Artificial Intelligence

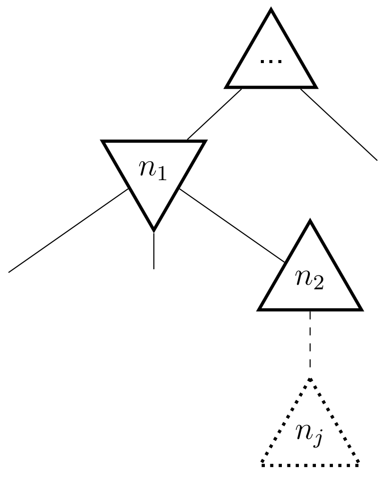
Assignment 8

Assignment due by: 21.12.2016, Discussion: 10.01.2017

Question 1 Alpha-Beta Pruning – Correctness [6 points]

The goal of this question is to prove the correctness of alpha-beta pruning.

Consider the following situation:



Prove that Alpha-Beta pruning implies that if n_j is pruned then the minimax value of n_1 is independant of the minimax value of n_j. At a given node n_j, when performing Alpha-Beta pruning, prove that your pruning decision at node n_j is optimal (i.e. you cannot prune more) provided your current knowledge of the tree. *Hint:* You can refer to the sibling nodes of n_i as $n_{i1}, \ldots, n_{ik}$. Furthermore, l_i refers to the minimum/maximum value under the left siblings of n_i and r_i to the minimum/maximum value under the right siblings.

Question 2 Alpha-Beta Pruning – KCell [14 points]

Implement a minimax search with alpha-beta pruning in Java. The agent is supposed to play KCell, a simple two-player game. Each player has n pieces on a game-board consisting of k cells in a line. The rules are the following, illustrated for $n = 2$ and $k = 7$:

- Initially, the players stones are placed at opposing ends of the board, one piece per cell (i.e. [oo___xx], with "o" denoting player 1's pieces, "x" denoting player 2's pieces and "_" denoting empty cells).
- Players alternate moves, starting with player 1.
- To win the game, a player has to move both his pieces all the way to the other side, i.e. cells 6 and 7 for player 1 or cells 1 and 2 for player 2.
- Each turn, the players may move exactly one of their pieces forward, i.e. player 1 may only move right and player 2 may only move left.

- The following moves are allowed (examples as player 1):
  - a single step forward, if the next cell is empty
    [oo___xx] $\mapsto$ [o_o__xx]
  - jump over a single piece (own or opposing)
    [_oox__x] $\mapsto$ [_o_xo_x]
    [oo___xx] $\mapsto$ [_oo__xx]
  - if, after a jump, a piece could perform another jump, it automatically does so. This counts as one move and continues until that piece can't make any more jumps
    [_oox_x_] $\mapsto$ [_o_xox_] $\mapsto$ [_o_x_xo]
- if a player has legal moves, he must take one of them.
- if (and only if) a player has no legal moves, he makes an empty move (he passes) and the other player goes again.
- if neither player has any legal moves left, the winner is the player who has reached the opposing side of the board with the most pieces (consecutive pieces from the opposing edge). Otherwise the game is declared a draw.

Thus, a legal move is completely defined by the number of a cell containing a piece that can be moved according to the rules above.

Download the file `kcell.zip` which contains relevant interfaces and files for you to put your code into. It also contains a `KCellHumanPlayer` that allows play via the terminal and an implementation of minimax search.

(a) Implement the game logic of KCell in the Class `de.cogsys.ai.kcell.KCell`. Make sure the logic only allows valid moves (if there are no other legal moves, passing a/the only valid move). Test your implementation by playing with two human or minimax players.

(b) Implement alpha-beta pruning in the class `de.cogsys.ai.game.AlphaBetaAgent`, which should perform minimax search with alpha-beta pruning without using a cutoff-test. Test your implementation against the minimax agent, it should play at the same strength. Compare the speed of the pure minimax and alpha-beta pruning by seeing how large you can make the size of the board and the number of stones while the agents take at most ~ 1 sec of time for each move.

(c) Finally, implement a cutoff-test. To do this, create a class that implements the `AlphaBetaHeuristic` interface and modify your `AlphaBetaAgent` to accept that heuristic. The class `RandomKCellHeuristic` contains a heuristic that assigns random values to each move, and your task is to create a heuristic that can reliably beat this heuristic. Have the two play against each other at 15 cells and 3 stones and make sure your heuristic can beat the random one as both player 1 and player 2. Additionally your heuristic must run in less than 5 seconds. Explain the idea behind your heuristic and document a typical game against the random heuristic.

Note: solutions that do not compile will be given **zero** points. You can use anything from the Java Class Libraries, and any features up to Java 8, but **no** other external dependencies. The included Ant build file should be able to compile all the classes.