CS310 – AI Foundations

Andrew Abel

February 2024

# Week 4: Pruning

# Welcome!

- Improve minimax with pruning
  - pruning a search to improve processing time
    - Refinement of minimax
  - α and ß pruning

# Previously

- We introduced game trees through minimax
- We call min and max alternatively
- Applies to zero sum games with turn taking
- Max is trying to maximise their reward
- Min is trying to minimise their reward
- Assumes that both players play optimally

# The Minimax Algorithm

- MaxValue(state) returns a utility value

  if Terminal-Test(state) then return Utility(state)

  v ← **MinimalGameValue ( = - ∞ )**

  for s in Successors(state) do

  v ← Max(v, MinValue(s))

  return v

- MinValue(state) returns a utility value

  if Terminal-Test(state) then return Utility(state)

  v ← **MaximalGameValue ( = + ∞ )**

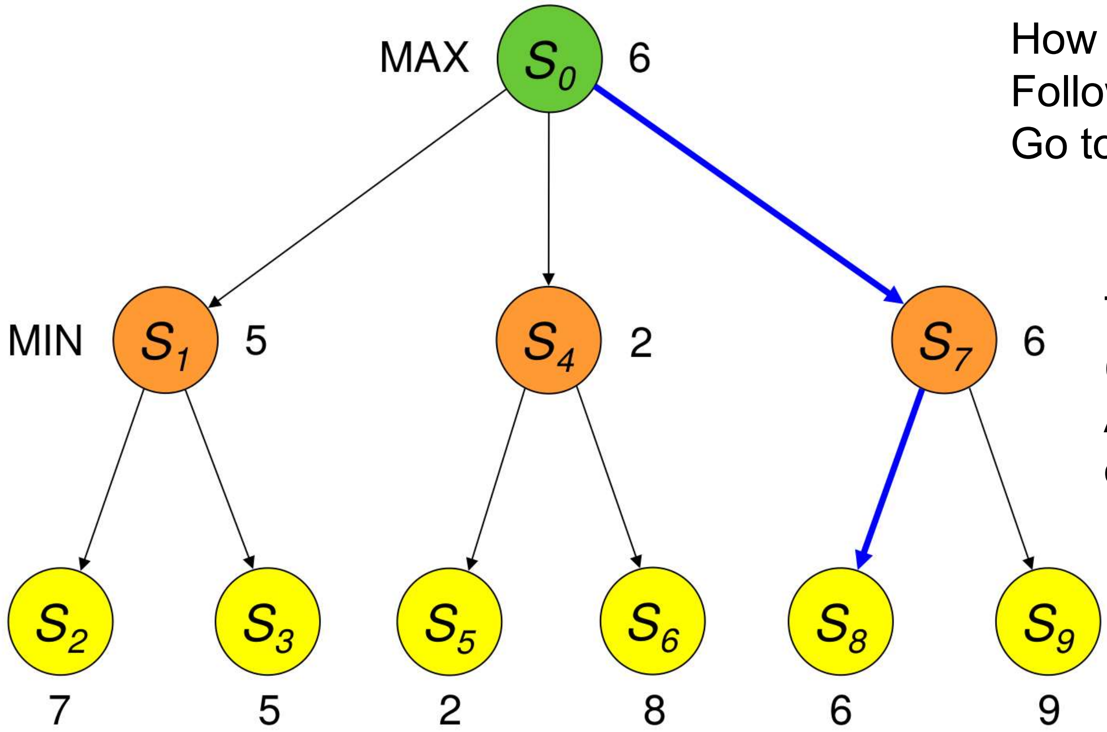
  for s in Successors(state) do

  v ← Min(v, MaxValue(s))

  return v

- If it's a terminal node, return the reward (utility)
- If its not, initially, not knowing the details, we can assume the worst for the player
- Can either calculate min value, or assume – infinity
- For successors, we compute the min values, and then take the max of the min value
- Vice versa for minValue

# The Minimax Algorithm:




How to find optimum path?
Follow all the '6' nodes
Go to successor

This maximises max reward (6 or 9)
Assuming min also plays optimally

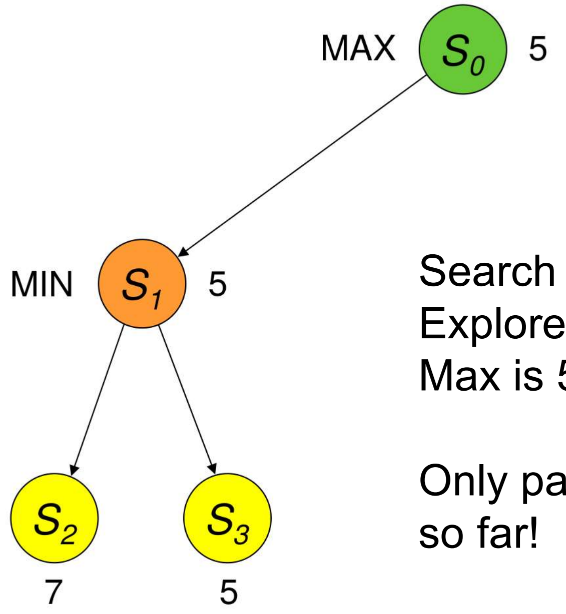
Min can only go to $S_8$ to minimise max's reward

Its not the highest reward, but it is the highest reward assuming that both players play optimally!

## Why prune the Search

- Minimax is an exhaustive search algorithm, so it is exponential in the number of moves
  - Every possible move
- Exhaustive search is not suitable
  - Even if the game takes only 20 moves, it quickly becomes unmanageable
- Even worse: inability to deal with loops/infinite plays.
  - Minimax relies on backwards induction from terminal nodes
  - How can you calculate it if there is a loop?
- We can only apply full blown minimax to very small games, or games which are close to a terminal state
- However, we can do better - sometimes we can “prune” the tree…

# Pruning the Search



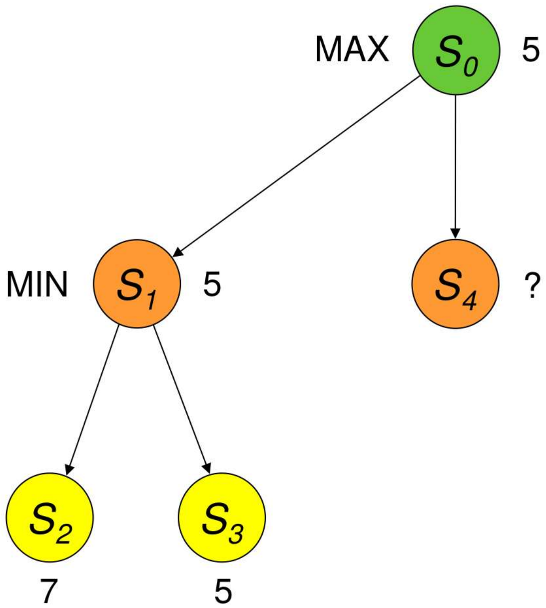

Search in progress,
Explored S1, S2, S3
Max is 5 (so far)

Only partial exploration
so far!

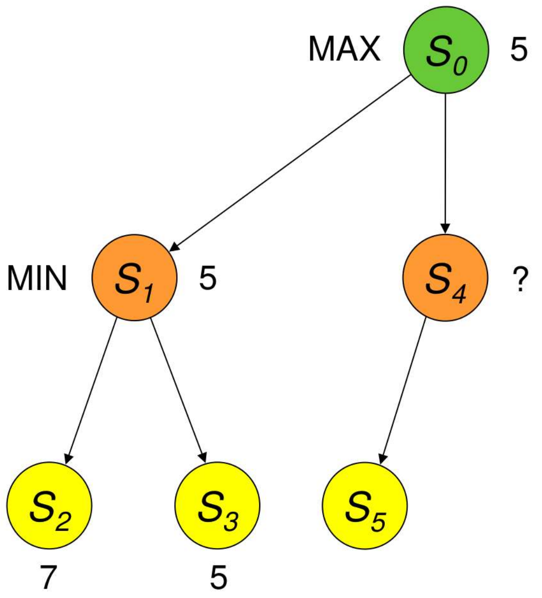
Min of 7 and 5 is 5

Max of 5 is 5

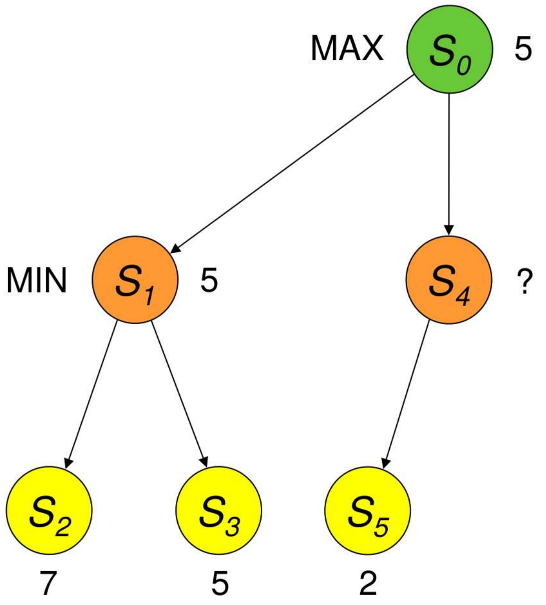
# Pruning the Search



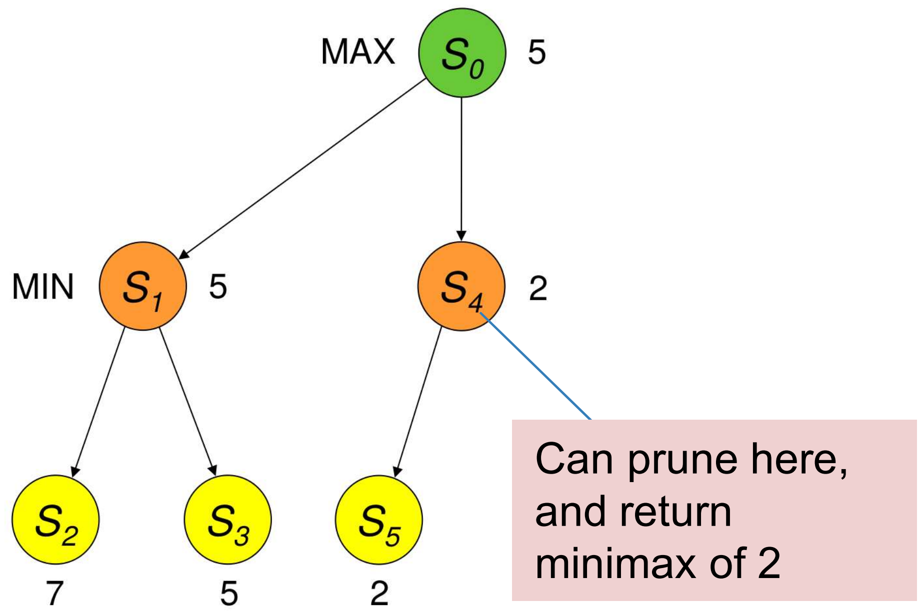
# Pruning the Search



# Pruning the Search



# Pruning the Search




As $S_5$ is below a value of 5, then it means that the value of $S_4$ will be a minimum value of 2 (or less)

Accordingly, when $S_0$ picks a max value, it will pick $S_1$

Min now has a way to limit the damage to a max of 2.

Max has a way of going to a minimum of 5

If Max chooses S4, then if Min is optimal, it will always limit the reward to 2. So why would Max choose this node, regardless of what any other children are?

So Max will NEVER choose S4

# Another Pruning Example

- Eight sticks in a row. At each turn, the player can pick up three sticks, two sticks or one stick.
- The player who picks up the last stick loses.
- Example game: Max picks up 2, then Min picks up 2,
- Max picks up 1, Min picks up 2, Max picks up the last stick and thus loses.
- Max plays first. What move should Max make?
- We can prune the search once we are sure that Max has a certain win at any node

## Another Pruning Example - Pointers

- If there are between 2 and 4 sticks left, you win
  - 4 left: Take 3, leave 1, other player loses
  - 3 left: Take 2, leave 1, other player loses
  - 2 left: Take 1, leave 1, other player loses
- Therefore, first move should be to have 4,3, or 2 sticks left by Max’s second move
- Should move from [8] to [5]
  - i.e. pick up 3 sticks
- At [5], min has to take either 1 [4], 2 [3] or 3 [2] sticks
  - Means that max will always win

## Informal Pruning

- Focus on winning positions
  - What conditions can a player have the game won?
  - Can help to remove other options from the tree
- But these are very informal!

# α and ß pruning

## Alpha-Beta Pruning

- Alpha-Beta pruning is a method for ignoring branches of the search tree, while still finding the optimal move
  - Formalises the examples in previous slides
  - Allows us to ignore irrelevant branches of the search tree
- Given a state being explored, alpha (α) is the value of the best alternative for Max along the path to the state
  - While searching, α is the current best max number on this path
  - In other words, Max can get at least α out of the game in this part of the game tree, i.e. along current path
  - If it doesn’t match α, then it can be pruned
- beta (ß) is the value of the best alternative for Min along the path to the terminal state
  - In other words, Min can minimise Max's reward to be at most ß in this part of the game tree
  - If min down a tree is higher than ß, can be pruned

## Recap: Minimax Algorithm

- MaxValue(state) returns a utility value

  if Terminal-Test(state) then return Utility(state)

  v ← **MinimalGameValue ( = - ∞ )**

  for s in Successors(state) do

      v ← Max(v, MinValue(s))

  return v

MaxValue
Takes the min value of the max values of the successors

- MinValue(state) returns a utility value

  if Terminal-Test(state) then return Utility(state)

  v ← **MaximalGameValue ( = + ∞ )**

  for s in Successors(state) do

      v ← Min(v, MaxValue(s))

  return v

MinValue
Takes the max value of the min values of the successors

## Function Max-Value (state, α , ß )

- Max-Value(state, α , ß ) returns a utility value

  if Terminal-Test(state) then return Utility(state)

  v ← MinimalGameValue (initialize as - ∞ )

  for s in Successors(state) do

  v’ ← Min-Value(s, α , ß )

  if v' > v, v ← v'

  v' ≥ ß then return v

  if v' > α then α ← v‘

  return v

# Function Max-Value (state, α , ß )

- Max-Value(state, α , ß ) returns a utility value

  if Terminal-Test(state) then return Utility(state)    (same as minimax!)

  v $\leftarrow$ MinimalGameValue

  for s in Successors(state) do

    v' $\leftarrow$ Min-Value(s, α , ß )

    if v' > v, v $\leftarrow$ v'    (update v if v' is better)

    v' ≥ ß then return v    (**prune!** Min will prefer ß)

    if v' > α then α $\leftarrow$ v'    (update α if v' is better)

  return v

If val is greater than alpha, then set a new alpha!

If value is greater than beta, then this means a higher min value. So why would max go down this path? We can prune!

# Function Min-Value (state, α , ß )

- Min-Value(state, α , ß ) returns a utility value

  if Terminal-Test(state) then return Utility(state)
  v ← MaximalGameValue
  for s in Successors(state) do

  v' ← Max-Value(s, α , ß )

  if v' < v, v ← v'

  v' ≤ α then return v

  if v' < ß then ß ← v'

  return v

If value is smaller than alpha, then reward for max is smaller, so max would never go down this route

Update beta if beta is smaller

## Function Min-Value (state, α , ß )

- Min-Value(state, α , ß ) returns a utility value

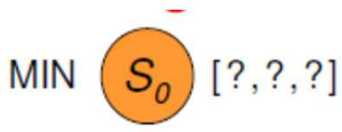
```
if Terminal-Test(state) then return Utility(state)
v ← MaximalGameValue
for s in Successors(state) do
    v’ ← Max-Value(s, α , ß )
    if v' < v, v ←  v'                (update v if v' is better)
    v' ≤ α then return v              (prune! Max will prefer α)
    if v' < ß then ß ← v‘             (update ß if v' is better)
return v
```

# Initial values for α and ß

- we start the algorithm by calling it with $\alpha = -\infty$ and $ß = +\infty$

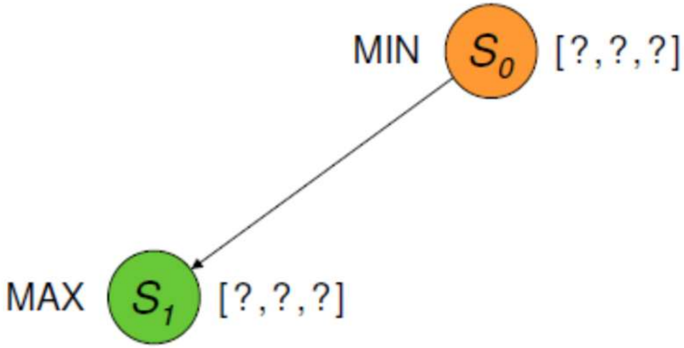
- We'll have a look at  detailed examples next!

# Example of ß-Pruning



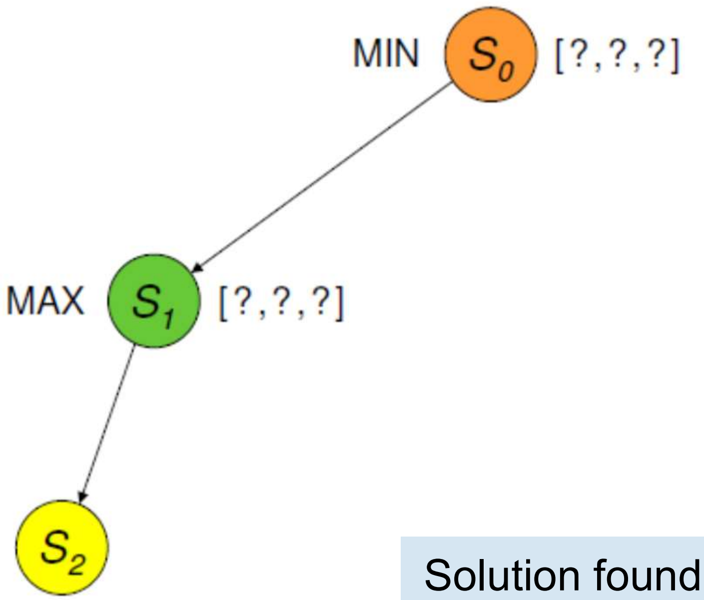

Min = unknown
Alpha = - ∞
Beta = + ∞

# Example of ß-Pruning



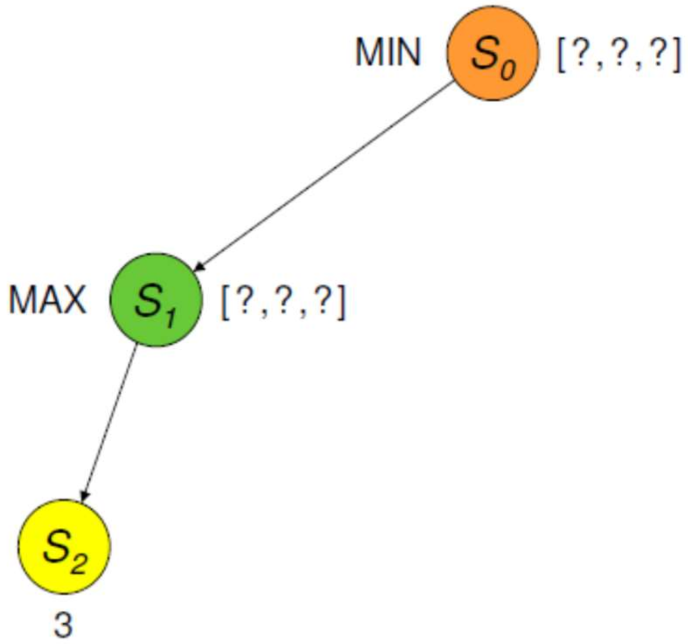

Started with min, so call max
Values unchanged

# Example of ß-Pruning

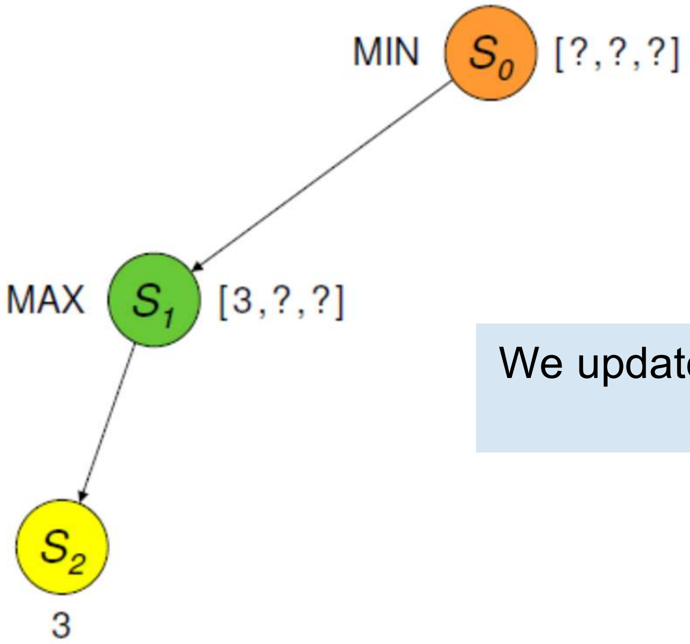



Solution found!
Can update some values

# Example of ß-Pruning

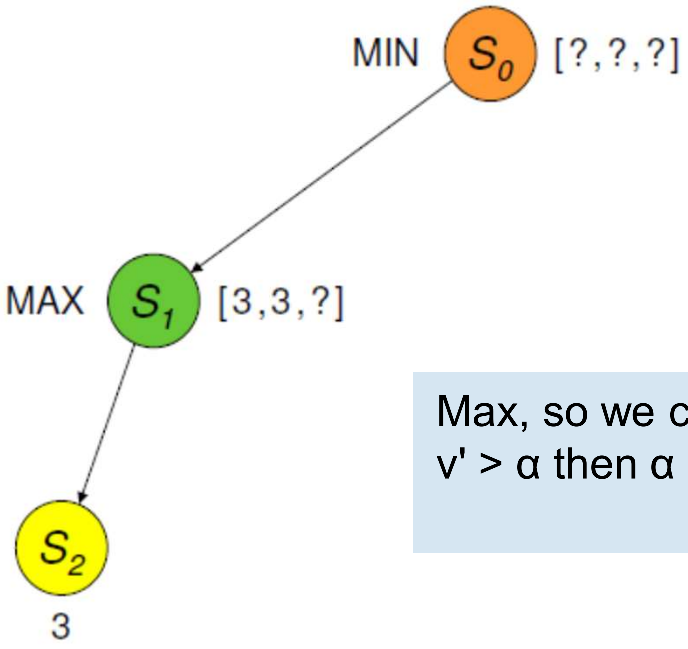


# Example of ß-Pruning



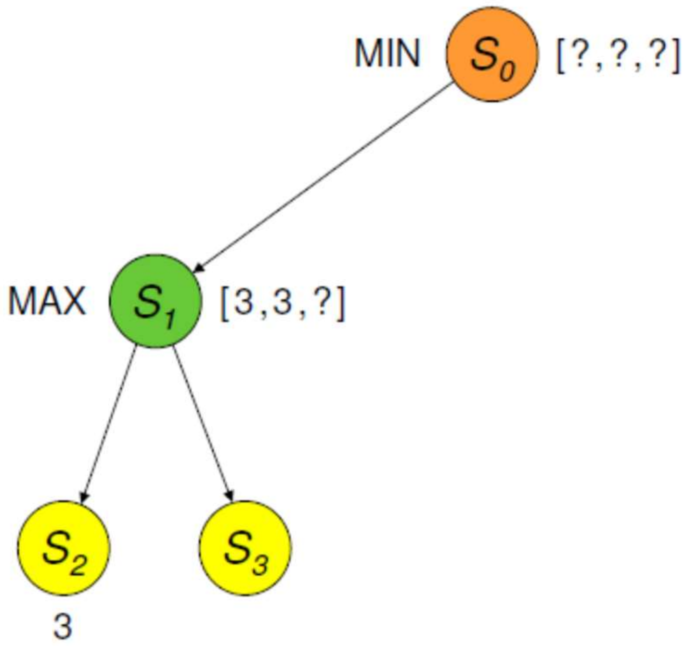

We update current value

# Example of ß-Pruning

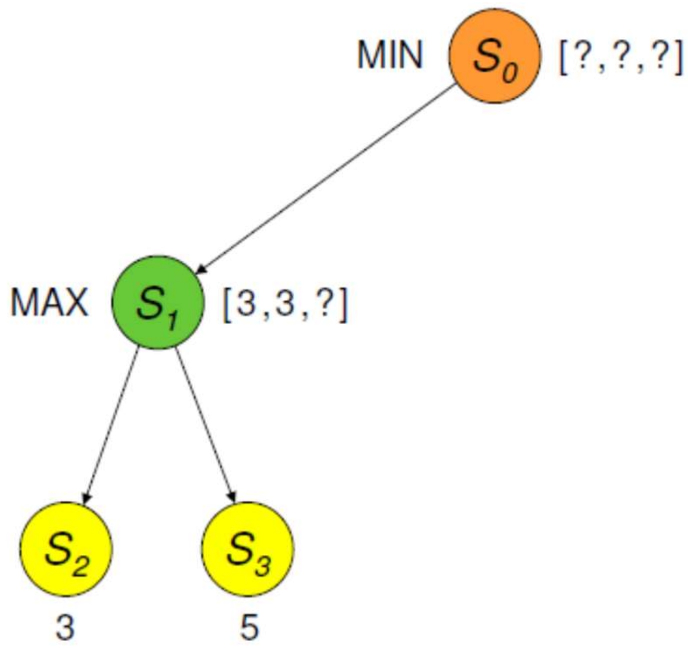



Max, so we can update alpha
v' > α then α ← v‘

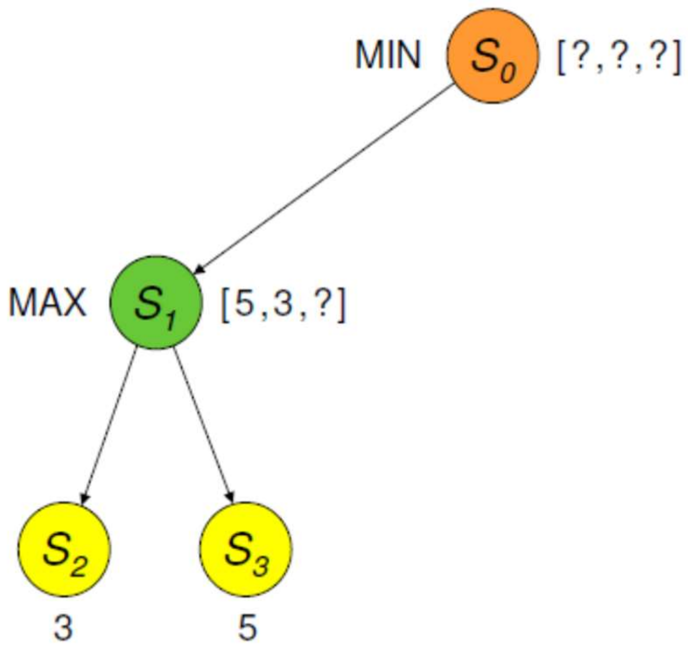
# Example of ß-Pruning



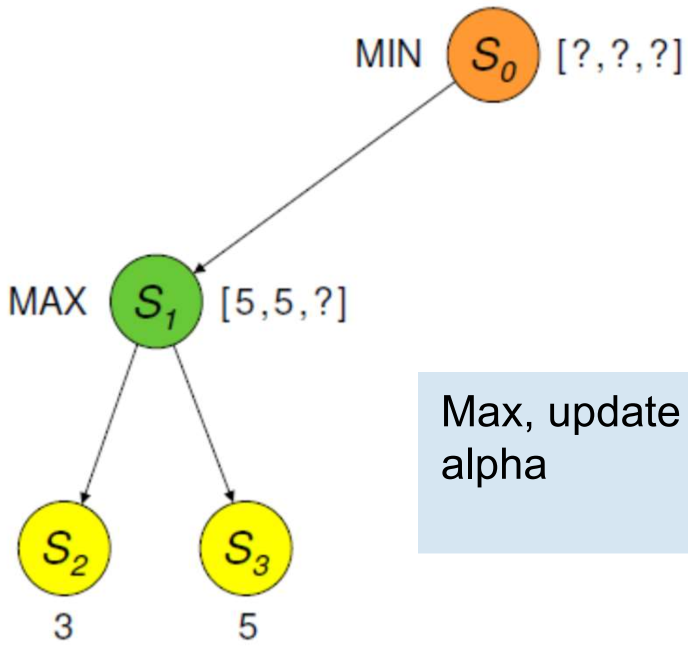
# Example of ß-Pruning



# Example of ß-Pruning

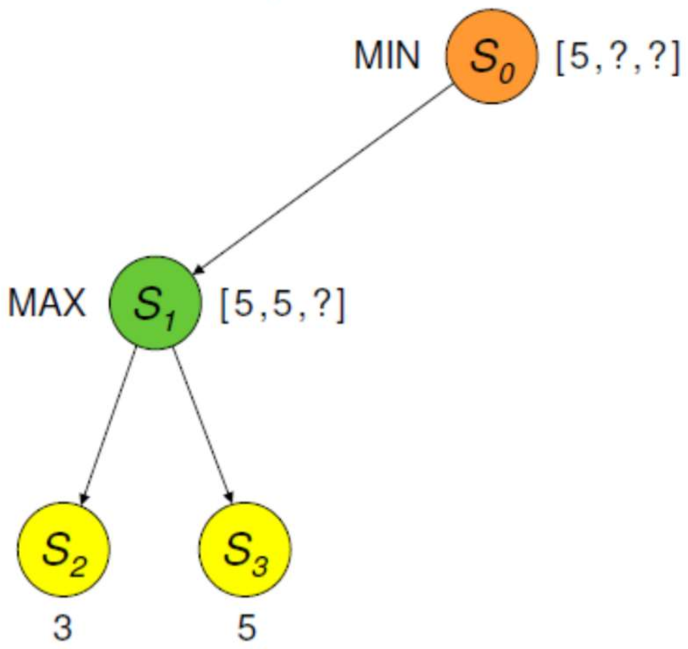


# Example of ß-Pruning



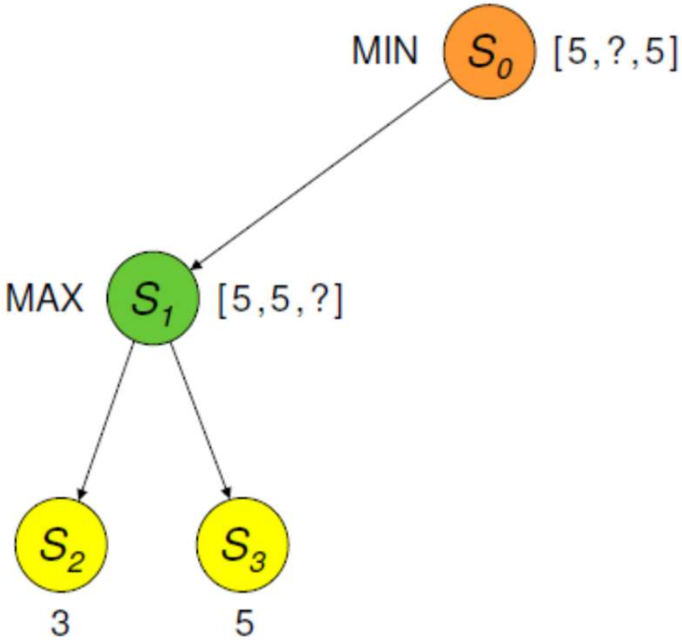

Max, update max value, and also alpha

# Example of ß-Pruning



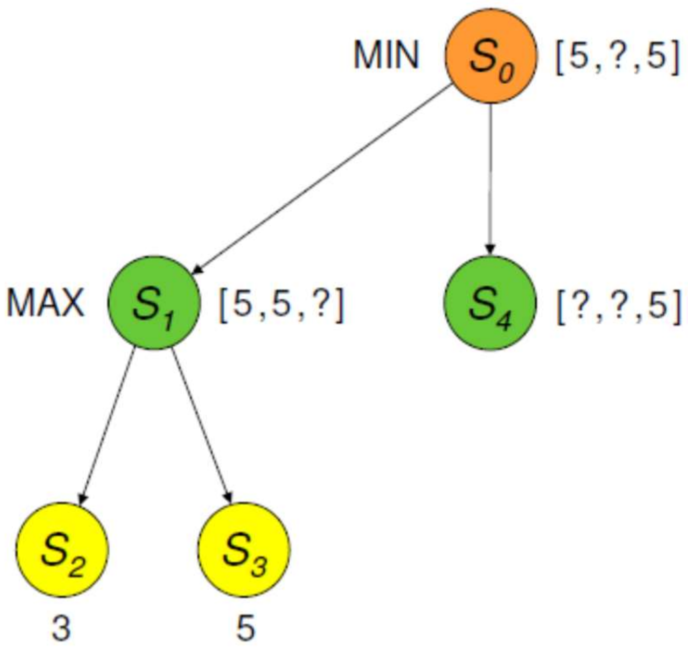

Min value of this branch is 5

# Example of ß-Pruning



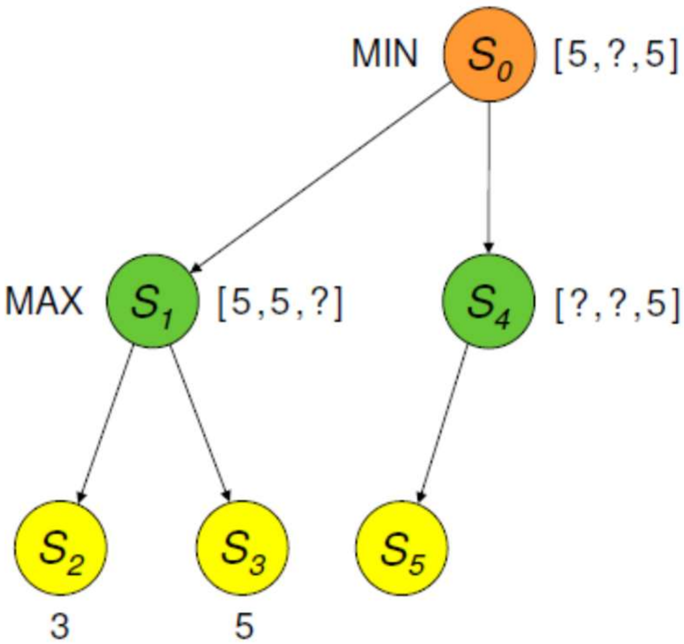

Can update beta

# Example of ß-Pruning

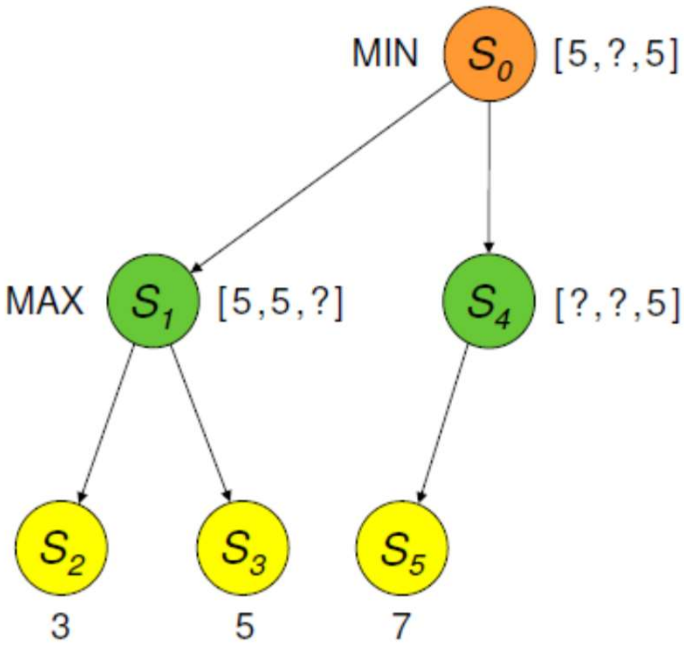



Improved beta can travel down path

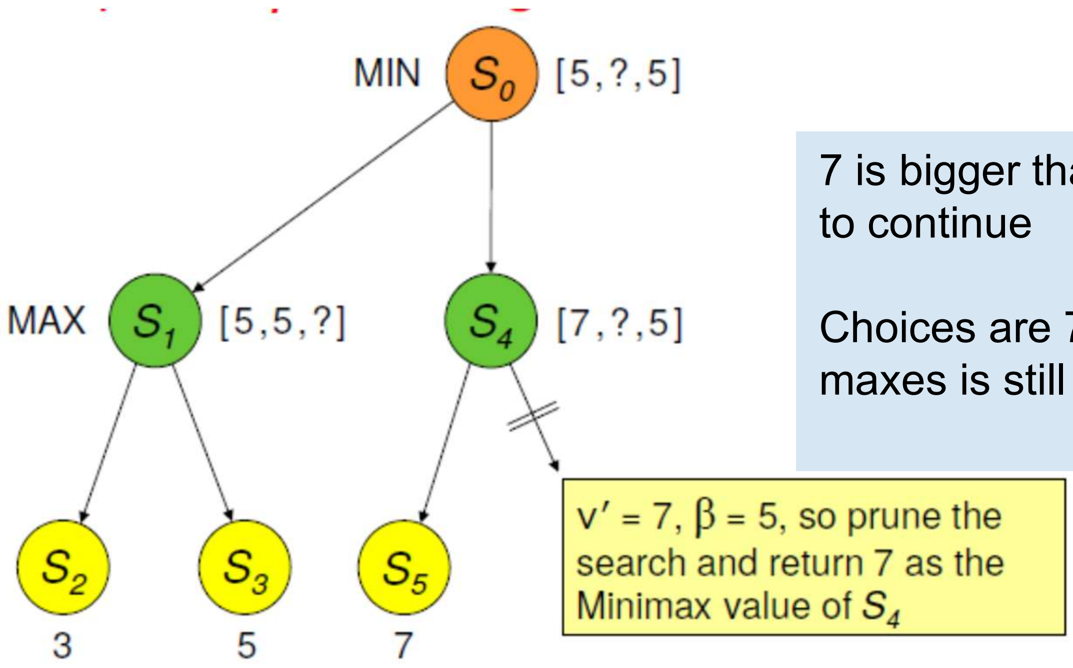
# Example of ß-Pruning



# Example of ß-Pruning



# Example of ß-Pruning




7 is bigger than beta, so no need to continue

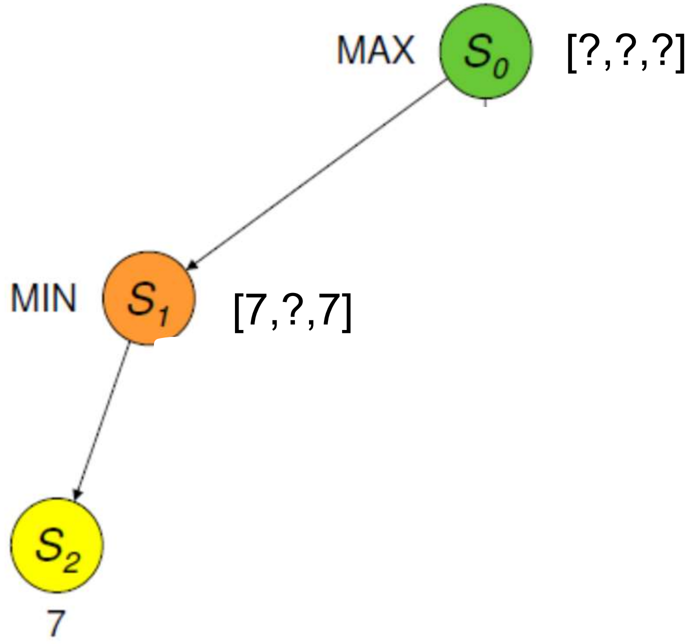
Choices are 7 and 5, min of maxes is still 5

# Example of α-pruning

- Similar, except start with max
  - i.e. max goes first

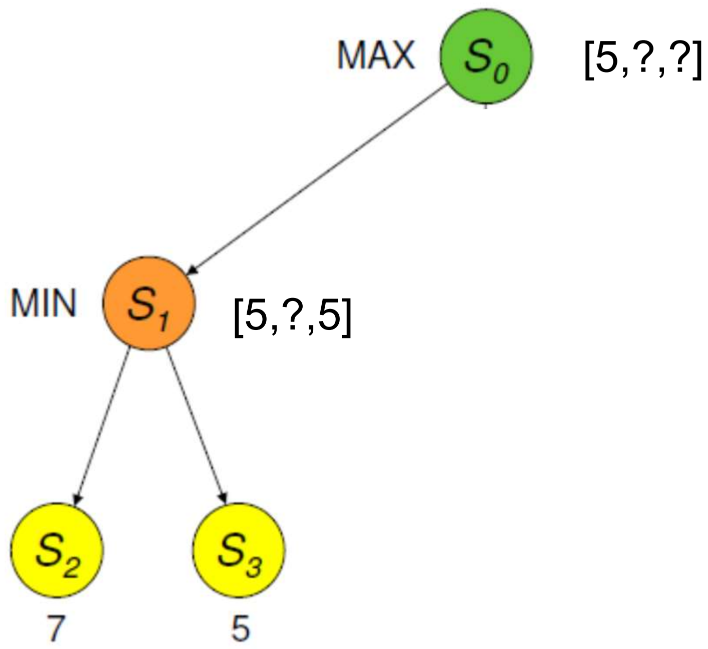
# Example of α-pruning

- First path, update beta



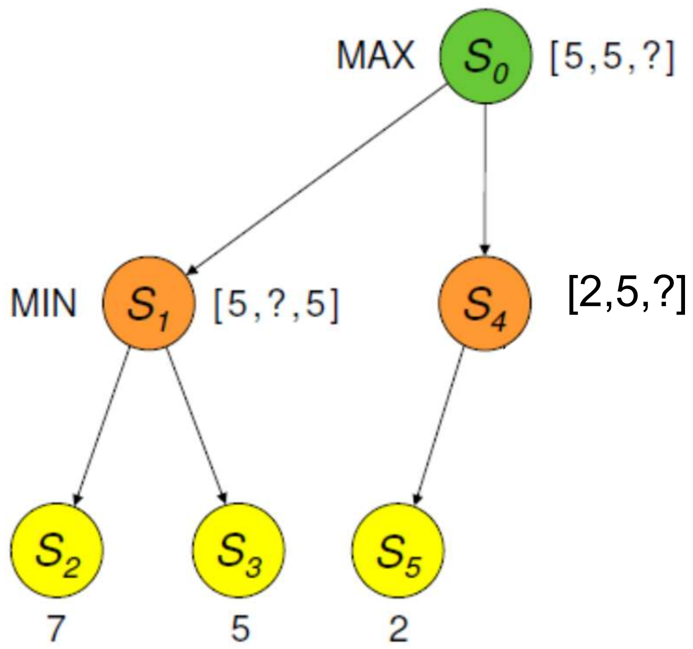
# Example of α-pruning

- 5 < 7, so update vals
- Can update min and beta



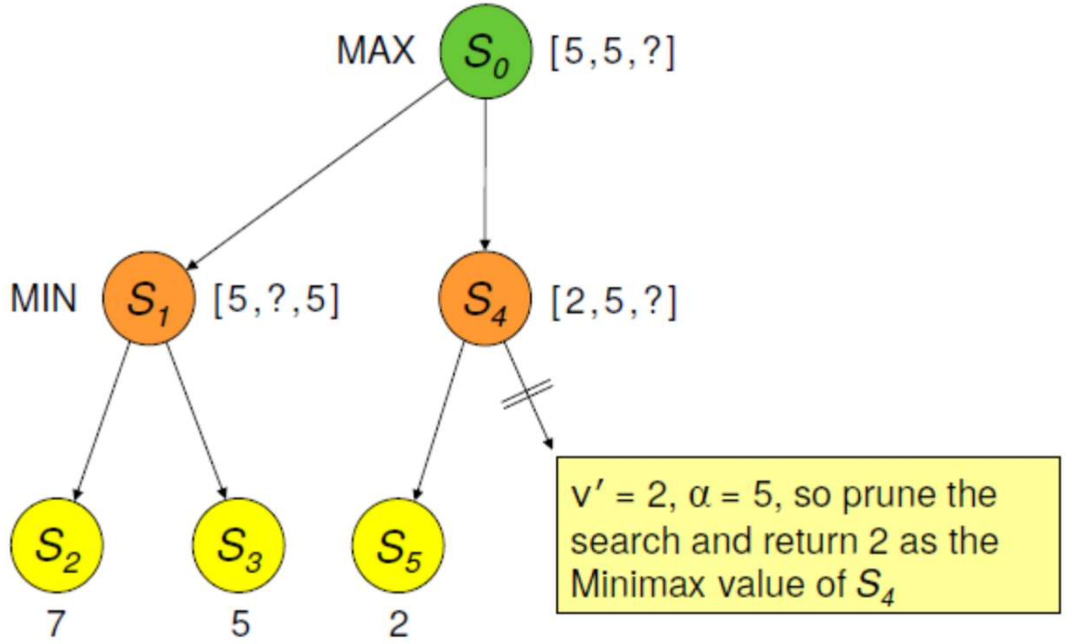
# Example of α-pruning

- Similar, except start with max
  - i.e. max goes first



# Example of α-pruning

- Similar, except start with max
  - i.e. max goes first



## How Effective is Pruning?

- The effectiveness of pruning is highly dependent on the order in which successors are examined
  - Optimal if optimal branch examined first!
- In the best case, we can get $O(b^{m/2})$ rather than $O(b^m)$
  - Moves, rather than depth
- This means that alpha-beta can look ahead roughly twice as far as minimax in the same amount of time
- However, we can still only apply this to small games, chess and 5 x 5 dots and boxes are still out of reach!
- Next week: how to cope if you've only got a limited amount of time to make your decision...

## Summary

- We've looked at two player zero-sum games of perfect information
- In these games, we can do a game tree search
- We use minimax and alpha beta pruning to search the game tree to find the optimal move