KI-Praktikum

M. Helmert, R. Mattmüller, T. Keller
Sommersemester 2009

Universität Freiburg
Institut für Informatik

Projekt 2: Spiele

Abgabe: 21. Juni 2009

Präsenzaufgaben

Aufgabe 1

Beim *Nim-Spiel* sind mehrere Reihen mit Streichhölzern vorhanden. Zwei Spieler nehmen abwechselnd Streichhölzer aus einer der Reihen weg. Wie viele sie nehmen, spielt keine Rolle, es muss aber mindestens ein Streichholz sein und es dürfen bei einem Zug nur Streichhölzer einer einzigen Reihe genommen werden. Derjenige Spieler, der den letzten Zug macht, also die letzten Streichhölzer wegnimmt, gewinnt.
In der Datei `nim.py` sind die Nim-Regeln kodiert, und `minimax_player.py` enthält die Implementierung eines einfachen Minimax-Spielers. In `main.py` werden einige Nim-Anfangszustände erstellt und Spiele zwischen Minimax-Spielern gestartet.

(a) Führen Sie `main.py` aus und tragen Sie Gewinner, Laufzeiten und Knotenexpansionen (rekursive Minimax-Aufrufe) für die vorgegebenen Nim-Spiele (`nim_1`, ..., `nim_5`) in die mit „Aufgabe 1“ überschriebene Spalte in der unten angegebenen Tabelle 1 ein. Gemeint sind jeweils Laufzeiten (in Sekunden) und Knotenexpansionen für den Top-Level-Minimax-Aufruf von Spieler MAX.

(b) Die Datei `nim.py` enthält bereits eine Unterklasse `OneHeapNim` von `Nim`, in der die Regeln der folgenden Nim-Variante kodiert werden sollen: Es gibt statt mehreren Reihen nur eine Reihe von Streichhölzern, aber dafür dürfen in jedem Zug nur maximal drei Streichhölzer weggenommen werden. Passen Sie die Methode `get_successors` von `OneHeapNim` so an, dass nur die entsprechenden Nachfolger erzeugt werden (`OneHeapNim` enthält bereits eine Implementierung von `get_successors`, es handelt sich dabei jedoch nur um eine exakte Kopie der Methode aus der Basisklasse `Nim`).

(c) Kommentieren Sie in `main.py` den Codeabschnitt ein, in dem die Instanzen von `OneHeapNim` gespielt werden, und führen Sie `main.py` erneut aus. Vervollständigen Sie den Teil von Tabelle 1, der mit „Aufgabe 1“ überschrieben ist.

Aufgabe 2

Die Datei `alpha_beta_player.py` enthält eine von `MinimaxPlayer` abgeleitete Klasse `AlphaBetaPlayer`, in der die Methoden `max_value` und `min_value` überschrieben werden. In der Datei, die Sie ausgecheckt haben, sind die Rümpfe der Methoden `AlphaBetaPlayer.max_value` und `AlphaBetaPlayer.min_value` exakte Kopien aus `MinimaxPlayer.max_value` und `MinimaxPlayer.min_value`.

(a) Passen Sie die Implementierung so an, dass tatsächlich α-β-Pruning stattfindet.

(b) Ändern Sie `main.py` so, dass nun statt einfachen Minimax-Spielern Spieler mit α-β-Pruning gegeneinander antreten, messen Sie erneut Laufzeiten und Knotenexpansionen bei den vorgegebenen Nim-Instanzen und vervollständigen Sie Tabelle 1.

Aufgabe 3

In `chess.py` ist eine Schachvariante (Minichess) implementiert, die sich vom bekannten Schach durch folgende Vereinfachungen unterscheidet:

- Das Schachbrett ist kleiner (bei uns 4×4 Felder).
- Es sind weniger Figuren auf dem Brett (bei uns höchstens Könige, Türme, Springer und Läufer).
- Die komplizierten Remis-Regeln, bei denen die Historie des Spiels eine Rolle spielt, fallen weg, dafür endet jede Partie nach einer vordefinierten Anzahl von Halbzügen (bei uns zwischen 5 und 8) mit Remis, falls sie nicht bereits zuvor beendet wurde.

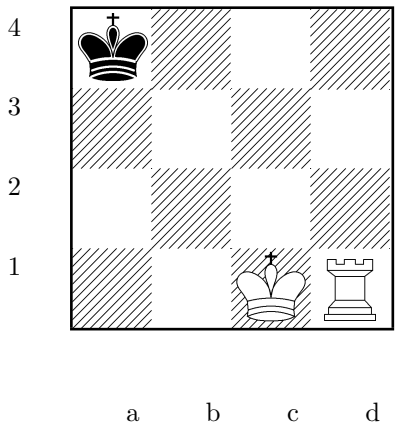
Ziel ist es weiterhin, mit den üblichen erlaubten Zügen den gegnerischen König mattzusetzen.

Drei Beispiele für Minichess-Instanzen sind in Abbildung 1 angegeben. In diesen Beispielen hat Weiß Gewinnstrategien in zwei bis vier Zügen, etwa in Abbildung 1(a):

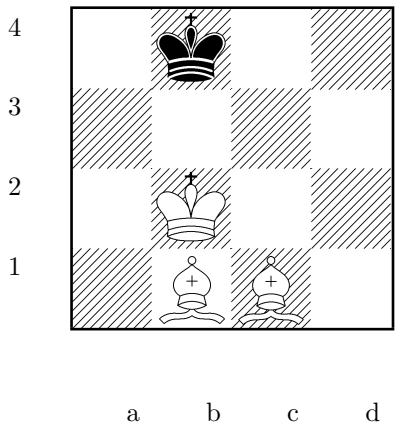
1. Kc1-b2 Ka4-b4
2. Td1-d4#

Wie in vielen anderen Spielen leidet der Minimax-Algorithmus auch bei Minichess darunter, dass identische Positionen (d. h. im Fall von Minichess identische Stellung der Figuren, gleicher Spieler am Zug, noch gleich viele verbleibende Züge bis zum Spielabbruch) entlang mehrerer Zugfolgen erreicht werden können und deshalb ganze Teilbäume des Spielbaumes unnötigerweise mehrfach untersucht werden. Abhilfe schaffen sogenannte *Transpositionstabellen*, also Hashtabellen, in denen für bereits untersuchte Zustände deren Werte und ggf. die besten Züge gespeichert werden.

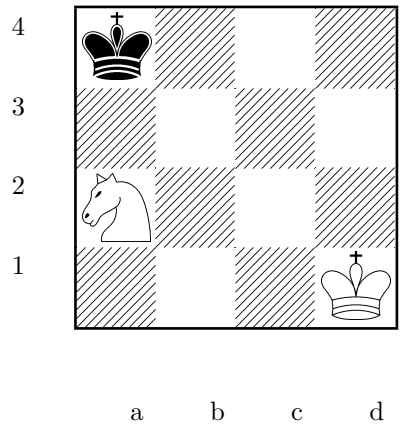
(a) Kommentieren Sie den entsprechenden Teil von `main.py` ein und messen Sie Laufzeiten und Knotenexpansionen von reinen Minimax-Spielern ohne



(a) `chess_1`: Matt in zwei Zügen



(b) `chess_2`: Matt in drei Zügen



(c) `chess_3`: Matt in vier Zügen

Abbildung 1: Minichess-Probleme. Weiß am Zug.

Transpositionstabellen bei den vorgegebenen Minichess-Instanzen. Tragen Sie die Messungen in die dafür vorgesehene Tabelle 2 ein. *Hinweis:* Sie können informativere Ausgaben über die Züge der Spieler erhalten, indem Sie in der Datei `game.py` das Flag `DEBUG` auf `True` setzen.

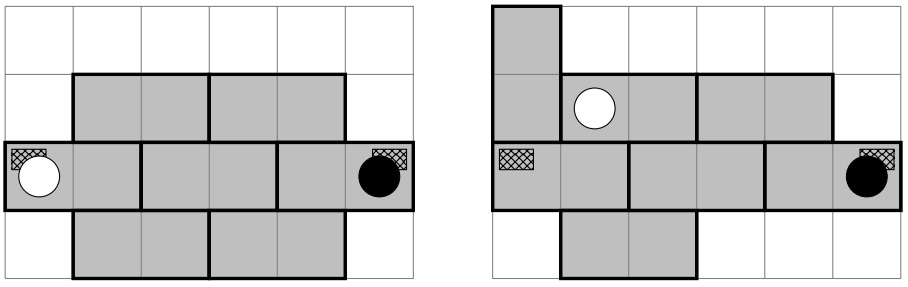
(b) In der Datei `transposition_table_player.py` befindet sich, analog zur vorherigen Aufgabe zum α-β-Pruning, eine unvollständig implementierte Klasse `TranspositionTablePlayer`. Passen Sie dort die Implementierung der Methoden `max_value` und `min_value` so an, dass das *Dictionary* `transposition_table` als Transpositionstabelle verwendet wird. Sie dürfen voraussetzen, dass alle Minichess-Zustände über eine Methode `hash_code()` verfügen, die einen eindeutigen Hashcode zurückliefert, unter dem die Zustände bzw. ihre Werte in der Hashtabelle abgelegt werden können.

(c) Führen Sie die Messungen aus dem ersten Aufgabenteil nun erneut mit `TranspositionTablePlayer`s anstelle von `MinimaxPlayer`s durch und vervollständigen Sie Tabelle 2.

Aufgabe 4

Im Spiel *Mouse Island* (Abb. 2) schlüpfen zwei Spieler in die Rolle je einer Maus, die beide auf derselben Insel festsitzen. Beiden ist es (vor Spielbeginn) gelungen, den Käse der jeweils anderen zu stehlen, aber sie stellen fest, dass dieser bei Weitem nicht so gut schmeckt wie ihr eigener. Das Ziel des Spieles ist es daher, den eigenen Käse (der noch im Besitz des Gegenspielers ist) zurückzuerobern. Jede Maus startet also am Ziel ihres Kontrahenten und versucht, die Insel schneller zu überqueren als der Gegenspieler. Die Insel wird dabei aus domino-ähnlichen Spielsteinen aufgebaut, die je zwei Felder bedecken.
Jeder Spielzug gliedert sich in zwei Teile:

- Der aktive Spieler zieht seine Maus ein Feld weiter. Dabei existieren die acht Möglichkeiten Nord (N), Nordost (NO), Ost (O), Südost(SO), Süd



(a) Anfangszustand (b) Möglicher Zustand nach Zug von Weiß

Abbildung 2: Mouse Island.

(S), Südwest(SW), West (W) und Nordwest (NW). Es darf nicht auf Felder im Meer oder auf das Feld gezogen werden, das mit der gegnerischen Maus besetzt ist.

- Nach dem Ziehen darf der aktive Spieler (muss aber nicht) zusätzlich einen Spielstein, auf dem sich weder eine Maus noch ein Stück Käse befindet, aus dem Spiel entfernen oder aber so umlegen, dass alle Dominosteine, die die Insel bilden, weiterhin eine Zusammenhangskomponente bilden (Verbindungen an einzelnen Eckpunkten zählen auch).

Das Spiel endet nach 50 Halbzügen mit einem Unentschieden, wenn es bis dahin keinen Gewinner gibt.
Eine Implementierung des Spiels finden Sie in der Datei `mouse_island.py`. Das Spiel ist zu komplex, um den Spielbaum bis zu einer Tiefe vollständig zu durchsuchen, die ausreicht, um den Sieger zu bestimmen. Es bleibt also nichts anderes übrig, als lediglich eine tiefenbeschränkte Minimax-Suche durchzuführen und auf Knoten am Suchhorizont eine geeignete Bewertungsfunktion anzuwenden, mit der wir abschätzen können, wie gut der Knoten für Spieler MAX ist.

(a) Passen Sie analog zu den vorigen Aufgaben die Methoden `max_value` und `min_value` der Klasse `DepthLimitedPlayer` so an, dass nach Erreichen einer vorgegebenen Tiefenbeschränkung die Rekursion abgebrochen und die Bewertungsfunktion `evaluation_function`, die im Konstruktor übergeben wurde, auf die Knoten am Suchhorizont angewendet wird.

(b) Solange bei der Erstellung eines `DepthLimitedPlayer`s keine geeignete Bewertungsfunktion übergeben wird, verwendet der Spieler die Funktion, die jeden Zustand mit 0 bewertet. Entwerfen und implementieren Sie eine für Mouse Island geeignete Bewertungsfunktion, lassen Sie damit zwei `DepthLimitedPlayer`s mit unterschiedlichen Tiefenbeschränkungen (gemäß der Vorgaben in Tabelle 3) gegeneinander antreten, und ergänzen Sie die fehlenden Einträge in der Tabelle. Testen Sie ferner einen Spieler mit Ihrer Bewertungsfunktion und Tiefenbeschränkung 2 im Vergleich zu einem Spieler mit Standard-Bewertungsfunktion und Tiefenschranke 1. *Hinweis:* Die Klasse `MouseIslandState`, deren Instanzen Ihre Bewertungsfunktion als Argument annehmen soll, ist am Anfang der Datei `mouse_island.py` definiert. Die Attribute `WIDTH` und `HEIGHT` sind bei uns fest (6 bzw. 4), `tiles` ist eine Liste der Spielsteine, wobei jeder Spielstein durch ein Paar von benachbarten Feldern $((x, y), (z, w))$ definiert ist, `board` ist eine zweidimensionale `WIDTH` $\times$ `HEIGHT`-Matrix mit Booleschen Einträgen (`True` gdw. an der entsprechenden Position Land und `False` gdw. an der entsprechenden Position Wasser ist), `mice` ist eine Liste mit zwei Einträgen, wobei `mice[0]` $= (x, y)$ die Position der weißen und entsprechend `mice[1]` die Position der schwarzen Maus ist, `player` ist der Spieler am Zug (`0` für Weiß, `1` für Schwarz) und `step` gibt an, wie viele Halbzüge schon gemacht wurden. Die Zielpositionen der Mäuse (Käsestücke) sind $(5, 1)$ für Weiß und $(0, 1)$ für Schwarz, die maximale Zahl der Halbzüge beträgt 50.

	Aufgabe 1 Ohne α-β-Pruning		**Aufgabe 2 Mit α-β-Pruning**	
Spiel	Knoten	Laufzeit	Knoten	Laufzeit
`nim_1`				
`nim_2`				
`nim_3`				
`nim_4`				
`nim_5`				
`one_heap_nim_1`				
`one_heap_nim_2`				
`one_heap_nim_3`				
`one_heap_nim_4`				
`one_heap_nim_5`				

Tabelle 1: Knotenerzeugungen und Laufzeiten ohne und mit α-β-Pruning.

	Ohne Transpositionstabelle		**Mit Transpositionstabelle**	
Spiel	Knoten	Laufzeit	Knoten	Laufzeit
`chess_1`				
`chess_2`				
`chess_3`				

Tabelle 2: Knotenerzeugungen und Laufzeiten ohne und mit Transpositionstabelle.

		Spieler 1		**Spieler 2**		
Spieler 1	Spieler 2	Knoten	Laufzeit	Knoten	Laufzeit	Gewinner
$DL(1)$	$DL(1)$					
$DL(1)$	$DL(2)$					
$DL(2)$	$DL(1)$					
$DL(2)$	$DL(2)$					

Tabelle 3: Tiefenbeschränkte Tiefensuche bei Mouse-Island. $DL(d)$: Tiefenbeschränkte Minimax-Suche mit Tiefenschranke $d > 0$. Knotenexpansionen und Laufzeiten jeweils für ersten Aufruf von `max_value` bzw. `min_value` des betreffenden Spielers.

Hausaufgaben

Aufgabe 5

Unter `http://www.chessvariants.com/ms.dir/congo.html` können die Spielregeln für ein weiteres, von Schach abgeleitetes Zweipersonenspiel namens `Congo` gefunden werden. Eine Basisklasse für einen `Congo`-Spieler ist durch die Klasse `CongoPosition` in `position.py` gegeben. Implementieren Sie darauf aufbauend einen Spieler, der zumindest den in `test_players.py` gegebenen Spieler zuverlässig schlagen sollte. Die meisten notwendigen Änderungen sollten sich auf Code in dem Modul `position.py` beziehen. Die Datei `advanced_player.py` enthält bereits ein Grundgerüst einer von `CongoPosition` abgeleiteten Klasse `AdvancedPlayer`. Implementieren Sie dort ihre Verbesserungen, indem Sie geeignete Basisklassen-Methoden überladen. Das Modul `position.py` enthält den Suchalgorithmus und die Bewertungsfunktion, die im Moment beide recht bescheiden gehalten sind: Der Suchalgorithmus ist eine einfache Minimax-Suche, die Bewertungsfunktion berechnet die Differenz zwischen der Zahl der weißen und schwarzen Figuren. In `position.py` ist der „Standardspieler“ implementiert. Beispiele für weitere von `CongoPosition` abgeleitete Spieler befinden sich im Modul `test_players.py`.
Um einen Spieler zu testen, verwenden Sie `match.py`. Ruft man `match.py` ohne Parameter auf, treten zwei Versionen des Standardspielers mit je einer Sekunde Bedenkzeit pro Zug gegeneinander an. Durch Übergabe von Argumenten über die Kommandozeile kann man die Spieler konfigurieren. So tritt z. B. mit dem Aufruf `./match.py "MinimaxPlayer(5)" "ZufimaxPlayer(10)"` der Standardspieler (der auf den Namen `MinimaxPlayer` hört) mit 5 Sekunden Bedenkzeit pro Zug als Weiß gegen den im Modul `test_players` implementierten ZufimaxPlayer mit 10 Sekunden Bedenkzeit pro Zug als Schwarz an. Eigene Spielerklassen sind automatisch verfügbar, sofern sie im Modul `test_players` definiert wurden. Ansonsten muss man die entsprechenden Klassen innerhalb von `match.py` importieren, so wie dies für die Klassen in `test_players` bereits geschieht.
Der Computer gerät häufiger in eine endlose Zugwiederholung; in diesem Fall einfach das Programm mit Strg+C abbrechen.
Welche in den Präsenzaufgaben vorgestellten Techniken (α-β-Pruning, Transpositionstabellen, Tiefenbeschränkter Minimax, Bewertungsfunktion) oder darüber hinausgehende wie z. B. Move-Ordering, History-Heuristik oder Quiescence Search dabei implementiert werden, bleibt Ihnen selbst überlassen. Die Gruppe, deren Spieler im direkten Vergleich aller Spieler untereinander am erfolgreichsten abschneidet, erhält einen Bonuspunkt.
Ein paar Ideen, die über die Präsenzaufgaben hinausgehen:

- Bessere Bewertungsfunktion: Ein Bauer sollte nicht denselben Wert haben wie ein Elefant. Dazu evtl. das Modul `pieces.py` modifizieren, um mit den Figuren unterschiedliche Werte zu assoziieren. Auch Aspekte, die nicht

direkt mit dem Figurenmaterial zu tun haben, könnten und sollten in die Bewertungsfunktion eingehen (z.B. Mobilität).

- Zufall: Oft werden Computerspieler allein schon dadurch besser, dass sie eine Zufallskomponente enthalten. Man könnte dazu entweder die Züge in zufälliger Reihenfolge betrachten (die Reihenfolge macht einen Unterschied im Zusammenhang mit α-β-Suche) oder bei der Bewertungsfunktion einen kleinen zufälligen Betrag addieren.

Aufgabe 6

Laden Sie unter `http://www.zillions-of-games.com/demo/` die Zillions-of-Games-Demoversion (ZoG) herunter und installieren Sie diese auf einem Windows-System. Folgen Sie nach dem Start der Anwendung den Anweisungen, welche in der Datei `zillions_key.txt` aus dem Repository angegeben sind, um die Vollversion von ZoG freizuschalten. Da die Universität nur über eine begrenzte Anzahl von Lizenzen verfügt, bitten wir Sie, ZoG nach Beendigung des Praktikums wieder zu deinstallieren.
Unter Help → Rules File Language Reference ist die Modellierungssprache von ZoG beschrieben. Modellieren Sie damit sowie den im Repository im Ordner `zog` gegebenen Grafiken (Spielbrett und Figuren) das Spiel „Baden vs. Schwaben“ mit den folgenden Regeln:

- Gespielt wird auf einem Brett mit 8×8 Feldern derselben Farbe.
- Jeder Spieler besitzt zu Beginn des Spieles 16 Wappenträger, die in den beiden untersten bzw. obersten Reihen platziert werden (vgl. Abb. 3)
- Die Spieler sind abwechselnd am Zug. Ein Wappenträger kann entweder um ein Feld nach vorne oder schräg nach vorne bewegt werden.
- Schlagen kann ein Wappenträger einen gegnerischen wie der Bauer im Schach, also nur schräg nach vorne.
- Gewonnen hat der Spieler, der zuerst mit einer seiner Figuren die hinterste Reihe des Gegenspielers erreicht.

Aufgabe 7

Modellieren Sie „Baden vs. Schwaben“ auch in Python, wofür Sie weite Teile des `Congo`-Codes aus Aufgabe 5 oder des Codes der Präsenzaufgaben wiederverwenden können, und implementieren Sie einen dazugehörigen Spieler. Starten Sie diesen sowie ZoG parallel, und lassen die beiden gegeneinander antreten. Dafür müssen Sie die Züge des jeweils anderen Programms von Hand über den menschlichen Spieler (der im Code der Präsenzaufgaben unter `human_player.py` gegeben ist) simulieren. Ihre Pythonimplementierung sollte bei 20 Sekunden Bedenkzeit pro Zug den ZoG-Spieler auf der schwächsten Stufe (Pushover) zuverlässig schlagen.
Checken Sie neben dem Code auch ein Protokoll des Spielablaufs zwischen Ihrem Spieler und ZoG als Textdatei ins Repository ein.

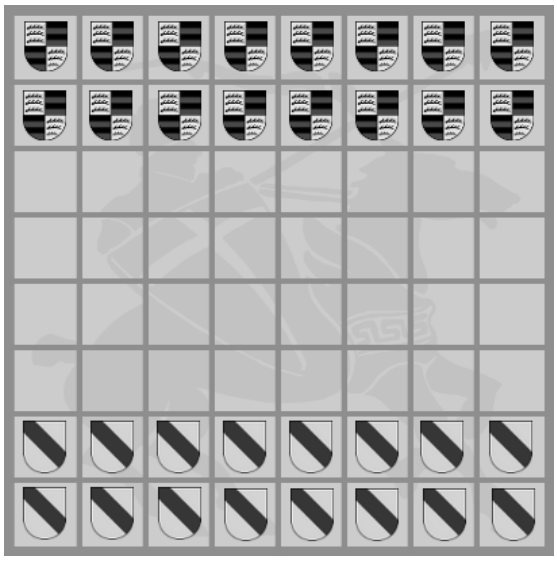


Abbildung 3: Spielfeld und Startaufstellung von „Baden vs. Schwaben“