Homework 3 – Artificial Intelligence

Task 1

1.1

According to the minimax algorithm, the best move the MAX player can do in the initial state is the move leading to state `b`. This is, because the minimax algorithm yields the value `60` (i.e. the best reachable node is node `n`).

1.2

According to the minimax algorithm, the root node has value `60`.

1.3

The following branches can be cut:

- branch to `m` can be cut because $\beta(a) == 35 <= 50 == \alpha(e)$
- branch to `q` can be cut because $\beta(g) == 60 <= 75 == \alpha(g)$
- branch to `i` can be cut because $\beta(c) == 40 <= 60 == \alpha(root)$

1.4

The following ordering maximizes the number of pruned branches:

- children of `MAX` nodes are ordered in decreasing values
- children of `MIN` nodes are ordered in increasing values

This ordering results in the following expansion order (this does not consider pruned branches): `bfnogqpadjkelmcituhsr`

1.5

If one had access to this perfect ordering function of moves, one could simply use it to make the perfect move. If we are expanding the first node, we do not know yet which move will be the best. This is exactly why we are running the algorithm. However, there are several options to improve the algorithm. E.g.:

- using iterative deepening
  - first use minimax with depth 1 and remember the computed values
  - then increase the depth but use the remembered values to order the potential moves
  - this increases the number of pruned branches because it allows us to order the discovered states
- use transposition tables to cache the result of already seen states
  - strictly spoken, this does not improve the pruning, but it ensures that we do not evaluate the same state multiple times

Task 2

2.1

In general, given the maximal height h and the branching factor b, the number of examined nodes n can be computed by $n = b^h$. If we apply alpha-beta pruning and assume random node ordering, we can reduce the number of nodes to $n = b^{3/4*h}$. If we further apply perfect node ordering, we could further reduce this to $n = b^{h/2}$.

Assuming random node ordering and given a number of examined states and given the maximal height, we can compute the branching factor using $b = \sqrt[(3/4*h)]{n}$. If we are using a transition table, each state is evaluated/examined only once because the result of each node is cached in the transition table. For this task I assume that there are in total $1.5*10^9$ states to be explored by the algorithm. This is taken from the fact, that the implementation of the *Fhourstones Benchmark* explores this amount of states while using a transition table.

I assume the following values:

- $h = 42$ since this is the maximal height/depth of the search tree
- $n = 1.5 * 10^9$ since this is the number of states to finish the search

$$b = \sqrt[3*4/h]{n} = \sqrt[3*4/42]{1.5 * 10^9} = \sqrt[31.5]{1.5 * 10^9} = 1.9557 \approx 1.96$$

2.2

Using the result from the previous task, I consider a branching factor of $b = 1.96$ because we can still use transition tables. However, this time we do not use alpha-beta pruning which effectively means that we consider all possible branches of the tree. Assuming height $h = 42$ and no alpha-beta pruning, we have in total $n = b^h = 1.96^{42} \approx 1.88 * 10^{12}$ states to evaluate. With a speed of 10^7 states per second it takes about $1.88 * 10^5$ seconds. This corresponds to approximately 52 hours.

2.3

In the following, I will compute the duration of the algorithm *with* and *without* the proposed heuristics. In both cases, the Minimax algorithm is used with alpha-beta pruning and transition tables. Using the transition tables leads to the assumption that the branching factor is $b = 1.96$. The maximal height of the search tree is $h = 42$.

Estimate without the heuristics

Without using the heuristics (but still using alpha-beta pruning) the number of examined states is computed via $n = b^{3/4*h}$.

- number of examined states: $1.5 * 10^9$
- states per second: 10^7
- total duration: $1.5 * 10^9 / 10^7 = 150s$

Estimate with improved heuristics

Using the heuristics allows us to reduce the number of examined states. The number of examined states is computed via $n = b^{h/2}$, since the heuristic is (almost) perfect.

- number of examined states: $n = b^{h/2} = 1.96^{42/2} = 1.3 * 10^6$
- states per second: $0.5 * 10^7 = 5^7$
- total duration: $\frac{1.3*10^6}{0.5*10^7} = 2.6 * 10^{-1} = 0.26s$

Although the application of the heuristic reduces the number of computed states per second by factor 0.5, the runtime of the Minimax algorithm is improved. This is, because the heuristics allows to order the states almost perfectly, which reduces the number of examined states by factor 10^3. It is worth adding the heuristic to the code.

Task 3

Consider Minimax to play a two-player, turn-taking game that is not zero-sum. In a zero-sum game, the goal of each player is to maximize their own payoff while minimizing the payoff of their opponent. Minimax uses this property to evaluate each game state by assuming that each player will play optimally and make the move that leads to the best possible outcome for them. If the game is not zero-sum, the rewards of the players may not add up to zero, and the outcome of the game may not be a win for one player or a draw. This can lead to suboptimal results and potentially incorrect decisions when using Minimax.

Consider Minimax to play a game that is not turn-taking. In this case, both players decide at the same time which move to take. Thus, the classical Minimax algorithm does not know, which move the other player will do. Since the algorithm uses this knowledge to build the game tree, it can not compute the next move without knowing the last move of the other player. To compute the next move of a player, the algorithm would need to guess which move the other player will do.