine auf der Bar sind, er diese aber nicht zurück ins Spiel bringen kann.

- `N_POINTS_WITH_CHECKERS`: Weiterhin wird die Zahl der „Zungen“ mit mindestens zwei eigenen Spielsteinen addiert. Dieser Wert wird vor der Addition noch durch 7 geteilt, da es maximal sieben solcher „Zungen“ geben kann. Dies ist sinnvoll, da viele Zungen mit mindestens zwei eigenen Spielsteinen aus zwei Gründen nützlich sind: Zum einen werden so die Zugmöglichkeiten des Gegners eingeschränkt und zum anderen ist es wahrscheinlicher, dass Spielsteine so gesetzt werden können, dass der Gegner nicht in der Lage ist einen zu schlagen.

- `BEARED_OFF`: Dieser Summand gibt einfach nur den Anteil eigener ausgewürfelter Spielsteine an.

- `DISTANCE_FROM_HOMEBOARD`: Dieser Summand gibt die durchschnittliche Entfernung der Spielsteine zum Homeboard geteilt durch 18 an. Der Wert wird zur Normierung auf eine Größe zwischen 0 und 1 durch 18 geteilt, denn weiter als 18 Felder kann ein Spielstein nicht vom Homeboard entfernt sein.

- `CHECKERS_IN_OP_HOMEBOARD`: Dieser Summand gibt die Zahl der eigenen Spiele im gegnerischen Homeboard und auf der Bar an (zur Normierung geteilt durch 15). Dies lasse ich mit in die Bewertung einfließen, da der Gegner mit 3 Punkten – einem Backgammon – gewinnen könnte, so lange sich dort Spielsteine befinden.

- `OP_DISTANCE_FROM_OFF_GAME`: Mit diesem Summand wird angegeben, wieviele „Zungen“ die Spielsteine des Gegners durchschnittlich noch gezogen werden müssen, bis diese ausgewürfelt sind. Zur Normierung auf einen Wert zwischen 0 und 1 wird durch 24 geteilt. Dieser Summand ändert sich, wenn Spielsteine des Gegners geschlagen werden und zwar um so mehr, je näher der geschlagene Spielstein bereits an der letzten „Zunge“ war.

Durch ein Anpassen der Bewertungsfaktoren in `m_rating_factors` lässt sich das Verhalten der KI ändern. In welfenlab_comp06/ai_std_config.h habe ich folgende Standardwerte deklariert:

```
35 /// Standardwerte für die Bewertungsfaktoren.
   const double AI_STD_RATING_FACTORS[ AIThread::NUM_RATING_FACTORS ] =
```

```
    {
      -400.0, // PROB_CHECKER_IS_HIT
39    -400.0, // PROB_CANNOT_MOVE
       100.0, // PROB_OP_CANNOT_MOVE
       100.0, // PROB_OP_CANNOT_MOVE_BAR
       100.0, // N_POINTS_WITH_CHECKERS
       100.0, // BEARED_OFF
44    -600.0, // DISTANCE_FROM_HOMEBOARD
      -100.0, // CHECKERS_IN_OP_HOMBEBOARD
       700.0  // OP_DISTANCE_FROM_OFF_GAME
    };
```

Diese habe ich vor allem dadurch ermittelt, dass ich wiederholt gegen die KI gespielt habe oder auch die KI gegen sich selbst habe spielen lassen. Mithilfe meiner Beobachtungen, an welchen Stellen die KI nicht so reagiert wie es opitmal wäre, habe ich die Werte dann schrittweise angepasst. Eine andere Methode, die ich mir überlegt habe, um die Werte automatisch anzupassen, hat leider nicht wie gewünscht funktioniert. Ich gehe dadrauf in Kapitel 6.4 ein.

Im Folgenden werde ich noch erklären, wie die Bewertungsfunktion bestimmte Wahrscheinlichkeiten berechnet.

6.2.1. Wahrscheinlichkeit, dass eine bestimmte „Zunge“ erreicht werden kann

Zur Berechnung der Wahrscheinlichkeit, dass der Spieler `player` eine bestimmte „Zunge“, im Folgendem mit `position` bezeichnet, in seinem nächsten Zug erreichen kann, wird die Funktion `prob_position_is_reached()`[19] verwendet. Dazu speichert sie in dem zweidimensionalen Array `possible_dice_combinations`, mit welchen Würfelwürfen `position` erreicht werden kann.

Um dies zu ermitteln, werden in einer ersten Schleife die möglichen Augenzahlen (also eins bis sechs) durchgegangen. Die „Zunge“, die entsprechend viele Felder vor `position` liegt, wird mit `checking_position` gespeichert. Dann wird geprüft, ob auf diesem Feld ein Spielstein von `player` ist, dieser also `position` erreichen kann. Wenn dies der Fall ist, werden die Werte von `possible_dice_combinations` entsprechend gesetzt. Ansonsten werden – sofern nicht mindestens zwei gegnerische Spielsteine auf `position` sind, in einer weiteren Schleife erneut die möglichen Augenzahlen durchgegangen und dabei `checking_position` jeweils auf die „Zunge“ gesetzt, die die Entfernung beider Augenzahlen vor `position` liegt. Wenn auf `checking_position` ein eigener Spielstein ist, kann `position` von `player` erreicht werden und `possible_dice_combinations` wird entsprechend gesetzt. Wenn dies nicht der Fall ist, aber beide Augenzahlen gleich sind und auf `checking_position` nicht mehr als ein gegnerischer Spielstein ist, wird dieses Schema fortgesetzt. Allerdings werden dabei dann keine Augenzahlen mehr durchgegangen, da es sich um ein Pasch handelt und somit alle drei oder vier Augenzahlen gleich sind.

Nachdem so sämtliche Werte in `possible_dice_combinations` gesetzt wurden, wird dieses Array erneut durchgegangen und für jede mögliche Kombination zweier Augenzahlen, mit denen `position` erreicht werden könnte, $\frac{1}{36}$ zum Rückgabewert addiert, da dies die Wahrscheinlichkeit dafür ist, dass eine bestimmte Kombination zweier Augenzahlen auftritt.

[19] `double AIThread::prob_position_is_reached( const BG::Backgammon *game, short int position, BG::Player player ) const;` welfenlab_comp06/aithread.cpp:567

6.2.2. Wahrscheinlichkeit, einen Spielstein nicht setzen zu können

Mit der Funktion `prob_player_cannot_move_at_pos()`[20] wird die Wahrscheinlichkeit bestimmt, dass ein Spieler im nächsten Zug einen Spielstein nicht setzen kann. Dazu werden einfach die möglichen Augenzahlen von eins bis sechs durchgegangen und jeweils geprüft, ob sich der Spielstein damit setzen ließe. Für jede Augenzahl, für die dies möglich ist, muss $\frac{1}{6}$ zum Rückgabewert addiert werden.

6.3. Verbesserungsideen

Während der Entwicklung der KI hatte ich vor allem zwei weitere Ideen, wie man diese noch weiter verbessern könnte, abgesehen von Verbesserungen an der Bewertungsfunktion und der Gewichtung deren Summanden. Allerdings fehlte mir die Zeit diese genauer auszuarbeiten oder gar zu implementieren.

Meine erste Idee war, dass es nicht gerade optimal ist, die KI die Zugkombinationen in starrer Reihenfolge durchgehen zu lassen. Sinnvoller wäre es, wenn anfangs solche Zugkombinationen geprüft werden, bei denen die Wahrscheinlichkeit, dass sie eine hohe Bewertung erhalten, relativ hoch ist und erst anschließend die restlichen Zugkombinationen. Dies wäre gerade bei kurzen Timeout-Werten von Vorteil. Allerdings stellt sich die Frage, wie man schnell die Zugkombinationen ermittelt, welche zuerst geprüft werden sollen, denn wenn dies zu lange dauert ist die jetzige Implementierung der KI weiterhin im Vorteil.

Dieses Problem könnte evtl. durch meine zweite Idee gelöst werden: So lange der gegnerische Spieler an der Reihe ist, tut die KI nichts. Diese Zeit könnte man aber bereits für Berechnungen nutzen. So könnten z. B. bereits Züge für meine erste Idee rausgesucht werden.

Andererseits stellt sich die Frage, ob der Nutzen dieser beiden Ideen so groß ist, dass der Implementierungsaufwand gerechtfertigt ist. So hat die KI auf meinen Testrechnern[21] in vielen Fällen innerhalb kürzester Zeit gezogen und das Timeout, das ich in der Regel auf 3 Sekunden eingestellt hatte, gar nicht ausgenutzt. Doch auch nicht gerade selten wurde die KI durch den Timeout unterbrochen. Gerade bei einem Pasch, wo sich die Zahl der Zugmöglichkeiten vervielfacht, ist dies eigentlich immer der Fall gewesen.

6.4. ai-evolver

Anfangs hatte ich vor, möglichst gute Gewichtungen für die Bewertungsfunktion mit einem genetischen Algorithmus zu finden. Im Endeffekt hat dies leider nicht funktioniert und die Gewichtungen wurden zunehmend schlechter, je länger der Algorithmus lief. Trotzdem möchte ich hier etwas näher auf diesen eingehen und im Anschluss ein paar Überlegungen anstellen, was der Grund des Versagens sein könnte.

Der genetische Algorithmus, den ich implementiert habe ist einer der einfachsten. Zunächst hat man ein Ausgangsindividuum (hier bestimmte Gewichtungen) mit dem man

[20] `double AIThread::prob_player_cannot_move_at_pos( BG::Position pos, const BG::Backgammon *game, BG::Player player ) const;` welfenlab_comp06/aithread.cpp:493

[21] Die Prozessoren meiner Testsysteme waren ein AMD Athlon XP 2600+ Prozessor mit ca. 2 GHz und ein Intel Celeron M mit ca. 1,6 GHz.

eine Population einer bestimmten Größe erstellt. Dies geschieht, indem man Varianten des Ausgangsindividuums erstellt. Jeder einzelne Gewichtungsfaktor kann mit einer gewissen Wahrscheinlichkeit verändert werden („Mutationswahrscheinlichkeit“). Wenn dies der Fall ist, wird der Wert um einen zufälligen Wert geändert, dessen maximaler Betrag festgelegt ist („Mutationsradius“). Anschließend bestimmt man, welches Individuum in der Population die größte „Fitness“ hat. In unserem Fall wäre das die Spielstärke der KI. Zur Bestimmung dieser lässt man jedes Individuum der Population gegen alle anderen ein oder mehrere Backgammon-Spiele austragen und das Individuum mit den meisten Punkten ist das mit der größten „Fitness“. Mit diesem Individuum bzw. diesen Gewichtungsfaktoren wird wieder eine neue Population erstellt.

Implementiert habe ich einen solchen Algorithmus mit dem Unterprojekt ai-evolver im Ordner ai-evolver. Dabei findet sich der Hauptcode in der Klasse `AIEvolver`[22]. Die Funktion `evolve()`[23] implementiert den genetischen Algorithmus. Ein Backgammon-Spiel zwischen zwei Individuen bzw. zwei KIs mit unterschiedlichen Gewichtungsfaktoren wird mithilfe der Funktion `simulate_game()`[24] ausgetragen.

6.4.1. Bedienung

Der ai-evolver ist ein Kommandozeilen-Programm, das mit 12 oder 13 Argumenten aufgerufen werden muss. Der Aufruf lautet:

```
$ ./ai–evolver <Populationsgröße> <Mutationswahrscheinlichkeit>
<Mutationsradius> <Spiele pro Match> <Faktor 1 (PROB_CHECKER_IS_HIT)>
<Faktor 2 (PROB_CANNOT_MOVE)> <Faktor 3 (PROB_OP_CANNOT_MOVE)>
<Faktor 4 (PROB_OP_CANNOT_MOVE_BAR)> <Faktor 5 (N_POINTS_WITH_CHECKER)>
<Faktor 6 (BEARED_OFF)> <Faktor 7 (DISTANCE_FROM_HOMEBOARD)>
<Faktor 8 (CHECKERS_IN_OP_HOMEBOARD)>
<Faktor 9 (OP_DISTANCE_FROM_OFF_GAME)> [Timeout]
```

Mit Strg + C wird das Programm beendet.

Jede Minute und immer, wenn bessere Gewichtungsfaktoren gefunden wurden, wird eine Statusmeldung ausgegeben, die auch die besten bis zu dem Zeitpunkt gefundenen Gewichtungsfaktoren enthält. Zudem wird für jedes gestartete Backgammon-Spiel eine öffnende eckige Klammer ([), für jedes beendete eine schließende eckige Klammer (]) ausgegeben. Für jeden Zug in einem Backgammon-Spiel wird ein Punkt (.) ausgegeben.

6.4.2. Überlegungen, warum der Algorithmus versagt

Das Problem ist wahrscheinlich, dass zum einen die Unterschiede zwischen den einzelnen Individuuen einer Population recht gering sind und zum anderen Backgammon eine nicht allzu geringe Zufallskomponente hat. Aufgrund des geringen Unterschieds der Gewichtungsfaktoren spielen die verschiedenen Indiviuuen fast gleich stark und durch die Zufallskomponente kann auch die KI mit den eigentlich schlechteren Gewichtungsfaktoren viele Spiele für sich entscheiden. Da die Wahrscheinlichkeit dafür wohl immer noch knapp unter 50 Prozent ist,

[22] `class AIEvolver : public QCoreApplication;` ai-evolver/aievolver.h:34
[23] `int AIEvolver::evolve( void );` ai-evolver/aievolver.cpp:122
[24] `int AIEvolver::simulate_game( double *factors1, double *factors2 );` ai-evolver/aievolver.cpp:290

kommt es häufig vor, dass die schlechteren Gewichtungsfaktoren als die besseren erkannt werden. Zudem ist die Wahrscheinlichkeit, dass die Faktoren nach einer „Mutation“ besser sind relativ gering, so dass von vornherein ein Überschuss an schlechteren Gewichtungsfaktoren besteht. So werden die Faktoren fortlaufend geringfügig schlechter, da sich nicht einwandfrei sagen lässt, welche denn nun wirklich besser sind.

Lösen könnte man das Problem unter Umständen indem man die Individuuen eine deutlich größere Anzahl an Spielen gegeneinander austragen lässt oder die Populationsgröße erhöht, was ja auch wiederum zu einer Erhöhung der Anzahl der gespielten Spiele führt.[25] Durch die Erhöhung der Anzahl der Spiele rückt die Zufallskomponente etwas in den Hintergrund bzw. das Verhältnis der gewonnenen Spiele nähert sich dem Verhältnis der theoretischen Gewinnwahrscheinlichkeiten an (Gesetz der großen Zahl). Die Zahl der Spiele, die gegeneinander ausgetragen werden müssten, ist aber wohl so hoch, dass die benötigte Rechenzeit ins unermessliche wächst und diese Lösung somit nicht praktikabel ist.

[25] Die maximalen Werte, die ich verwendet habe waren eine Populationsgröße von 5 und jeweils 5 Spielen sowie eine Populationsgröße von 3 und jeweils 7 Spielen.

Teil III.

Anhang

A. Schlusswort und Rückblick

Nachdem ich nun zum vierten und letzten Mal an der Welfenlab Competition teilgenommen habe ist es Zeit für einen kurzen Rückblick. Mir hat die Bearbeitung der Aufgaben jedesmal viel Spaß gemacht, auch wenn es manchmal doch recht anstrengend und zeitaufwändig war. Aber ich fand, dass die Aufgaben – wenn auch anspruchsvoll – nie zu schwer waren. Interessanterweise fand ich sie jedes Jahr sogar ein bisschen leichter. Aber das liegt vielleicht auch daran, dass ich dazu gelernt habe. Nicht nur, aber auch bei der Welfenlab Competition. Gerade zu Anfang war der Lerneffekt sehr hoch. So gibt es aus meiner heutigen Sicht bei den Aufgaben von vor drei Jahren viele Dinge, die ich jetzt anders und hoffentlich besser machen würde.

Hiermit möchte ich auch alle Schüler, die dies vielleicht lesen, dazu ermutigen an der Welfenlab Competition oder anderen Schüler-Wettbewerben teilzunehmen. Man lernt einfach eine Menge und es macht eine Menge Spaß. Selbst wenn es mal nicht ganz so glatt läuft, sollte man nicht aufgeben. Dieses Jahr musste ich über zwei Wochen nach einem Bug suchen! Und wenn man beim ersten Versuch keinen der ersten drei Plätze erreicht, klappt es vielleicht beim nächsten Mal. Ich habe es auch nicht gleich auf Anhieb geschafft.

Ich konnte jedenfalls eine Menge Erfahrungen sammeln und bin stolz auf das, was ich bei der Welfenlab Competition geschafft habe. Jetzt bin ich gespannt welche Herausforderungen mich in der Zukunft erwarten ...

B. Bekannte Bugs des Backgammon-Servers auf werefkin.gdv.uni-hannover.de

Im Folgenden dokumentiere ich die von mir gefundenen Bugs des Backgammon-Servers auf werefkin.gdv.uni-hannover.de und deren Auswirkungen auf mein Programm.

- **Beschreibung:** In folgendem Fall erkennt der Server nicht, dass der Spieler nach einem Zug zugunfähig ist:

  ```
  > BOARD:0 0 6 0 −1 0 2 0 3 0 0 0 0 0 0 0 0 0 −3 −2 −2 −3 −1 −3 2 2 0 0
  > DICE:6 1
  < TURN:1 25 24;
  > ERROR:TURN;not a valid Turn
  > DICE:6 1
  ```

  Es sind zwei Steine auf der Bar, es kann allerdings nur einer mit der Augenzahl 1 zurück ins Spiel gebracht werden, weil das 6. Feld mit drei gegnerischen Steinen belegt ist.

  Auswirkung: Das Spiel kann nicht fortgesetzt werden.

- **Beschreibung:** In folgendem Fall erkennt der Server nicht, dass nur noch für einen Würfel gezogen werden kann, da nur noch ein Spielstein im Spiel ist:

  ```
  > BOARD:14 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 −2 0 −13 0
  < CONFIRM
  > DICE:4 6
  < TURN:6 1 0;
  > ERROR:TURN;not a valid Turn
  > DICE:4 6
  ```

  Auswirkung: Es wird die normale Meldung angezeigt, dass das Spiel zu Ende wäre, da dies vom Programm selbstständig erkannt wird. Wenn der entsprechende Spieler allerdings durch die KI gesteuert wird, wird die Meldung wiederholt angezeigt, weil der Server immer wieder die Aufforderung sendet zu ziehen, somit mehrmals der letzte Zug ausgeführt wird und jedesmal die Meldung angezeigt wird, dass das Spiel beendet wurde.

- **Beschreibung:** In folgendem Fall erkennt der Server nicht, dass der Spieler nach dem einen Zug zugunfähig ist:

  ```
  > BOARD:0 2 9 0 0 0 2 0 0 −1 0 0 0 0 0 0 0 −3 −2 −2 −2 −2 0 −2 2 0 0 −1
  < CONFIRM
  > DICE:6 1
  < TURN:1 2 1;
  > ERROR:TURN;not a valid Turn
  > DICE:6 1
  ```

Es sind nicht alle Steine im Homeboard, daher kann keiner von diesen ausgewürfelt werden. Die beiden Steine ganz hinten können aber auch nicht gezogen werden. Daher kann nur die eins gesetzt werden.

Auswirkung: Das Spiel kann nicht fortgesetzt werden.

- **Beschreibung:** In folgendem Fall erkennt das Spiel zwar die Zugunfähigkeit, geht aber erst nach einer längeren Wartezeit weiter:

```
> BOARD:0 −2 2 2 2 2 2 3 0 0 0 0 0 0 0 0 0 −2 −3 0 −3 −3 −2 2 0 0 0
> DICE:3 6
< TURN:3 7 4;
```

Dieser Bug scheint immer dann aufzutreten, wenn nur für eine Augenzahl gesetzt werden kann, aber der Server noch erkennt, dass der Spieler nach Setzen des einen Zuges zugunfähig ist. Im Unterschied zum vorhergehenden Bug befinden sich keine Steine auf der Bar.

Auswirkung: Das Spiel kann nach der Wartezeit normal weitergeführt werden.

- **Beschreibung:** Der letzte Zug in einem Spiel wird vom Server falsch codiert an den Client übertragen, der nicht am Zug ist:

```
> BOARD:14 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 −2 0 −13 0
< CONFIRM
> INFO:DICE;6 4
> INFO:TURN;6 1 0;4 1 0;
> ENDGAME:LOSE;1;0 1
> INFO:ENDGAME;1 gosmann2 1 1 0
```

Korrekt müsste der vom Server übermittelte Zug für das Beispiel in der Debug-Ausgabe `INFO:TURN;6 24 26;4 24 26;` sein.

Auswirkung: Der letzte Zug in einem Spiel wird nicht korrekt in die Zugliste eingetragen und die Verteilung der Spielsteine auf dem Spielfeld wird nicht korrekt angezeigt.

B.1. Anscheind bereits behobene Bugs

Hier liste ich die Bugs aus früheren Server-Versionen auf, die anscheinend bereits behoben wurden. Da es kein Changelog gab, kann ich mir dessen leider nicht ganz sicher sein. Dies ist auch der Grund für diesen Abschnitt.

- **Beschreibung:** Der Server sendet zu Beginn des Spieles ein Pasch als Würfelwurf. Dies ist bei einem normalen Backgammon-Spiel eigentlich nicht möglich, da durch den ersten Würfelwurf entschieden wird, wer das Spiel beginnt. Bei einem Pasch würde neu gewürfelt werden.

  Auswirkung: Das Spiel kann nicht gestartet werden und es muss ein neues erstellt werden.