Games

1 GENERAL RULES

You should work in pairs, and you can discuss and divide the work within your pair, as you wish. However, you will be assessed individually. This means that both people have to be able to explain the theory as well as the details of the implementation.
You can also discuss the assignment with others, but you are not allowed to exchange any parts of the implementation. This should be done within the pair. You will be assessed based on the solution produced by your pair. Any parts of the solution which are not produced by your pair should be clearly stated and will not be considered when assessing your work.

Deadline
For grades A and B you must have presented your homework not later than 30th of September, 2016. Any homework presented after this will result in max. grade C.

2 INTRODUCTION TO GAME THEORY

The main object of study in game theory is a system where two or more agents are trying to maximize their own gain by changing a shared state. In this assignment, we are interested in a special class of such problems, namely those where two players are in direct conflict (which are commonly called zero-sum games).
To put it formally, a game consists of the following parts:

$P = \{A, B\}$ is the list of players (which will always be the same in our case because we will only analyze two-player games).

S is the list of possible game states.

s_0 is the initial state, the game *always starts* in the initial state.

$\mu : P \times S \to \mathscr{P}(S)$ (where $\mathscr{P}(S)$ is the set of all subsets of S) is a function that given a player and a game state returns the possible game states that the player may achieve with one *legal* move by the current player.

$\gamma : P \times S \to \mathbb{R}$ is a *utility function* that given a player and a state says how "useful" the state is for the player.

A game is called a zero-sum game, if $\gamma(A, s) + \gamma(B, s) = 0$ or equivalently $\gamma(A, s) = -\gamma(B, s)$. Informally, each component of the specification has a role in structuring the gameplay. More concretely we have:

Not all states in S need to be reachable by a legal sequence of moves.

μ is the part of the specification that dictates the *rules* of the game (by dictating which are the legal moves).

γ is the part of the specification that drives the game forward, and it is not unique (any rescaling of γ is also a utility function).

We say that the game is over when it reaches a state s_t such that $\mu(p, s) = \emptyset$ and player p is about to play. In this case s is called a *terminal state*, and we say that:

Player A wins if $\gamma(A, s) > \gamma(B, s)$.

Player B wins if $\gamma(B, s) > \gamma(A, s)$.

The game is tied if $\gamma(A, s) = \gamma(B, s) = 0$.

Hence the objective of player A is to reach a terminal state s_t that maximizes $\gamma(A, s_t)$ and since $\gamma(B, s) = -\gamma(A, s)$, the objective of player B is to reach a terminal state s'_t that minimizes $\gamma(A, s'_t)$. Finally we say that player A wins if, upon reaching a terminal state s, $\gamma(A, s) > \gamma(B, s)$, and it is a tie if $\gamma(A, s) = \gamma(B, s)$.

EXAMPLE: CHECKERS

- S is any board where only the black squares are occupied, and there are at most 12 checkers of each color.
- s_0 is the board with 12 checkers of each color positioned in opposite sides of the board, as shown in figure 2.1.
- μ assigns to a game state (s, A) the set of states s' that can be obtained from s by moving a white checker (or a red checker in case of B).

Describe the possible states, initial state, transition function.

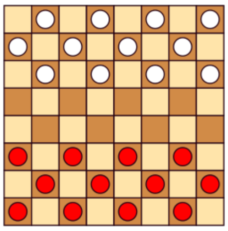

Figure 2.1: Initial game state for checkers.

Describe the terminal states of both checkers and tic-tac-toe.

2.1 Heuristic functions

Note that in the previous example we did not describe any utility function. This is because utility functions are used as a theoretical device and intuitively they describe how a perfect player would play (at each step the perfect player would choose the next state with the highest utility). It is natural to assume that such a function is difficult to obtain, so instead one generally uses *heuristic functions*.
Simply put, a heuristic function is one that approximates a utility function. Usually the process for obtaining such a function is by analyzing a simpler problem, so that the resulting function is easier to compute. With that said, a rather simple heuristic function for checkers could be given by:

$$v(A,s) = \#\{\text{white checkers}\} - \#\{\text{red checkers}\}$$

This function v belongs to a class heuristic functions called *evaluation functions* meaning that it can be efficiently computed using only the game state.

Why is $v(A,s) = \#\{\text{white checkers}\} - \#\{\text{red checkers}\}$ a suitable heuristic function for checkers (knowing that A plays white and B plays red)?

Can you provide an example of a state s where $v(A,s) > 0$ and B wins in the following turn? (Hint: recall the rules for jumping in checkers.)

2.2 Game trees and more heuristics

From the questions in the previous section (specifically the last one) it should be evident that a naive heuristic function is, in general not enough to guarantee a victory against a smart (and

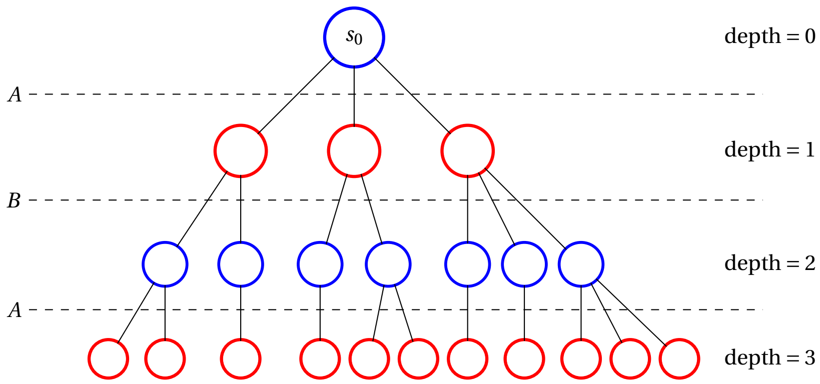

Figure 2.2: Each node corresponds to a game state, The blue nodes have even depth, the red nodes have odd depth, and there is an edge from a blue node to a red node (with corresponding states s_b, s_r, respectively) if in state s_b, player A can perform a move that will change the state into s_r and there is an edge from a red node to a blue node (with corresponding states s'_r, s'_b, respectively) if in state s'_r, player B can perform a move that transforms state into s'_b.

sometimes, even random) player. This shortcomming is shared by most evaluation functions[1].
In this section we introduce the minimax algorithm that, given a naive heuristic function produces a heuristic function that better fits a utility function. To this end we start by introducing the following heuristic function:

$$\eta(A,s) = |w(s,A)| - |l(s,A)| \qquad \eta(B,s) = |w(s,B)| - |l(s,B)|$$

Where $w(s,p)$ is the set of terminal states at which player p wins and which are reachable from state s, and $l(s,p)$ is the set of terminal states at which player p loses and which are reachable from state s (and $|\cdot|$ denotes the list length).
It is easy to see that η satisfies $\eta(A,s)+\eta(B,s) = 0$, and for a terminal state s_t $\eta(A,s_t) > 0$ if player A wins, $\eta(A,s_t) < 0$ if player B wins, and $\eta(A,s_t) = 0$ if it is a draw. Also, note that η is not an evaluation function, as it's computation requires one to filter through valid sequences of moves.
Now, for simplicity assume that player A always starts the game. In order to calculate $\eta(A,s)$ it is convenient to represent the game as a *game tree*, $GT = (V,E)$ with a set of nodes V, each of which corresponds to a game state[2] and a set of edges $E \subseteq V \times V$ satisfying:
Given $s, s' \in V$, the edge (s,s') (also denoted $s \to s'$) is in E if and only if $s' \in \mu(A,s)$ and depth(s) is even, or $s' \in \mu(B,s)$ and depth(s) is odd (see illustration in figure 2.2).
A game tree can be used to search for a winning strategy at each point of the game. Particularly, the *complete game tree* is a game tree with s_0 as root and where every possible move is

[1] Note that given any heuristic function it would be, theoretically possible to create a hash-table associating each possible state with its utility function, and this would constitute an evaluation function.
[2] Note that there may be more than one node with the same state.

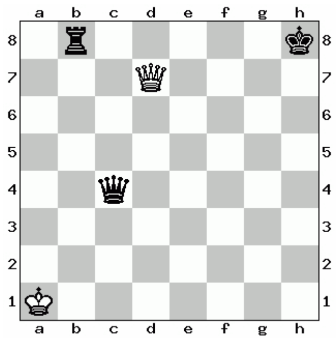

Figure 2.3: Consider this chessboard setup, with black to play. While playing white an algorithm that only uses a heuristic function such as η may treat the black queen moving horizontally to a4 in the same way as moving diagonally to a6 while any player would be able to see that moving the queen to a4 would result in the loss of the queen, and most likely the loss of any chance of mate by the black player, whereas moving the queen diagonally would result in mate in two turns.

expanded until it reaches a terminal state. We can use game trees to solve games, i.e., to find a sequence of moves that one of the players can follow to guarantee a win. Thus, if we have access to the complete game tree, we can calculate $\eta(A, s)$ by checking all terminal states that are reachable from s by a path of strictly increasing depth.

Will η suffer from the same problem (referred to in the last question) as the evaluation function v?

2.3 Minimax algorithm

Now even though η represents a step in the right direction (as we take information about how the game might proceed from every point), it still has a large drawback that is, it gives the same weight to moves of player B that are beneficial to it, as it would give if they were actually prejudicial (see Figure 2.3 for an example). This amounts to assuming that player B is playing randomly, which is generally not the case.

In the specific case of η, $\eta(A, s)$ just means that there are 10 more reachable terminal states reachable that result in a victory for player A than for player B, however it tells nothing of the distribution of such states.

The way around this problem is assuming that the opponent is also aware that it has to optimize it's chances of winning, and therefore when calculating a heuristic function that requires traversing the tree, we should only consider the transitions that maximize the heuristic for player B (and therefore minimize the heuristic for player A). This can be done by employing the *minimax algorithm*, the pseudocode is written in Algorithm 1 (where we assume that our heuristic function γ is an integer function).

Algorithm 1: Pseudocode for minimax algorithm

```
int minimax (state, player)
    // state: the current state we are analyzing
    // player: the current player
    // returns a heuristic value that approximates a utility function of the state

    if μ(state, player) = ∅  // terminal state
        return γ(player, state)
        // γ is a function that returns 1,0 or −1 depending on whether the current state
        // is a win, tie or loss for the player

    else // can search deeper

        if player = A
            bestPossible = −∞
            for each child in μ(A, state)
                v = minimax(child, B)
                bestPossible = max(bestPossible, v)
            return bestPossible

        else // player = B
            bestPossible = +∞
            for each child in μ(B, state)
                v = minimax(child, A)
                bestPossible = min(bestPossible, v)
            return bestPossible
```

Note that Algorithm 1 requires one to evaluate the tree until it reaches a terminal state. However, given an evaluation function, we can use it to truncate the search at a given depth and use the evaluation function at that state.

2.4 Alpha-Beta pruning

Now, Algorithm 1 requires one to search the complete gametree (at least until a given depth). However, this may not be the case. The idea behind the *alpha-beta pruning* algorithm is that in some situations, parts of the tree may be ignored if we know a priori that no better result may be achieved.

To understand how this is possible, consider that player A is traversing the tree in order to find the next best move using the minimax algorithm. Instead of just keeping track of the best node found so far, the player can also keep track of the best heuristic value computed so far, α as well as the worst such value β (which yields the best result for B). At each transition in the tree α should be updated, if it is player A's turn, and otherwise β should be updated. Finally, the remainder of a branch should be disregarded whenever α is greater then β since that indicates the presence of a non-desirable state.

Algorithm 2: Pseudocode for minimax algorithm with alpha-beta pruning

```
int alphabeta (state, depth, α, β, player)
    // state: the current state we are analyzing
    // α: the current best value achievable by A
    // β: the current best value acheivable by B
    // player: the current player
    // returns the minimax value of the state

    if depth = 0 or μ(state, player) = ∅
        // terminal state
        v = γ(player, state)

    elseif player = A
        v = −∞
        for each child in μ(A, state)
            v = max( v, alphabeta(child, depth − 1, α, β, B) )
            α = max(α,v)
            if β ≤ α
                break // β prune

    else // player = B
        v = ∞
        for each child in μ(B, state)
            v = min(v, alphabeta(child, depth − 1, α, β, A))
            β = min(β, v)
            if β ≤ α
                break // α prune
    return v
```

3 ASSIGNMENT: N-DIMENSIONAL TIC-TAC-TOE

In this assignment we consider a special case of n-dimensional generalization of Tic-Tac-Toe game. The rules are simple. There is an n-dimensional hypercube H, consisting of 4^n cells. Two players, A and B, take turns marking blank cells in H. The first player to mark 4 cells along a row wins. Here by a row we mean any 4 cells, whose centres lie along a straight line in H. For instance, in the case $n = 2$ any horizontal, vertical and diagonal row is winning. In the case $n = 3$, winning rows lie along the 48 orthogonal rows (those which are parallel to one of the edges of the cube), the 24 diagonal rows, or the 4 main diagonals of the cube, making 76 winning rows in total. Player A always starts the game.
In this assignment, our goal is to find the best possible move for player X given a particular state of the game. The number n and the maximal execution time of the program depend on the grade level.

4 ASSIGNMENT: CHECKERS

For obtaining a higher grade, the student is expected to be able to use the general framework developoed for tic-tac-toe and adapt it to play checkers.

5 Grade level E and D

In order to get E or D, one has to write a program computing the best possible next move for player X, given a particular state of the game for $n = 2$. In this case, it is enough to implement Minimax algorithm without alpha-beta pruning. Due to time constraints one cannot analyse the complete game tree. Therefore the student should investigate the influence of the maximum search depth and come up with a suitable evaluation function. The total execution time for 100 different board states should not exceed 25 s. As an example of a simple evaluation function, one can consider the following:

$$eval(Board) = \sum_{r \in Rows} myMarks(r) + \sum_{c \in Columns} myMarks(c) + \sum_{d \in Diagonals} myMarks(d)$$

The problem can be found on Kattis:
`https://kth.kattis.com/problems/kth.ai.tictactoe2d`

To pass, you have to get at least 94 points out of 100 on Kattis.

6 Grade level C

In order to get C, one has to write a program computing the best possible next move for player X, given a particular state of the game for $n = 3$. The total execution time for 100 different board states should not exceed 50 s.
The problem can be found on Kattis:
`https://kth.kattis.com/problems/kth.ai.tictactoe3d`
To pass, you have to get at least 97 points out of 100 on Kattis.

7 Grade level B and A

In order to get B or A, the student is expected to have completed levels E through C and adapt the algorithms where necessary to the checkers problem, which can be found on Kattis:
`https://kth.kattis.com/problems/kth:ai:checkers`
To pass, you have to get at least 20 **points out of** 25 **on Kattis.**