# Alpha Beta Pruning with the Selection Monad

Johannes Hartmann[1] and Jeremy Gibbons[1]

University of Oxford, Department of Computer Science, UK `firstname.lastname@cs.ox.ac.uk`

**Abstract.** The selection monad is widely studied in the context of sequential games. Most prominently it provides an elegant implementation of the minimax algorithm that is commonly used to determine the best move in a two-player game. However the minimax algorithm is not efficient for large game trees. The alpha beta pruning algorithm is a well-known optimization of the minimax algorithm. In this paper we show how the selection monad can be extended to support the alpha beta pruning algorithm. We provide a general implementation of the alpha beta pruning algorithm utilizing the selection monad and show how it can be applied to a simple example tree. We also show how the alpha beta pruning algorithm can be implemented using the generalized selection monad.

**Keywords:** Selection monad · Functional programming · Algorithm design · Performance Optimisation · Monads · Alpha Beta Pruning.

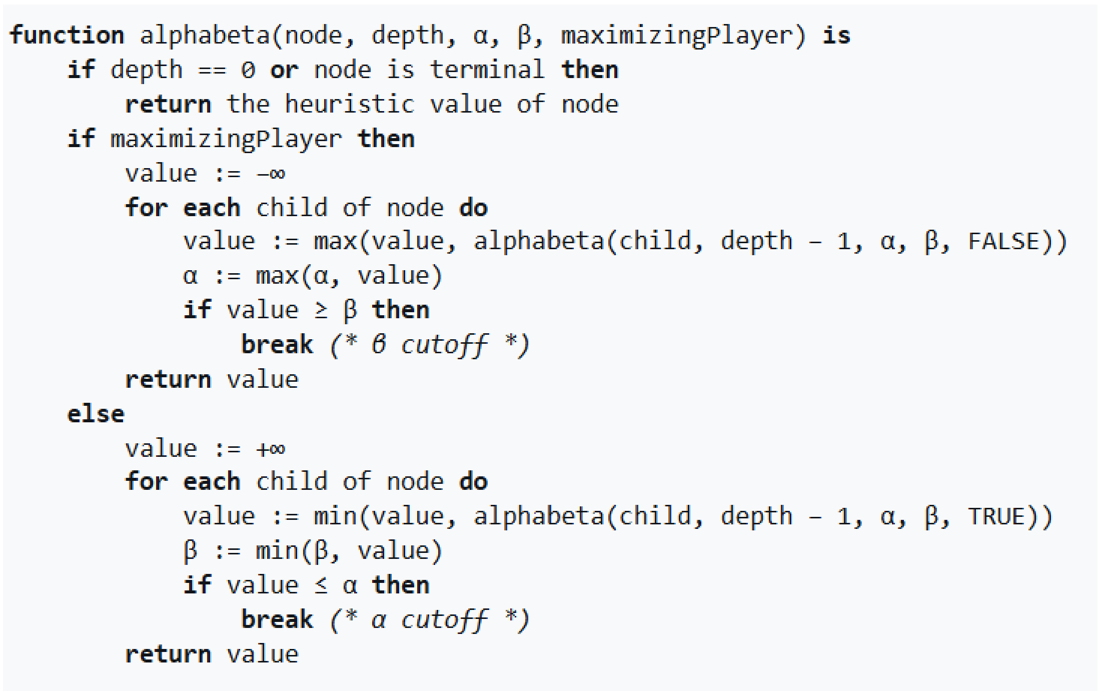
## 1 Introduction

This chapter introduces the selection monad and how it can be used to solve games with perfect information.
*TODO:* write the following:

- introduction to the selection monad and how it is used to calculate solutions to games with perfect information.
- introduction to the minimax algorithm
- teaser the use of selection functions to implement the minimax algorithm
- motivate inefficiencies and teaser alpha beta pruning

## 2 Alpha Beta Pruning

This chapter explains the alpha beta pruning algorithm.
Alpha beta pruning is a way of reducing the search space of the minimax algorithm by pruning branches that are not relevant, while still guaranteeing an optimal result.
Todo: Write the chapter and explain the algorithm
The following code haskell implementation is very similar to the one provided in the wikipedia article on alpha beta pruning:
https://en.wikipedia.org/wiki/Alpha%E2%80%93beta_pruning
*TODO:* compile this into a proper introduction to alpha beta pruning and quote original source for the algorithm and pseudo code.



```
function alphabeta(node, depth, α, β, maximizingPlayer) is
    if depth == 0 or node is terminal then
        return the heuristic value of node
    if maximizingPlayer then
        value := −∞
        for each child of node do
            value := max(value, alphabeta(child, depth − 1, α, β, FALSE))
            α := max(α, value)
            if value ≥ β then
                break (* β cutoff *)
        return value
    else
        value := +∞
        for each child of node do
            value := min(value, alphabeta(child, depth − 1, α, β, TRUE))
            β := min(β, value)
            if value ≤ α then
                break (* α cutoff *)
        return value
```

**Fig. 1.** Pseudo code of the alpha beta pruning algorithm

## 3 Alpha Beta Pruning with the Selection Monad

In this chapter we show how the selection monad can be extended to support the alpha beta pruning algorithm. We provide a general implementation of the alpha beta pruning algorithm utilizing the selection monad and show how it can be applied to a simple example tree.
First we show how the selection monad J can be utilized to implement a simple minimax algorithm

### 3.1 General J setup:

A selection function J is a function with the following type signature:

```
type J r a = (a -> r) -> a
```

Selection functions form a monad in the following way:

```
(>>=) :: J r a -> (a -> J r b) -> J r b
(>>=) e f p = f (e (p . flip f p)) p

return :: a -> J r a
return x p = x
```

Now the haskell standard library provides a function sequence function for monad that given a list of monadic values, returns a monadic list of these values. In the case of the selection function J, the sequence function can be defined as follows:

```
sequence :: [J r a] -> J r [a]
sequence [] = return []
sequence (e:es) = e >>=
                  \x -> sequence es >>=
                  \xs -> return (x:xs)
```

In the context of sequential games, we can now think of a selection function as a function that selects the best move based on a given predicate. Further, we usually have a list of previously played moves, and given that history we can calculate the current options and select the best one. We therefore extend the sequence function to take a history as an additional argument:

```
hSequenceJ :: [a] -> [[a] -> J r a] -> J r [a]
hSequenceJ h []       = return []
hSequenceJ h (ma:mas) = ma h >>=
                        \x -> hSequenceJ (h ++ [x]) mas >>=
                        \xs -> return (x:xs)
```

Now lets consider the following two selection functions, that select the optimal value from a list of values according to a given function:

```
maxWithJ :: Ord r => [a] -> J r a
maxWithJ xs f = foldr1 (\x y -> if f x >= f y then x else y) xs

minWithJ :: Ord r => [a] -> J r a
minWithJ xs f = foldr1 (\x y -> if f x < f y then x else y) xs
```

A minimax algorithm can now be implemented as the sequence of min and max selection functions:

```
optimalGame = hSequenceJ [] (take 5 all) p
  where
    all = (minWithJ . getNextMoves) : (maxWithJ . getNextMoves) : all
```

where get moves is a function that returns the possible moves from a given history of previously played moves:

```
getNextMoves :: [a] -> [a]
getNextMoves = undefined
```

This is basically performing a brute force search of the game tree, selecting the path through that game tree that represents a winning strategy.
Due to the nature of a brute force search, exploring every possible move and counter move, the minimax algorithm is not efficient for large game trees, resulting in exponential runtime complexity. The alpha beta pruning algorithm is a well-known optimization of the minimax algorithm that can reduce the search space significantly.

### 3.2 Alpha Beta pruning min and max functions:

To implement alpha beta pruning with the selection monad, we need to cary around alpha and beta values in addition to the current value. Due to the limited access to values outside of the selection function, we need to attach the alpha and beta values to the current value. We therefore define a maxWithABJ function that given a list of values with their corresponding alpha and beta values $x : (R, R, A)$ and a predicate function $p : (R, R, A) \to R$ judging each individual value, returns the value with the highest value according to the predicate function. The alpha beta pruning functionality now comes from checking for each individual value if their corresponding $R$ value is within the alpha and beta bounds. If it is not, the search will be aborted and the current best value will be returned. It further updates the alpha and beta values accordingly.

```
maxWithABJ :: Ord r => [(r, r, a)] -> J r (r, r, a)
maxWithABJ ((a, b, x) : xs) p = let r = p (a, b, x) in
            if r >= b then (a,b,x) else go (max a r, b) (r, (a, b, x)) xs
  where
   go _ best [] = snd best
   go (a, b) (rBest, best) ((_, _, x') : xs) =
     let r                = p (a, b, x')
         (rNew, newBest) = if rBest >= r then (rBest, best) else (r, (a, b, x'))
     in if r >= b then newBest else go (max a r, b) (rNew, newBest) xs

minWithABJ :: Ord r => [(r, r, a)] -> J r (r, r, a)
minWithABJ ((a, b, x) : xs) p = let r = p (a, b, x) in
            if r <= a then (a,b,x) else go (a, min b r) (r, (a, b, x)) xs
  where
   go _ best [] = snd best
   go (a, b) (rBest, best) ((_, _, x') : xs) =
     let r                = p (a, b, x')
         (rNew, newBest) = if rBest < r then (rBest, best) else (r, (a, b, x'))
     in if r < b then newBest else go (a,min r b) (rNew, newBest) xs
```

Note that the expensive operation that we want to avoid is the calculation of the predicate function. The above implementation is therefore making sure to avoid unnecessary calls to the predicate function.
The second part of the alpha beta pruning lies in making sure the correct alpha and beta values are passed to the next selection function. To understand how this is happening we will now consider the following basic example.

## 4 Example

In this example we will consider a simple game tree without any game logic attached to it. The tree is constructed as follows: Tree Setup: ———

```
data Tree a = Branch a [Tree a] | Leaf a deriving Show
```

A tree is either a branch with a label and a list of child trees, or a leaf with a value. While labels are not strictly necessary, we will later use them to identify paths through the tree. To utilize the minimax algorithm we need to define a function that returns the possible next moves from a given path through the tree. Given a path from the game start, the function getNextNodes returns the possible next moves from that path.

```
getNextNodes :: Eq a => Tree a -> [a] -> [a]
getNextNodes (Leaf x) []              = [x]
getNextNodes (Leaf x) _               = []
getNextNodes (Branch a cs) []         = [a]
getNextNodes (Branch a cs) [x]        = if x == a then getValues cs else []
getNextNodes (Branch a cs) (x:y:xs) = if x /= a then [] else getNextNodes (findNode y
```

We further need a function that given a label and a list of trees, returns the tree with that label:

```
findNode :: Eq a => a -> [Tree a] -> Tree a
findNode a ((Leaf x):xs)          | a == x     = Leaf x
                                  | otherwise  = findNode a xs
findNode a ((Branch x cs):xs)     | a == x     = Branch x cs
                                  | otherwise  = findNode a xs
```

And a function that given a list of trees, returns the values of the leafs:

```
getValues :: [Tree a] -> [a]
getValues []                = []
getValues ((Branch x _):xs) = x : getValues xs
getValues ((Leaf x ):xs)    = trace "." $ x : getValues xs
```

Now lets consider the following tree as an example:

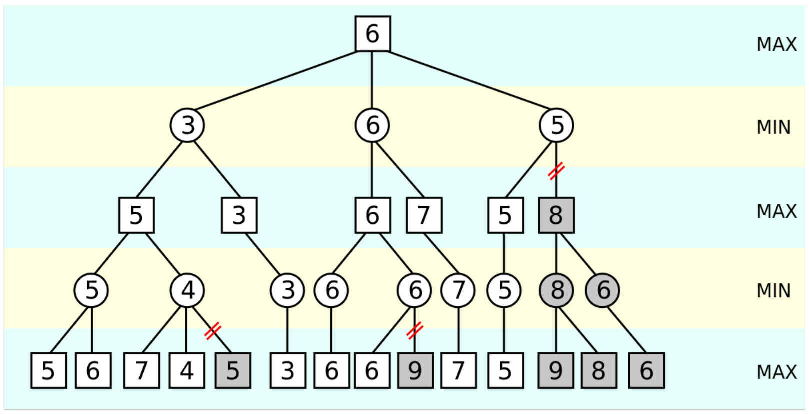


**Fig. 2.** Example tree

https://upload.wikimedia.org/wikipedia/commons/9/91/AB_pruning.svg
Which can be represented in Haskell as follows:

```
x :: Tree Int
x = Branch 1 [
       Branch 2 [
         Branch 3 [
           Branch 4 [Leaf 5, Leaf 6],
           Branch 5 [Leaf 7, Leaf 4, Leaf 5]],
         Branch 6 [
           Branch 7 [Leaf 3]]],
       Branch 8 [
         Branch 9 [
           Branch 10 [Leaf 6],
           Branch 11 [Leaf 6, Leaf 9]],
         Branch 12 [
           Branch 13 [Leaf 7]]],
       Branch 14 [
         Branch 15 [
           Branch 16 [Leaf 5]],
         Branch 17 [
           Branch 18 [Leaf 9, Leaf 8],
           Branch 19 [Leaf 6]
         ]
       ]]
```

### 4.1 Finding best path without alpha beta pruning

We now first explore how we would usually utilize the selection monad to find the best path through the tree, that would lead to an optimal outcome given that player 1 wants to maximise the outcome and player 2 wants to minimise the outcome.
To do that we will first define a predicate function that given a path through the tree, returns the value of the leaf at the end of that path:

```
p ::  [Int] -> Int
p = last
```

Given this predicate we can construct a list of alternating minimum and maximum functions where the length of the list represents the depth of the tree. Utilizing the hSequenceJ function we can then calculate the best path through the tree, and then utilizing p again on that path to get the outcome value of that optimal path.

```
exampleComp :: Int
exampleComp = p $ hSequenceJ [] (take 5 all) p
  where
    all :: Ord r => [[Int] -> J r Int]
    all = (minWithJ . getNextNodes x) : (maxWithJ . getNextNodes x) : all
```

### 4.2 Alpha Beta Pruning:

Similar to the above example, we can now utilize the alpha beta pruning functions maxWithABJ and minWithABJ to find the best path through the tree. The only difference is that we now need to attach alpha and beta values to the current value. Our predicate function now needs to be adjusted slightly to take into account the attached alpha and beta values, however as they are irrelevant to the final outcome, we can simply ignore them.

```
p' :: [(Int, Int, Int)] -> Int
p' x = let (_ , _ , r) = last x in r
```

Now similar to the above example we can construct a list of alternating min and max functions, and utilize the hSequenceJ function to find the best path through the tree. However, we need to make sure that the alpha and beta values are passed correctly to the next selection function. This is done by extending the getNodesFunction to now return all possible moves with their corresponding alpha and beta values.
So given an empty history, i.e. the start of the game, we will initialize the alpha and beta values to the minimum and maximum possible values respectively, and attach them to all possible moves at that stage. In the case where there has already been a history of moves, we will attach the alpha and beta values of the last move to the current possible moves.

```
f :: Bounded r => [(r, r, Int)] -> [(r, r, Int)]
f [] = map (minBound,maxBound,) (getNextNodes x [])
f h  = map (((\(a,_,_) -> a) . last) h,((\(_,b,_) -> b) . last) h,)
         (getNextNodes x (map (\(_,_,x) -> x) h))
```

With this extended function $f$ we can now utilize the minWith and maxWith selection functions that we previously optimized for alpha beta pruning to find the best path through the tree.

```
exampleComp' :: Int
exampleComp' = p' $ hSequenceJ [] (take 5 all) p'
```

```
  where
    all :: [[(Int, Int, Int)] -> J Int (Int, Int, Int)]
    all = (minWithABJ . f) : (maxWithABJ . f) : all
```

I was quickly verify with the haskell debug trace tool that this indeed prunes the search space as we would expect it to. It never evaluates the predicate function for the nodes that are pruned.
However due to other performance issues that come with the design of the selection monad, there are still a lot of unnecessary calls to the predicate function. They are explained in more detail in: Towards a more efficient selection function [7].

## 5 Alpha Beta Pruning with the Generalized Selection Monad

The previously presented implementation of alpha beta pruning can easily be extended to utilize the more general type for selection functions $GK$ to further benefit from the performance optimizations that come with it. These optimizations are explained in more detail in: Towards a more efficient selection function [7].
The type for selection functions $GK$ is defined as follows:

```
type GK r a = forall b. (a -> (r,b)) -> (r,b)
```

### 5.1 MinWith and MaxWith functions

For reference we will start with the unoptimized implementation of the minimax algorithm for the generalized selection function $GK$. Given the equivalent history dependend sequence function for $GK$

```
hSequenceGK :: [a] -> [[a] -> GK r a] -> GK r [a]
hSequenceGK h [] p     = p []
hSequenceGK h (e:es) p = e h (\x -> hSequenceGK (h ++ [x]) es (\xs -> p (x:xs)))
```

And the corresponding minWith and maxWith functions:

```
minWithGK :: Ord r => [x] -> GK r x
minWithGK xs f = foldr1 (\(r,x) (s,y) -> if r <= s then (r,x) else (s,y) ) ↩
(map f xs)

maxWithGK :: Ord r => [x] -> GK r x
maxWithGK xs f = foldr1 (\(r,x) (s,y) -> if r >= s then (r,x) else (s,y) ) ↩
(map f xs)
```

we can define a predicate function that accounts for the new type of $GK$:

```
pGK ::  [Int] -> (Int, [Int])
pGK [x]    = (x, [x])
pGK (x:xs) = let (r,v) = pGK xs in (r,x:v)
```

and computing an optimal path through the tree is as simple as before:

```
exampleCompGK :: (Int, [Int])
exampleCompGK = hSequenceGK [] (take 5 all) pGK
  where
    all :: Ord r => [[Int] -> GK r Int]
    all = (minWithGK . getNextNodes x) : (maxWithGK . getNextNodes x) : all
```

### 5.2 Alpha Beta Pruning with GK:

We can now further adjust the previously defined alpha beta pruning functions to work with the generalized selection function *GK*:

```
alphaMaxWith :: Ord r => [(r, r, x)] -> ((r, r, x) -> (r, y)) -> (r, y)
alphaMaxWith ((a, b, x) : xs) p = let (r,y) = p (a, b, x) in
             if r >= b then (r,y) else go (max a r, b) (r,y) xs
  where
    go _ best [] = best
    go (a, b) (r, y) ((_, _, x) : xs) =
      let (r', y') = p (a, b, x)
          newBest  = if r >= r' then (r, y) else (r', y')
      in if r >= b || r' >= b then newBest else go (max r' a, b) newBest xs

betaMinWith :: Ord r => [(r, r, x)] -> ((r, r, x) -> (r, y)) -> (r, y)
betaMinWith ((a, b, x) : xs) p = let (r,y) = p (a, b, x) in
            if r <= a then (r,y) else go (a, min b r) (r,y) xs
  where
    go _ best [] = best
    go (a, b) (r, y) ((_, _, x) : xs) =
     let (r', y') = p (a, b, x)
         newBest = if r <= r' then (r, y) else (r', y')
      in if r <= b || r' <= b
           then newBest
           else go (min r' a, b) newBest xs
```

And a predicate function that accounts for the new type of *GK*:

```
pGK' :: [(Int, Int, Int)] -> (Int, [Int])
pGK' [(a, b ,x)]     = (x,[x])
pGK' ((a, b, x):xs) = let (r,v) = pGK' xs in (r,x:v)
```

The optimal path through the tree can now be calculated as follows:

```
exampleCompGK' :: (Int, [Int])
exampleCompGK' = hSequenceGK [] (take 5 all) pGK'
  where
        es :: [(Int,Int,Int)] -> GK Int (Int, Int, Int)
        es h = alphaMaxWith (f h)
        es' :: [(Int,Int,Int)] -> GK Int (Int, Int, Int)
        es' h = betaMinWith (f h)
        all :: [[(Int,Int,Int)] -> GK Int (Int, Int, Int)]
        all = es' : es : all
```

A quick analysis with the haskell debug trace tool shows that this implementation indeed prunes the search space as expected. Attaching the debug tool to the leaf nodes, we can indeed confirm that the unoptimized implementation is visiting every leaf and does a total of 14 calls to the predicate function for the given example tree, and the optimized implementation only visits 9 leaf nodes and does a total of 9 calls to the predicate function.

## 6 Performance Analysis

This chapter is analyzing the performance improvements of alpha beta pruning with the selection monad and the generalized selection monad. To do so it will compare the number of calls to the

predicate function for the unoptimized and optimized implementation of the minimax algorithm as well as the total time spent on the computation and the memory usage.
We will run these versions of trees of different size and complexity with different potential for pruning to see how the performance of the different implementations scales with the size of the tree and the potential for pruning.
We further test it on a more complex example game to see how the performance of the different implementations scales with the complexity of the game.
*TODO:* do the performance analysis

## 7 Related and Future Work

This work extends already existing work on the selection monad in the context of sequential games. While there is extensive research on how to use the selection monad to solve games with perfect information [2][3][4], and how the selection monad can be otherwise utilized in game theory [8] [9] [1], there is only basic work on how to efficiently use the selection monad [7] [6] [5]. This work is covering various ways on how to optimize the selection monad for different use cases, but so far was lacking a concrete implementation of the alpha beta pruning algorithm with the selection monad. This work is therefore closing that gap.
Potential future work could explore how the provided implementation could be further optimized using the state or reader monad. This could very well also be part of this paper, but there exists no concrete work at this draft stage of the paper.

## 8 Conclusion

The selection monad is an elegant way of describing algorithms that explore a search tree. In its most basic form, it does so purely by exploring the tree in a depth-first manner, examine every node without any context and then choosing the optimal way. Various efforts have been made to pass around some information to have more context to base decisions on. One example would be extending the sequence function to keep track of previously chosen nodes. This way the selection monad can be used to implement the minimax algorithm. However, in order to implement alpha beta pruning, a min/max function would also need to receive information from further down the tree, to prune appropriately. This paper shows a way of obtaining that functionality and therefore enabling alpha beta pruning.
This paper further shows how alpha beta pruning can now be utilized with the selection monad, which enables a more efficient way of solving games with perfect information.

## References

1. Bolt, J., Hedges, J., Zahn, P.: Sequential games and nondeterministic selection functions. arXiv preprint arXiv:1811.06810 (2018)
2. Escardó, M., Oliva, P.: Selection functions, bar recursion and backward induction. Math. Struct. Comput. Sci. **20**(2), 127–168 (2010)
3. Escardó, M., Oliva, P.: What sequential games, the Tychonoff Theorem and the double-negation shift have in common. In: Proceedings of the third ACM SIGPLAN workshop on Mathematically structured functional programming. pp. 21–32 (2010)
4. Escardó, M., Oliva, P.: Sequential games and optimal strategies. Proceedings of the Royal Society A: Mathematical, Physical and Engineering Sciences **467**(2130), 1519–1545 (2011)

5. Hartmann, J.: Finding optimal strategies in sequential games with the novel selection monad. arXiv preprint arXiv:2105.12514 (2018)
6. Hartmann, J., Gibbons, J.: Algorithm design with the selection monad. In: International Symposium on Trends in Functional Programming. pp. 126–143. Springer (2022)
7. Hartmann, J., Schrijvers, T., Gibbons, J.: Towards a more efficient selection monad. Trends in Functional Programming, Proceedings (2024)
8. Hedges, J.: The selection monad as a cps transformation. arXiv preprint arXiv:1503.06061 (2015)
9. Hedges, J.M.: Towards compositional game theory. Ph.D. thesis, Queen Mary University of London (2016)

## Appendix