Assignment: Game Search

2ID90, edition 2018-2019 Q3

1 Overview

In this assignment you are going to build a computer player for the game of INTERNATIONAL DRAUGHTS. In short[1], International Draughts is a two player strategy game played on a 10x10 board of which only the dark fields are used to initially position 20 white pieces and 20 black pieces. The players in turn move a piece of their own color diagonally over the board. The white player starts the game. Pieces of the opponent may be captured by jumping over them to an empty field[2]. Finally, the player without any remaining moves looses the game. See figure 1 for a picture of a draughts board in the initial position. You are challenged to write a draughts playing agent that will compete in a small competition in KILLER DRAUGHTS[3] against agents made by your fellow students. The overall best playing agent will play a game of INTERNATIONAL DRAUGHTS against a former Dutch champion.

In global terms, you need to

1. write an alpha-beta search algorithm (Section 2),
2. write a function that can evaluate a state of the draughts game (Section 3), and
3. integrate these two in a framework for playing a (computer) draughts tournament (Section 4 and Section 5).

For this to work out you are given some functionality specific to the international draughts game (Section 6).

[1] If this game is unfamiliar to you you can find a full explanation of its relatively simple rules at wikipedia.

[2] A player who can capture a piece, is *obliged* to do so.

[3] Killer draughts only differs in the following: during a capture move by a king, this king has to land immediately after the last captured piece, if that piece is a king. This rule considerably reduces the draw margin of the game.

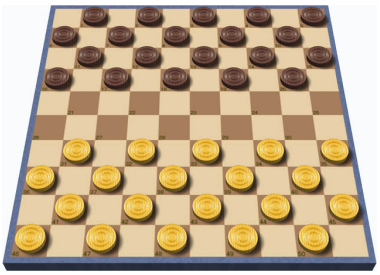

Figure 1: An international draughts board with pieces in initial position.

2 Alpha-beta search

For a general ALPHA-BETA algorithm the following classes/interfaces are useful[4]:

- `class GameState` with methods:
  1. `getMoves()`: for getting all valid moves in this state,
  2. `doMove(Move)`: for making a move,
  3. `undoMove(Move)`: for undoing a move.
- `class Move`, a class encapsulating the data for doing and undoing a valid move in a certain state.
- `class GameNode` with methods
  1. `getGameState()`: to return the game state of this node, and
  2. `setbestMove(Move)` and `getBestMove()`: to set or get the best move found for the game state of this node.

[4] In your implementation they are DraughtsState, Move, and DraughtsNode, respectively. All of them are provided to you.

A rather simple and surely incomplete sketch of an alpha-beta algorithm then becomes something like this:

```
int alphaBeta(GameNode node, int alpha, int beta) {
  GameState state = node.getState();
  List<Move> moves = state().getMoves();
  for (Move move : moves) {
    state.doMove(move);
    ... // recursive call
    state.undoMove(move);
  }
  node.setBestMove(bestMove);
  ...
```

It should be possible to stop an alpha-beta method call that takes too much time[5]. Technically, this boils down to throwing an `exception`, after a boolean *stopped* has been set to true in the `stop()` method:

[5] For fair play in the Contest environment of section 4, it is required that the computer players use the same amount of time.

```
private boolean stopped = false;
public void stop() { stopped = true; }

int alphaBeta(GameNode node, int alpha, int beta)
  throws AIStoppedException {
  if (stopped) {
    stopped = false;
    throw new AIStoppedException();
  }
  ...
```

3 *Draughts Evaluation function*

In leaf nodes of the game tree the state of the game has to be evaluated. To implement an evaluation the state of the game has to be inspected. A simple evaluation might count the number of pieces on the board. A skeleton of an evaluation function is given below. Note that the index of the pieces array starts at index 1; this is conform the International Draughts field numbering that goes from 1 to 50 as shown in figure 2.

Note that the gametree may have two types of leaf nodes: 1) actual leaves without successor states, and 2) nodes at which the maximal search depth is reached. These two nodes need a different treatment.

```
import static nl.tue.s2id90.draughts.DraughtsState.*;
...
int evaluate(DraughtsState ds) {
  // obtain pieces array
  int[] pieces = ds.getPieces();
  // compute a value for this state, e.g.
  // by comparing p[i] to WHITEPIECE, WHITEKING, etc
  ...
  return computedValue;
}
```

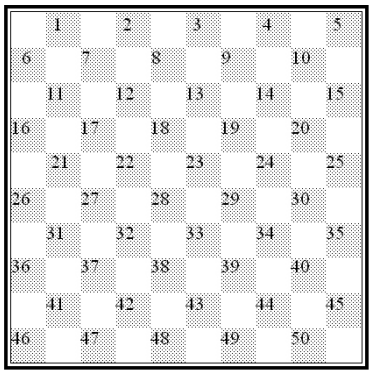

Figure 2: The field numbers used on a draughts board. Initially, the white pieces are on the fields 31- 50 and the black pieces on the fields 1-20.

4 *Tournament framework*

As you are writing your own AI player you need to test it in the AICompetition tool (see figure 3) against other AI players or

against a human player. In order to do so you need to implement the Player interface by inheriting from the DraughtsPlayer class. From the Player interface you need to implement the getMove(DraughtsState) method, that should return a valid move in the argument game state. Furthermore, a stop method has to be implemented, similarly to the method discussed in section 2. For a player that plays random moves this is very simple as shown in the code below, where the stop method is empty, because the getMove function is almost instanteneous.

```
public class RandomDraughtsPlayer extends DraughtsPlayer {
  public Move getMove(DraughtsState s) {
    List<Move> moves = s.getMoves();
    Collections.shuffle(moves);
    return moves.get(0);
  }
  public Integer getValue() { return 0; }
  public void stop() { }
}
```

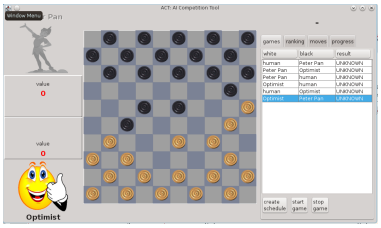

Figure 3: *A screenshot of the AICompetition tool.*

In your own implementation of a strong DraughtsPlayer the getMove method should contain a call to alpha-beta.

To get you quickly on track we gave you a sample implementation of an AI player that already has implemented:

1. calling an alphabeta method;
2. catching and throwing AIStoppedExceptions;
3. handling incomplete searches;
4. returning both best move and corresponding game value to the AICompetition tool.

5 *Installation*

Two NetBeans projects and a library folder are provided:

- AICompetition: a project for the AICompetition tool,
- DraughtsPlugin: a project containing an example Draughts player, and
- a lib folder with some additional libraries.

Branding the player project by replacing *NN* by *XY, your group's number in canvas,* gives you your own *unique* player project:

1. In the src folder rename package `nl.tue.s2id90.groupNN` to `nl.tue.s2id90.groupXY`[6].
2. Optionally, rename class `nl.tue.s2id90.groupXY.MyDraughtsPlayer` to a name of your liking[6]; this name will appear in the AICompetition tool.
3. In the arguments options[7] of the AICompetition project, write `../DraughtsPlugin/dist/` .

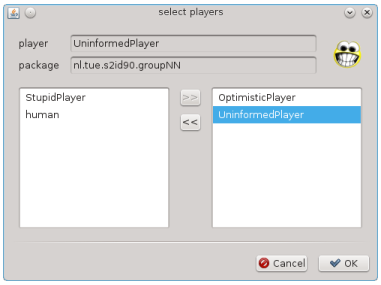

Figure 4: Popup of the AICompetition tool for selecting the players in a contest.

[6] Do this using the NetBeans functionality for renaming packages/classes, available via the context menu Refactor/rename.

[7] The command line options are specified in the run tab of the project properties.

After *first* compiling the AICompetion project and *then* your DraughtsPlugin project, you can run the AICompetition tool. In that tool pressing the CREATE-SCHEDULE button should give you a popup similar to that in figure 4, but then containing your own player.

6 *International Draughts Library*

The java library file AICompetition.jar contains several classes for International Draughts. Below the source code of some of them are given. The relevant classes and their source code and javadoc are provided.

6.1 *Gamestate*

This interface is implemented by the `DraughtsState` class.

```
package nl.tue.s2id90.game;
import java.util.List;

/** class that keeps the state information of a Game. */
public interface GameState<Move> {
  /** @return a list of valid moves in this state. */
  List<Move> getMoves();

  /** @return true iff valid moves in this state exist. */
  boolean isEndState();

  /** applies move m in this state. */
  void doMove(Move m);

  /** undoes the effect of move m. */
  void undoMove(Move m);

  /** @return whether or not the white player is to move. */
  boolean isWhiteToMove();

  /** resets state to initial game state. */
  void reset();
}
```

6.2 *Player interface*

This interface is implemented by the `DraughtsPlayer` class. Implementations of the latter class may be provided to the AICompetition tool where they can be selected for playing a game.

```
package nl.tue.s2id90.game;
import nl.tue.win.competition.util.Identity;
public interface Player<Move,State extends GameState<Move>>
      extends Identity {
  /** computes a valid move in the given game state s.
  * This method should be prepared
  * to immediately return a Move when stop() has been called.
  * @return a valid move in State s.
  **/
  Move getMove(State s);

  /** @return the computed value of the last Move.
  * Returns null if that value is not available.
```

```
  **/
15  Integer getValue();

17  /** as a result of this call the Player should immediately
     * return a value in getMove().
19  **/
    void stop();
21 }
```

7 What to do next?

To come to a basic draughts player you need to do the following.

- Implement alpha-beta.
- Implement an initial evaluation function that just counts pieces.
- implement Iterative Deepening.

From here on you can repeatedly try to improve your player in several ways, for example:

- enhance evaluation function.
- improve search function.
- improve stop criterion for searching.
- reuse results in a previous iteration of Iterative Deepening.

Improvements that hinder your opponent are clearly not allowed[8].

[8] So, it is not allowed to search while the opponent is searching; it is not allowed to keep results between two calls to Player::getMove().

8 Appendix

8.1 Tournament rules

1. In the tournament matches are played according to *killer draughts* rules.
2. A computer player that makes an illegal move looses the match immediately.
3. The number of seconds per move is unknown to a computer player: If a move is not supplied within a reasonable amount of time after receiving a call to its stop method, a computer player looses a match immediately.
4. Decisions of the lecturers of 2ID90 always prevail and are beyond discussion.
5. The tournament consists of one or more GROUP phases followed by a KNOCK-OUT phase.

6. In a group phase there are X groups of maximum Y teams. In each group a single round-robin tournament is played, i.e. all teams play each other once in a match. In each match a total two points can be earned: A winning team gets 2 points, in case of a tie each team gets 1 point. For each group the team that collects most points goes to the next phase. In case of a tie, the team with the most points in matches between the tied teams is the winner. If this is insufficient, the drawing of lots will determine the winner.

7. The knock-out phase consists of several final rounds, e.g. 4 quarter finals, 2 semi finals, and a final. If matches in these finals are a draw, a rematch is played with swapped colors. If the rematch is a draw, the winner is determined by drawing of lots. For the finals, if time permits an additional rematch may be played instead.