Hajo Schulz

Denkmaschine

Brettspiel-KI selbst gemacht

Dass man ein Schachprogramm, das menschlichen Großmeistern ernsthaft Konkurrenz machen könnte, nicht mal eben am Wochenende schreibt, sollte intuitiv klar sein. Für etwas weniger anspruchsvolle Spiele ist das aber mit unserem C#-Framework und dem Know-how aus diesem Artikel durchaus drin.

Schon seit Urzeiten sind Menschen fasziniert von der Idee, Maschinen zu bauen, die menschlichen Gegnern in Denkspielen wie Schach Paroli bieten können. Im 18. Jahrhundert ging diese Faszination sogar so weit, dass man einen angeblichen Schachroboter baute, in dem aber tatsächlich ein menschlicher Spieler steckte.

Mittlerweile haben Computer die Rolle solcher Maschinen übernommen, und selbst wenn sie auf Taschenspielertricks verzichten, haben menschliche Großmeister kaum noch Chancen gegen gute Schachprogramme. Sogar Programme für das asiatische Brettspiel Go, das lange Zeit als von Computern nicht beherrschbar galt, schlagen mittlerweile sehr gute Amateure. Superrechner sind dazu nicht mehr notwendig, handelsübliche PCs reichen aus. Der Leidenschaft, mit der solche Programme entwickelt werden, hat das keinen Abbruch getan – das zeigen regelmäßig stattfindende Computer-Weltmeisterschaften für alle möglichen Spiele. So traten bei der Computer-Olympiade der International Computer Games Association zuletzt 85 Entwicklerteams in 25 Disziplinen gegeneinander an [1].

So unterschiedlich Spiele wie Schach, Go, „Vier gewinnt“ oder Othello auf den ersten Blick aussehen mögen, sie (und viele andere Spiele) haben doch etliche Gemeinsamkeiten: Sie alle werden von zwei Spielern gespielt, die abwechselnd am Zug sind. Abgesehen davon, dass das Recht des ersten (oder letzten) Zuges einen Vorteil bedeuten kann, hängen die Siegchancen beider Spieler ausschließlich von ihren eigenen Zügen ab – es gibt keine Glückskomponenten wie Würfel oder Karten. Ein Beobachter, der die jeweiligen Spielregeln kennt, kann aus der Stellung auf dem Brett und der Information, welcher Spieler als Nächstes am Zug ist, die Spielsituation eindeutig beurteilen – es gibt keine verdeckten Gegenstände wie die Rohstoffkarten bei „Die Siedler von Catan“ oder die umgedrehten Kärtchen beim Memory.

© Copyright by Heise Zeitschriften Verlag
Persönliches PDF für Frank Baethge aus

Bei allen Spielen, auf die diese Merkmale zutreffen, funktioniert die Suche nach dem besten nächsten Zug auf recht ähnliche Art und Weise. Wir haben diese Mechanismen daher in eine in C# implementierte .NET-Klassenbibliothek gegossen. Wie man sie verwendet, demonstriert ein Windows-Programm, das das Spiel Othello beherrscht (Spielregeln im Kasten unten) und das Sie samt Bibliothek und Quelltext über den c't-Link unter diesem Artikel herunterladen können.

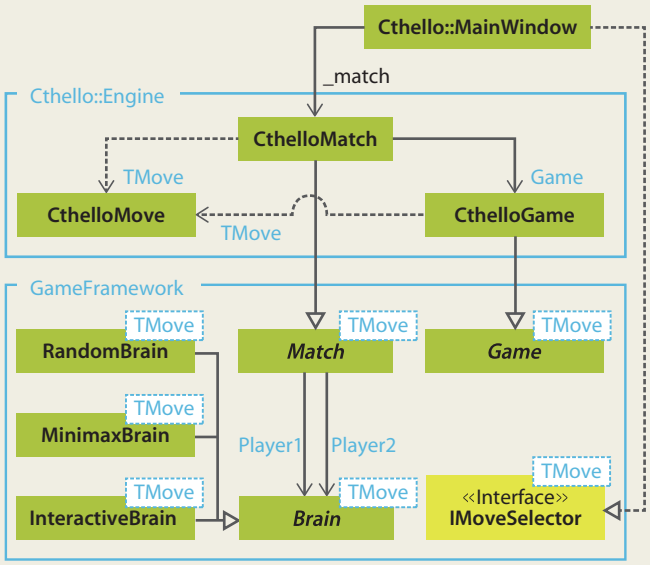
Regelwerk

Die grundlegenden Klassen residieren im Namensraum GameFramework. Da wären zunächst einmal die abstrakten Basisklassen Game und Match. Game selbst speichert nur den bisherigen Spielverlauf. Ableitungen implementieren die jeweiligen Spielregeln, können also unter anderem Züge auf Gültigkeit prüfen und durchführen, eine Liste aller möglichen Züge des aktuellen Spielers erzeugen und den Spielstand beurteilen.

Match beziehungsweise die für ein konkretes Spiel davon abgeleitete Klasse kümmert sich darum, welche beiden Spieler sich duellieren, und weiß, wer gerade am Zug ist. Außerdem bildet diese Klasse das Bindeglied zwischen der Game-Klasse und der Bedienschnittstelle.

Das Framework

Die Brain-Klassen im Namensraum GameFramework implementieren unabhängig vom eigentlichen Spiel verschiedene Strategien, um den nächsten Spielzug auszuwählen.



Die Spieler werden durch Ableitungen der abstrakten Klasse Brain dargestellt, die die Aufgabe haben, den jeweils nächsten Zug auszuwählen. Sie sind so aufgebaut, dass sie unabhängig vom konkreten Spiel funktionieren – das Wissen darüber steckt komplett in der Game-Klasse. Wenn Sie mit dem Framework ein anderes Spiel als Othello bauen wollen, müssen Sie die mitgelieferten Brain-Klassen nicht anfassen oder davon ableiten. In der vorliegenden Version enthält das Framework drei Brain-Ableitungen: RandomBrain ist vor allem für Testzwecke gedacht und spielt gültige, aber zufällig ausgewählte Züge. In MinimaxBrain steckt die eigentliche Spiel-Intelligenz – auf seine Funktionsweise gehen wir weiter unten noch ausführlich ein. Schließlich gibt es noch einen InteractiveBrain, der über sein Spiel nicht selbst entscheidet, sondern darauf wartet, dass ein menschlicher Spieler in einem Client-Programm den nächsten Zug auswählt.

Alle bisher vorgestellten Klassen sind generisch und brauchen den Typparameter TMove, hinter dem eine spielspezifische Klasse stecken muss, die einen Spielzug darstellt. Die muss aber nicht von einer speziellen Basisklasse erben.

Das Beispielprogramm zu diesem Artikel – es heißt übrigens c'thello – besteht abgesehen von der Bedienschnittstelle aus den drei Klassen CthelloGame, CthelloMatch und CthelloMove. Letztere ist eine primitive Datenklasse, die nur speichert, welcher Spieler diesen Zug spielt (Player ist ein int; 0 steht für Schwarz, 1 für Weiß), welches Feld er besetzt (Row und Column) oder ob er passt (Pass). CthelloGame erbt von Game<CthelloMove> und CthelloMatch von Match<CthelloMove>.

Spielerhirn

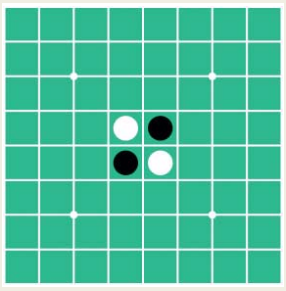
Damit ein Computerspiel den Anschein von Intelligenz erweckt, braucht es eine Komponente, die immer dann, wenn der Rechner am Zug ist, aus den regelkonformen Spielzügen den besten auswählt. In der einfachsten Variante spielt das Programm jeden Zug probeweise und bewertet dann die dadurch

Othello

Das Spiel, das unser Beispielprogramm spielt, ist unter den Markennamen Othello und – mit wenigen Regelabweichungen – Reversi bekannt. Es wird auf einem Spielbrett mit acht mal acht Feldern und mit flachen Spielsteinen gespielt, die unterschiedlich gefärbte Vorder- und Rückseiten haben; die sichtbare Seite gibt an, welcher Partei ein Stein momentan gehört. Zu Beginn liegt in den vier Feldern der Spielfeldmitte je ein Stein, je zwei für Schwarz und für Weiß. Schwarz beginnt. Ein gültiger Zug besteht darin, dass man einen Stein der eigenen Farbe so auf ein freies Feld platziert, dass ein oder mehrere gegnerische Steine in einer ununterbrochenen, geraden Linie senkrecht, waagerecht oder diagonal zwischen dem neuen und einem bereits auf dem Brett liegenden eigenen Stein eingeschlossen werden; das kann auch in mehreren Richtungen gleichzeitig der Fall sein.

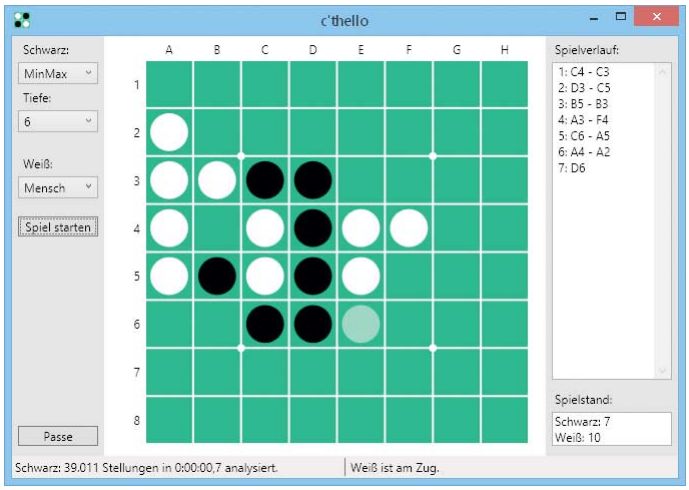
Alle so umschlossenen Steine dreht man auf seine eigene Farbe um. Es besteht Zugzwang; passen darf man nur, wenn man keinen gültigen Zug hat. Das Spiel endet, wenn beide Spieler nacheinander passen müssen, spätestens wenn das Spielfeld komplett gefüllt ist. Gewonnen hat, wer dann mehr Steine der eigenen Farbe auf dem Brett hat.

Taktisch ist Othello anspruchsvoller, als man auf den ersten Blick meint. So ist es relativ unwichtig, möglichst früh möglichst viele eigene Steine zu haben: Ein gut aufgestellter Gegner kann das Bild in den letzten paar Zügen komplett wenden. Wichtiger ist zum Beispiel, die Eckfelder zu gewinnen: Sind sie einmal besetzt, können sie nicht mehr geschlagen werden und dienen im Endspiel oft als Basis zum Überrollen des Gegners. Die direkt an die Ecken angrenzenden Felder sind dagegen eher zu vermeiden, weil sie häufig dem Gegner die Möglichkeit geben, die Ecke zu erobern. Steine am Rand sind schwerer zu schlagen als in der Mitte. Unbedingt im Auge behalten sollte man die Anzahl der möglichen eigenen Züge – wer im Endspiel vor dem letzten Zug passen muss, hat meist verloren.



Bei Othello beginnt jeder Spieler mit zwei Steinen seiner eigenen Farbe.

© Copyright by Heise Zeitschriften Verlag
Persönliches PDF für Frank Baethge aus



Optisch ist unser Beispielprogramm c'thello zwar nicht wirklich ein Hingucker. Dafür spielt es ganz passabel Othello und demonstriert den Einsatz der universellen C#-Bibliothek.

entstehenden Stellungen. Die Bewertungsfunktion zählt bei einem Othello-Spiel beispielsweise die eigenen Steine und vergibt dabei für eine eroberte Ecke ein paar Extra-Punkte. Von den so errechneten eigenen Wertungspunkten werden die auf dieselbe Weise ermittelten Punkte des Gegners abgezogen, sodass am Ende ein einziger Zahlenwert übrigbleibt. Je höher der ist, desto günstiger steht es für den aktuellen Spieler, desto besser ist also der Zug, der zu dieser Stellung führt.

Besonders stark wird ein Programm auf diese Weise aber nie spielen. Das liegt zum einen daran, dass die beschriebene Bewertungsfunktion ziemlich primitiv ist. Aber selbst wenn man sie verfeinert, wird es schwierig, taktische Finessen eines Spiels zu erkennen, also etwa Züge, die den Gegner zu nachteiligen Erwiderungen zwingen oder die einen entscheidenden Schlag im nächsten Zug vorbereiten. Bessere Spielprogramme bewerten daher bei ihren Entscheidungen nicht die Stellung unmittelbar nach dem eigenen nächsten Zug, sondern schauen einige Spielzüge in die Zukunft: Sie berechnen zu jedem regelkonformen eigenen Zug die möglichen Antworten des Gegners und in jeder dadurch entstehenden Stellung wieder die eigenen Möglichkeiten. Erst die Bewertung der Stellungen nach – in diesem Beispiel – drei Zügen entscheidet dann darüber, welcher Zug als Nächstes gespielt wird. Genau genommen spricht man übrigens bei einer Angabe, wie weit so ein Algorithmus in die Zukunft schaut, nicht von Zügen, sondern von Halbzügen: Die Aktion eines Spielers plus die Erwiderung des Gegners macht zwei Halbzüge.

Nun wäre es naiv, einfach den Zug zu spielen, der im weiteren Verlauf die beste Fortsetzung verspricht: Dazwischen ist ja immer auch der Gegner am Zug, und wenn der mindestens genauso stark spielt wie man selbst, wird er wohl verhindern, dass es zu der gewünschten Position kommt. Die nebenstehende Grafik verdeutlicht das: Weiß ist am Zug. Zwei Halbzüge weiter wäre der beste Zug von Weiß 12 Punkte wert, dazu müsste jetzt der obere Zug gespielt werden. Dann wäre aber Schwarz an der Reihe und hätte die Wahl zwischen einem Zug, der Weiß bis zu 12 Punkte bringt, und einem, der für Weiß höchstens 5 Punkte wert ist. Schwarz würde also den unteren Zug spielen, somit stünde Weiß nach zwei Halbzügen bestenfalls mit 5 Punkten da. Spielt Weiß im ersten Schritt den unteren Zug, lässt er Schwarz die Wahl zwischen drei Möglichkeiten, die aus weißer Sicht 10, 7 und 8 Punkte wert sind; Schwarz spielt dann vermutlich den mittleren, was Weiß nach drei Halbzügen insgesamt 7 Punkte beschert. Somit ist für Weiß der untere Zug um 2 Punkte günstiger als der obere.

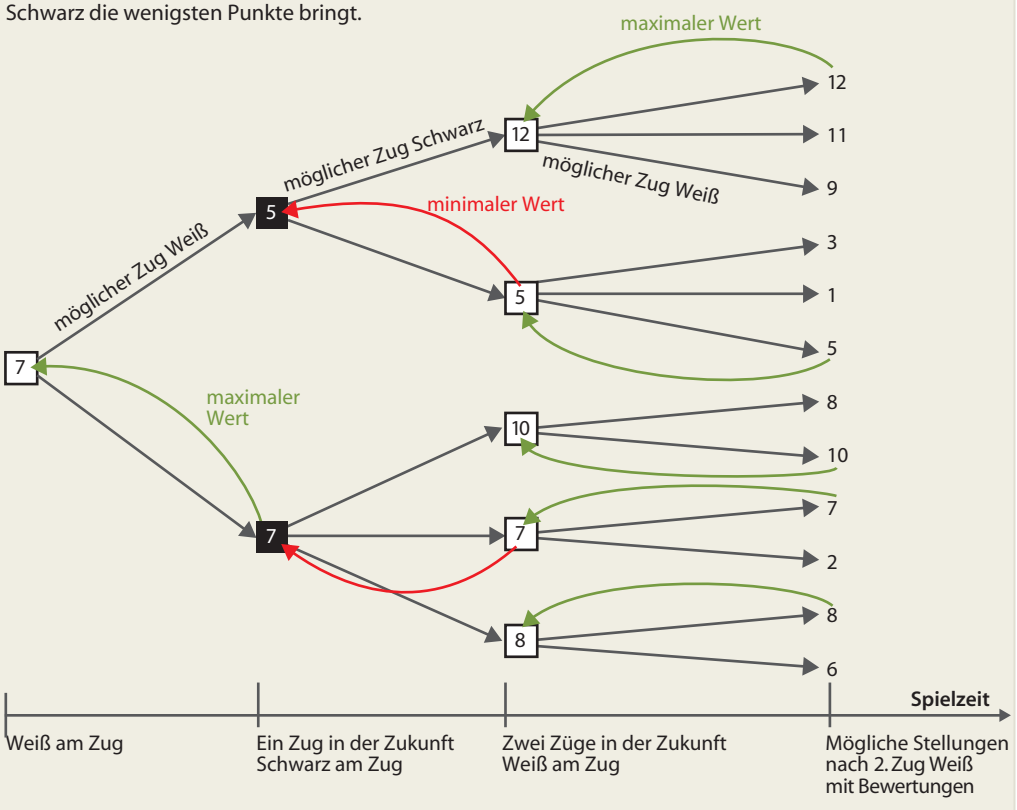
Beim Durchsuchen des Spielbaums nach der besten Fortsetzung geht es also darum, auf jeder Stufe, bei der man selbst an der Reihe ist, die maximale Punktzahl zu finden, und bei jedem Gegnerzug anzunehmen, dass er einem die minimale Punktzahl überlässt. Die so entstehende Rechenvorschrift heißt daher Minimax-Algorithmus.

Es handelt sich dabei um eine Tiefensuche: Bevor der Algorithmus auf einer Stufe einen zweiten Baumknoten betrachtet, rechnet er zuerst den ersten bis in die letzte Verästelung durch. Eine Stellungsbewertung mit Blick aufs Spielbrett findet nur in der untersten Stufe statt. Die Knoten darüber erben jeweils den höchsten beziehungsweise den niedrigsten Wert ihrer Kindknoten.

Gießt man dieses Vorgehen in Programmcode, kommt dabei

Der Minimax-Algorithmus

Die Suche nach dem besten Zug schaut einige Züge in die Zukunft. Wenn Weiß am Zug ist, trachtet sie danach, auf jeder Stufe den Zug zu finden, der Weiß die meisten und Schwarz die wenigsten Punkte bringt.



© Copyright by Heise Zeitschriften Verlag
Persönliches PDF für Frank Baethge aus

```
int MiniMax(Game<TMove> game, int player, int depth)
{
  if(depth == 0)
    return GameValue(game, player);
  int bestValue = -int.MaxValue;
  foreach(var move in game.MovesFor(player)) {
    int value = -MiniMax(game.MakeMove(move), 1 - player, depth - 1);
    if(value > bestValue)
      bestValue = value;
  }
  return bestValue;
}
```

Als C#-Code besteht der Minimax-Algorithmus aus einer einzigen rekursiven Funktion. Sie macht sich die Tatsache zunutze, dass die Stellungsbewertung symmetrisch ist und beim Umrechnen der Wertung von Schwarz' auf Weiß' Sicht nur das Vorzeichen getauscht werden muss.

eine rekursive Funktion heraus, also eine, die sich selbst aufruft. Das Ergebnis zeigt das Listing oben: Die Funktion MiniMax() prüft zunächst, ob depth gleich null ist. Dann ist die letzte Rekursionsstufe erreicht und der Wert der aktuellen Stellung durch einen Aufruf von GameValue() zu berechnen. Anderenfalls iteriert eine foreach-Schleife über sämtliche Züge, die der aktuelle Spieler in der gegenwärtigen Stellung spielen kann. Für jeden ruft die Zeile

```
value = -MiniMax(game.MakeMove(move),
                 1 - player, depth - 1);
```

die MiniMax()-Funktion noch einmal auf, wobei als Spieler 1 – player, also die ID des Gegners, und als depth die um 1 verminderte Rekursionstiefe übergeben wird.

Eine etwas ausführlichere Erklärung verdient das erste Argument game.MakeMove(move). Um es zu verstehen, muss man wissen, dass Instanzen der Game-Klasse als unveränderliche Objekte (engl. Immutables) ausgelegt sind. Das heißt, dass sie über ihre gesamte Lebensdauer die Eigenschaften beibehalten, die sie einmal bei der Erzeugung zugewiesen bekommen. Das hat unter anderem den Vorteil, dass man Zugriffe auf die Eigenschaften nicht synchronisieren muss, wenn sie aus mehreren Threads parallel stattfinden. Nun soll aber ein Game-Objekt den aktuellen Spielstand speichern, und der ändert sich ja mit jedem Zug. Als Ausweg aus diesem Dilemma speichert MakeMove() den neuen Zug nicht in dem vorliegenden Game-Objekt, sondern liefert eine Kopie desselben zurück, in der dieser Zug ausgeführt wurde.

Abgeschaut haben wir dieses Verhalten aus der Immutable-Collections-Bibliothek, die Microsoft seit Kurzem als NuGet-Paket zur Verfügung stellt [2] und die auch in der Game-Klasse selbst zum Einsatz kommt: Die Variable _history speichert hier den bisherigen Spielverlauf in einer ImmutableList<TMove>. Die kennt praktisch alle Methoden, die auch eine gewöhnliche List<T> beherrscht, aber sie funktionieren ein bisschen anders als gewohnt. So fügt etwa Add() nicht der aktuellen Liste das gewünschte Element hinzu, sondern liefert eine neue Liste, an die das Element angefügt wurde. Analog verhalten sich alle anderen Funktionen wie Insert(), Remove(), Replace() und so weiter. Das hört sich zunächst nach übler Speicherverschwendung an, aber das, was diese Funktionen zurückliefern, ist keine plumpe Kopie der Ausgangsliste, sondern eine trickreich über Pointer realisierte Struktur, die einen Großteil des ursprünglichen Inhalts der Liste wiederverwendet.

Solche Klimmzüge sind beim Speichern der aktuellen Belegung eines Othello-Spielfelds in der Klasse CthelloGame nicht nötig: Sie passt in zwei 64-Bit-Ganzzahlen, bei denen gesetzte Bits Felder markieren, auf denen ein schwarzer beziehungsweise ein

Anzeige

© Copyright by Heise Zeitschriften Verlag
Persönliches PDF für Frank Baethge aus

weißer Stein liegt. Diese Werte kopiert MakeMove() einfach in die neue Instanz und schaltet anschließend die betroffenen Bits um.

Neben der Thread-Sicherheit ergibt sich aus der Verwendung des Immutable-Entwurfsmusters für die Anwendung in einer Spiele-Engine noch ein weiterer Vorteil: Wenn das Ausführen eines Zugs den bestehenden Spielstand nicht ändert, braucht man auch keine Züge zurückzunehmen. Der rekursive Aufruf von MiniMax() bekommt einfach den neuen Spielstand übergeben; anschließend wird der nicht mehr benötigt und fällt irgendwann der Garbage-Collection anheim.

Von den Werten aller möglichen Züge hebt MiniMax() in der Variablen bestValue den größten auf und gibt ihn schließlich zurück. In der Rekursionsstufe über dem aktuellen Aufruf kommt also der Wert der Stellung aus der Sicht des Gegners an. Daraus die eigene Bewertung zu berechnen erfordert nur den Wechsel des Vorzeichens: Die Stellungsbewertung ist symmetrisch; eine Stellung, die aus der Sicht von Weiß 5 Punkte wert ist, beurteilt Schwarz mit –5 Punkten. Deshalb steht vor dem rekursiven Aufruf noch ein –.

Baumschnitt

Je weiter ein Programm vorausberechnen kann, wie sich ein Spielzug auswirkt, desto besser wird es spielen. Beliebig lässt sich die Suchtiefe allerdings nicht erhöhen: Ein Othello-Spieler hat beispielsweise im Schnitt jedes Mal rund acht Zugmöglichkeiten, wenn er an der Reihe ist. Das bedeutet, dass sich die Anzahl der zu bewertenden Stellungen mit jedem Halbzug, den man weiter in die Zukunft sieht, etwa verachtfacht. Auf einem Core-i7-Rechner mit 3,4 GHz bewertet das Programm c'thello gut 15 000 Stellungen pro Sekunde, wenn es auf einem Kern läuft. Schon bei einer Suchtiefe von fünf Halbzügen muss man also mit einer Antwortzeit von circa zwei Sekunden rechnen (8^5 = 32 768); lässt man das Programm sechs Halbzüge tief rechnen, wird ein menschlicher Spieler es bereits als langsam empfinden.

Um die Zuganalyse zu beschleunigen, kann man das Programm natürlich so ändern, dass es auf Multi-Core-CPUs alle Kerne nutzt. Im Falle von Othello gewinnt man damit aber frühestens mit acht (logischen) Kernen einen Halbzug Rechentiefe bei gleicher Antwortzeit. Trotzdem haben wir diese Strategie probehalber umgesetzt – in den Quelltexten im Download-Paket zu diesem Artikel ist der entsprechende Aufruf auskommentiert.

Zielführender ist es in jedem Fall, dafür zu sorgen, dass das Programm keine unnötigen Operationen durchführt. Wenn man zum Beispiel das Diagramm auf Seite 178 kritisch betrachtet, dann erkennt man, dass die Bewertung der Stellung ganz rechts unten überflüssig ist: Nachdem das Programm die ersten beiden Zugmöglichkeiten von Schwarz im unteren Ast des Baumes durchgerechnet hat, weiß es, dass Schwarz einen Zug hat, der aus weißer Sicht 7 Punkte wert ist. Würde Schwarz seine dritte Zugmöglichkeit wählen, wäre schon die erste weiße Antwort mehr, nämlich 8 Punkte wert. Schwarz wird diesen Zug also nicht spielen und es ist daher völlig unerheblich, welchen Wert mögliche andere Antworten von Weiß haben.

Zugegeben: Eine von zwölf Zugbewertungen einzusparen wird im Programm keinen Turbo zünden. Wenn man aber eine Prüfung auf derartige Verhältnisse konsequent durchimplementiert, erwischt man auch Stellen weiter oben im Zugbaum und kann sich dann das Bewerten ganzer Äste sparen. Wie das funktioniert, verdeutlicht die nebenstehende Grafik; das dazugehörige Listing steht darüber: Jeder Zugbewertung übergibt man das Intervall, innerhalb dessen der Stellungswert liegen muss, um im darüber liegenden Knoten überhaupt eine Veränderung herbeizuführen. Anfangs werden die Grenzen dieses Intervalls auf minus und plus Unendlich initialisiert. In der Literatur haben sich für die beiden Schranken die Namen Alpha und Beta etabliert; der hier beschriebene Algorithmus heißt „Minimax-Baumsuche mit Alpha-Beta-Cut". In der Grafik zeigen die beiden Zahlen oben links und rechts an jedem Knoten, welchen Wert Alpha und Beta beim Eintritt in die Berechnung dieses Knotens haben.

Wie bei der normalen Minimax-Suche ruft sich die AlphaBeta()-Funktion selbst rekursiv auf, übergibt dabei aber noch die zwei Parameter alpha und beta, und zwar bei jedem Aufruf vertauscht und negiert – MinimaxBrain verwendet ja ein und dieselbe Funktion zum Bewerten von Zügen aus der Sicht von Schwarz und von Weiß.

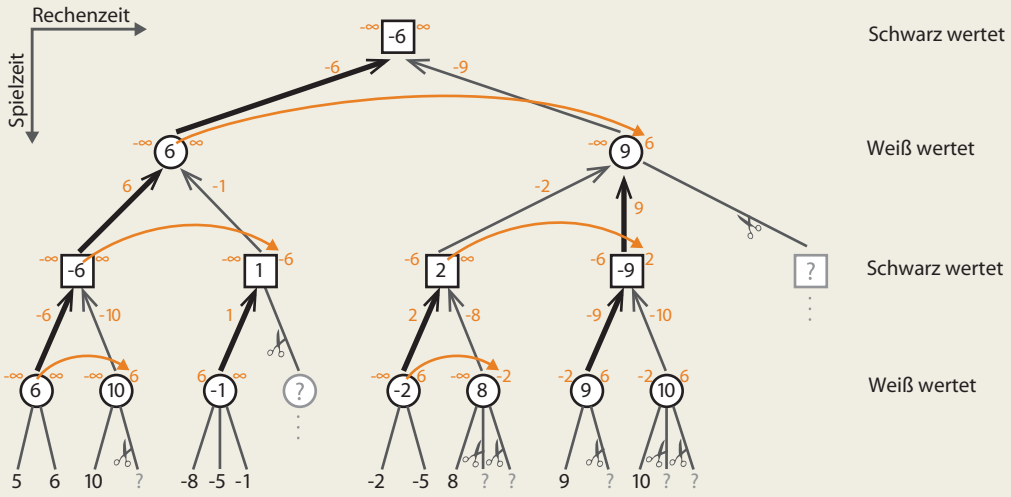
In dem in der Grafik gezeigten Beispiel ist der erste bewertete Knoten der weiße (runde) ganz

```
int AlphaBeta(Game<TMove> game, int player, int depth, int alpha, int beta)
{
  if(depth == 0)
    return GameValue(game, player);
  int bestValue = -int.MaxValue;
  foreach(var move in game.MovesFor(player)) {
    int value = -AlphaBeta(game.MakeMove(move), 1 - player, depth - 1,
                           -beta, -alpha);
    if(value > bestValue)
      bestValue = value;
    if(value > alpha) {
      alpha = value;
      if(alpha >= beta)
        break;
    }
  }
  return bestValue;
}
```

Im Vergleich mit der einfachen Minimax-Suche braucht die Alpha-Beta-Suche zwei Parameter mehr, die sie jeweils vertauscht und mit umgekehrtem Vorzeichen an die nächste Rekursionsstufe weitergibt.

Alpha-Beta-Cut

Mit dem sogenannten Alpha-Beta-Cut kann man die Minimax-Suche deutlich beschleunigen, indem man auf die Berechnung von Zügen verzichtet, die der Gegner bei bester Spielweise ohnehin nicht machen würde.



© Copyright by Heise Zeitschriften Verlag
Persönliches PDF für Frank Baethge aus

links unten, der für Weiß zwei Fortsetzungen mit den Werten 5 und 6 bietet, selbst also 6 Punkte wert ist. Aus der Sicht von Schwarz sind das –6 Punkte. Das ist mehr als das aktuelle Alpha, das noch minus unendlich enthält. Daher erhöht Schwarz dieses Alpha auf –6 und merkt sich damit: Sollte es zu dieser Spielsituation kommen, macht Weiß im nächsten Zug mindestens –6 Punkte. Mit anderem Vorzeichen versehen und als Beta an die nächste weiße Bewertung übergeben, weiß Weiß somit: Schwarz hat auf dieser Ebene einen anderen Zug, der mir 6 Punkte bringt. Die erste Fortsetzung der weißen Stellung ist 10 Punkte und damit mehr als das aktuelle Beta wert. Folglich wird Schwarz diesen Zug nicht spielen; es lohnt sich also nicht, die weiteren Fortsetzungszüge zu betrachten.

Eine Ebene höher finden dann dieselben Überlegungen mit vertauschten Vorzeichen und vertauschten Farben statt: Weiß merkt sich, dass Schwarz einen 6-Punkte-Zug hat, Schwarz bekommt bei seinem als Nächstes zu bewertenden Zug ein Beta von –6 übergeben. Er stellt fest, dass der erste eigene Zug mehr, nämlich 1 Punkt wert ist, und verzichtet daher auf das Durchrechnen der weiteren möglichen Züge. In der rechten Hälfte des Spielbaums findet ein Beta-Cut sogar noch eine Etage höher statt und schließt so noch mehr potenzielle Züge von der Berechnung aus.

In unserem c'thello-Programm leistet die Optimierung der Zugbewertung durch den Alpha-Beta-Cut Erstaunliches: Im Schnitt muss das Programm nur noch etwa ein Zwanzigstel der Züge bewerten, die es bei simpler Minimax-Suche abarbeitet. Noch einmal fast halbieren lässt sich die Anzahl der bewerteten Stellungen, indem man die jeweils möglichen Züge eines Spielers sinnvoll vorsortiert: Statt sie naiv von links oben nach rechts unten zu prüfen, verarbeitet man sie besser von innen nach außen. Dadurch werden auf jeder Stufe zunächst die Züge bewertet, die voraussichtlich eine recht ähnliche Punktzahl erreichen. Ein Schnitt im Suchbaum findet ja auch schon dann statt, wenn ein Zug dieselbe Bewertung erhält wie ein früherer Zug auf gleicher Ebene.

Eine Parallelisierung der Suche mit Alpha-Beta-Cut lohnt sich kaum: Startet man die Bewertung mehrerer Züge einer Ebene gleichzeitig, weiß man ja noch gar nicht, ob man den Wert des zweiten überhaupt benötigen wird. So lastet man dann zwar alle Prozessorkerne aus, aber der Löwenanteil der Arbeit, die sie verrichten, ist für die Katz. Lohnenswert ist es allenfalls, die Zugbewertung der ersten Ebene parallel zu starten, weil hier ja keine Schnitte zu erwarten sind. Man muss dann aber trotzdem damit leben, dass die Berechnung einiger Züge länger dauert als bei anderen, weil sie mehr Erwiderungen ermöglichen. Ein Ergebnis liegt immer erst dann vor, wenn die letzte Berechnung ein Ergebnis hat.

Spieltrieb

Wenn Sie sich das c'thello-Paket für eigene Experimente herunterladen, werden Sie feststellen, dass das Programm in der vorliegenden Fassung akzeptabel, aber nicht wirklich stark spielt. Das liegt vor allem an der einigermaßen primitiven Bewertungsfunktion für die einzelne Stellung – sie ist stark inspiriert von dem Artikel zu Computer-Othello in der deutschen Wikipedia [3]. Wenn Sie eine stärkere Bewertung implementieren wollen, müssen Sie sich die Funktion `CthelloGame.GetEvaluator()` vornehmen; die im Ausgangszustand verwendete Bewertung steckt im `case 2` der dort vorhandenen `switch`-Abfrage.

Eine noch größere Herausforderung ist es natürlich, ein komplett eigenes Spiel zu programmieren. Zumindest die grundlegenden Denk-Mechanismen können Sie dabei aber eins zu eins aus unserem Framework übernehmen. (hos)

Literatur

[1] Computer-Olympiade 2013 der International Computer Games Association (ICGA) in Yokohama, Japan: www.grappa.univ-lille3.fr/icga/event.php?id=44

[2] Preview of Immutable Collections Released on NuGet: http://blogs.msdn.com/b/bclteam/archive/2012/12/18/preview-of-immutable-collections-released-on-nuget.aspx

[3] Computer-Othello: http://de.wikipedia.org/wiki/Computer-Othello

www.ct.de/1410176 c't

Anzeige

© Copyright by Heise Zeitschriften Verlag
Persönliches PDF für Frank Baethge aus