TU Berlin, Fakultät IV
Blankertz/Röhr/Schlegel
MAR 4-3

Aufgabenblatt 4

Alpha-Beta Suche

Abgabe (bis 29.05.2022 23:59 Uhr)

Die folgenden Dateien müssen für eine erfolgreiche Abgabe im git Ordner eingecheckt sein:

`Blatt04/src/Board.java`	Aufgabe 3.1
`Blatt04/src/TicTacToe.java`	Aufgabe 3.2 und 3.3

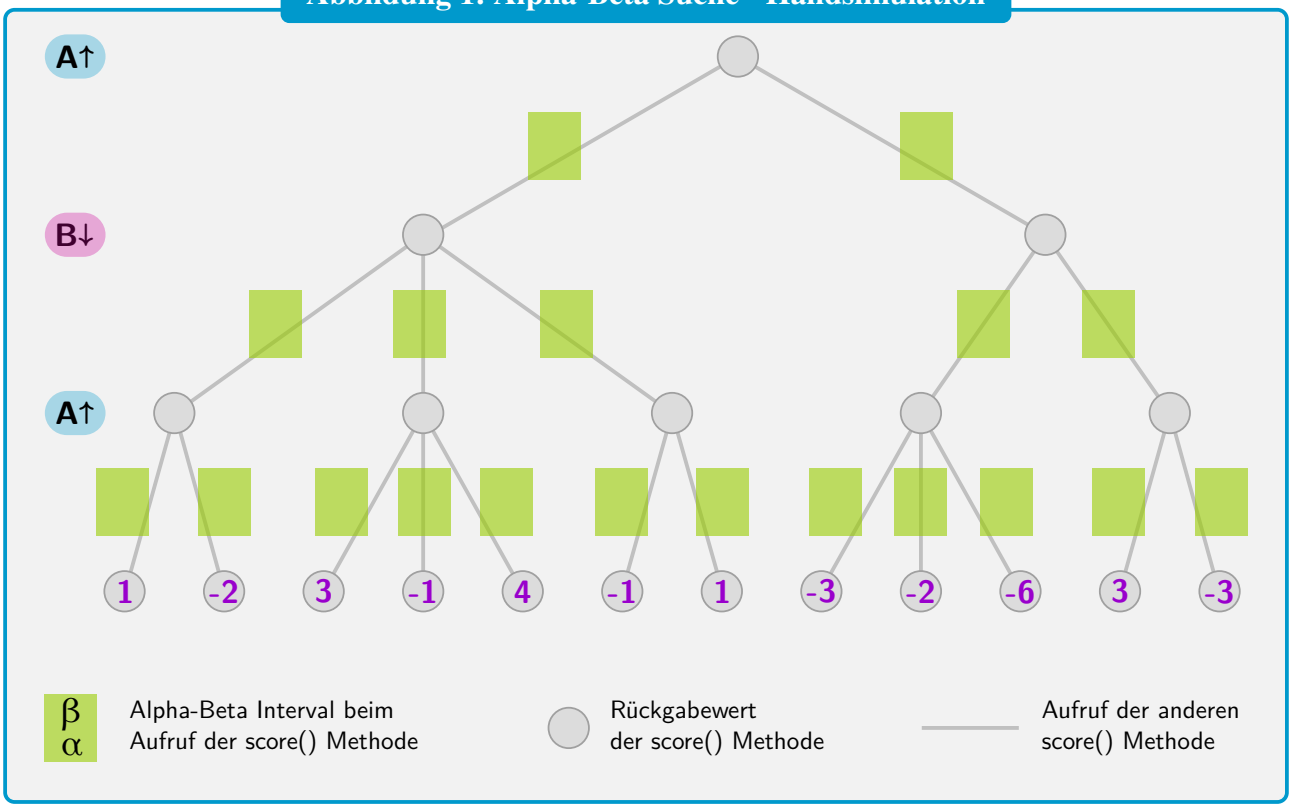
Als Abgabe wird jeweils nur die letzte Version im git gewertet.

Was Sie nach diesem Blatt wissen sollten:

- Wie funktioniert der Minimax Algorithmus?
- Wie hängen die Alpha-Beta Suche und der Minimax Algorithmus zusammen?
- Was macht die score() Methode in der Alpha-Beta Suche? Wofür stehen α und β?
- Wie funktioniert die Alpha-Beta Suche?
- Was ist die Negamax-Variante der Alpha-Beta Suche?
- Wie funktioniert Branch and Bound?

Aufgabe 1: Alpha-Beta Suche (Klausurvorbereitung)

Abbildung 1: Alpha-Beta Suche - Handsimulation



1.1 Wie funktioniert die Alpha-Beta Suche? Was beschreiben α und β? Was beschreibt jeweils der α- oder β-Cutoff?

1.2 Vervollständigen Sie Abbildung 1.

Aufgabe 2: Branch and Bound (Klausurvorbereitung)

2.1 Welche Konzepte von Algorithmen haben Sie bereits kennengelernt für Bäume der Teillösungen?

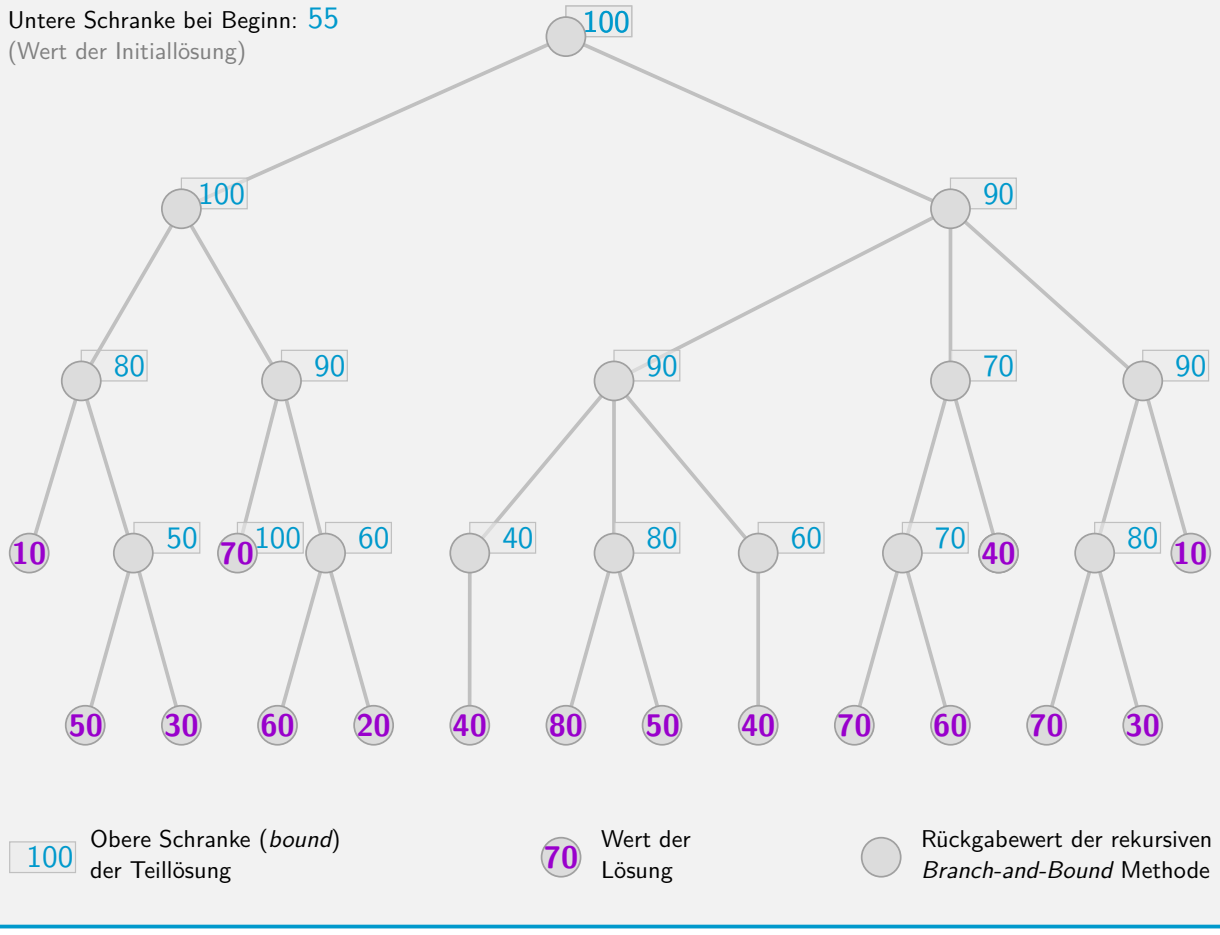
2.2 Wie funktioniert Branch and Bound? Erklären Sie die grundlegenden Ideen.

2.3 Vervollständigen Sie Abbildung 2.

Aufgabe 3: Tic-Tac-Toe (Hausaufgabe)

Es soll ein Programm geschrieben werden, das beliebige Stellungen im Tic-Tac-Toe Spiel bewertet. Bei Gewinnpositionen ist zu berücksichtigen, nach wieviel Zügen der Gewinn (bei optimal spielender Gegnerin) erreicht werden kann. Analoges gilt für Verlustpositionen. Eine genauere Erklärung wird unten gegeben.

Abbildung 2: Branch and Bound - Handsimulation



Tic-Tac-Toe wird üblicherweise auf einem 3×3 Brett gespielt. Startend von einem leeren Brett markieren die Spielenden abwechelnd jeweils ein freies Feld mit ‘x’ bzw. ‘o’. Es gewinnt, wer zuerst eine Dreierreihe (horizontal, vertikal oder diagonal) vervollständigt, siehe Abb. 1.

Die Implementation soll beliebige $n \times n$ Bretter zulassen, wobei $n \leq 10$ vorausgesetzt werden kann und das Spiel gewonnen ist, wenn n gleiche Steine eine Reihe bilden.

3.1 Implementation der Klasse `Board` (40 Punkte)

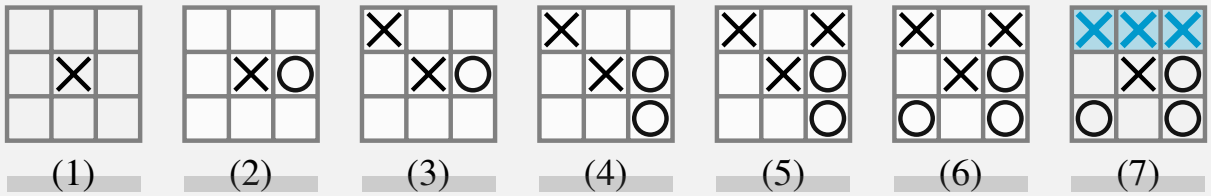
Betrachten Sie die gegebene Klasse `Position` und implementieren Sie die Klasse `Board` gemäß der folgenden API:

public class Board		
	`Board(int n)`	Erzeugt `Board` Objekt, ein Spielbrett der Größe $n \times n$ für $1 \leq n \leq 10$.
`int`	`nFreeFields()`	Gibt die Anzahl freier Felder zurück.
`void`	`setField(Position pos, int token)`	Setzt `token` auf Spielfeld `pos`, wobei `token` die Werte `0` (freies Feld), `-1` (Spielstein ’o’) oder `1` (Spielstein ’x’) sein kann.
`int`	`getField(Position pos)`	Gibt Inhalt des Feldes `pos` zurück.
`void`	`doMove(Position pos, int player)`	Führt einen Spielzug von Spielerin `player` (`1` oder `-1`) auf Feld `pos` aus.
`void`	`undoMove(Position pos)`	Macht den Zug auf Feld `pos` rückgängig, indem sie das Feld `pos` leert.
`Iterable<Position>`	`validMoves()`	Gibt alle Felder, auf die gezogen werden kann, zurück.
`boolean`	`isGameWon()`	Wurde das Spiel durch den letzten Zug per `doMove()` gewonnen?

Bemerkungen:

- Der Konstruktor `Board(int n)` sollte eine `InputMismatchException` werfen, wenn die Dimensionen des Spielbretts größer sind als 10 oder mit diesen Dimensionen kein Spielbrett erstellt werden kann.
- Die Methoden `setField()` und `getField()` müssen eine `InputMismatchException` werfen, wenn die Eingabeargumente keine validen Werte haben. Die Methode `setField()` überprüft explizit nicht, ob sich an der Position `pos` ein freies Feld befindet. Es handelt sich hierbei eher um eine klassische Setter-Methode für das Board. Das heißt, es muss ein valides Token, auf eine valide Position gesetzt werden, die Spielregeln werden hier nicht überprüft. Ob die Position frei ist, können Sie in der doMove()-Methode überprüfen und dann eine Exception werfen, das aber wird nicht von den Tests überprüft.
- Aus Effizienzgründen sollte die Anzahl der freien Felder oder die Anzahl der gesetzten Steine in einer Variable gespeichert werden, die in der Methode `setField()` aktualisiert wird. Dann müssen nicht bei jedem Aufruf von `nFreeFields()` alle Felder durchsucht werden. Beachten Sie, dass das Tauschen von Steinen durch die Methode `setField` erlaubt ist und die Anzahl der freien Felder dadurch nicht beeinträchtigt werden sollte.
- Die Methode `undoMove()` braucht kein Gedächtnis zu haben.
- Die Methode `validMoves()` braucht nicht zu prüfen, ob das Spiel bereits beendet ist. Sie kann einfach alle Positionen freier Felder zurückgeben.
- Sie sollten auch überlegen, wie Anfragen von `isGameWon()` effizient beantwortet werden können.
- Zum Debuggen kann es hilfreich sein, zusätzlich eine Methode `print()` zu implementieren, die das aktuelle Brett ausgibt.

Abbildung 3: Tic-Tac-Toe Spiel



Es wird abwechselnd gezogen, mit dem Ziel, eine Dreierreihe horizontal, vertikal oder diagonal zu besetzen. Hier endet das Spiel nach sieben Zügen. Da 2 Felder freibleiben, bekommt die Endstellung (7) in unserer Bewertung 3 Punkte und somit gilt dies aus Sicht von 'x' auch für Stellung (6). Ebenso bekommt Stellung (5) aus Sicht von 'o' -3 Punkte, da ein Sieg von 'x' mit 2 freien Feldern nicht vermieden werden kann. Stellung (1) hat hingegen aus Sicht von 'o' 0 Punkte und nicht -3, da es einen besseren Zug für 'o' gibt, mit dem ein Untentschieden erreicht werden kann.

3.2 Implementation der Klasse `TicTacToe` (50 Punkte)

Implementieren Sie die Alpha-Beta-Suche zur Bewertung einer Spielsituation im Tic-Tac-Toe Spiel als statische Methode

```
public static int alphaBeta(Board board, int player)
```

der Klasse `TicTacToe`. Dabei wird die Spielsituation als `board` übergeben, und die Position soll aus Sicht der Spielerin `player` (1 oder `-1`) bewertet werden. Gemäß der Motivation in der Einleitung benutzen wir die folgende Bewertung einer Spielstellung, siehe auch Abb. 3:

0	offene Position: keine der Spielerinnen kann einen Sieg erzwingen
$p+1$	mit $p \geq 0$: Gewinnposition für `player`, wobei das Spiel mit p freien Feldern gewonnen werden kann
$-p-1$	mit $p \geq 0$: Gewinnposition für `-player` (Gegenspielerin von `player`), wobei das Spiel mit p freien Feldern gewonnen werden kann

Es ist sinnvoll, die eigentliche Methode mit der Signatur

```
public static int alphaBeta(Board board, int player,
int alpha, int beta, int depth)
```

zu implementieren, und diese aus der Methode mit der zuerst genannten Signatur mit geeigneten Werten für `alpha`, `beta` und `depth` aufzurufen.

3.3 Bewertung aller möglichen Züge (10 Punkte)

Implementieren Sie in der Klasse `TicTacToe` eine statische Methode

```
public static void evaluatePossibleMoves(Board board, int player)
```

die zu einer gegebenen Spielposition `board` und Spielerin `player`, alle Zugmöglichkeiten mit der `alphaBeta()` Methode bewertet und wie in dem folgenden **Beispiel 1** auf dem Bildschirm ausgibt.

Beispiel 1: Bewertung aller Zugmöglichkeiten

```
Evaluation for player 'x':
0  o  3
0  3 -2
3  0  x
```

Die Spielsituation mit jeweils einem Zug von 'x' und 'o' ist vorgegeben, 'x' ist am Zug. Die besetzten Felder werden entsprechend ausgegeben. Für die freien Felder wird die Zugbewertung angezeigt.

Die Ausgabe der Bewertung muss über `System.out` erfolgen und genau wie in dem Beispiel formatiert sein (automatisierte Überprüfung). Genau heißt hier, Sie sollten die gleiche Anzahl an Zeilen ausgeben und an den richtigen Stellen Zeilenumbrüche einfügen, ob aber nach dem letzten Zeilenumbruch noch eine weitere Leerzeile kommt, ist egal. Auch die genauen Anführungszeichen und ob zwischen den einzelnen Werten ein Lehrzeichen oder zwei oder ein Tabulator stehen, ist nicht relevant, es **muss** aber eine Form von *space* zwischen den Werten sein.
Bei dem Spielbrett gilt der `x` Wert der `Position` in waagerechter Richtung. Das 'o' in Beispiel 1 ist also auf `Position(1, 0)`.

Nutzen Sie die Methode, um die folgenden Fragen zum 3×3 Tic-Tac-Toe zu beantworten (wird nicht bewertet):

- Gibt es einen ungünstigen ersten Zug, der die zweite Spielerin in eine Gewinnposition versetzt?
- Welcher erste Zug ist am günstigsten insofern, als dass er der zweiten Spielerin möglichst viele Verlustpositionen bereitet.

Hinweise:

Zum Testen und Debuggen Ihrer Implementation sollten Sie für jede Klasse eine `main()` Methode schreiben, die ein Objekt der Klasse erzeugt und die unterschiedlichen Methoden aufruft. Alternativ können Sie auch die mitgegebenen JUnit-Tests schreiben, um die einzelnen Methoden zu testen. Für die `Board` Klasse können Sie z.B. ein leeres Brett erzeugen, Züge ausführen, rückgängig machen, und anhand der Ausgabe der Zwischenergebnisse mit `Board.print()` die korrekte Funktionsweise überprüfen.

Für die `TicTacToe` Klasse können Sie ein Brett erzeugen, mehrere Züge ausführen und dann die `alphaBeta()` Methode aufrufen. Erzeugen Sie die Spielpositionen so, dass Sie den korrekten Rückgabewert kennen. Fangen Sie mit einer Position an, in der die ziehende Spielerin direkt mit dem nächsten Zug gewinnt. Funktioniert Ihr Programm in diesem Fall, testen Sie noch eine andere direkte Gewinnposition und fahren Sie dann mit schrittweise schwierigeren Spielpositionen fort.

Sie können `alphaBeta` auch für ein leeres 2×2 Brett aufrufen. Welche Ausgabe erwarten Sie?