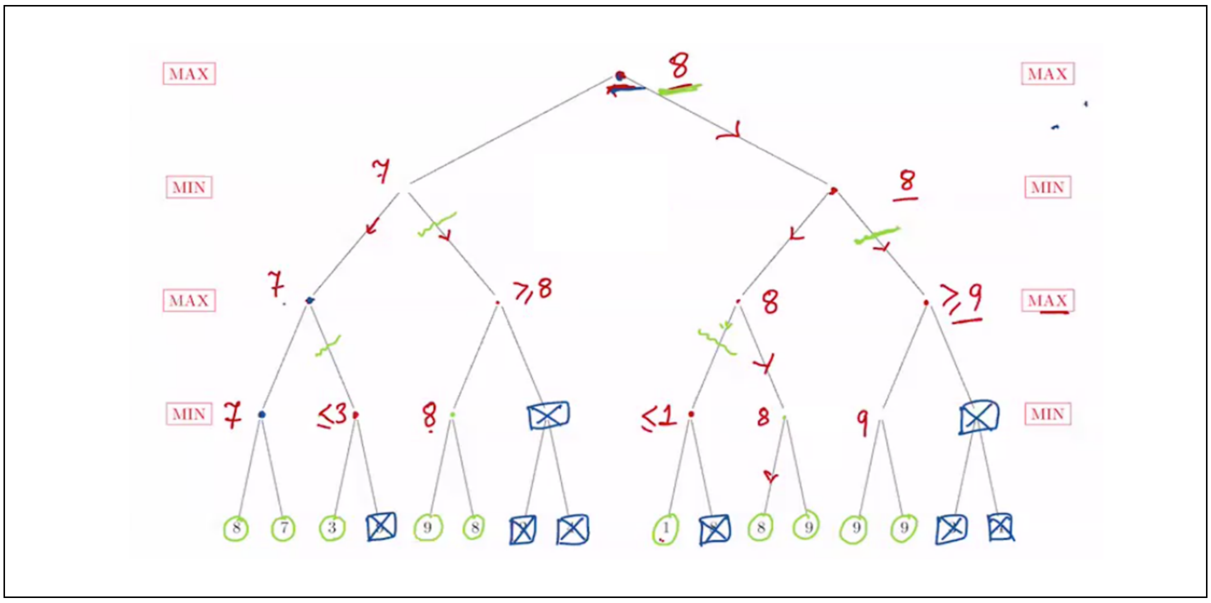
CS3243: Introduction to Artificial Intelligence
Lecture Notes 5: Games and Local Search

1. $\alpha - \beta$ Search

In the **Alpha-Beta Search** algorithm, for each game state within the tree:

- α represents the best value from the perspective of the **MAX** player.
- β represents the best value from the perspective of the **MIN** player.

Running the α-β search algorithm on a game tree like the one below would result in each of the nodes being assigned the following utility values.



The terminal states circled in green are those that were evaluated.

The pseudocode for the α-β search algorithm is as follows:

```
MAX_VALUE(s, α, β):
      if TERMINAL(s) return UTILITY(s)
      v ← −∞
      for each a in ACTIONS(s)
            v ← MAX(v, MIN_VALUE(RESULT(s, a), α, β))
            if (v ≥ β) return v
            α ← MAX(α, v)
      return v

MIN_VALUE(s, α, β):
      if TERMINAL(s) return UTILITY(s)
      v ← +∞
      for each a in ACTIONS(s)
            v ← MIN(v, MAX_VALUE(RESULT(s, a), α, β))
            if (v ≤ α) return v
            β ← MIN(v, β)
```

```
        return v

ALPHA-BETA-SEARCH(state)
        v ← MAX_VALUE(s, −∞, +∞)
        return the action in ACTIONS(state) with value v
```

We can compare the above pseudocode to that of the **Minimax Algorithm**, whose `MAX−VALUE` function is of the following:

```
MAX_VALUE(s):
        if TERMINAL(s) return UTILITY(s)
        v ← −∞
        for each a in ACTIONS(s)
                v ← MAX(v, MIN_VAL(RESULT(s, a)))
        return v
```

With a game tree of depth d,

- the minimax algorithm has a time complexity of $O(b^d)$,
- the α-β search algorithm has a time complexity of $O(b^{\frac{d}{2}})$.

2. Evaluation Function

How do we come up with an algorithm that is faster than the alpha-beta search? We shall attempt to do so by finding a way to not generate all the nodes of the game tree up to depth d. Instead, we will try to generate the nodes to only a certain depth $l < d$.

Since the terminal nodes are not evaluated in this case, we will need to have an **evaluation function**, which acts as a heuristic and allows us to calculate the utility of each node.

The evaluation function can be of the form

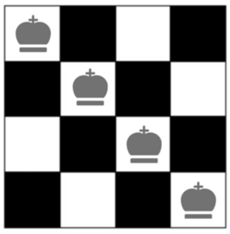
$$EVAL(s) = w_1 f_1(s) + w_2 f_2(s) + \cdots + w_n f_n(s)$$

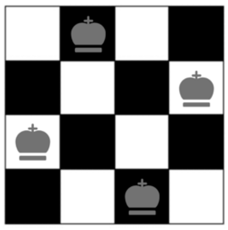
In the context of a game of chess, $f_1(s)$ can refer to the number of bishops on the chessboard at state s, $f_2(s)$ can denote the number knights, and so on.

3. N-Queen Puzzle

In the **N-Queen Puzzle**, we have an $n \times n$ chess board and n queens. We want to arrange the queens on the chess board such that they do not attack each other.

For the case of a 4×4 chess board and 4 queens, in the configuration at the left, all queens are able to attack each other. In the one on the right, however, no queens are able to attack each other.





Let us make the assumption that each queen on the chess board is only allowed to move along the *column* that it resides in.

- $\boldsymbol{N(s)}$ as the set of neighbors of a given state s, where a state can be a neighbor of s if one of its queens has moved along its column.
- $\boldsymbol{Val(s)}$ is the "value" or "quality" of a state s, and it is equal to the heuristic $h(s)$. This means that $Val(u) = 0$ if u is a goal state. Let $Val(s)$ be the number of pairs of queens that attack each other in state s.

To solve the puzzle, we will conduct a repeated calling of the Hill-Climbing Algorithm, which is made use of in the following pseudocode.

```
SOLVE_N_QUEEN:
      s ← initial state
      while (true):
            s ← HILL_CLIMB(s)
            if Val(s) = 0 break
```

4. Hill-Climbing Algorithm

The Hill-Climbing Algorithm is found below. It is similar to the Gradient Descent algorithm.

```
HILL_CLIMB(s):
      minVal ← Val(s)
      minState ← null
      for u in N(s):
            if Val(u) < minVal:
                  minState = u
                  minVal = Val(u)
      return minState
```