Programming Assignment: Search

Deadline
For grade **A** you must have your implementations accepted by Kattis with the required scores by the **24th of September 2021, 17:00pm sharp** and presented within the regular examination period. Any assignment accepted by Kattis/presented after these deadlines will result in max. grade B.

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 work individually, but you cannot work in groups larger than two people.

1 THE KTH FISHINGDERBY

In this assignment we will take a look at the KTH fishing derby game, inspired by the famous fishing derby arcade game.
The players in this turn-based game are two fishing boats out at sea (see Figure 1.1). Imagine that you are in command of the green boat (on the left) while the red one (on the right) is the opponent. The boats can move horizontally and have a fishing line attached. There is a hook at the end of the fishing line which can be moved vertically. Consequently, the possible actions for each player are LEFT, RIGHT, UP, DOWN and STAY. Note that we consider a 20×20 2D scenario, and the boats cannot cross each other. If the boats are next to each other, for instance with the green at the left position from the red one, and the green boat applies action RIGHT, the action will just result in the same behavior as action STAY. However, the world is round, and action LEFT applied at the leftmost position of the screen means that the boat will appear on the rightmost position (unless the other boat occupies that position).
Each player has to catch as many as valuable fish as possible. A fish is caught once its position coincides with the hook's position. Some fish are more valuable than others resulting in different score points per fish. A penalty (negative points) is given for types of fish that should not be caught. The correspondence between the type of fish and number of points is known at the beginning of the game. A player wins if he or she has obtained a higher number of points than the opponent. The game ends either if no fish is left or the game time has passed (the fishing day is over).

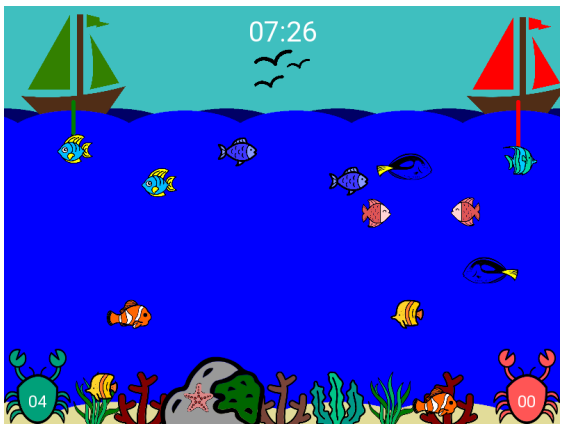

Figure 1.1: Graphical interface of the Fishingderby game (Minimax). The crabs in the corner show the current score. 07:26 is the time remaining till the end of the fishing day (7 minutes and 26 seconds).

Throughout this document, we will use the KTH fishing derby game as an example of a two-player zero-sum game. Your task in this assignment will be to develop an AI agent that decides which move the green boat should realize using a variant of the *minimax* algorithm.

2 INTRODUCTION TO GAME THEORY

Game theory studies systems where two or more agents try to maximize their own gain by operating in a shared environment. In this assignment, we are interested in a special class of such problems, namely one 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, that is, any possible set of game states belongs to $\mathscr{P}(S)$) 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 two-player 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 game-play. More specifically, we have:

> μ is the part of the specification that dictates the *rules* of the game (by dictating which are the legal moves). Notice that not all states in S need to be reachable by a legal sequence of 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 $\mu(p, s_t) = \emptyset$ and player p is about to play. In this case, s_t 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 $\gamma(A, s'_t)$.

Q1. Describe the possible states, initial state, transition function of the KTH fishing derby game.

Q2. Describe the terminal states of the KTH fishing derby game.

2.1 Heuristic functions

Note that in the KTH fishing derby 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 the KTH fishing derby could be given by:

$$v(A, s) = \text{Score(Green boat)} - \text{Score(Red boat)}$$

This function v belongs to a class of heuristic functions called *evaluation functions* meaning that it can be efficiently computed using only the game state[1].

Q3. Why is $v(A, s) = \text{Score(Green boat)} - \text{Score(Red boat)}$ a good heuristic function for the KTH fishing derby (knowing that A plays the green boat and B plays the red boat)?

Q4. When does v best approximate the utility function, and why?

Q5. Can you provide an example of a state s where $v(A, s) > 0$ and B wins in the following turn? (Hint: recall that fish from diferent types yield different scores).

[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.

2.2 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 sometimes, even random) player. This shortcoming is shared by most evaluation functions. 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, that is, the number of such states).

It is easy to see that η satisfies $\eta(A,s) + \eta(B,s) = 0$, and for a terminal state s_t we have that $\eta(A,s_t) > 0$ 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 two nodes from the game tree $s, s' \in V$, the edge (s,s') (also denoted $s \to s'$) is in E if and only if s' can be reached from s with a legal move, i.e., $s' \in \mu(A,s)$ and depth(s) is even, or $s' \in \mu(B,s)$ and depth(s) is odd (see illustration in Figure 2.1).*

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 the initial state s_0 as root and where every possible move is 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 their 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.

Q6. Will η suffer from the same problem (referred to in **Q5**) as the evaluation function v? If so, can you provide with an example? (Hint: note how such a heuristic could be problematic in the game of chess, as shown in Figure 2.2).

[2] Note that there may be more than one node with the same state.

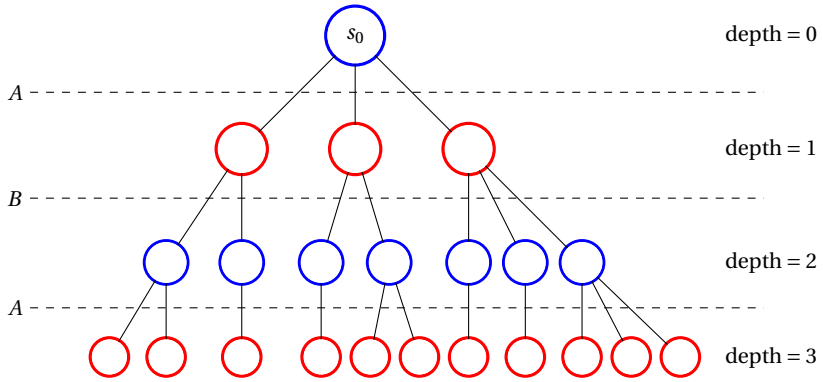

Figure 2.1: Each node corresponds to a game state, The blue nodes have even depth, the red nodes have odd depth, and there is an edge from a blue node to a red node (with corresponding states s_b, s_r, respectively) if in state s_b, player A can perform a move that will change the state into s_r and 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.

2.3 Minimax algorithm

Now, even though η represents a step in the right direction, as it takes information about how the game might proceed from every point, it still has a large drawback. It assumes player B plays randomly, and assigns the same weight to moves of player B independent of their outcome (see again Figure 2.2 for an example).

In the specific case of η, consider for instance $\eta(A, s) = 10$ for a given state s. This just means that there are 10 more reachable terminal states that result in a victory for player A than for player B, however it tells nothing of the distribution of such states, and how much B can do to counteract those moves.

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 function for player B (therefore minimizing the heuristic for player A). This can be done by employing the *minimax algorithm*, which pseudocode is written in Algorithm 1.

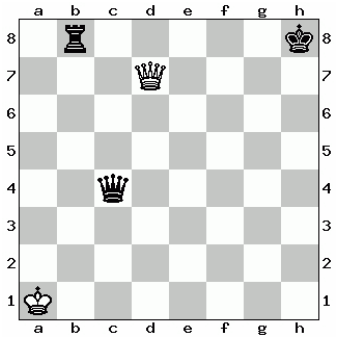

Figure 2.2: Consider this chessboard setup, with black to play. While playing white an algorithm that only uses a heuristic function such as η may treat the black queen moving horizontally to $a4$ in the same way as moving diagonally to $a6$ while any player would be able to see that moving the queen 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.

Algorithm 1: Pseudocode for minimax algorithm

```
1  int minimax (state, player)
2      // state: the current state we are analyzing
3      // player: the current player
4      // returns a heuristic value that approximates a utility function of the state
5
6      if μ(state, player) = ∅  // terminal state
7          return γ(player, state)
8          // γ is an evaluation function
9
10     else // can search deeper
11
12         if player = A
13             bestPossible = −∞
14             for each child in μ(A, state)
15                 v = minimax(child, B)
16                 bestPossible = max(bestPossible, v)
17             return bestPossible
18
19         else // player = B
20             bestPossible = +∞
21             for each child in μ(B, state)
22                 v = minimax(child, A)
23                 bestPossible = min(bestPossible, v)
24             return bestPossible
```

Note that Algorithm 1 requires one to evaluate the tree until it reaches a terminal state. However, given any evaluation function which can assign a heuristic for the game, we can truncate the search at a given depth by including the number of levels left to analyze as an argument and decreasing it in each step.

2.4 Alpha-Beta pruning

Now, Algorithm 1 requires one to search the complete game tree (at least until a given depth). However, this may not be necessary. 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.

Formally, if α gets updated in player's A turn and $\alpha \geq \beta$, then we already know that

$$\max(\alpha, \min(\beta, v)) = \alpha,$$

regardless of which is the value v, and since α is already the best that player A can do, it is worthless to explore any other state. Similarly, if β gets updated in player's B turn and $\alpha \geq \beta$, then we already know that

$$\min(\beta, \max(\alpha, v)) = \beta,$$

regardless of which is the value of v.

2.5 Further optimizations

Until now, we saw that the *minimax* algorithm allows us to search through the game tree for the optimal strategy (assuming a perfect opponent) based on a given heuristic. Then, we observed how *alpha-beta pruning* allows us to reduce the game tree search by skipping entire branches that are guaranteed to not benefit on our optimal strategy search.

However, alpha-beta pruning is not the only addition we can perform to the minimax algorithm in order to make our game tree search more efficient. It is now up to **you** to check on other gimmicks such as *iterative deepening search*, *move ordering*, *repeated states checking*, *quiescence search*, *forward pruning*, and more (see [1, Chapter 5]) in order to incorporate them in your algorithm implementation for this assignment.

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

```
1  int alphabeta (state, depth, α, β, player)
2      // state: the current state we are analyzing
3      // α: the current best value achievable by A
4      // β: the current best value achievable by B
5      // player: the current player
6      // returns the minimax value of the state
7
8      if depth = 0 or μ(state, player) = ∅
9          // terminal state
10         v = γ(player, state)
11
12     elseif player = A
13         v = −∞
14         for each child in μ(A, state)
15             v = max( v, alphabeta(child, depth-1, α, β, B) )
16             α = max(α, v)
17             if β ≤ α
18                 break // β prune
19
20     else // player = B
21         v = ∞
22         for each child in μ(B, state)
23             v = min(v, alphabeta(child, depth-1, α, β, A))
24             β = min(β, v)
25             if β ≤ α
26                 break // α prune
27     return v
```

3 THE ASSIGNMENT

In this assignment, you have to write a program that determines the next move for the green player in the KTH fishing derby game. You will be tested against a minimax opponent capable of reaching large depths, which in the testing scenarios will result in a perfect minimax.

But don't worry!, in all scenarios you will be put in a position in which if you perform the right moves, you are guaranteed to win. Therefore the deciding factor for your victory will be the depth that your algorithm is capable of reaching within the predefined deadline (i.e., how efficient is your minimax implementation) and how well your heuristic characterizes the value of each state.

3.1 GRADE LEVEL E AND D

In order to get pass the grade level E or D, you are supposed to implement the minimax algorithm to find a strategy for your player. Due to the large branching factor of the game tree you will have to limit the depth of the game tree and/or enhance your minimax algorithm with alpha-beta pruning to compute actions within reasonable time. Consequently, you will have to develop a suitable heuristic which approximates the utility of states.

The problem can be found on Kattis:

https://kth.kattis.com/problems/kth.ai.search

To achieve this grade the student must obtain at least 10 points out of 25 on Kattis with a tree search of depth >= 2. Note: The Kattis score does not guarantee the student is given the corresponding grade level, the grade level will depend on later oral examination.

3.2 GRADE LEVEL C

Now, in order to get pass the grade level C, the depth that your game state search algorithm can reach with vanilla minimax will not be enough to win against the red opponent, which has a minimax implementation capable of reach further in the tree than you do. In order to obtain C the student is expected to have a minimax implementation with further avenues, such as iterative deepening search or move ordering (see, e.g, [1, Chapter 5]).

The problem can be found on Kattis:

https://kth.kattis.com/problems/kth.ai.search

To achieve this grade the student must obtain at least 15 points out of 25 on Kattis. Note: The Kattis score does not guarantee the student is given the corresponding grade level, the grade level will depend on later oral examination.

3.3 GRADE LEVEL B AND A

Finally, in order to get pass grade level B-A, you are expected to implement a version of the minimax algorithm capable of reaching depths of at least 7 with an appropiate heuristic, so that is capable of beating the red opponent in yet more difficult scenarios. For this purpose you will need to enhance your minimax implementation with yet more gimmicks, such as repeated states checking (see, e.g., [1, Chapter 5]). In other words, it might be possible to get to the required score by fiddling the depth and the evaluation heuristics, but this is not the point of this exercise and will not yield B-A grades. For B-A grades, it is expected that you will deploy some advanced techniques on top of alpha-beta pruning.

The problem can be found on Kattis:

`https://kth.kattis.com/problems/kth.ai.search`

To achieve this grade the student must obtain at least 20 points out of 25 on Kattis. Note: The Kattis score does not guarantee the student is given the corresponding grade level, the grade level will depend on later oral examination.

REFERENCES

[1] S. Russell and P. Norvig, *Artificial intelligence: a modern approach*. Prentice Hall, 3 ed., 2010.