

Alpha-Beta Search

CMPUT 355: Games, Puzzles, and Algorithms

Lecture Outline

1. Logistics & Recap
2. Pruning
3. Alpha-beta search
4. Examples

Logistics

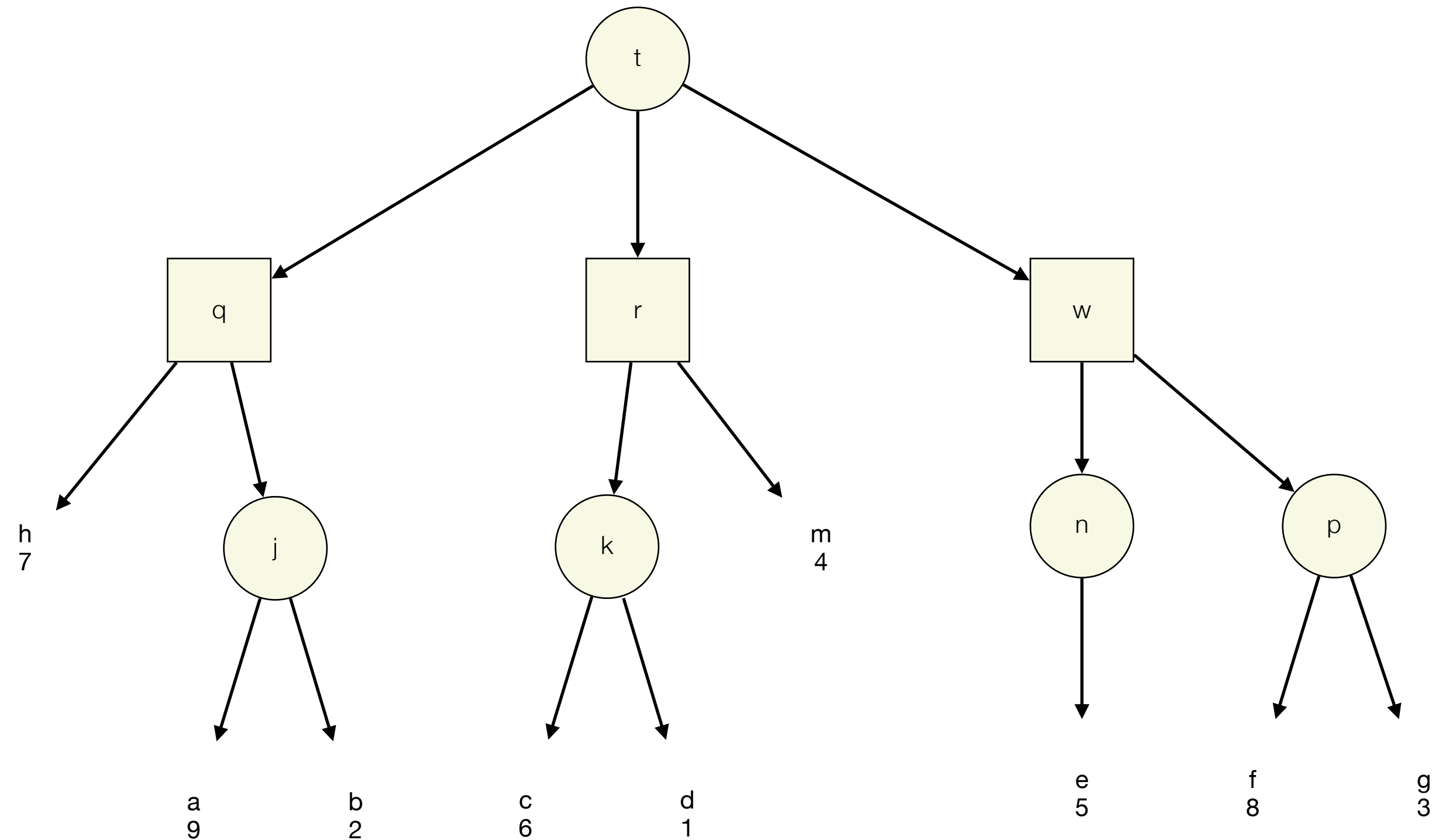
- **Practice quiz questions #2:**
 - Now available
 - Answers posted yesterday
- **Quiz 1 marks available** (average: approximately 14/20)
 - Solutions have been posted
 - Scans are on Canvas
 - Any concerns about grading should be raised in a comment on the Canvas submission
- **Quiz 2:** Friday, **Feb 6**
 - In-class, full 50 minutes
 - **No need to email** if you have to miss it;
up to 3 missed quizzes replaced by final exam **automatically**
 - Coverage: up to the end of **last Friday's lecture** (Sliding tiles & subgoals)
 - Questions will be very similar to practice questions

Recap: Minimax Search

```
assume P1 plays at root
assume players alternate turns

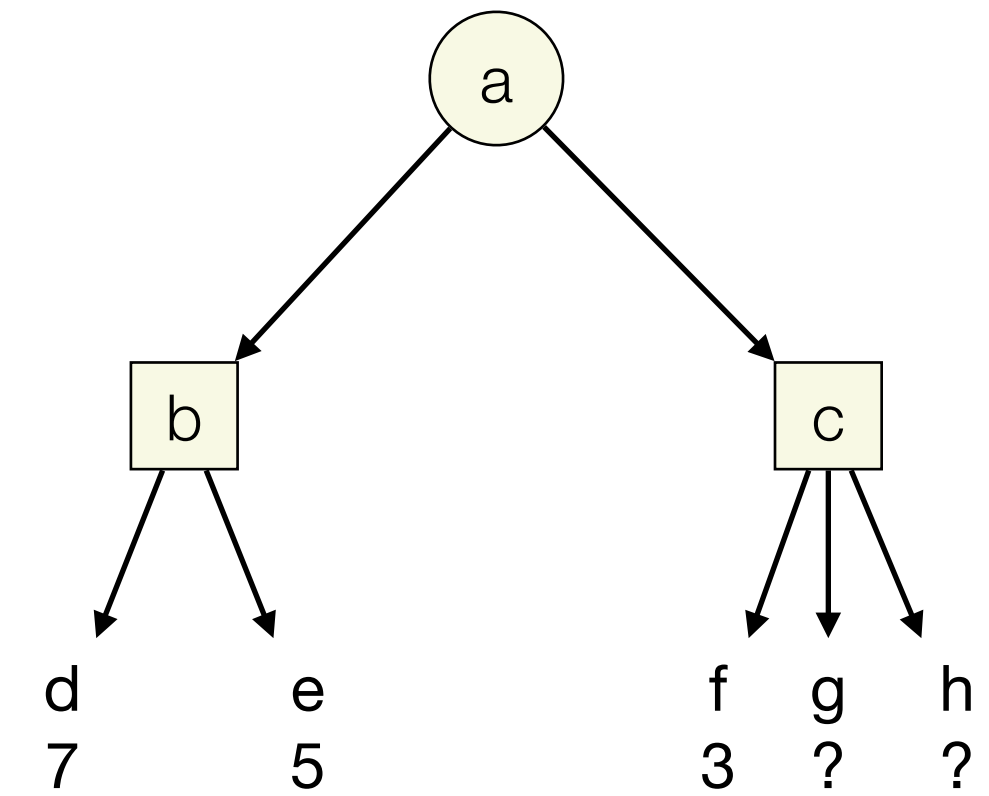
def score(s):
    return P1's score at state s

def minimax(s):
    if terminal(s):
        return score(s)
    if player(s) == 1:
        return max{minimax(c) for
all c in children(s)}
    if player(s) == 2:
        return min{minimax(c) for
all c in children(s)}
```



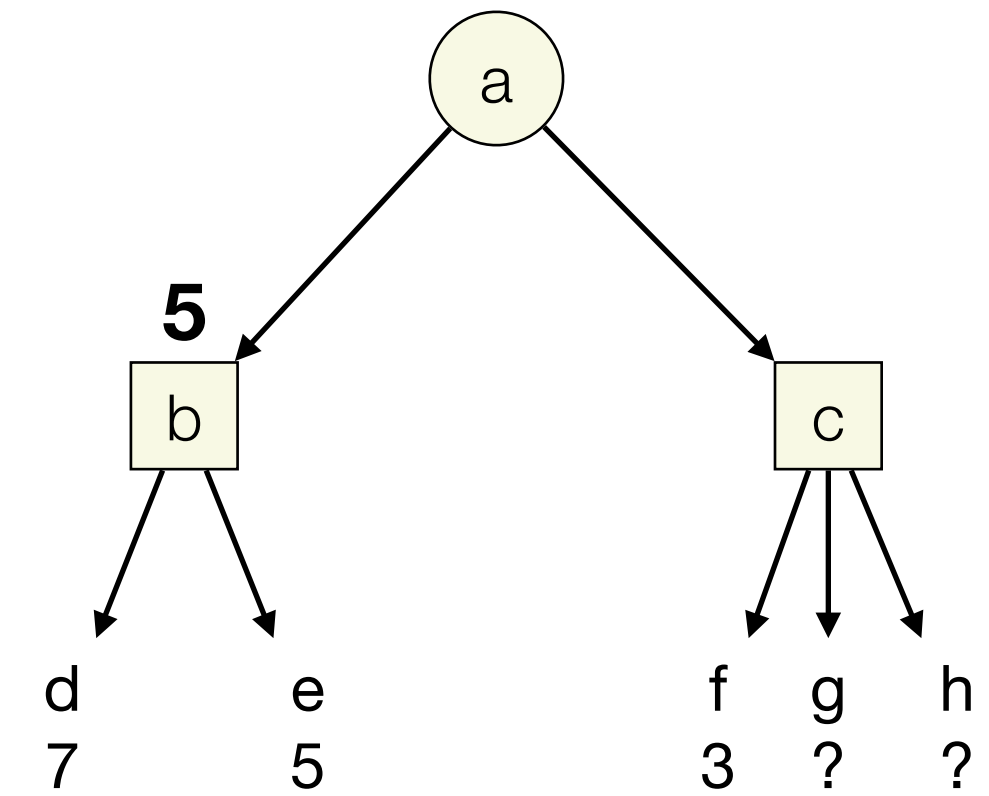
Pruning

- Plain minimax search visits every node in depth-first order
- But it's sometimes possible to avoid searching some subtrees based on information partway through the algorithm
- Once we have explored the **b**'s subtree and the first child of **c**, we know:
 - minimax value of **b** is 5
 - minimax value of **c** is **no larger than 3** (**why?**)
- But **a** is a max node!
 - **No matter what** minimax value of **c** turns out to be, P1 will choose **b** instead (**why?**)
 - So minimax value of **a** is 5
 - ...even though we haven't explored **g** or **h** yet
 - those nodes were **pruned**
- **Alpha-beta search** checks as it goes for these pruning opportunities



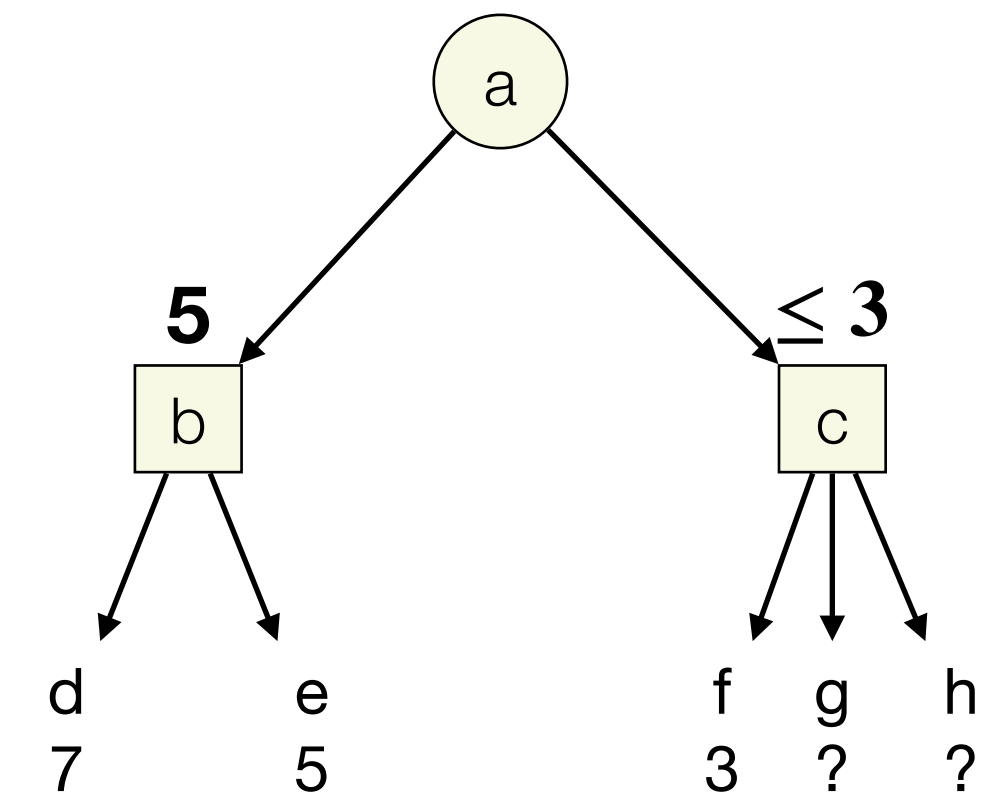
Pruning

- Plain minimax search visits every node in depth-first order
- But it's sometimes possible to avoid searching some subtrees based on information partway through the algorithm
- Once we have explored the **b**'s subtree and the first child of **c**, we know:
 - minimax value of **b** is 5
 - minimax value of **c** is **no larger than 3** (**why?**)
- But **a** is a max node!
 - **No matter what** minimax value of **c** turns out to be, P1 will choose **b** instead (**why?**)
 - So minimax value of **a** is 5
 - ...even though we haven't explored **g** or **h** yet
 - those nodes were **pruned**
- **Alpha-beta search** checks as it goes for these pruning opportunities



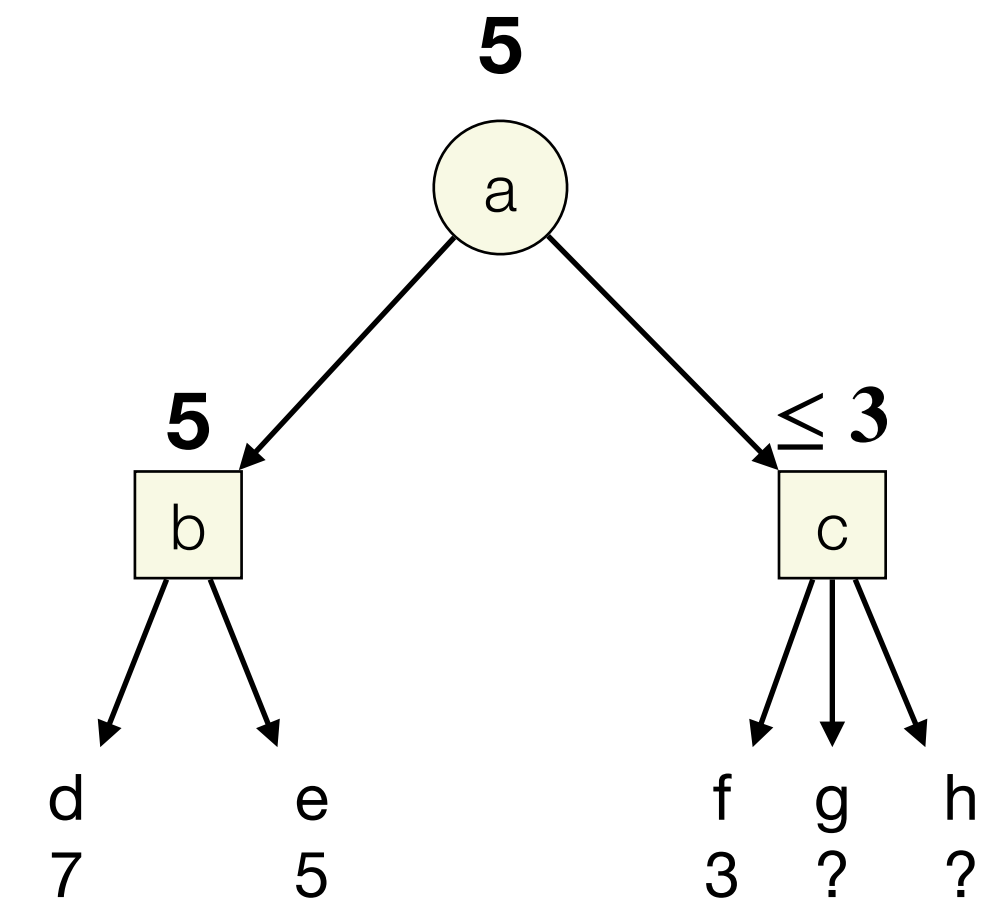
Pruning

- Plain minimax search visits every node in depth-first order
- But it's sometimes possible to avoid searching some subtrees based on information partway through the algorithm
- Once we have explored the **b**'s subtree and the first child of **c**, we know:
 - minimax value of **b** is 5
 - minimax value of **c** is **no larger than 3** (**why?**)
- But **a** is a max node!
 - **No matter what** minimax value of **c** turns out to be, P1 will choose **b** instead (**why?**)
 - So minimax value of **a** is 5
 - ...even though we haven't explored **g** or **h** yet
 - those nodes were **pruned**
- **Alpha-beta search** checks as it goes for these pruning opportunities



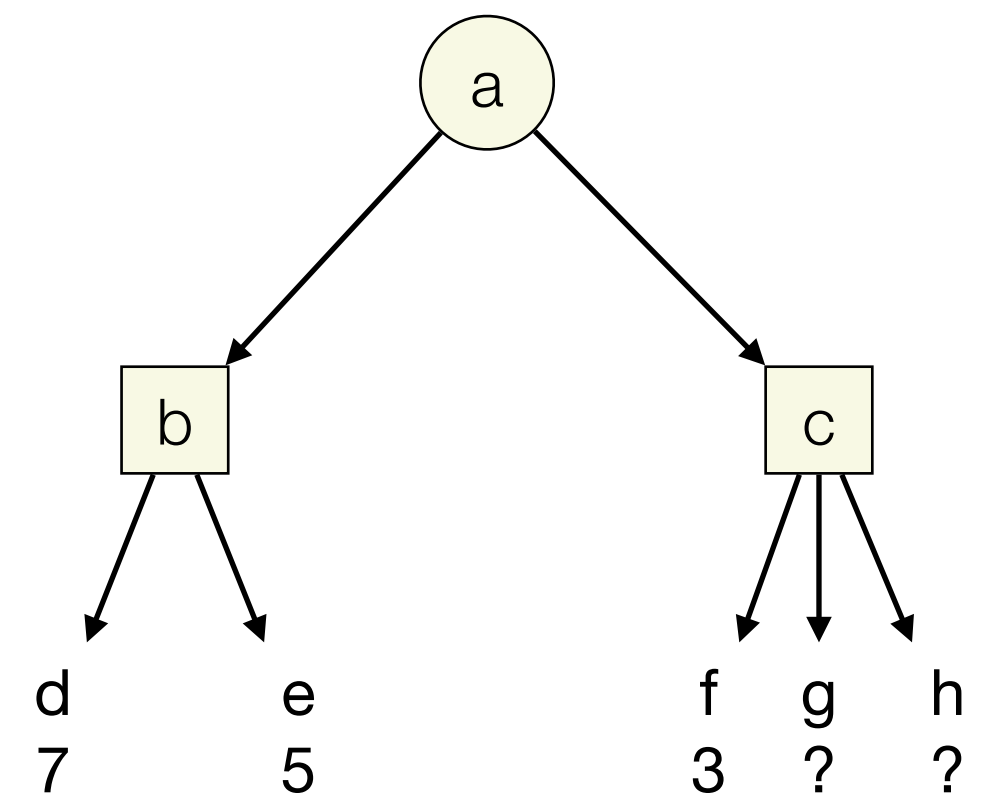
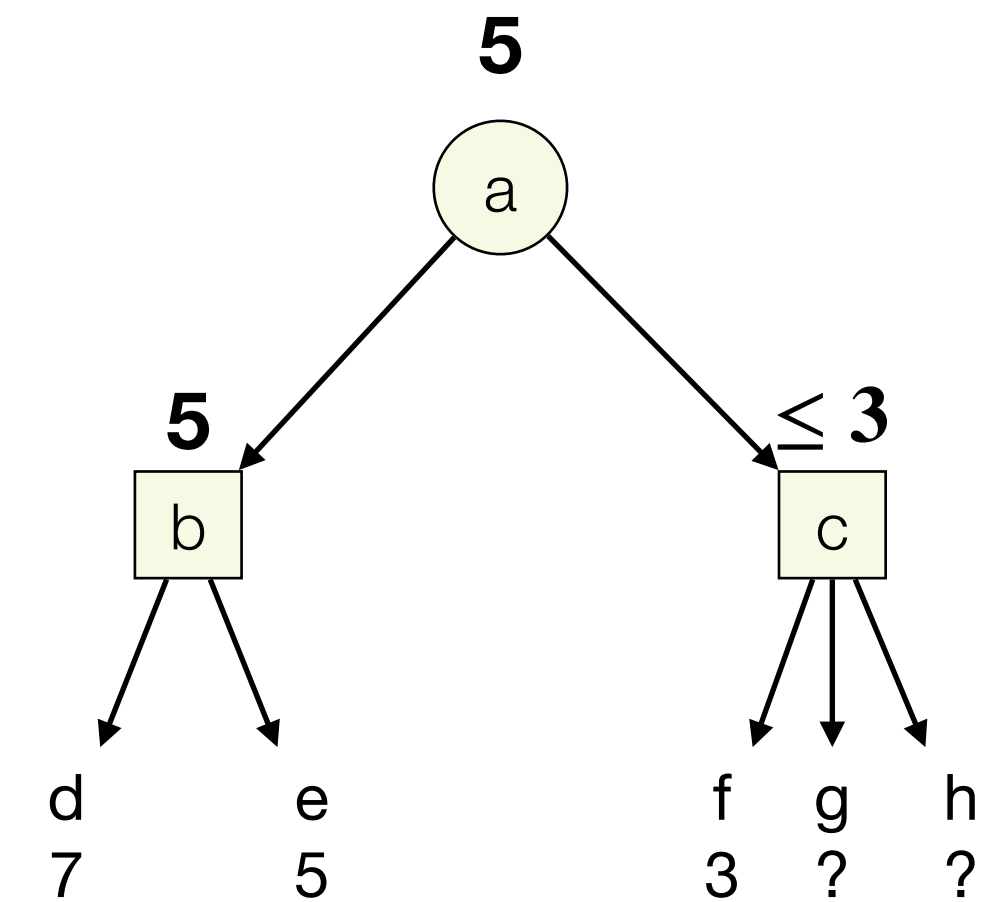
Pruning

- Plain minimax search visits every node in depth-first order
- But it's sometimes possible to avoid searching some subtrees based on information partway through the algorithm
- Once we have explored the **b**'s subtree and the first child of **c**, we know:
 - minimax value of **b** is 5
 - minimax value of **c** is **no larger than 3** (**why?**)
- But **a** is a max node!
 - **No matter what** minimax value of **c** turns out to be, P1 will choose **b** instead (**why?**)
 - So minimax value of **a** is 5
 - ...even though we haven't explored **g** or **h** yet
 - those nodes were **pruned**
- **Alpha-beta search** checks as it goes for these pruning opportunities



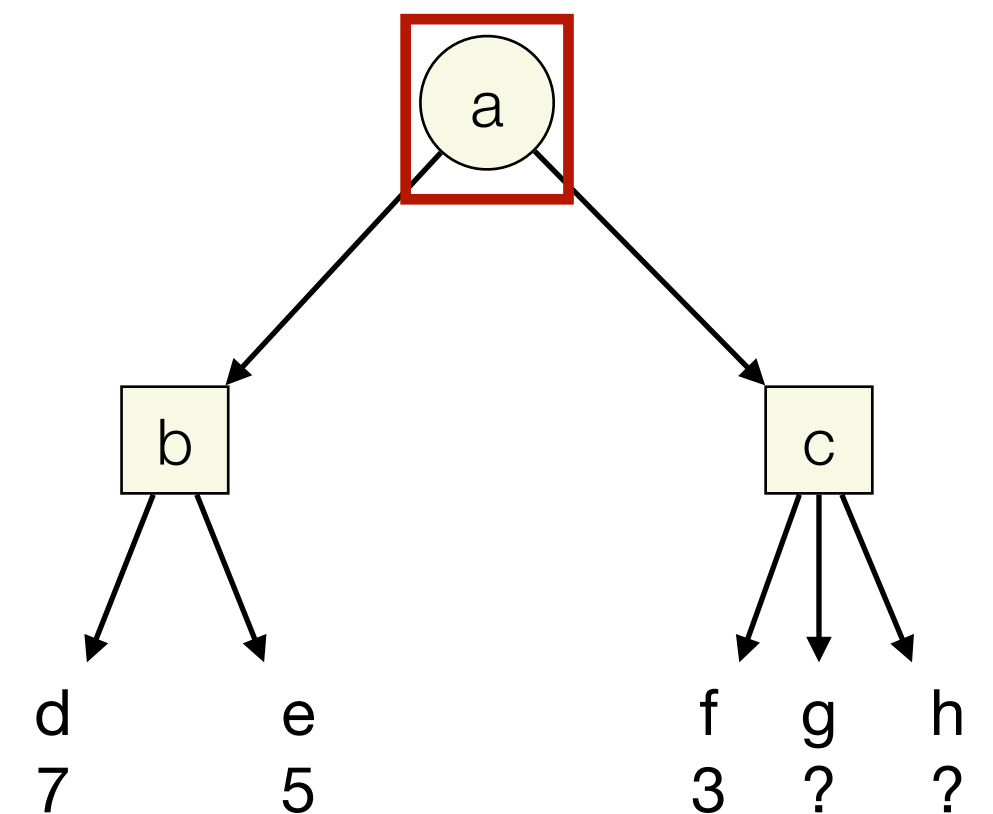
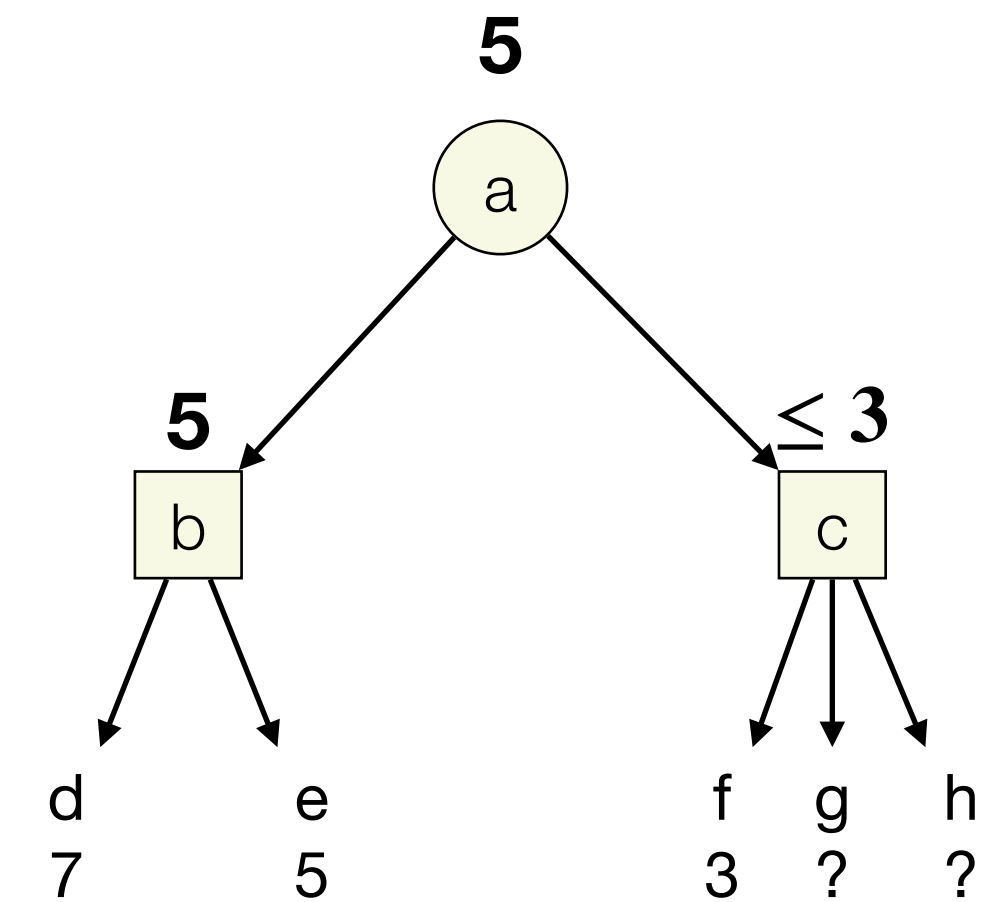
Pruning

- Plain minimax search visits every node in depth-first order
- But it's sometimes possible to avoid searching some subtrees based on information partway through the algorithm
- Once we have explored the **b**'s subtree and the first child of **c**, we know:
 - minimax value of **b** is 5
 - minimax value of **c** is **no larger than 3** (**why?**)
- But **a** is a max node!
 - **No matter what** minimax value of **c** turns out to be, P1 will choose **b** instead (**why?**)
 - So minimax value of **a** is 5
 - ...even though we haven't explored **g** or **h** yet
 - those nodes were **pruned**
- **Alpha-beta search** checks as it goes for these pruning opportunities



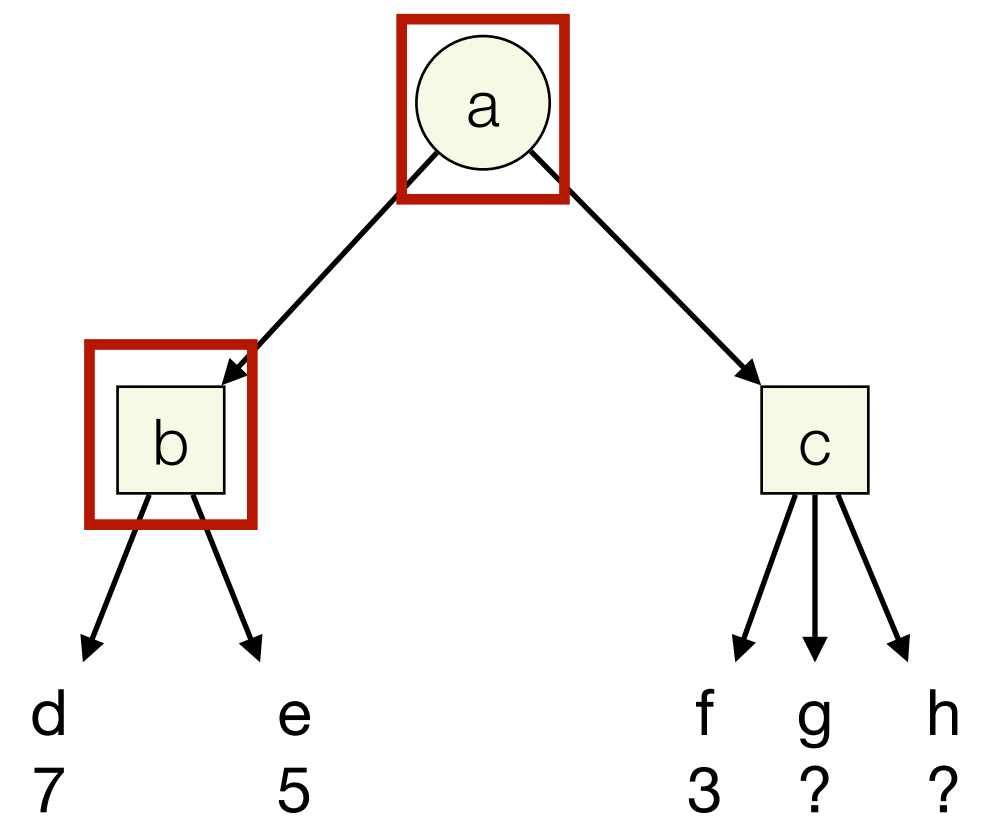
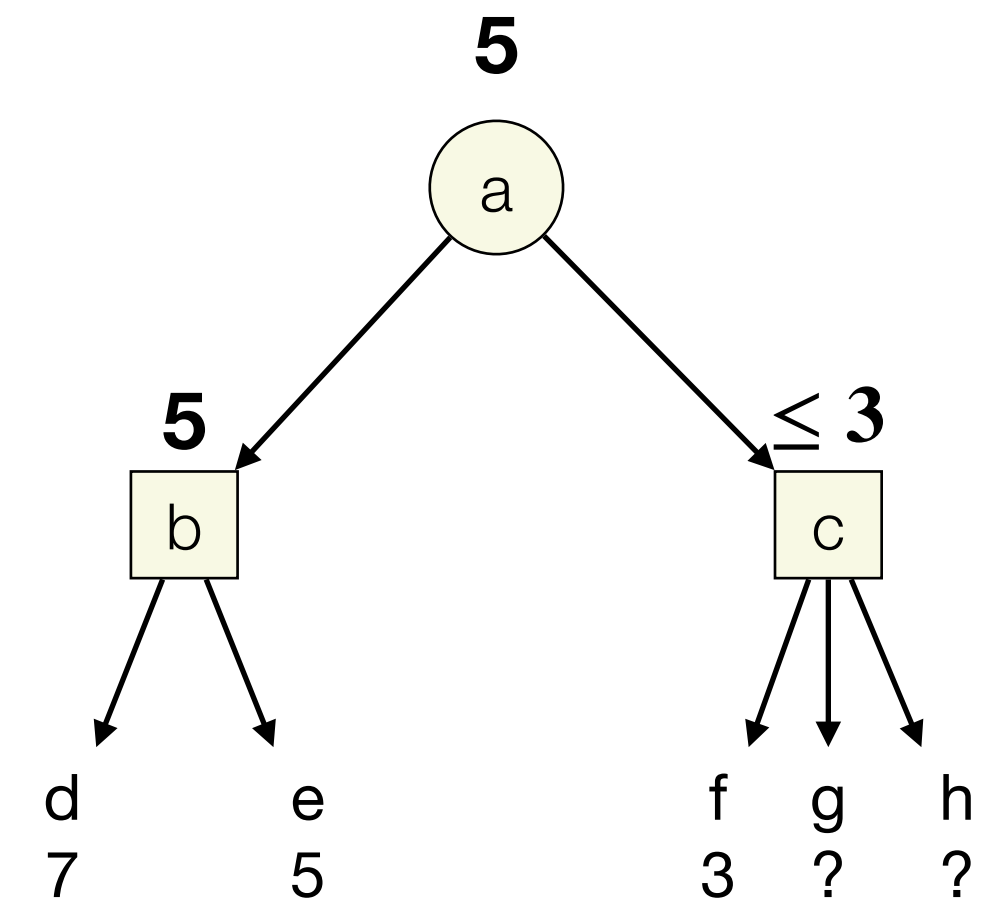
Pruning

- Plain minimax search visits every node in depth-first order
- But it's sometimes possible to avoid searching some subtrees based on information partway through the algorithm
- Once we have explored the **b**'s subtree and the first child of **c**, we know:
 - minimax value of **b** is 5
 - minimax value of **c** is **no larger than 3** (**why?**)
- But **a** is a max node!
 - **No matter what** minimax value of **c** turns out to be, P1 will choose **b** instead (**why?**)
 - So minimax value of **a** is 5
 - ...even though we haven't explored **g** or **h** yet
 - those nodes were **pruned**
- **Alpha-beta search** checks as it goes for these pruning opportunities



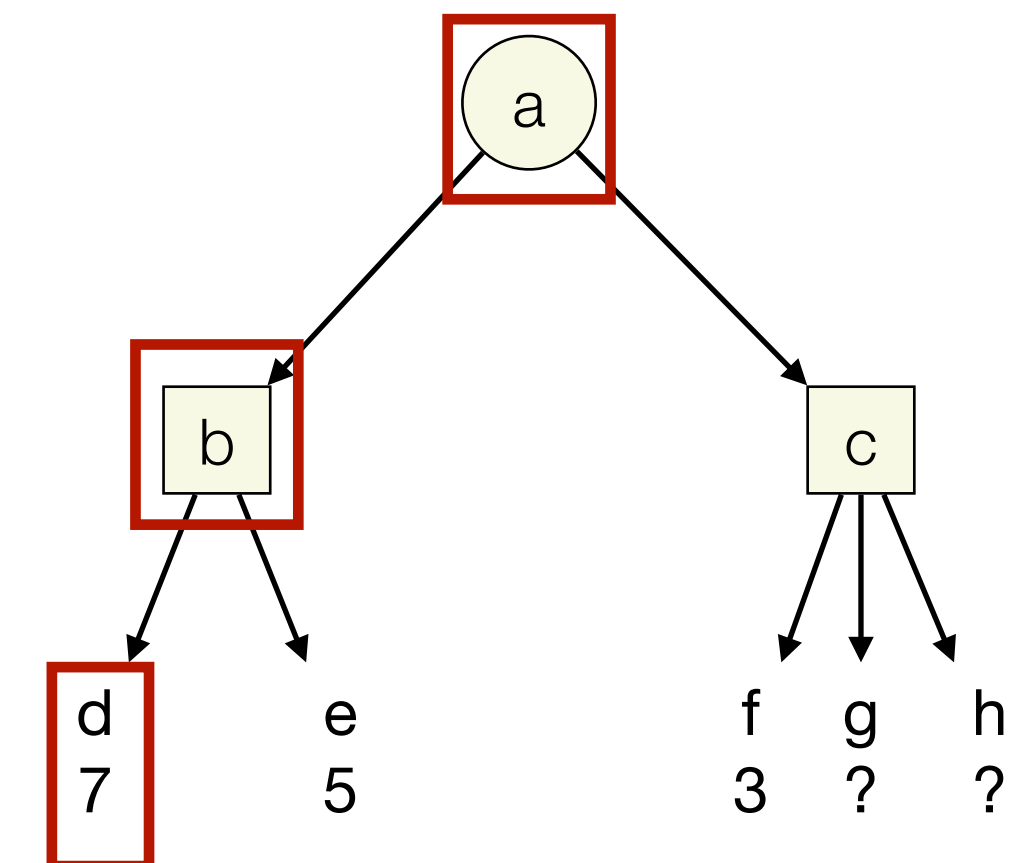
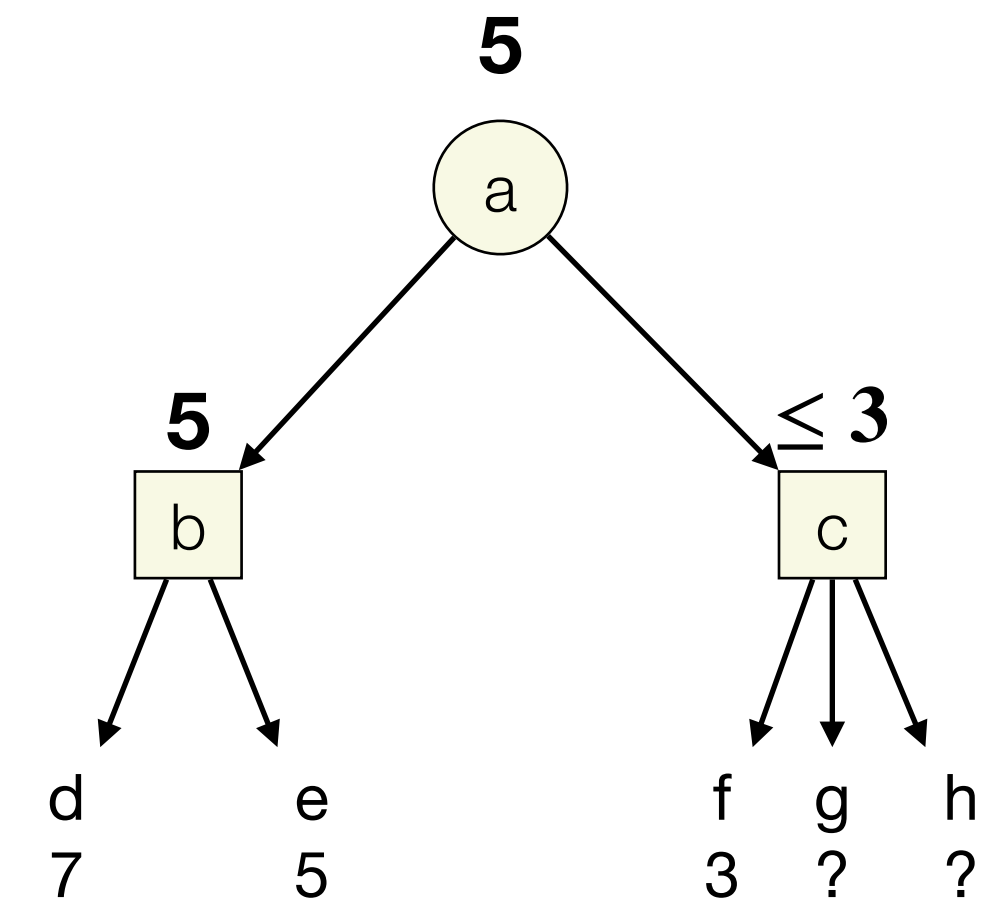
Pruning

- Plain minimax search visits every node in depth-first order
- But it's sometimes possible to avoid searching some subtrees based on information partway through the algorithm
- Once we have explored the **b**'s subtree and the first child of **c**, we know:
 - minimax value of **b** is 5
 - minimax value of **c** is **no larger than 3** (**why?**)
- But **a** is a max node!
 - **No matter what** minimax value of **c** turns out to be, P1 will choose **b** instead (**why?**)
 - So minimax value of **a** is 5
 - ...even though we haven't explored **g** or **h** yet
 - those nodes were **pruned**
- **Alpha-beta search** checks as it goes for these pruning opportunities



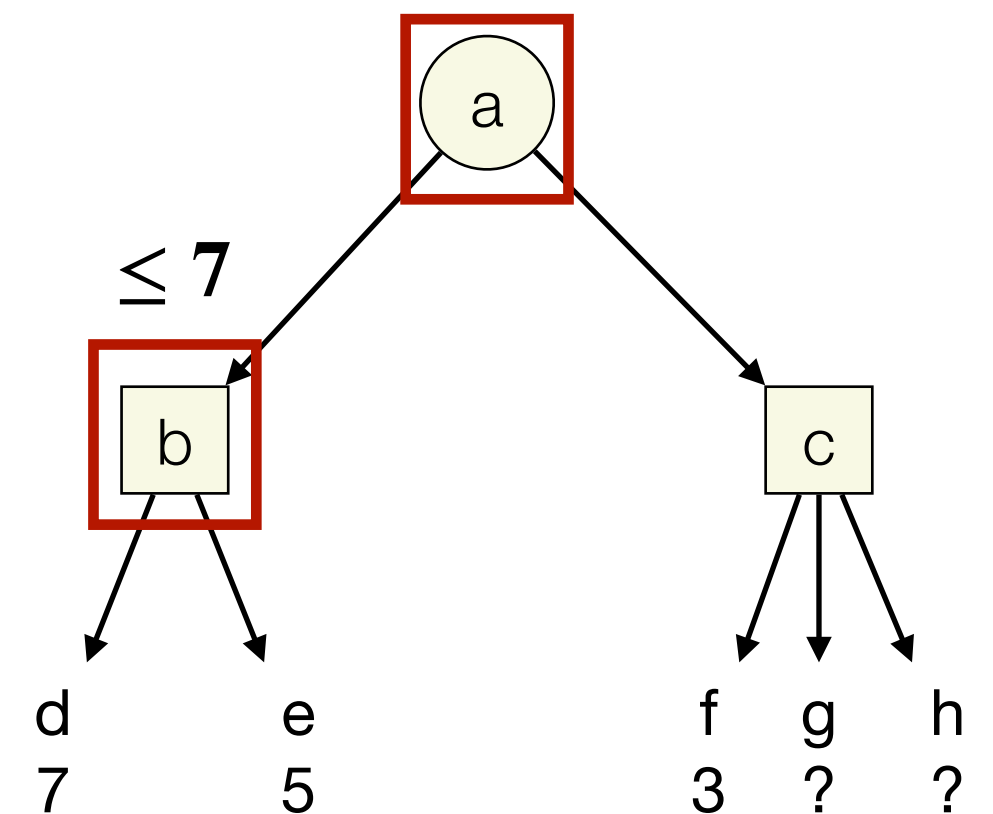
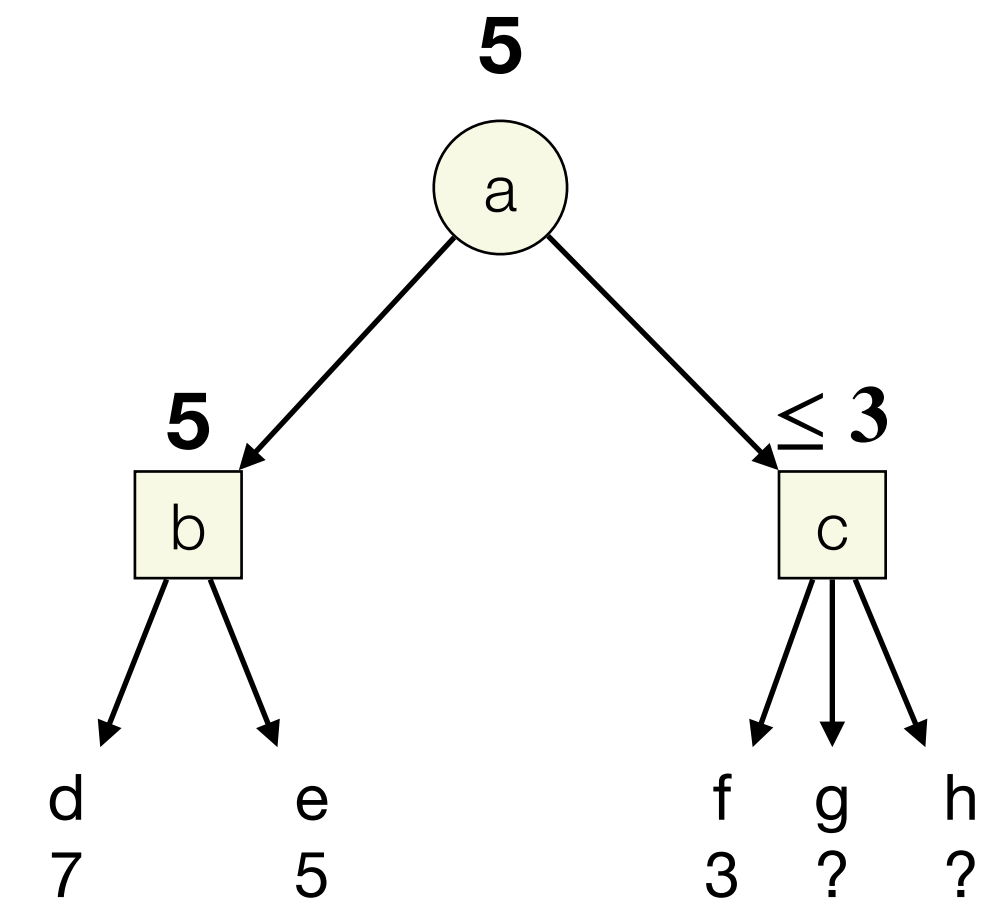
Pruning

- Plain minimax search visits every node in depth-first order
- But it's sometimes possible to avoid searching some subtrees based on information partway through the algorithm
- Once we have explored the **b**'s subtree and the first child of **c**, we know:
 - minimax value of **b** is 5
 - minimax value of **c** is **no larger than 3** (**why?**)
- But **a** is a max node!
 - **No matter what** minimax value of **c** turns out to be, P1 will choose **b** instead (**why?**)
 - So minimax value of **a** is 5
 - ...even though we haven't explored **g** or **h** yet
 - those nodes were **pruned**
- **Alpha-beta search** checks as it goes for these pruning opportunities



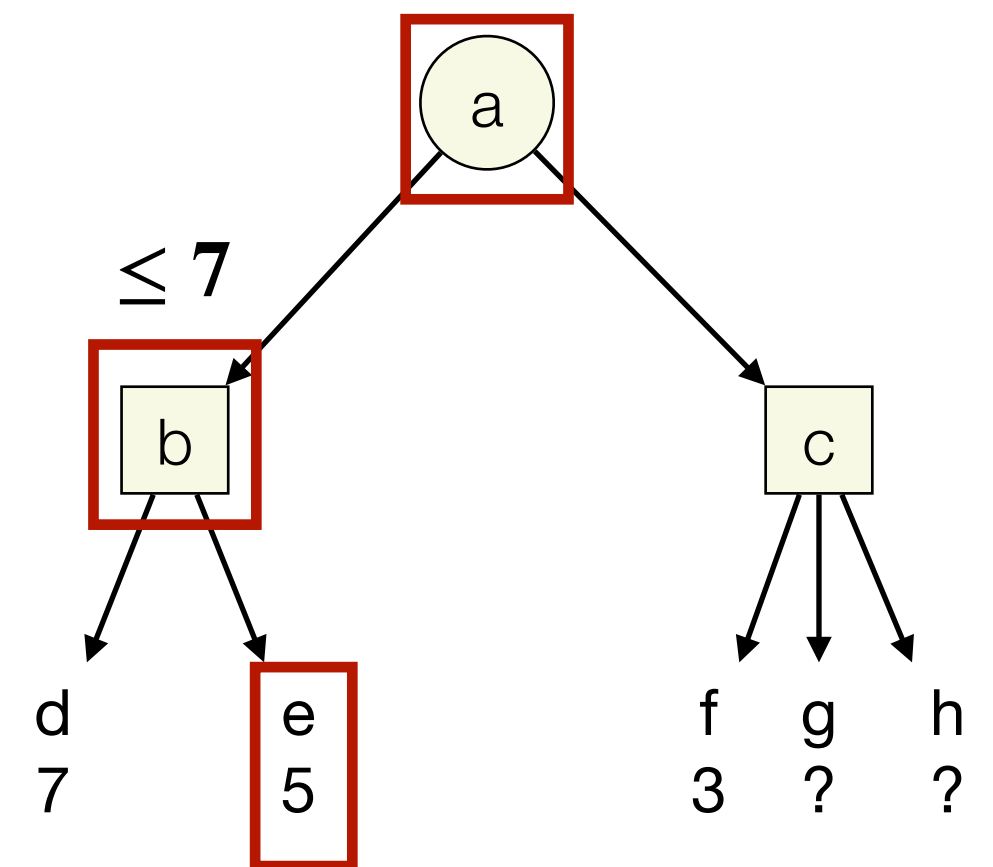
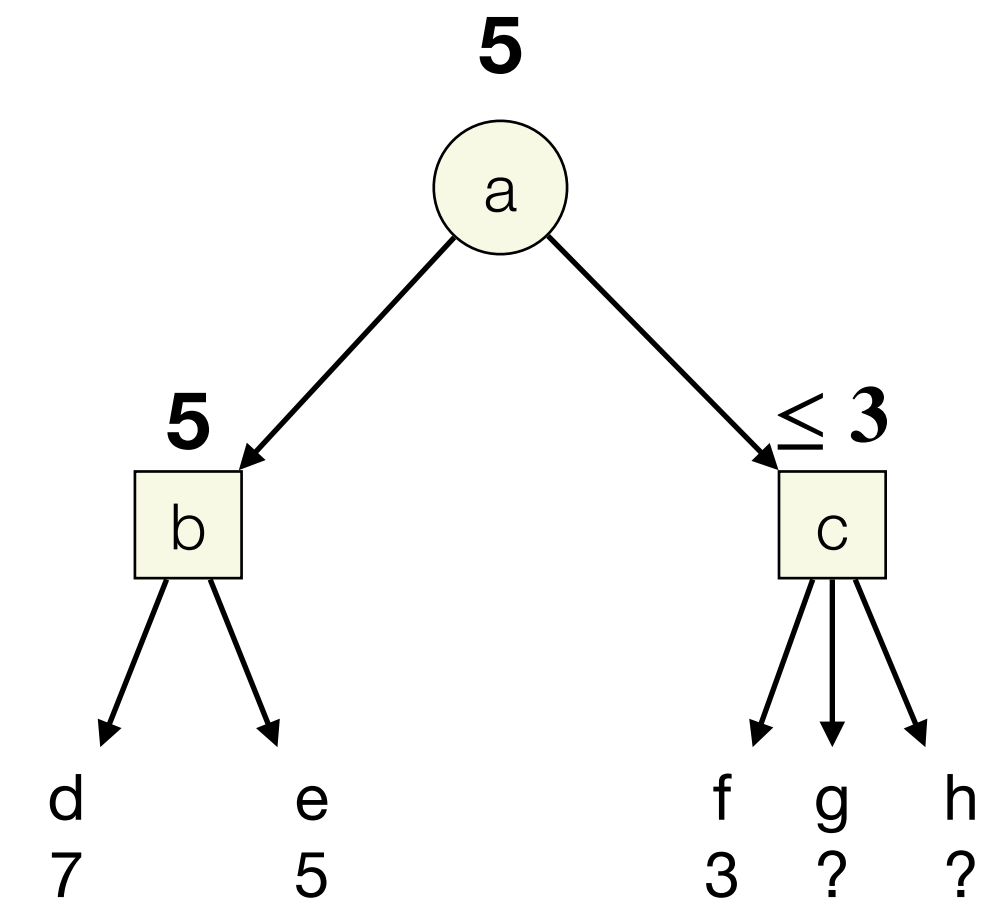
Pruning

- Plain minimax search visits every node in depth-first order
- But it's sometimes possible to avoid searching some subtrees based on information partway through the algorithm
- Once we have explored the **b**'s subtree and the first child of **c**, we know:
 - minimax value of **b** is 5
 - minimax value of **c** is **no larger than 3** (**why?**)
- But **a** is a max node!
 - **No matter what** minimax value of **c** turns out to be, P1 will choose **b** instead (**why?**)
 - So minimax value of **a** is 5
 - ...even though we haven't explored **g** or **h** yet
 - those nodes were **pruned**
- **Alpha-beta search** checks as it goes for these pruning opportunities



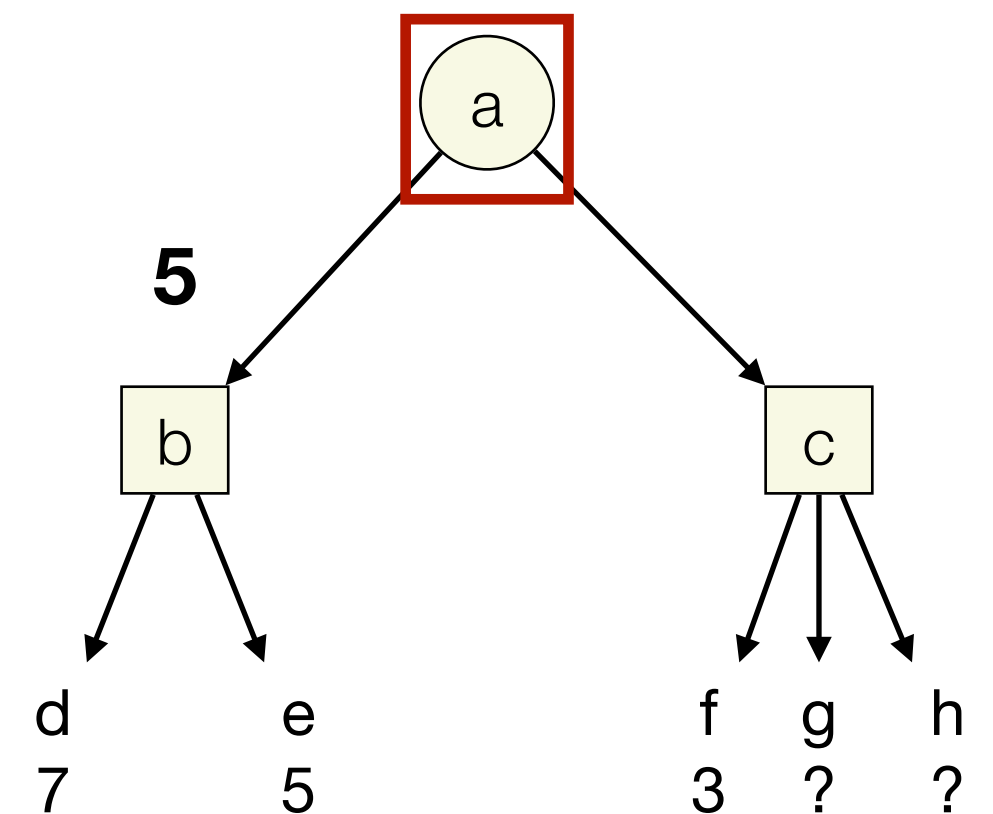
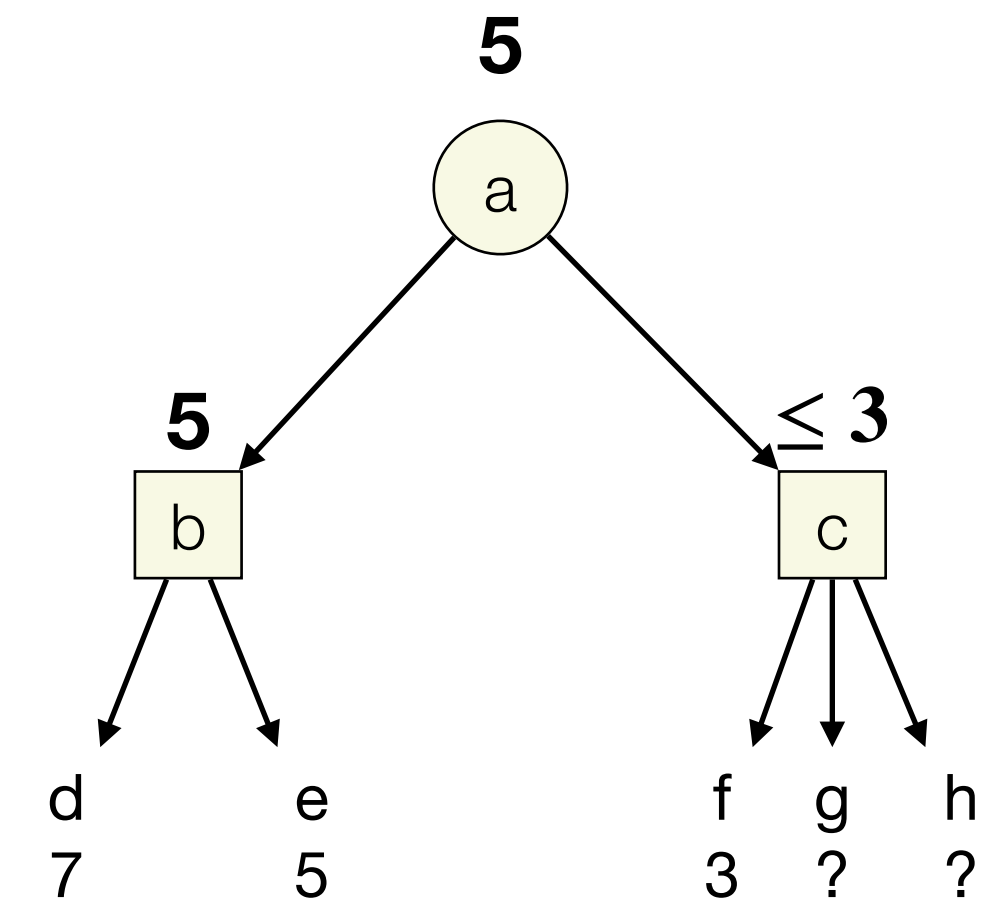
Pruning

- Plain minimax search visits every node in depth-first order
- But it's sometimes possible to avoid searching some subtrees based on information partway through the algorithm
- Once we have explored the **b**'s subtree and the first child of **c**, we know:
 - minimax value of **b** is 5
 - minimax value of **c** is **no larger than 3** (**why?**)
- But **a** is a max node!
 - **No matter what** minimax value of **c** turns out to be, P1 will choose **b** instead (**why?**)
 - So minimax value of **a** is 5
 - ...even though we haven't explored **g** or **h** yet
 - those nodes were **pruned**
- **Alpha-beta search** checks as it goes for these pruning opportunities



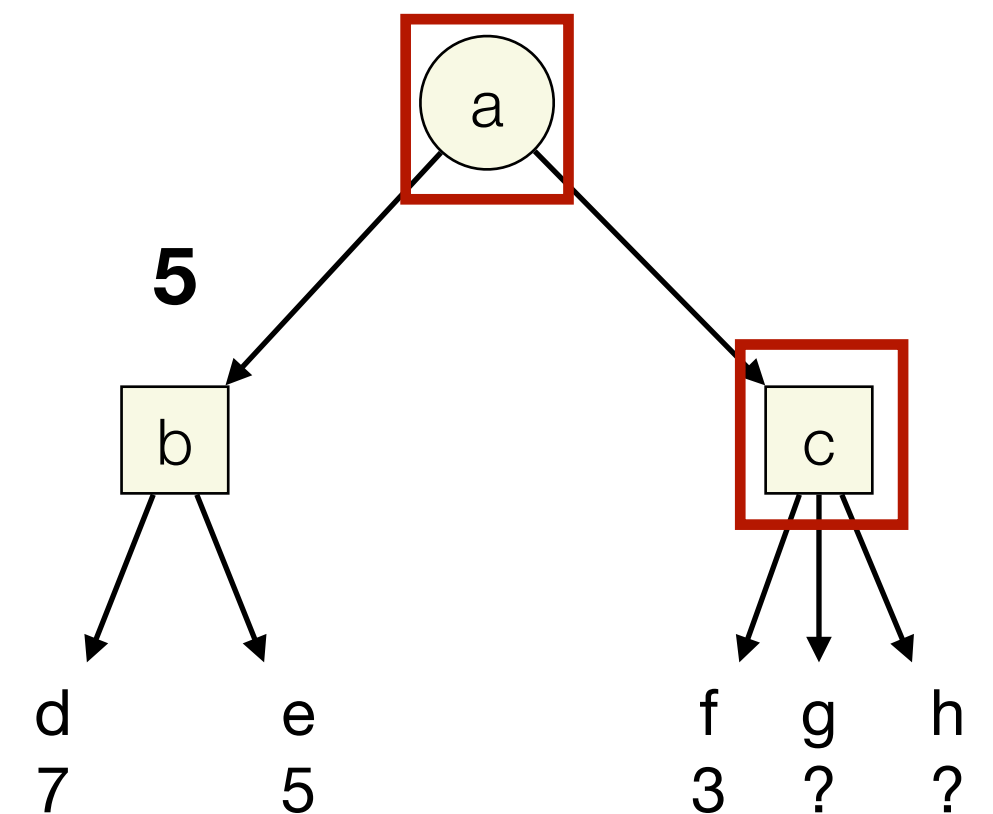
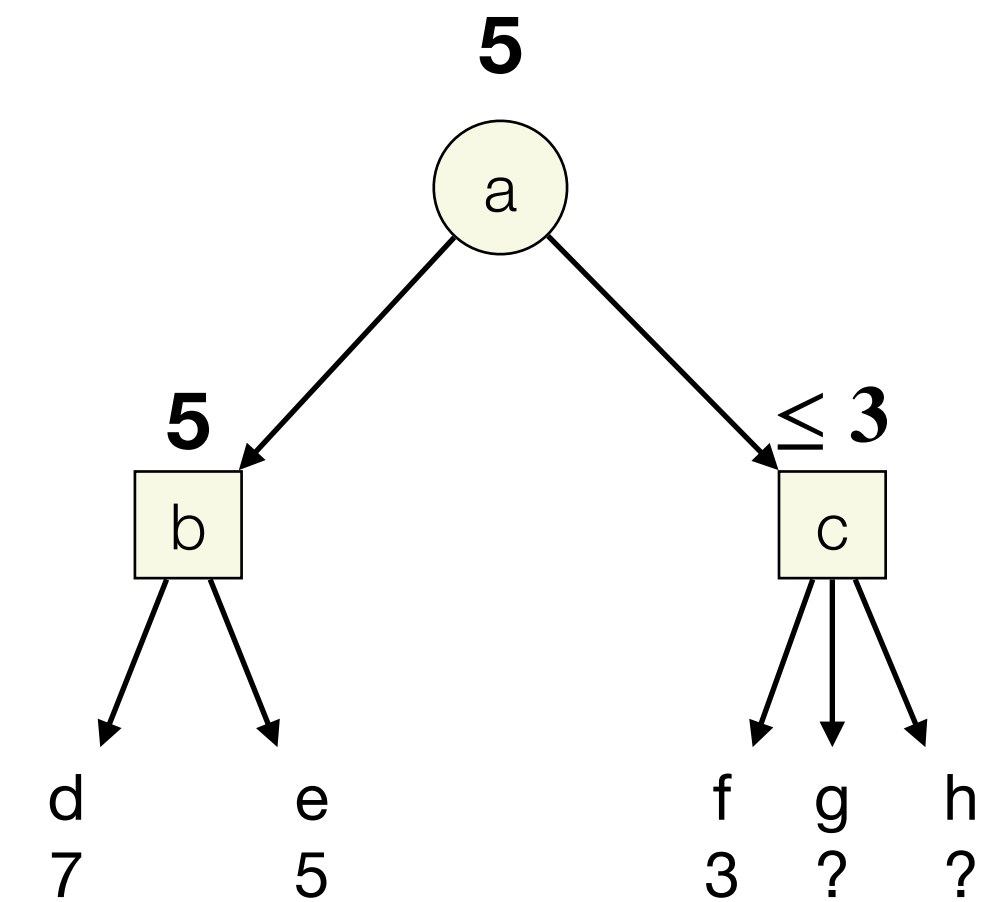
Pruning

- Plain minimax search visits every node in depth-first order
- But it's sometimes possible to avoid searching some subtrees based on information partway through the algorithm
- Once we have explored the **b**'s subtree and the first child of **c**, we know:
 - minimax value of **b** is 5
 - minimax value of **c** is **no larger than 3** (**why?**)
- But **a** is a max node!
 - **No matter what** minimax value of **c** turns out to be, P1 will choose **b** instead (**why?**)
 - So minimax value of **a** is 5
 - ...even though we haven't explored **g** or **h** yet
 - those nodes were **pruned**
- **Alpha-beta search** checks as it goes for these pruning opportunities



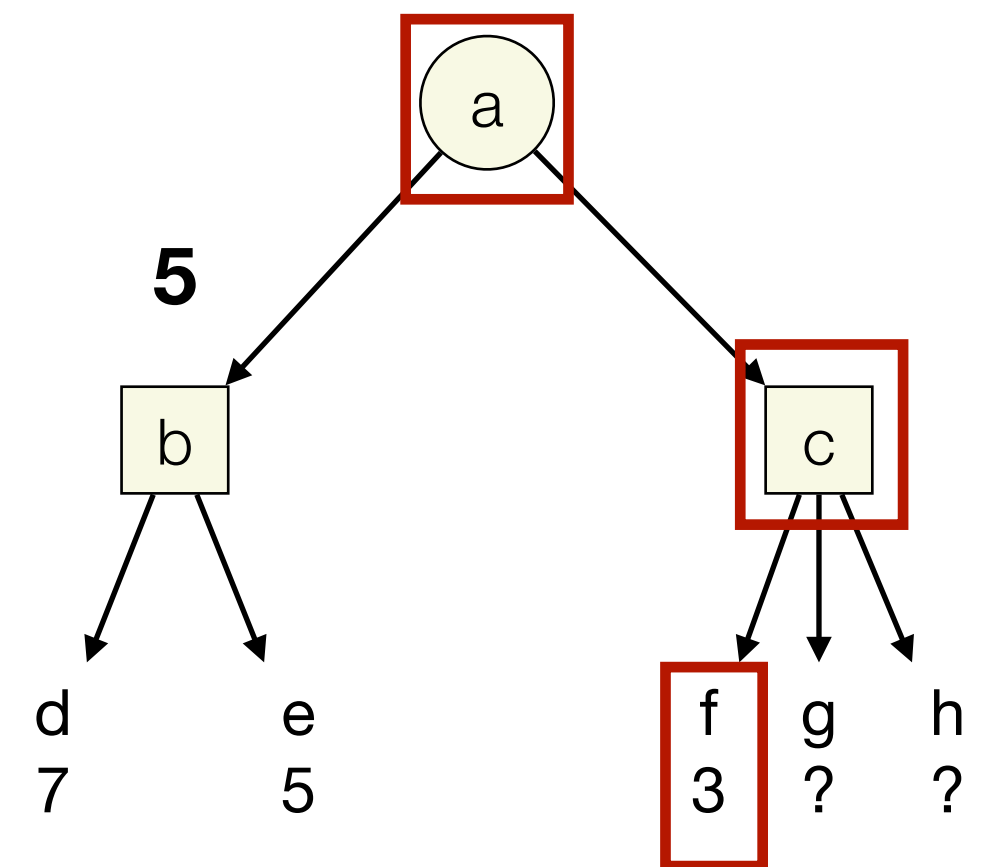
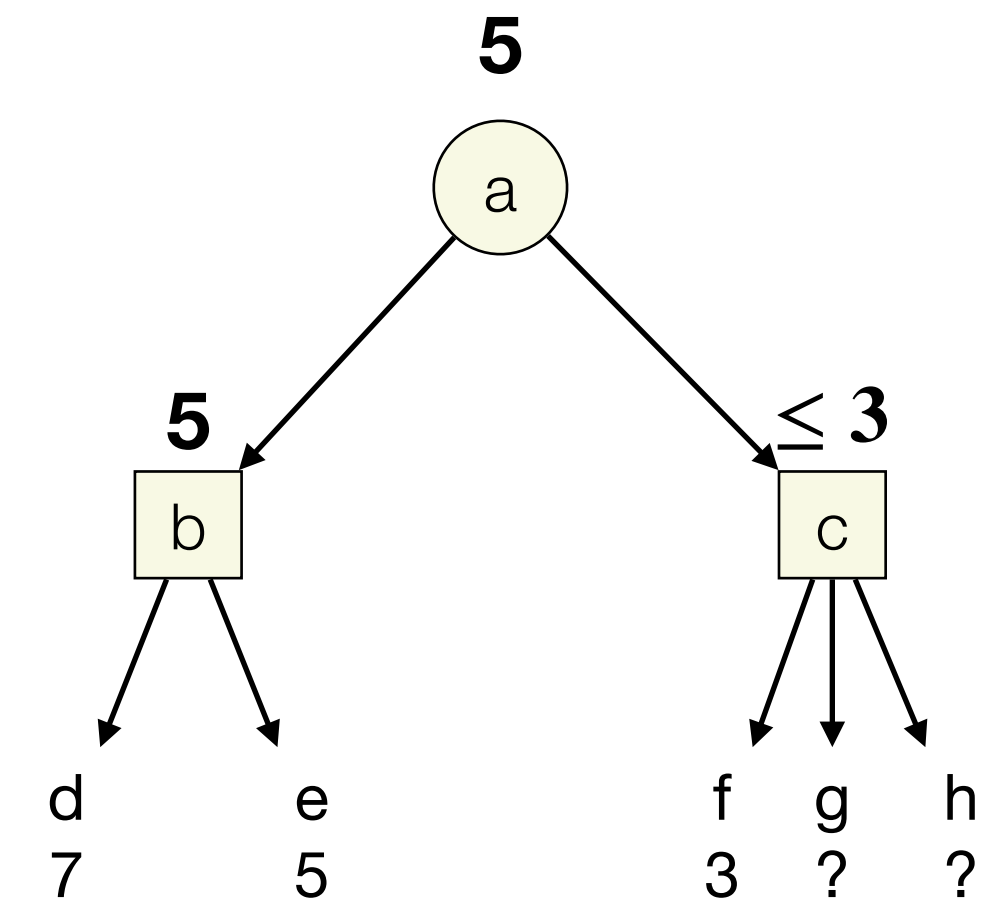
Pruning

- Plain minimax search visits every node in depth-first order
- But it's sometimes possible to avoid searching some subtrees based on information partway through the algorithm
- Once we have explored the **b**'s subtree and the first child of **c**, we know:
 - minimax value of **b** is 5
 - minimax value of **c** is **no larger than 3** (**why?**)
- But **a** is a max node!
 - **No matter what** minimax value of **c** turns out to be, P1 will choose **b** instead (**why?**)
 - So minimax value of **a** is 5
 - ...even though we haven't explored **g** or **h** yet
 - those nodes were **pruned**
- **Alpha-beta search** checks as it goes for these pruning opportunities



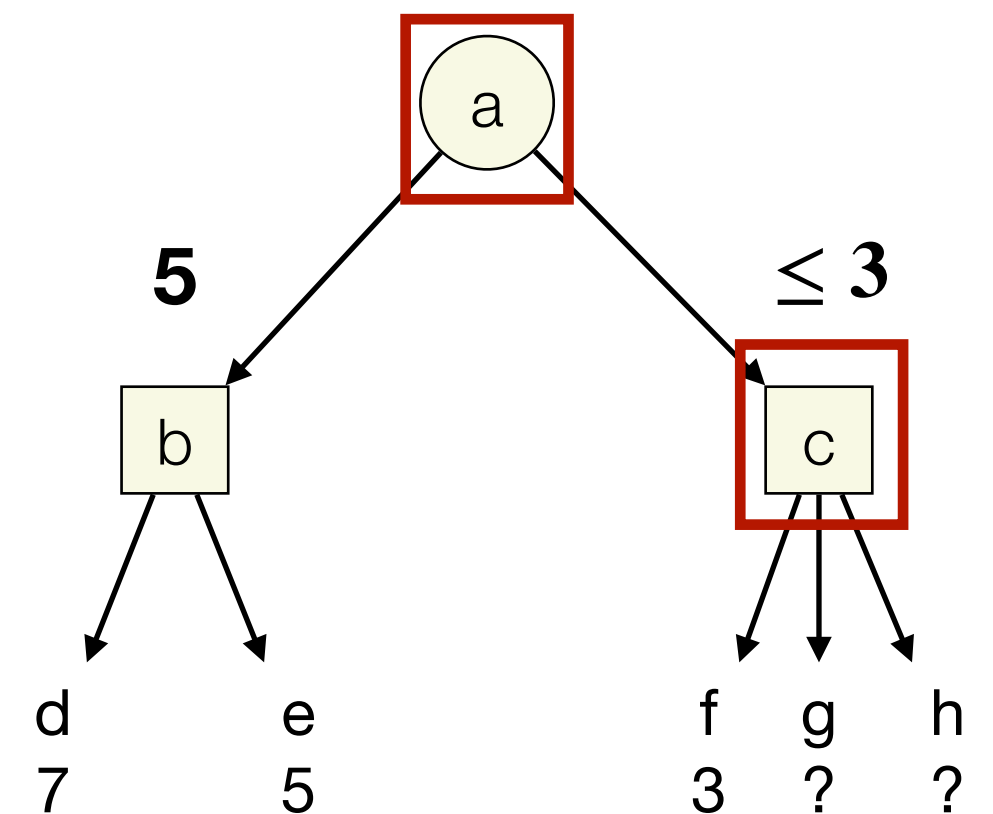
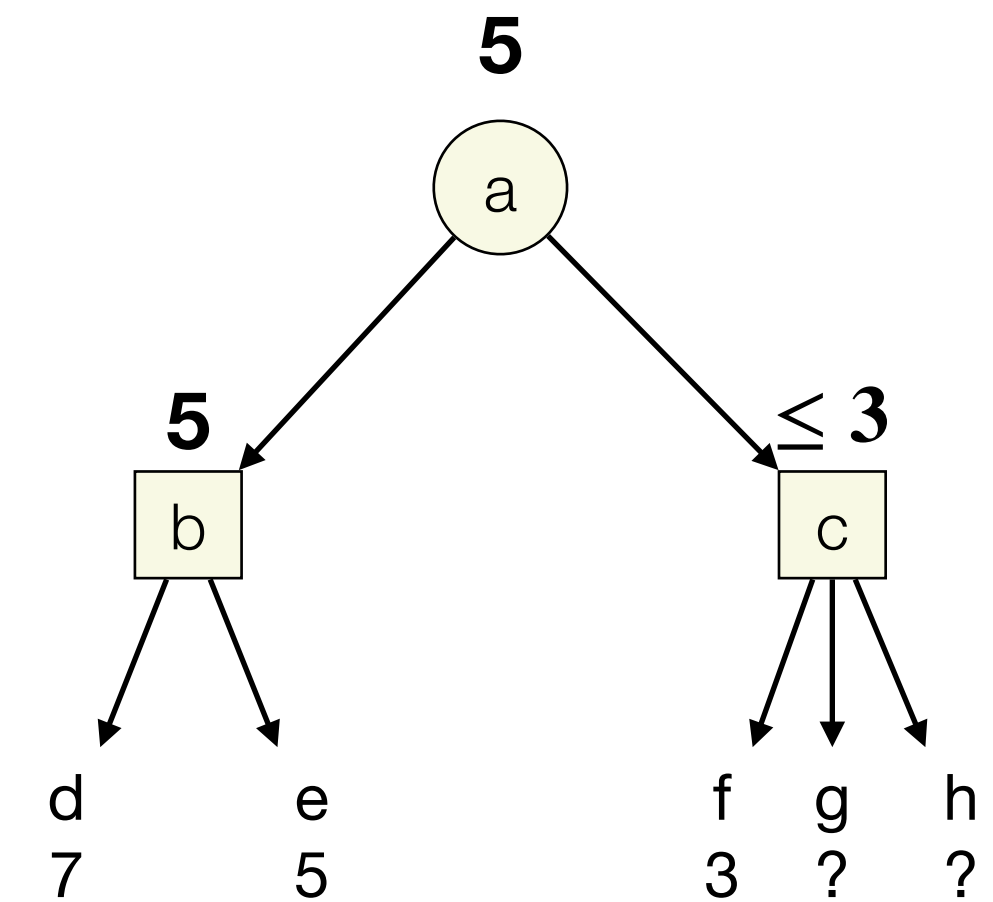
Pruning

- Plain minimax search visits every node in depth-first order
- But it's sometimes possible to avoid searching some subtrees based on information partway through the algorithm
- Once we have explored the **b**'s subtree and the first child of **c**, we know:
 - minimax value of **b** is 5
 - minimax value of **c** is **no larger than 3** (**why?**)
- But **a** is a max node!
 - **No matter what** minimax value of **c** turns out to be, P1 will choose **b** instead (**why?**)
 - So minimax value of **a** is 5
 - ...even though we haven't explored **g** or **h** yet
 - those nodes were **pruned**
- **Alpha-beta search** checks as it goes for these pruning opportunities



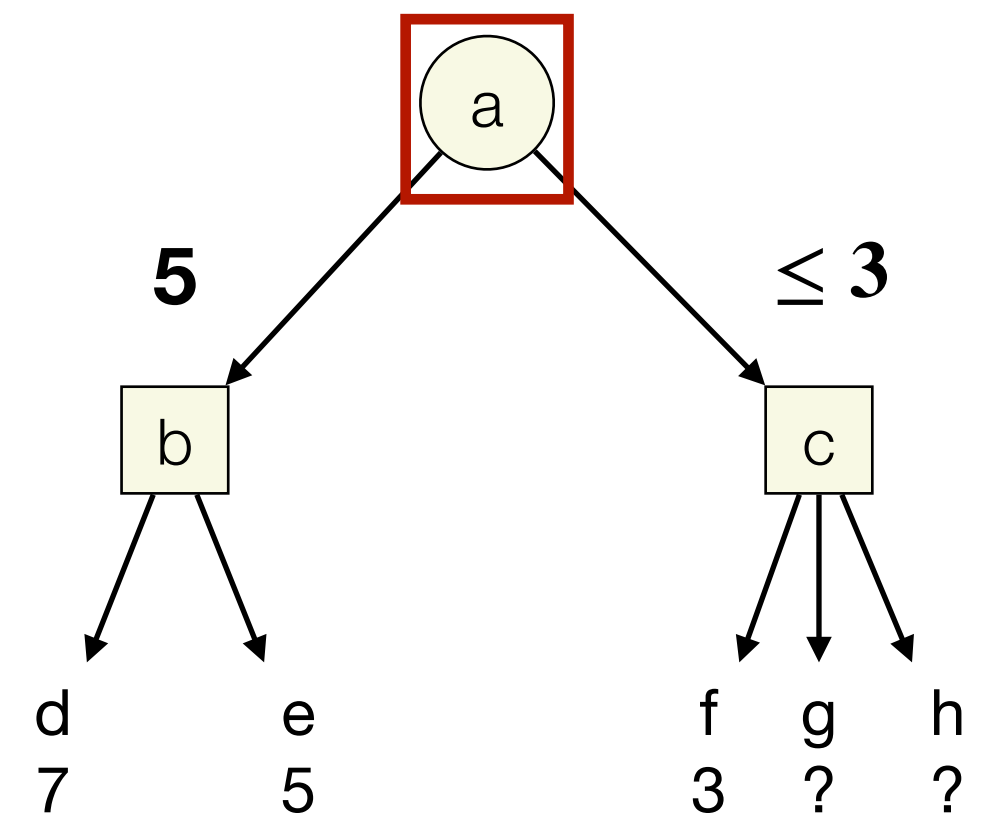
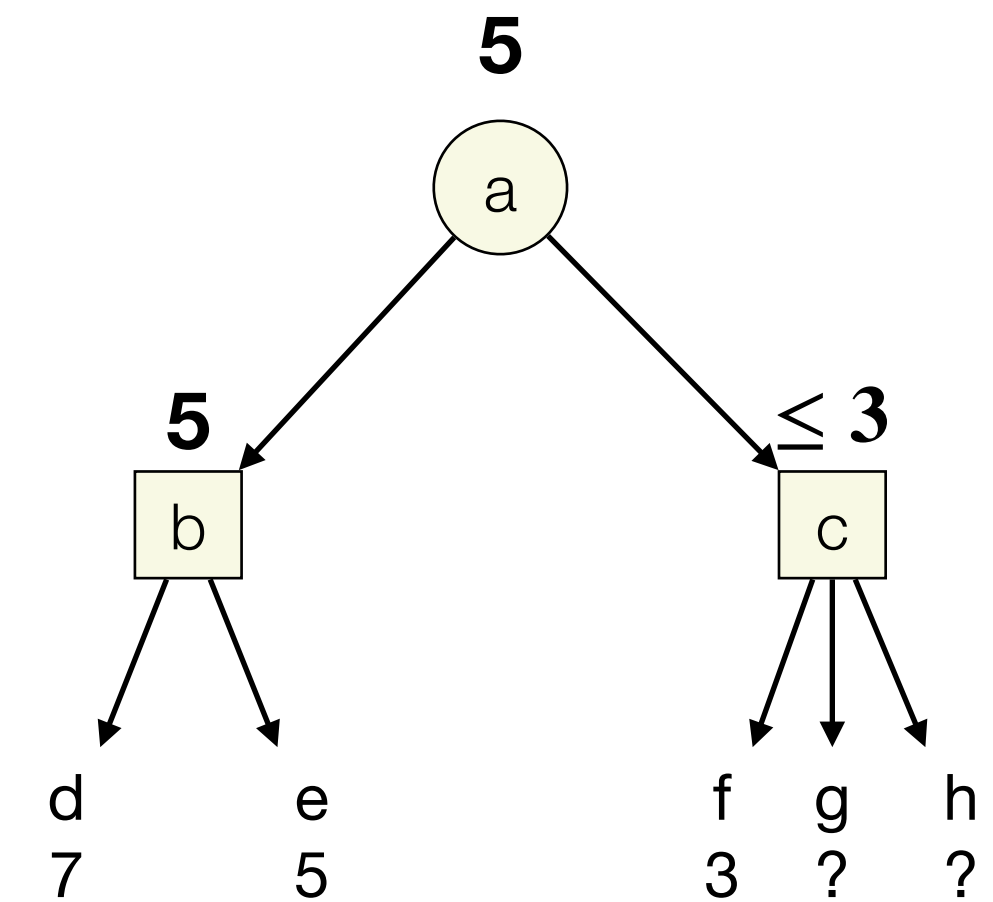
Pruning

- Plain minimax search visits every node in depth-first order
- But it's sometimes possible to avoid searching some subtrees based on information partway through the algorithm
- Once we have explored the **b**'s subtree and the first child of **c**, we know:
 - minimax value of **b** is 5
 - minimax value of **c** is **no larger than 3** (**why?**)
- But **a** is a max node!
 - **No matter what** minimax value of **c** turns out to be, P1 will choose **b** instead (**why?**)
 - So minimax value of **a** is 5
 - ...even though we haven't explored **g** or **h** yet
 - those nodes were **pruned**
- **Alpha-beta search** checks as it goes for these pruning opportunities



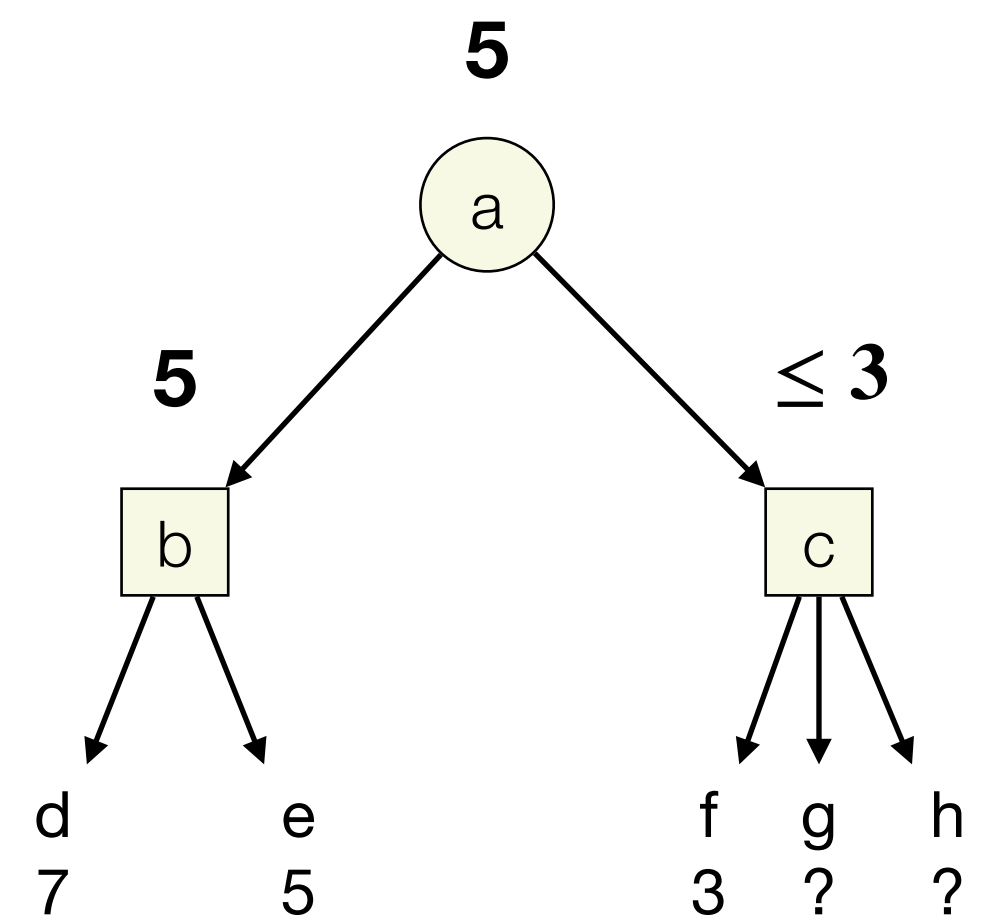
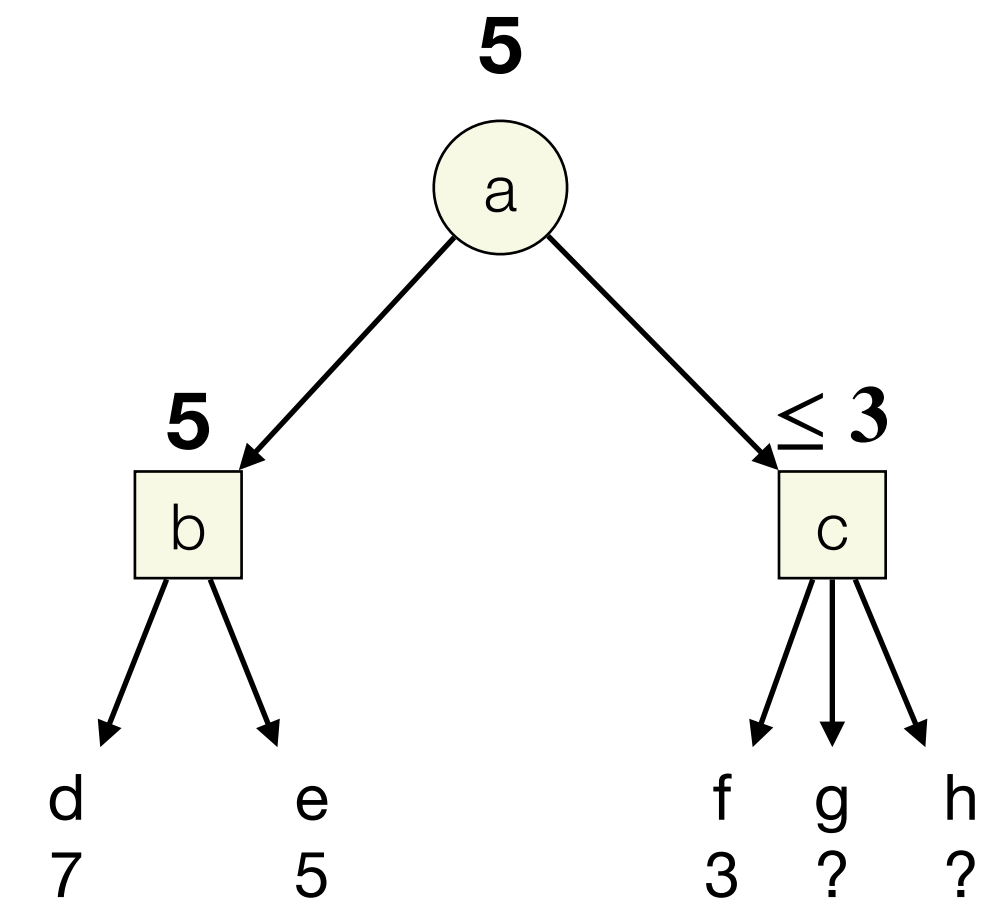
Pruning

- Plain minimax search visits every node in depth-first order
- But it's sometimes possible to avoid searching some subtrees based on information partway through the algorithm
- Once we have explored the **b**'s subtree and the first child of **c**, we know:
 - minimax value of **b** is 5
 - minimax value of **c** is **no larger than 3** (**why?**)
- But **a** is a max node!
 - **No matter what** minimax value of **c** turns out to be, P1 will choose **b** instead (**why?**)
 - So minimax value of **a** is 5
 - ...even though we haven't explored **g** or **h** yet
 - those nodes were **pruned**
- **Alpha-beta search** checks as it goes for these pruning opportunities



Pruning

- Plain minimax search visits every node in depth-first order
- But it's sometimes possible to avoid searching some subtrees based on information partway through the algorithm
- Once we have explored the **b**'s subtree and the first child of **c**, we know:
 - minimax value of **b** is 5
 - minimax value of **c** is **no larger than 3** (**why?**)
- But **a** is a max node!
 - **No matter what** minimax value of **c** turns out to be, P1 will choose **b** instead (**why?**)
 - So minimax value of **a** is 5
 - ...even though we haven't explored **g** or **h** yet
 - those nodes were **pruned**
- **Alpha-beta search** checks as it goes for these pruning opportunities



Alpha-Beta Search, informally

- **Main idea:** Learn **enough** about the current node guarantee that minimax strategies **never play here**

Alpha-Beta Search, informally

- **Main idea:** Learn **enough** about the current node guarantee that minimax strategies **never play here**
- What can guarantee that?
 1. Current node is a **max** node, *and* minimax value of current node is **at least α** , *and* a **min** node **earlier in the tree** has a choice that guarantees **less than α**

Alpha-Beta Search, informally

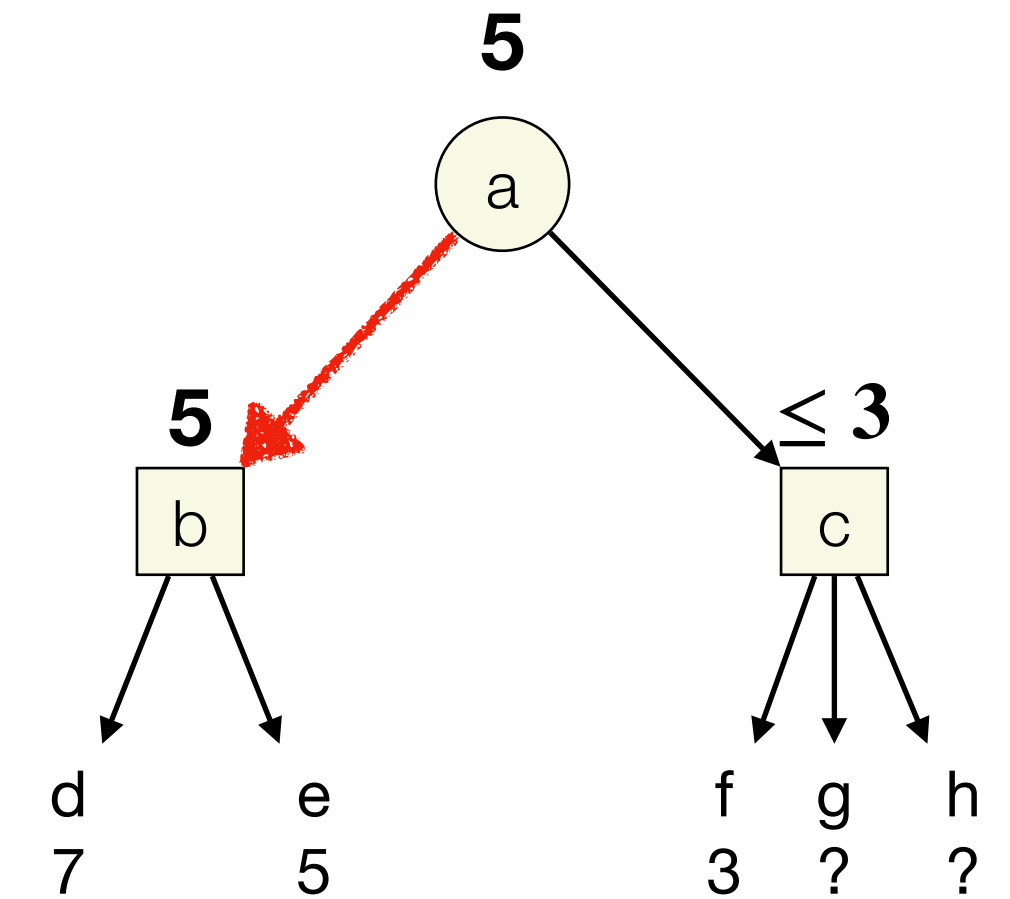
- **Main idea:** Learn **enough** about the current node guarantee that minimax strategies **never play here**
- What can guarantee that?
 1. Current node is a **max** node, *and*
minimax value of current node is **at least α** , *and*
a **min** node **earlier in the tree** has a choice that guarantees **less than α**
 2. Current node is a min node, *and*
minimax value of current node is **less than β** , *and*
a **max** node **earlier in the tree** has a choice that guarantees **more than β**

Alpha-Beta Search, informally

- **Main idea:** Learn **enough** about the current node guarantee that minimax strategies **never play here**
- What can guarantee that?
 1. Current node is a **max** node, *and*
minimax value of current node is **at least α** , *and*
a **min** node **earlier in the tree** has a choice that guarantees **less than α**
 2. Current node is a min node, *and*
minimax value of current node is **less than β** , *and*
a **max** node **earlier in the tree** has a choice that guarantees **more than β**
- **Question:** Why is earlier in the tree important?

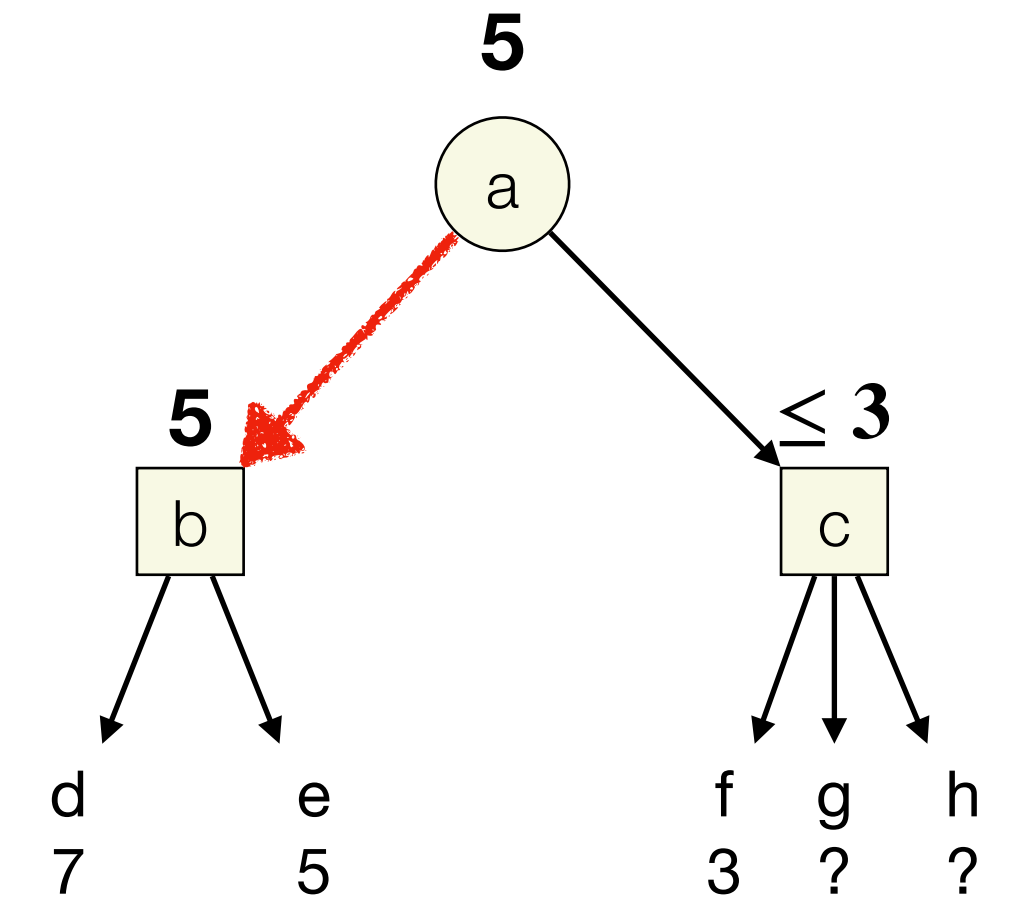
Alpha-Beta Search, informally

- **Main idea:** Learn **enough** about the current node guarantee that minimax strategies **never play here**
- What can guarantee that?
 1. Current node is a **max** node, *and* minimax value of current node is **at least α** , *and* a **min** node **earlier in the tree** has a choice that guarantees **less than α**
 2. Current node is a min node, *and* minimax value of current node is **less than β** , *and* a **max** node **earlier in the tree** has a choice that guarantees **more than β**
- **Question:** Why is earlier in the tree important?



Alpha-Beta Search, informally

- **Main idea:** Learn **enough** about the current node guarantee that minimax strategies **never play here**
- What can guarantee that?
 1. Current node is a **max** node, *and* minimax value of current node is **at least α** , *and* a **min** node **earlier in the tree** has a choice that guarantees **less than α**
 2. Current node is a min node, *and* minimax value of current node is **less than β** , *and* a **max** node **earlier in the tree** has a choice that guarantees **more than β**
- **Question:** Why is earlier in the tree important?
- At each stage, alpha-beta search tracks:
 - α : **Highest** value available on **path from current node to root** (including both current node and root)
 - β : **Lowest** value available on **path from current node to root** (including both current node and root)



Alpha-Beta Search Pseudocode

```
def alphabeta(s, alpha, beta):
```

Alpha-Beta Search Pseudocode

```
def alphabeta(s, alpha, beta):  
    if terminal(s):  
        return score(s)
```

Alpha-Beta Search Pseudocode

```
def alphabeta(s, alpha, beta):  
    if terminal(s):  
        return score(s)  
    if player(s) = 1: # MAX player  
        val = -inf  
        for c in children(s):  
            ab = alphabeta(c, alpha, beta)  
            alpha = max(alpha, ab)  
            val = max(val, ab)  
            if alpha >= beta:  
                return val # prune remaining children
```

Alpha-Beta Search Pseudocode

```
def alphabeta(s, alpha, beta):  
    if terminal(s):  
        return score(s)  
    if player(s) = 1: # MAX player  
        val = -inf  
        for c in children(s):  
            ab = alphabeta(c, alpha, beta)  
            alpha = max(alpha, ab)  
            val = max(val, ab)  
            if alpha >= beta:  
                return val # prune remaining children  
    if player(s) = 2:  
        val = inf  
        for c in children(s):  
            ab = alphabeta(c, alpha, beta)  
            beta = min(beta, ab)  
            val = min(val, ab)  
            if alpha >= beta:  
                return val # prune remaining children
```

Alpha-Beta Search Pseudocode

```
def alphabeta(s, alpha, beta):
    if terminal(s):
        return score(s)
    if player(s) = 1: # MAX player
        val = -inf
        for c in children(s):
            ab = alphabeta(c, alpha, beta)
            alpha = max(alpha, ab)
            val = max(val, ab)
            if alpha >= beta:
                return val # prune remaining children
    if player(s) = 2:
        val = inf
        for c in children(s):
            ab = alphabeta(c, alpha, beta)
            beta = min(beta, ab)
            val = min(val, ab)
            if alpha >= beta:
                return val # prune remaining children
    return val
```

Alpha-Beta Search Pseudocode

```
def alphabeta(s, alpha, beta):
    if terminal(s):
        return score(s)
    if player(s) = 1: # MAX player
        val = -inf
        for c in children(s):
            ab = alphabeta(c, alpha, beta)
            alpha = max(alpha, ab)
            val = max(val, ab)
            if alpha >= beta:
                return val # prune remaining children
    if player(s) = 2:
        val = inf
        for c in children(s):
            ab = alphabeta(c, alpha, beta)
            beta = min(beta, ab)
            val = min(val, ab)
            if alpha >= beta:
                return val # prune remaining children
    return val
```

Questions:

1. Why do we initialize `val` with `inf` or `-inf`?

Alpha-Beta Search Pseudocode

```
def alphabeta(s, alpha, beta):
    if terminal(s):
        return score(s)
    if player(s) = 1: # MAX player
        val = -inf
        for c in children(s):
            ab = alphabeta(c, alpha, beta)
            alpha = max(alpha, ab)
            val = max(val, ab)
            if alpha >= beta:
                return val # prune remaining children
    if player(s) = 2:
        val = inf
        for c in children(s):
            ab = alphabeta(c, alpha, beta)
            beta = min(beta, ab)
            val = min(val, ab)
            if alpha >= beta:
                return val # prune remaining children
    return val
```

Questions:

1. Why do we initialize `val` with `inf` or `-inf`?
2. With what arguments should we call `alphabeta` on the root node? (**why?**)

Alpha-Beta Search Pseudocode

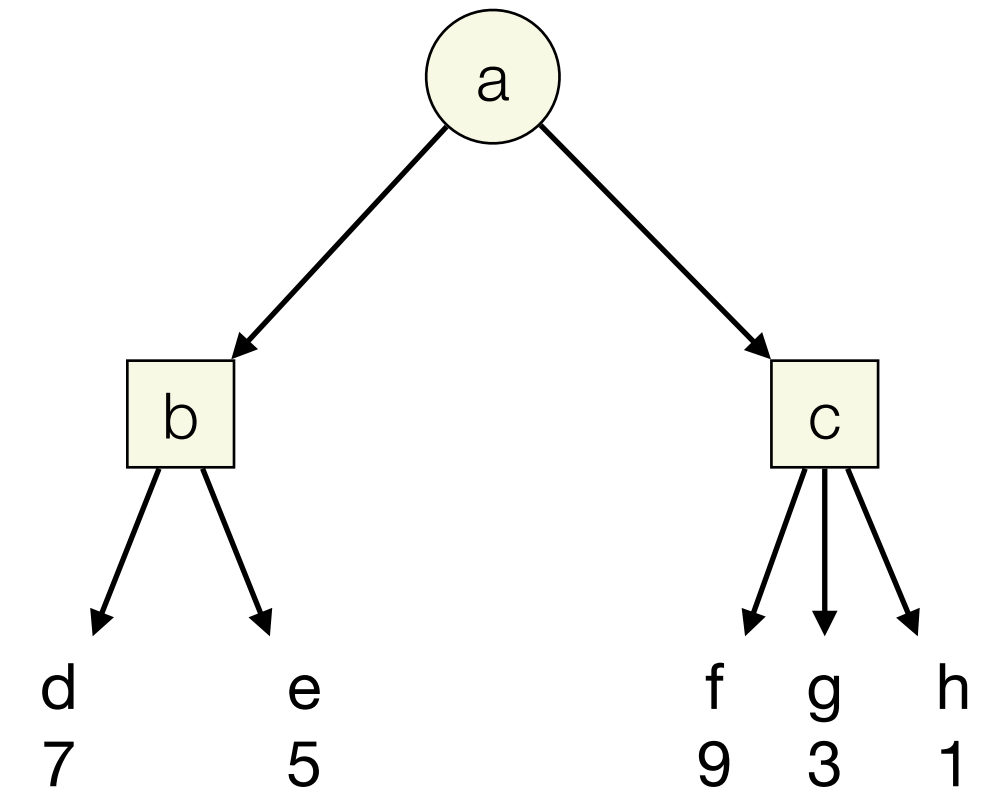
```
def alphabeta(s, alpha, beta):
    if terminal(s):
        return score(s)
    if player(s) = 1: # MAX player
        val = -inf
        for c in children(s):
            ab = alphabeta(c, alpha, beta)
            alpha = max(alpha, ab)
            val = max(val, ab)
            if alpha >= beta:
                return val # prune remaining children
    if player(s) = 2:
        val = inf
        for c in children(s):
            ab = alphabeta(c, alpha, beta)
            beta = min(beta, ab)
            val = min(val, ab)
            if alpha >= beta:
                return val # prune remaining children
    return val
```

Questions:

1. Why do we initialize `val` with `inf` or `-inf`?
2. With what arguments should we call `alphabeta` on the root node? (**why?**)
3. Why do we not return updated `alpha` / `beta` values as well as `val`?

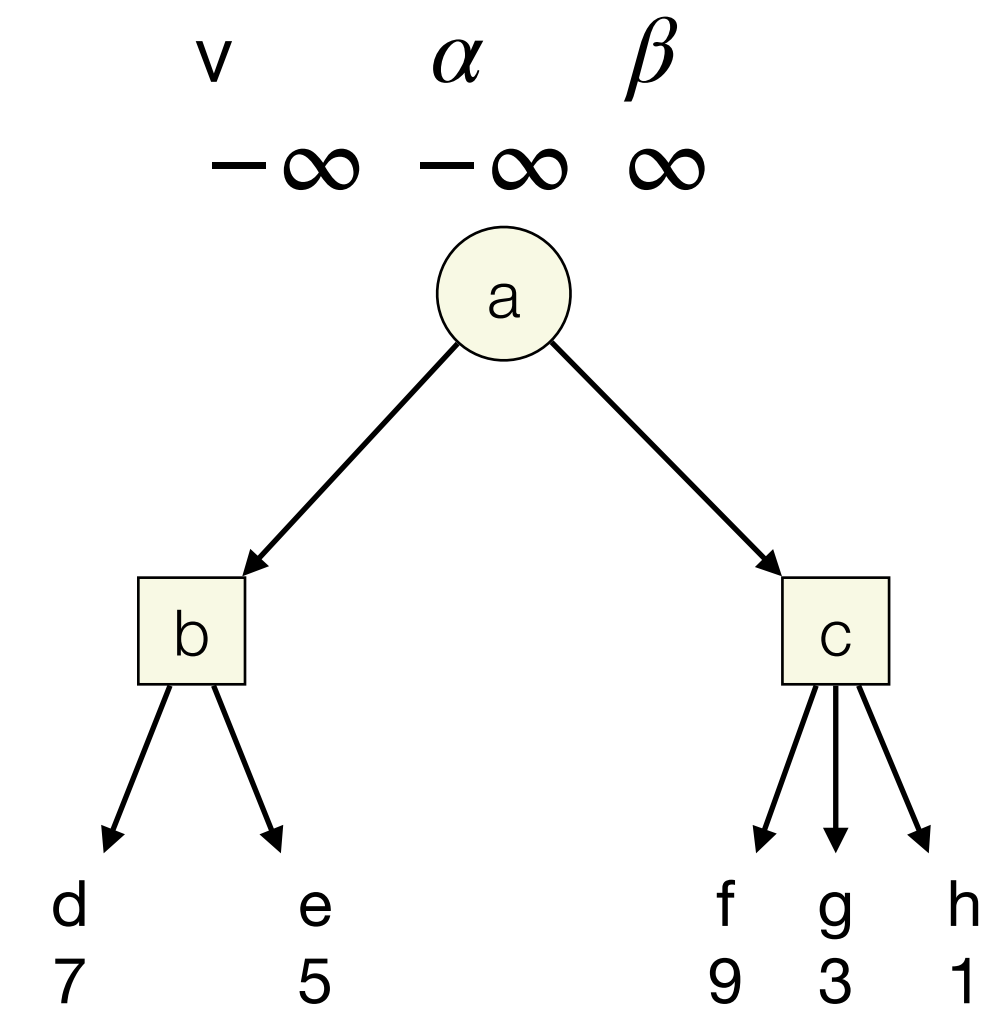
Alpha-Beta Search Example

```
def alphabeta(s, alpha, beta):
    if terminal(s):
        return score(s)
    if player(s) == 1: # MAX player
        val = -inf
        for c in children(s):
            ab = alphabeta(c, alpha, beta)
            alpha = max(alpha, ab)
            val = max(val, ab)
            if alpha >= beta:
                return val # prune remaining children
    if player(s) == 2:
        val = inf
        for c in children(s):
            ab = alphabeta(c, alpha, beta)
            beta = min(beta, ab)
            val = min(val, ab)
            if alpha >= beta:
                return val # prune remaining children
    return val
```



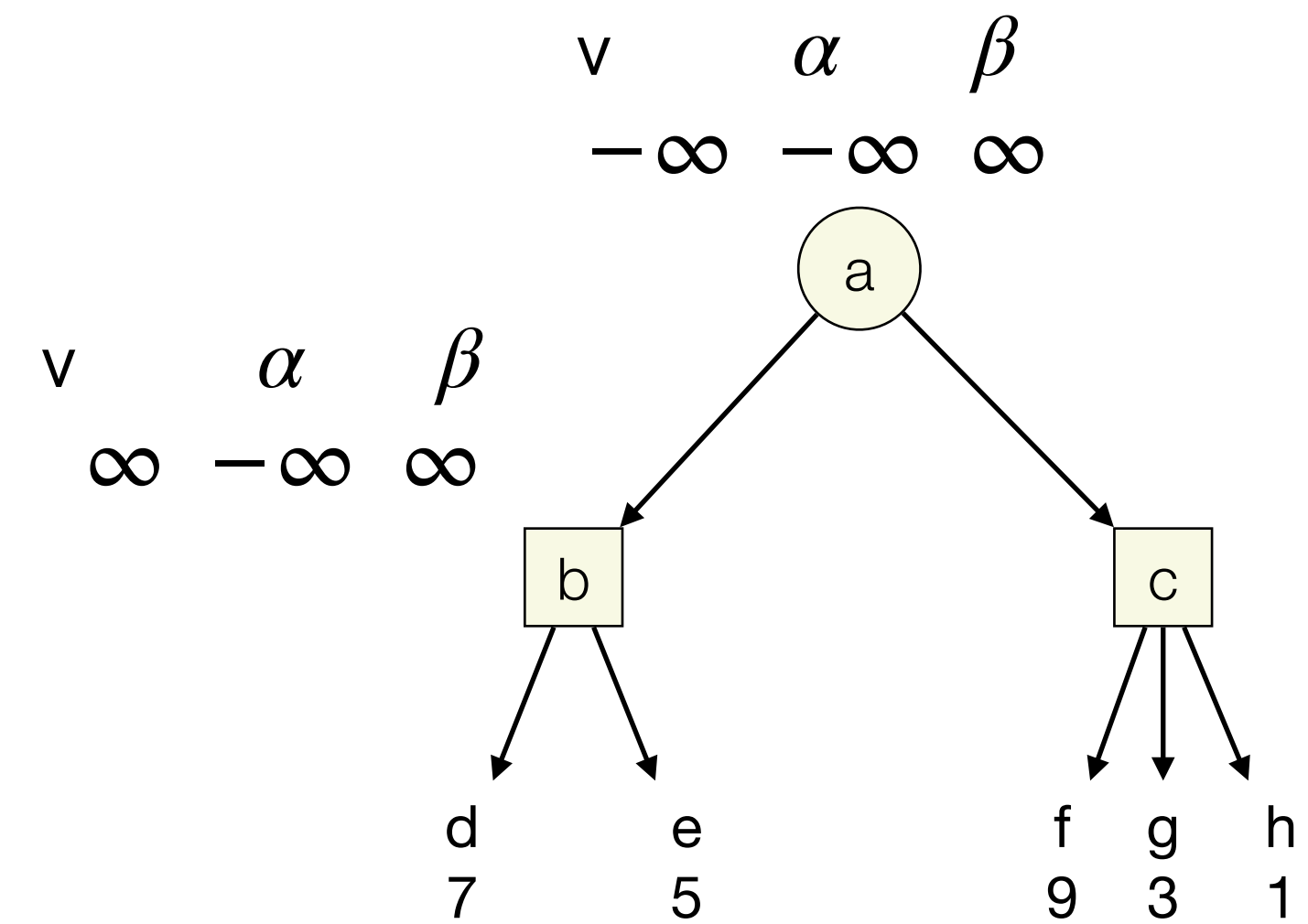
Alpha-Beta Search Example

```
def alphabeta(s, alpha, beta):
    if terminal(s):
        return score(s)
    if player(s) == 1: # MAX player
        val = -inf
        for c in children(s):
            ab = alphabeta(c, alpha, beta)
            alpha = max(alpha, ab)
            val = max(val, ab)
            if alpha >= beta:
                return val # prune remaining children
    if player(s) == 2:
        val = inf
        for c in children(s):
            ab = alphabeta(c, alpha, beta)
            beta = min(beta, ab)
            val = min(val, ab)
            if alpha >= beta:
                return val # prune remaining children
    return val
```



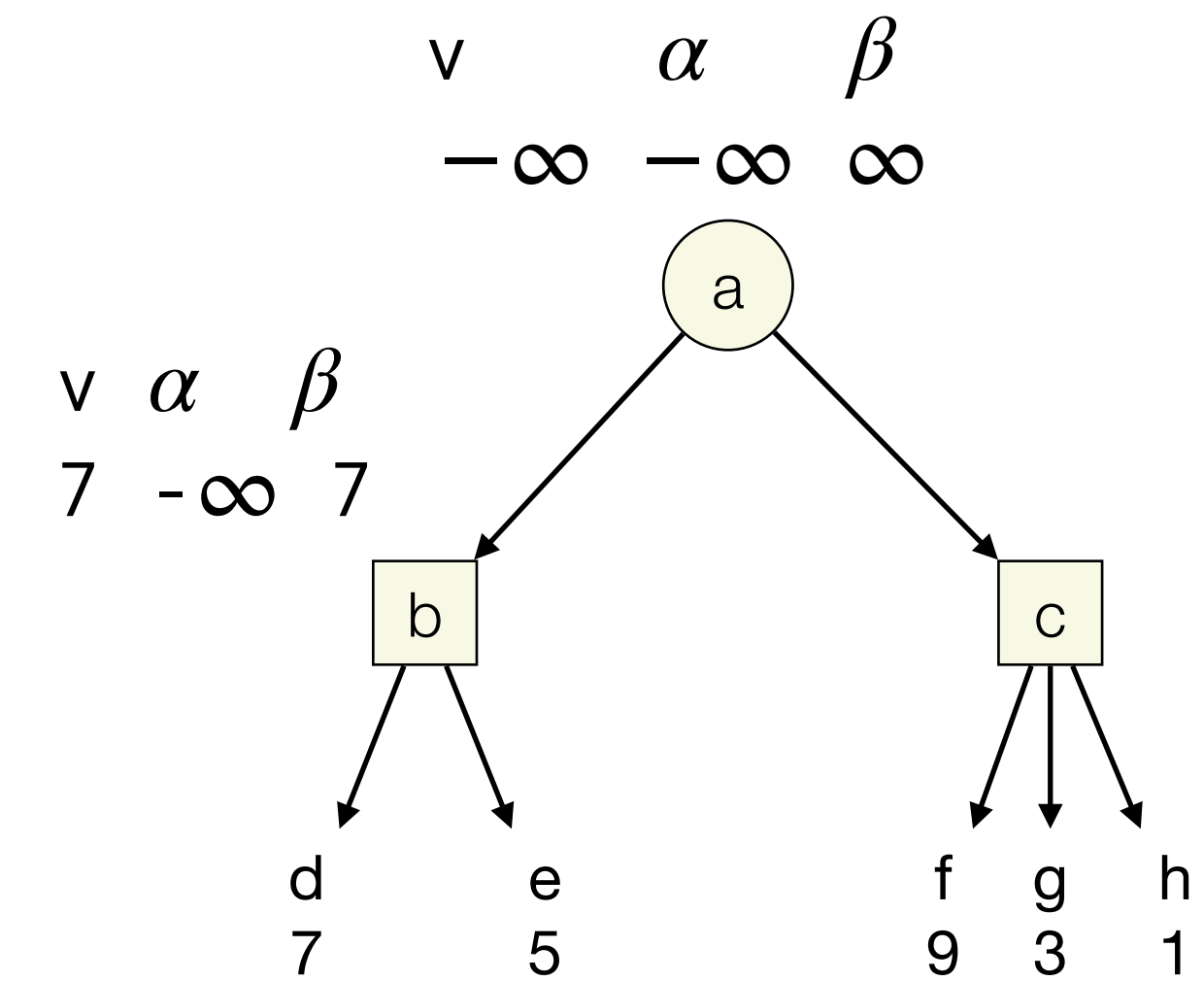
Alpha-Beta Search Example

```
def alphabeta(s, alpha, beta):  
    if terminal(s):  
        return score(s)  
    if player(s) = 1: # MAX player  
        val = -inf  
        for c in children(s):  
            ab = alphabeta(c, alpha, beta)  
            alpha = max(alpha, ab)  
            val = max(val, ab)  
            if alpha >= beta:  
                return val # prune remaining children  
    if player(s) = 2:  
        val = inf  
        for c in children(s):  
            ab = alphabeta(c, alpha, beta)  
            beta = min(beta, ab)  
            val = min(val, ab)  
            if alpha >= beta:  
                return val # prune remaining children  
    return val
```



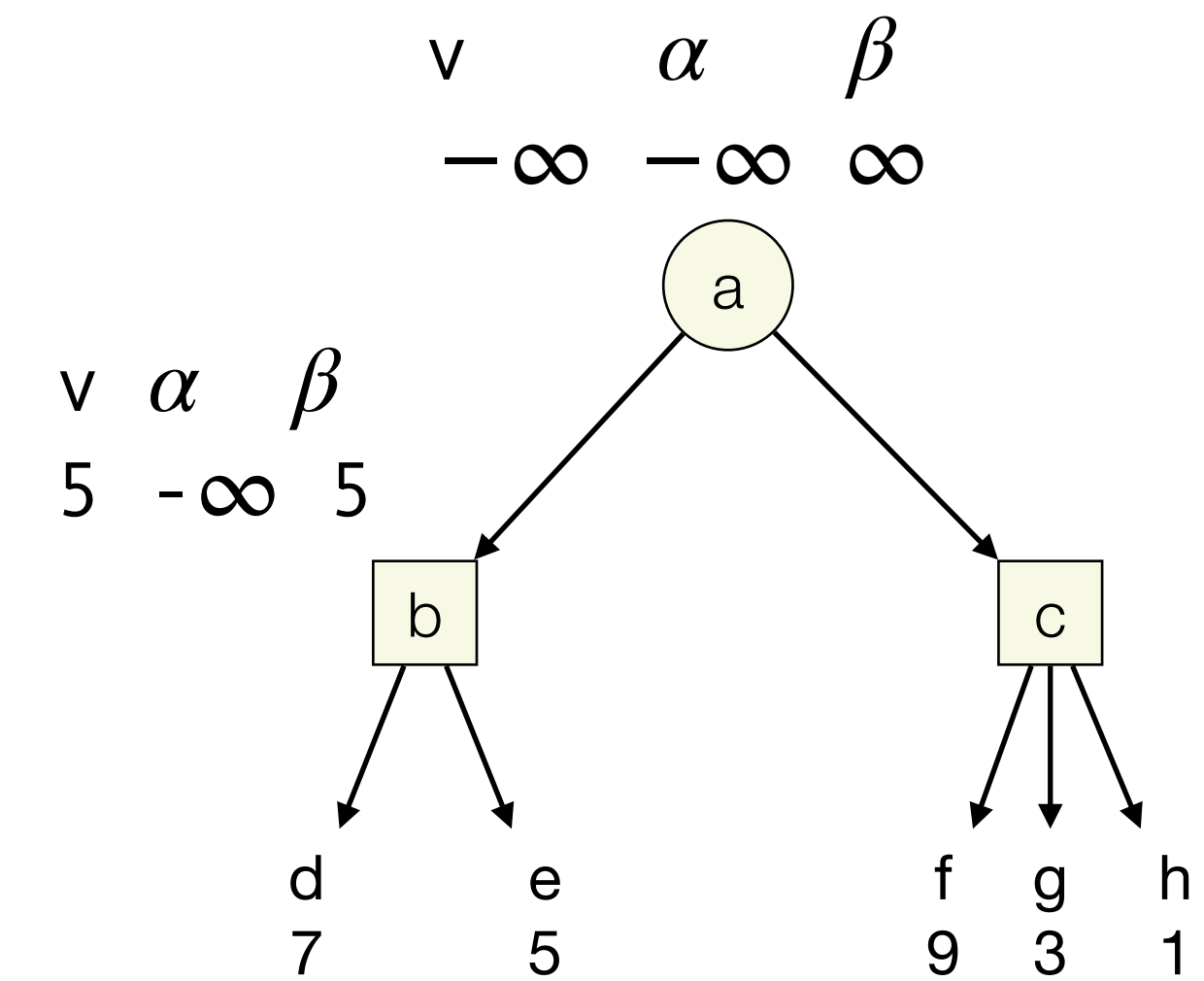
Alpha-Beta Search Example

```
def alphabeta(s, alpha, beta):  
    if terminal(s):  
        return score(s)  
    if player(s) == 1: # MAX player  
        val = -inf  
        for c in children(s):  
            ab = alphabeta(c, alpha, beta)  
            alpha = max(alpha, ab)  
            val = max(val, ab)  
            if alpha >= beta:  
                return val # prune remaining children  
    if player(s) == 2:  
        val = inf  
        for c in children(s):  
            ab = alphabeta(c, alpha, beta)  
            beta = min(beta, ab)  
            val = min(val, ab)  
            if alpha >= beta:  
                return val # prune remaining children  
    return val
```



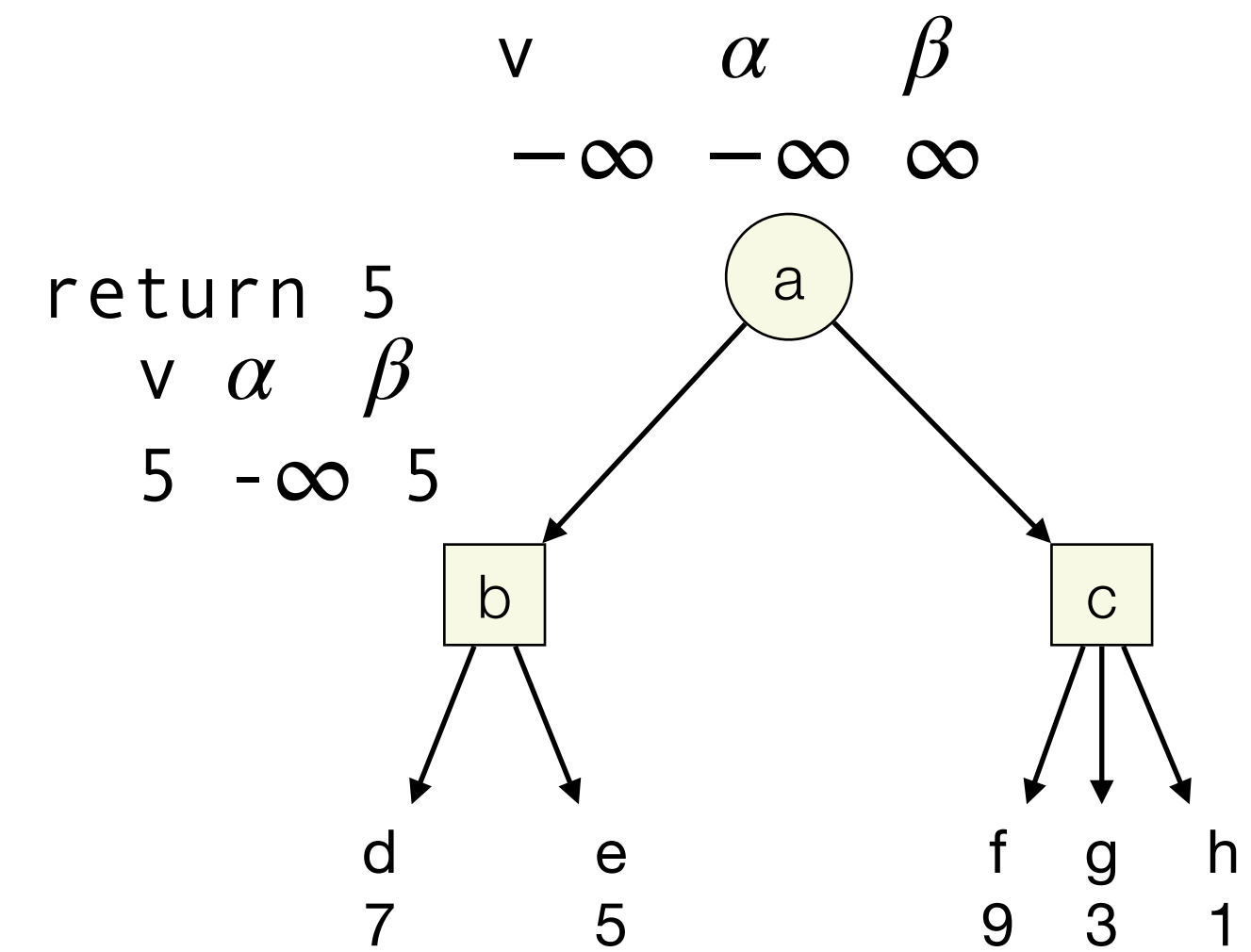
Alpha-Beta Search Example

```
def alphabeta(s, alpha, beta):
    if terminal(s):
        return score(s)
    if player(s) == 1: # MAX player
        val = -inf
        for c in children(s):
            ab = alphabeta(c, alpha, beta)
            alpha = max(alpha, ab)
            val = max(val, ab)
            if alpha >= beta:
                return val # prune remaining children
    if player(s) == 2:
        val = inf
        for c in children(s):
            ab = alphabeta(c, alpha, beta)
            beta = min(beta, ab)
            val = min(val, ab)
            if alpha >= beta:
                return val # prune remaining children
    return val
```



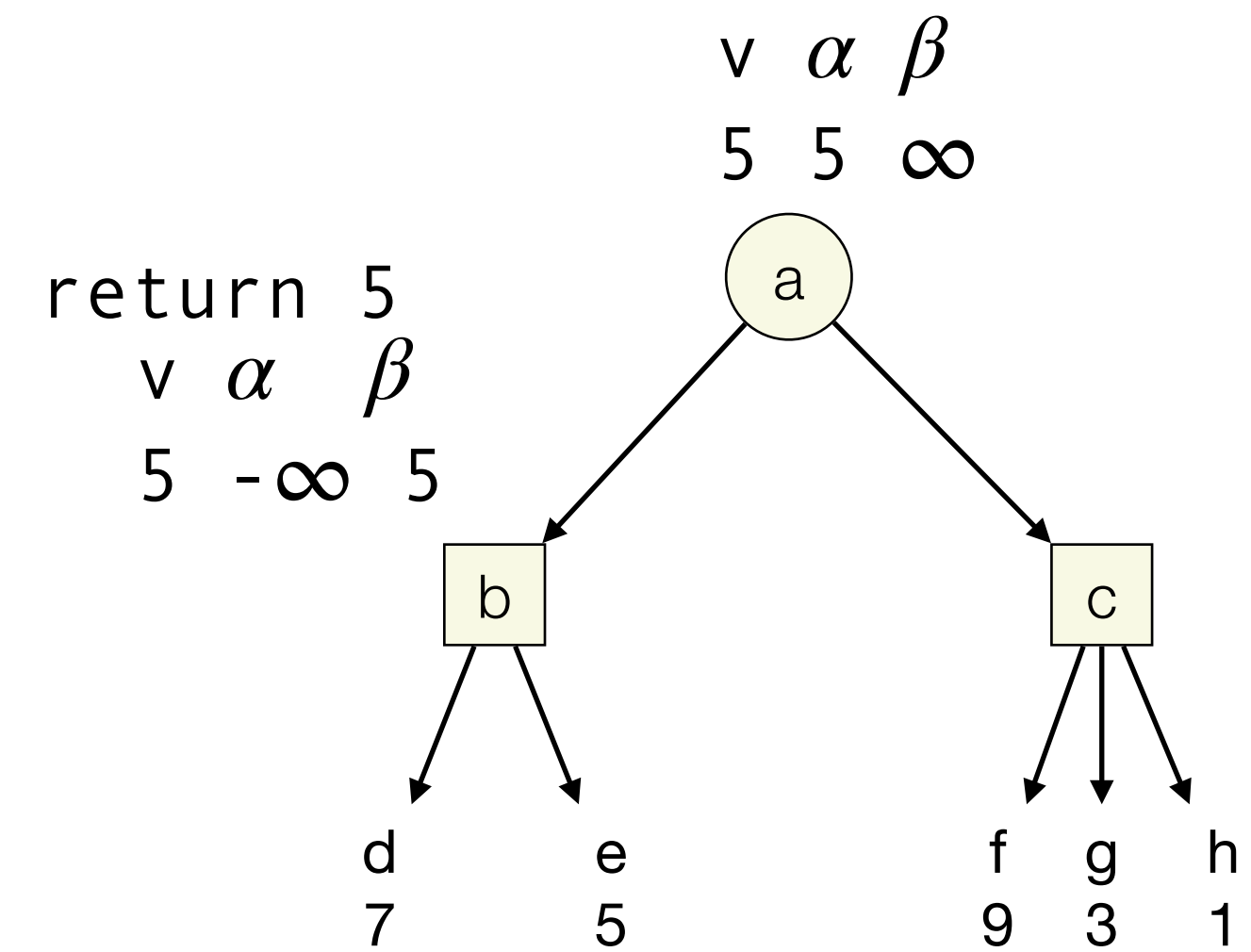
Alpha-Beta Search Example

```
def alphabeta(s, alpha, beta):  
    if terminal(s):  
        return score(s)  
    if player(s) = 1: # MAX player  
        val = -inf  
        for c in children(s):  
            ab = alphabeta(c, alpha, beta)  
            alpha = max(alpha, ab)  
            val = max(val, ab)  
            if alpha >= beta:  
                return val # prune remaining children  
    if player(s) = 2:  
        val = inf  
        for c in children(s):  
            ab = alphabeta(c, alpha, beta)  
            beta = min(beta, ab)  
            val = min(val, ab)  
            if alpha >= beta:  
                return val # prune remaining children  
    return val
```



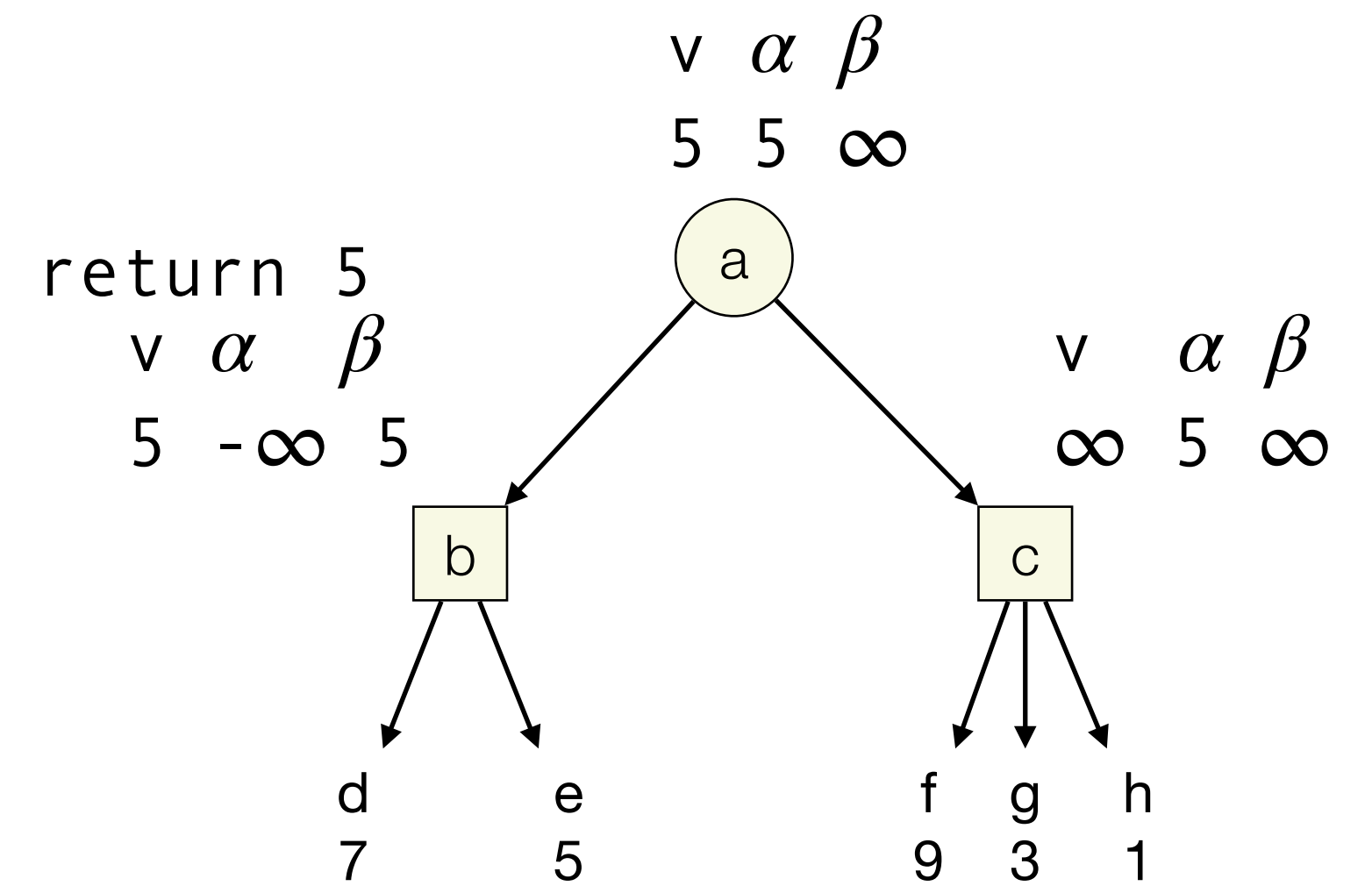
Alpha-Beta Search Example

```
def alphabeta(s, alpha, beta):
    if terminal(s):
        return score(s)
    if player(s) == 1: # MAX player
        val = -inf
        for c in children(s):
            ab = alphabeta(c, alpha, beta)
            alpha = max(alpha, ab)
            val = max(val, ab)
            if alpha >= beta:
                return val # prune remaining children
    if player(s) == 2:
        val = inf
        for c in children(s):
            ab = alphabeta(c, alpha, beta)
            beta = min(beta, ab)
            val = min(val, ab)
            if alpha >= beta:
                return val # prune remaining children
    return val
```



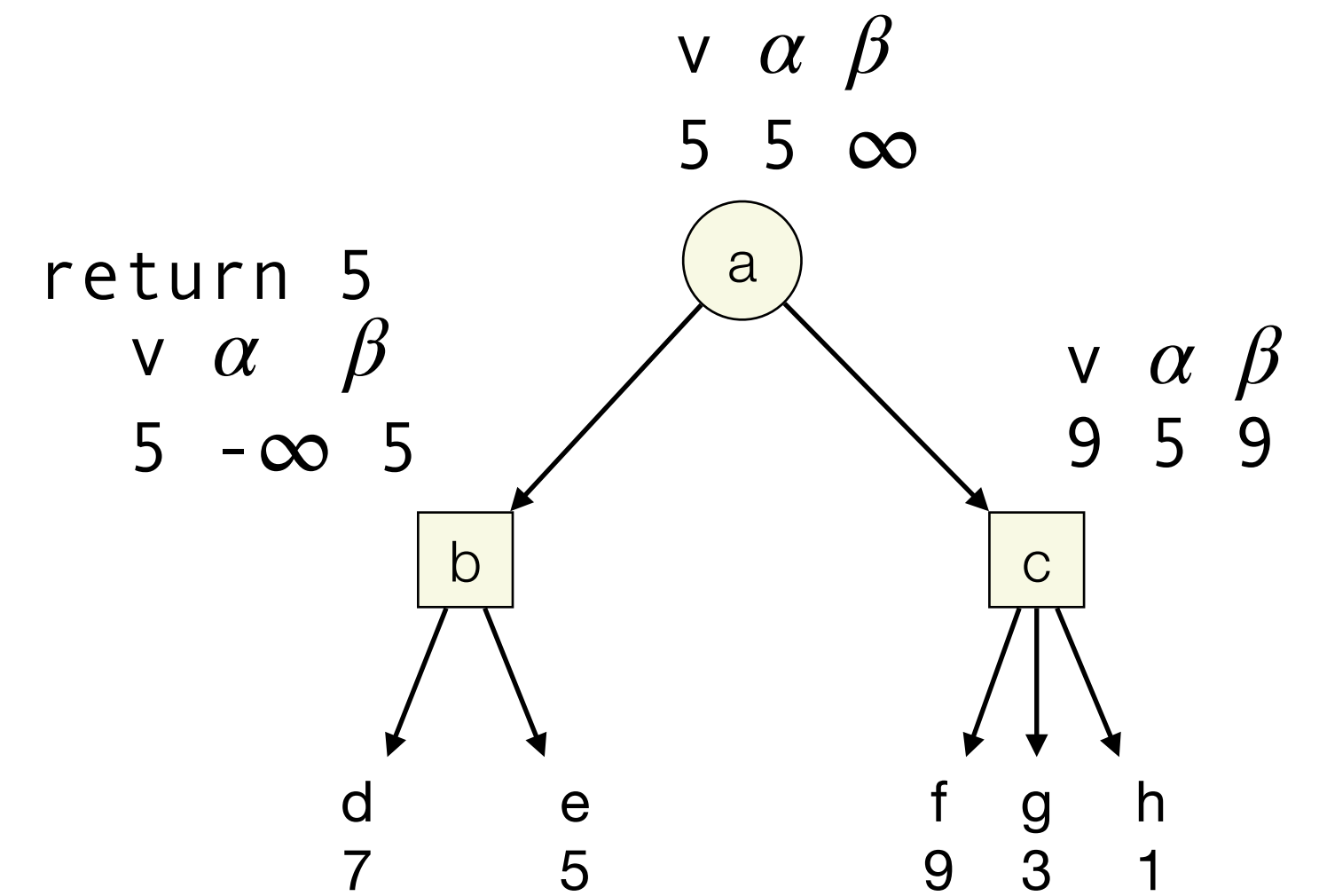
Alpha-Beta Search Example

```
def alphabeta(s, alpha, beta):  
    if terminal(s):  
        return score(s)  
    if player(s) = 1: # MAX player  
        val = -inf  
        for c in children(s):  
            ab = alphabeta(c, alpha, beta)  
            alpha = max(alpha, ab)  
            val = max(val, ab)  
            if alpha >= beta:  
                return val # prune remaining children  
    if player(s) = 2:  
        val = inf  
        for c in children(s):  
            ab = alphabeta(c, alpha, beta)  
            beta = min(beta, ab)  
            val = min(val, ab)  
            if alpha >= beta:  
                return val # prune remaining children  
    return val
```



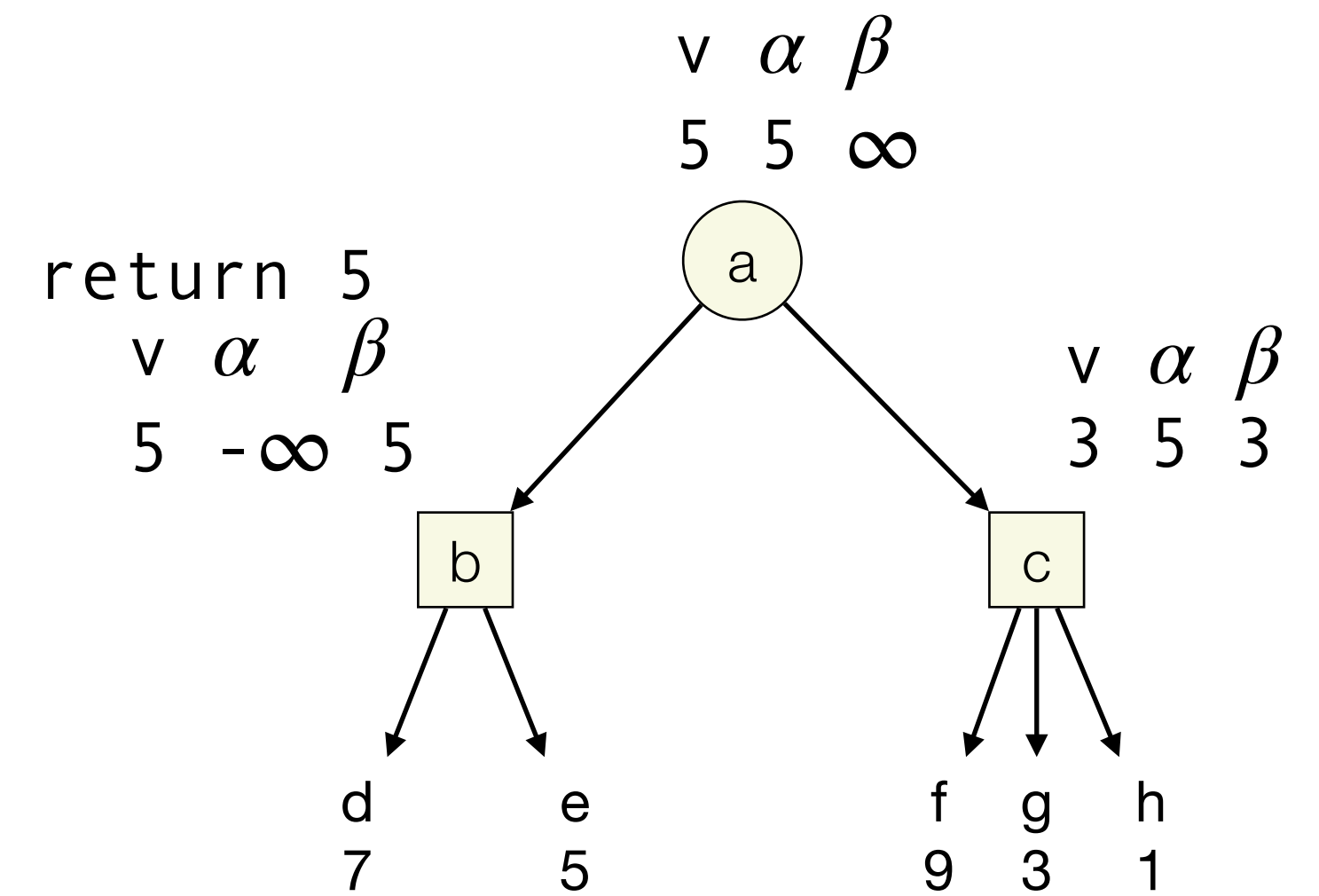
Alpha-Beta Search Example

```
def alphabeta(s, alpha, beta):
    if terminal(s):
        return score(s)
    if player(s) == 1: # MAX player
        val = -inf
        for c in children(s):
            ab = alphabeta(c, alpha, beta)
            alpha = max(alpha, ab)
            val = max(val, ab)
            if alpha >= beta:
                return val # prune remaining children
    if player(s) == 2:
        val = inf
        for c in children(s):
            ab = alphabeta(c, alpha, beta)
            beta = min(beta, ab)
            val = min(val, ab)
            if alpha >= beta:
                return val # prune remaining children
    return val
```



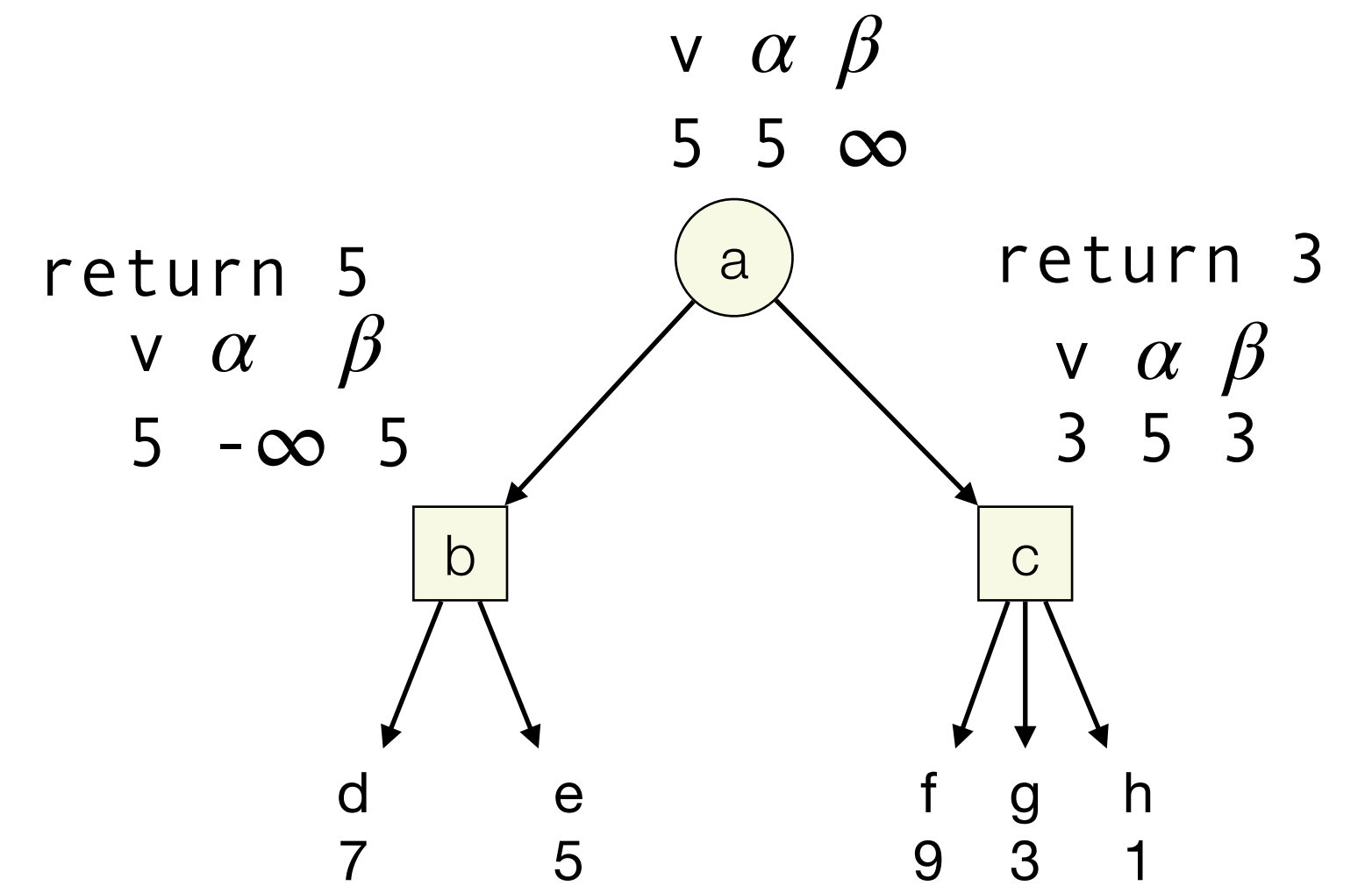
Alpha-Beta Search Example

```
def alphabeta(s, alpha, beta):
    if terminal(s):
        return score(s)
    if player(s) == 1: # MAX player
        val = -inf
        for c in children(s):
            ab = alphabeta(c, alpha, beta)
            alpha = max(alpha, ab)
            val = max(val, ab)
            if alpha >= beta:
                return val # prune remaining children
    if player(s) == 2:
        val = inf
        for c in children(s):
            ab = alphabeta(c, alpha, beta)
            beta = min(beta, ab)
            val = min(val, ab)
            if alpha >= beta:
                return val # prune remaining children
    return val
```



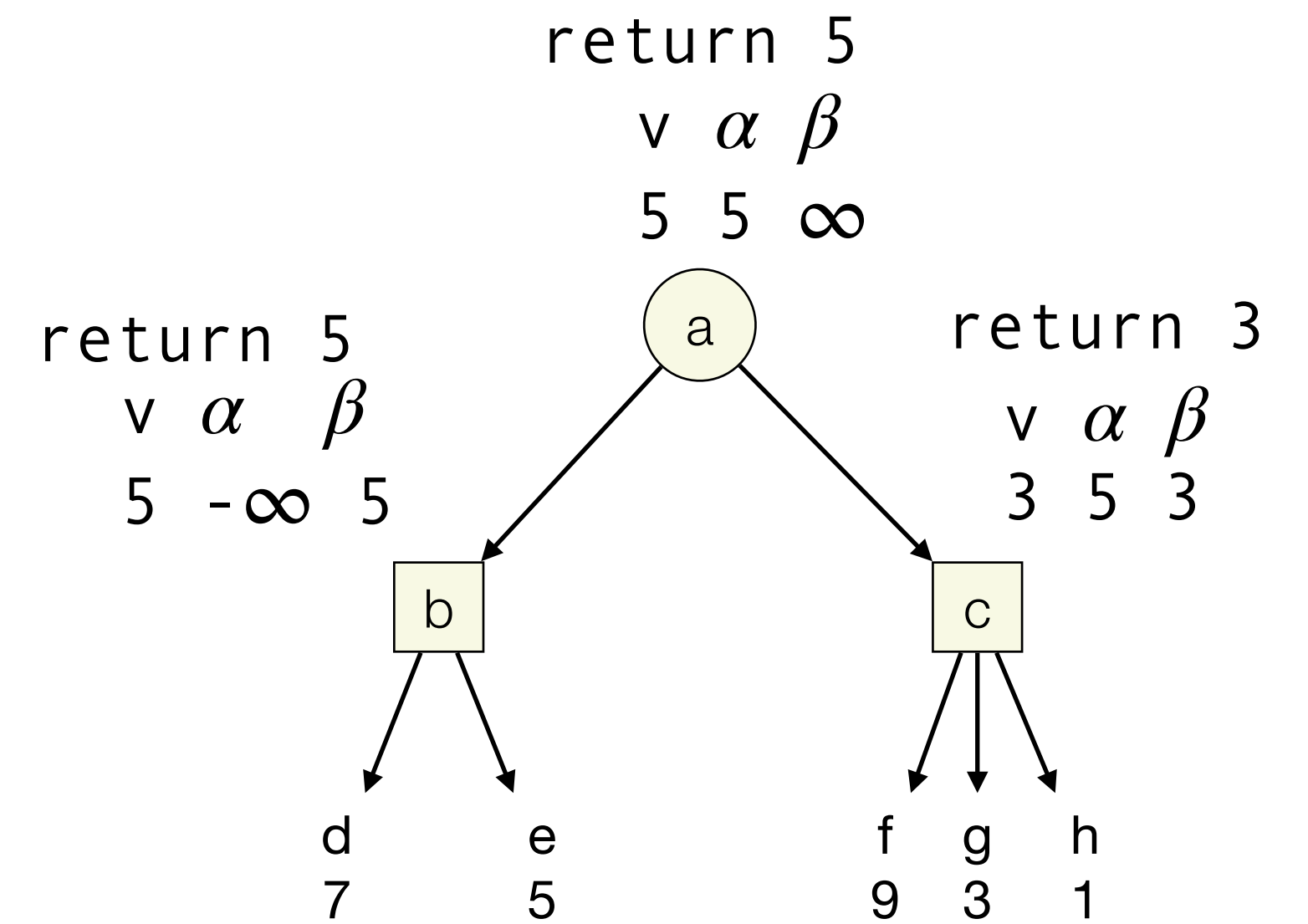
Alpha-Beta Search Example

```
def alphabeta(s, alpha, beta):  
    if terminal(s):  
        return score(s)  
    if player(s) = 1: # MAX player  
        val = -inf  
        for c in children(s):  
            ab = alphabeta(c, alpha, beta)  
            alpha = max(alpha, ab)  
            val = max(val, ab)  
            if alpha >= beta:  
                return val # prune remaining children  
    if player(s) = 2:  
        val = inf  
        for c in children(s):  
            ab = alphabeta(c, alpha, beta)  
            beta = min(beta, ab)  
            val = min(val, ab)  
            if alpha >= beta:  
                return val # prune remaining children  
    return val
```



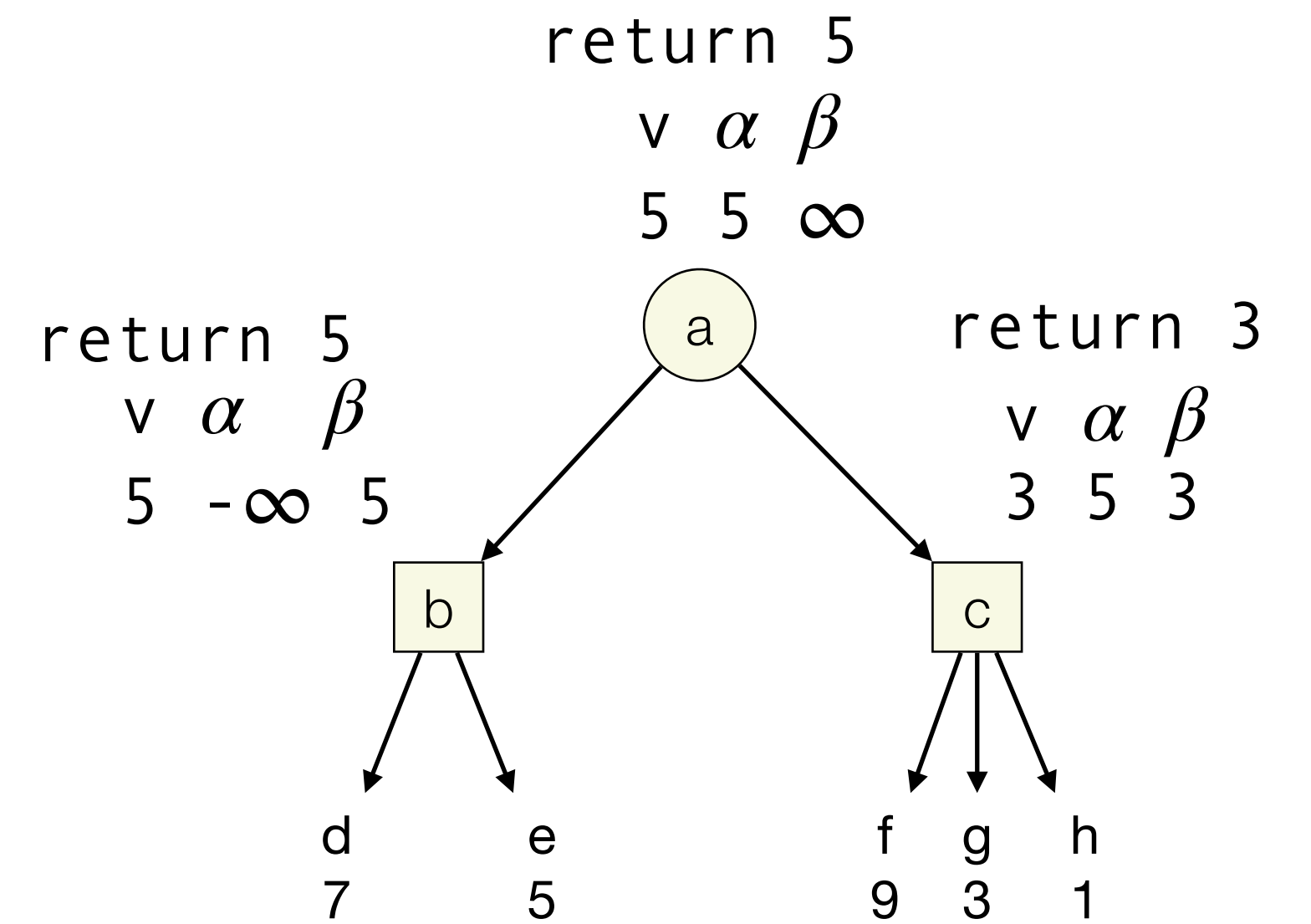
Alpha-Beta Search Example

```
def alphabeta(s, alpha, beta):  
    if terminal(s):  
        return score(s)  
    if player(s) = 1: # MAX player  
        val = -inf  
        for c in children(s):  
            ab = alphabeta(c, alpha, beta)  
            alpha = max(alpha, ab)  
            val = max(val, ab)  
            if alpha >= beta:  
                return val # prune remaining children  
    if player(s) = 2:  
        val = inf  
        for c in children(s):  
            ab = alphabeta(c, alpha, beta)  
            beta = min(beta, ab)  
            val = min(val, ab)  
            if alpha >= beta:  
                return val # prune remaining children  
    return val
```



Alpha-Beta Search Example

```
def alphabeta(s, alpha, beta):
    if terminal(s):
        return score(s)
    if player(s) == 1: # MAX player
        val = -inf
        for c in children(s):
            ab = alphabeta(c, alpha, beta)
            alpha = max(alpha, ab)
            val = max(val, ab)
            if alpha >= beta:
                return val # prune remaining children
    if player(s) == 2:
        val = inf
        for c in children(s):
            ab = alphabeta(c, alpha, beta)
            beta = min(beta, ab)
            val = min(val, ab)
            if alpha >= beta:
                return val # prune remaining children
    return val
```



Demo:

```
% cd abeta/  
% python3 alphabeta.py < t1.in
```

Implementation: `abeta/alphabet.py`

Implementation: `abeta/alpha.py`

```
L, T, V, root = readtree()  
alpha(0, T, V, root, float('-inf'), float('inf'))
```

Implementation: `abeta/alphabeta.py`

```
L, T, V, root = readtree()
alphabeta(0, T, V, root, float('-inf'), float('inf'))
```

```
def alphabeta(d, T, V, v, alpha, beta):
    if v in V: # V is the set of leaves
        val = V[v]
        return val
```

Implementation: abeta/alphabeta.py

```
L, T, V, root = readtree()
alphabeta(0, T, V, root, float('-inf'), float('inf'))
```

```
def alphabeta(d, T, V, v, alpha, beta):
    if v in V: # V is the set of leaves
        val = V[v]
        return val
```

```
    if 0 == d%2: # MAX node
        val = float('-inf')
        for c in T[v]:
            ab = alphabeta(d+1, T, V, c, alpha, beta)
            if ab > val: # have improved current mmax value
                alpha, val = ab, ab
            if alpha >= beta:
                break
        return val
```

Implementation: `abeta/alphabeta.py`

```
L, T, V, root = readtree()
alphabeta(0, T, V, root, float('-inf'), float('inf'))
```

```
def alphabeta(d, T, V, v, alpha, beta):
    if v in V: # V is the set of leaves
        val = V[v]
        return val
```

```
    if 0 == d%2: # MAX node
        val = float('-inf')
        for c in T[v]:
            ab = alphabeta(d+1, T, V, c, alpha, beta)
            if ab > val: # have improved current mmax value
                alpha, val = ab, ab
            if alpha >= beta:
                break
        return val
```

```
    #else a MIN node
    val = float('inf')
    for c in T[v]:
        ab = alphabeta(d+1, T, V, c, alpha, beta)
        if ab < val:
            beta, val = ab, ab
        if alpha >= beta:
            break
```

Implementation: abeta/alphabeta.py

```
L, T, V, root = readtree()
alphabeta(0, T, V, root, float('-inf'), float('inf'))
```

```
def alphabeta(d, T, V, v, alpha, beta):
    if v in V: # V is the set of leaves
        val = V[v]
        return val
```

```
    if 0 == d%2: # MAX node
        val = float('-inf')
        for c in T[v]:
            ab = alphabeta(d+1, T, V, c, alpha, beta)
            if ab > val: # have improved current mmax value
                alpha, val = ab, ab
            if alpha >= beta:
                break
        return val
```

```
    #else a MIN node
    val = float('inf')
    for c in T[v]:
        ab = alphabeta(d+1, T, V, c, alpha, beta)
        if ab < val:
            beta, val = ab, ab
        if alpha >= beta:
            break
```

Implementation: abeta/alphabeta.py

```
L, T, V, root = readtree()
alphabeta(0, T, V, root, float('-inf'), float('inf'))
```

```
def alphabeta(d, T, V, v, alpha, beta):
    if v in V: # V is the set of leaves
        val = V[v]
        return val
```

```
    if 0 == d%2: # MAX node
        val = float('-inf')
        for c in T[v]:
            ab = alphabeta(d+1, T, V, c, alpha, beta)
            if ab > val: # have improved current mmax value
                alpha, val = ab, ab
            if alpha >= beta:
                break
        return val
```

Question:

1. Why are we updating alpha if $ab > val$ instead of $ab > alpha$?

```
    #else a MIN node
    val = float('inf')
    for c in T[v]:
        ab = alphabeta(d+1, T, V, c, alpha, beta)
        if ab < val:
            beta, val = ab, ab
        if alpha >= beta:
            break
```

Implementation: abeta/alphabeta.py

```
L, T, V, root = readtree()
alphabeta(0, T, V, root, float('-inf'), float('inf'))
```

```
def alphabeta(d, T, V, v, alpha, beta):
    if v in V: # V is the set of leaves
        val = V[v]
        return val
```

```
    if 0 == d%2: # MAX node
        val = float('-inf')
        for c in T[v]:
            ab = alphabeta(d+1, T, V, c, alpha, beta)
            if ab > val: # have improved current mmax value
                alpha, val = ab, ab
            if alpha >= beta:
                break
        return val
```

Question:

1. Why are we updating alpha if $ab > val$ instead of $ab > alpha$?
2. How does this line tell us we're a MAX node?

```
#else a MIN node
val = float('inf')
for c in T[v]:
    ab = alphabeta(d+1, T, V, c, alpha, beta)
    if ab < val:
        beta, val = ab, ab
    if alpha >= beta:
        break
```

Implementation: abeta/alphabeta.py

```
L, T, V, root = readtree()
alphabeta(0, T, V, root, float('-inf'), float('inf'))
```

```
def alphabeta(d, T, V, v, alpha, beta):
    if v in V: # V is the set of leaves
        val = V[v]
        return val
```

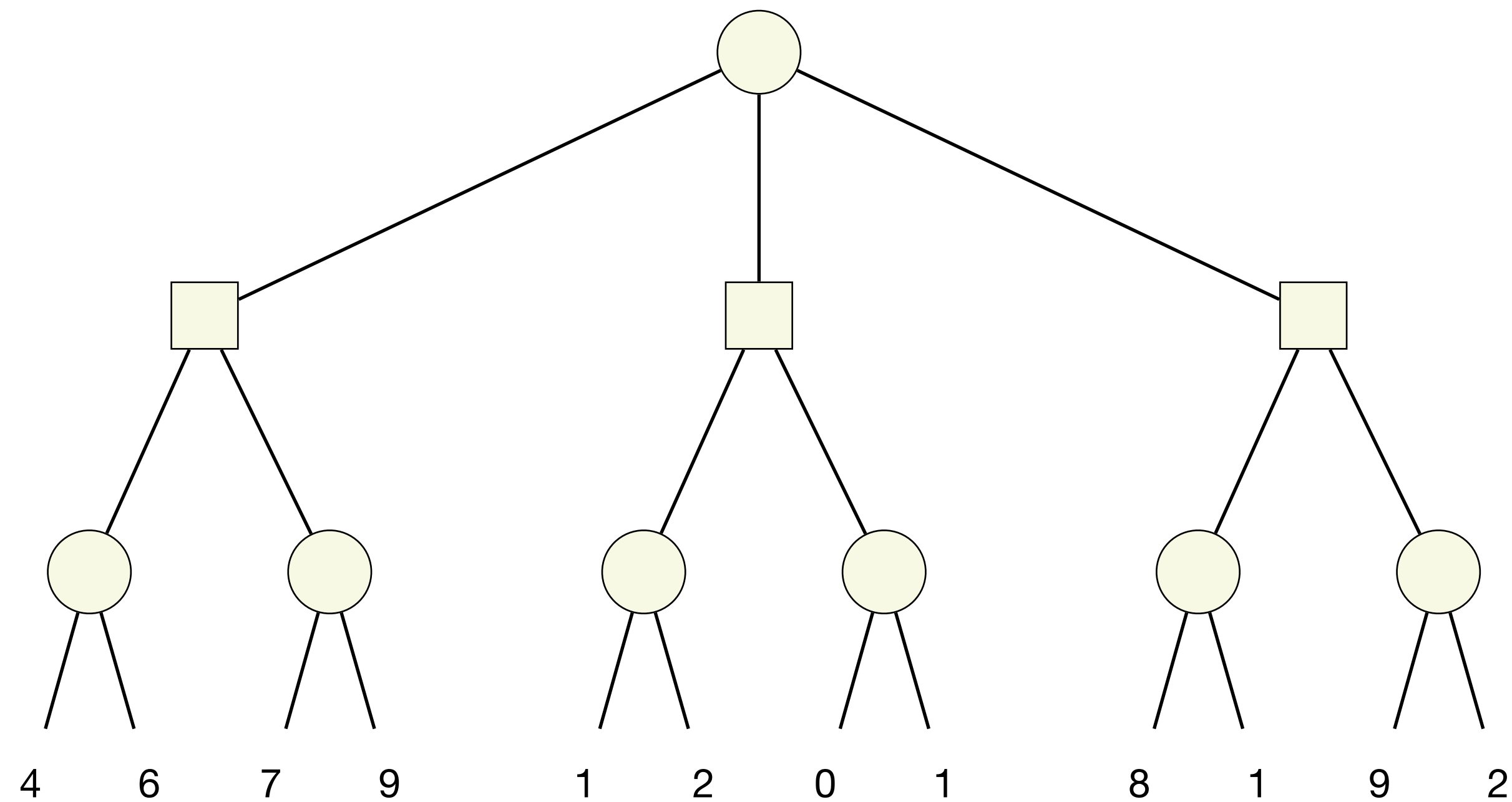
```
    if 0 == d%2: # MAX node
        val = float('-inf')
        for c in T[v]:
            ab = alphabeta(d+1, T, V, c, alpha, beta)
            if ab > val: # have improved current mmax value
                alpha, val = ab, ab
            if alpha >= beta:
                break
        return val
```

Question:

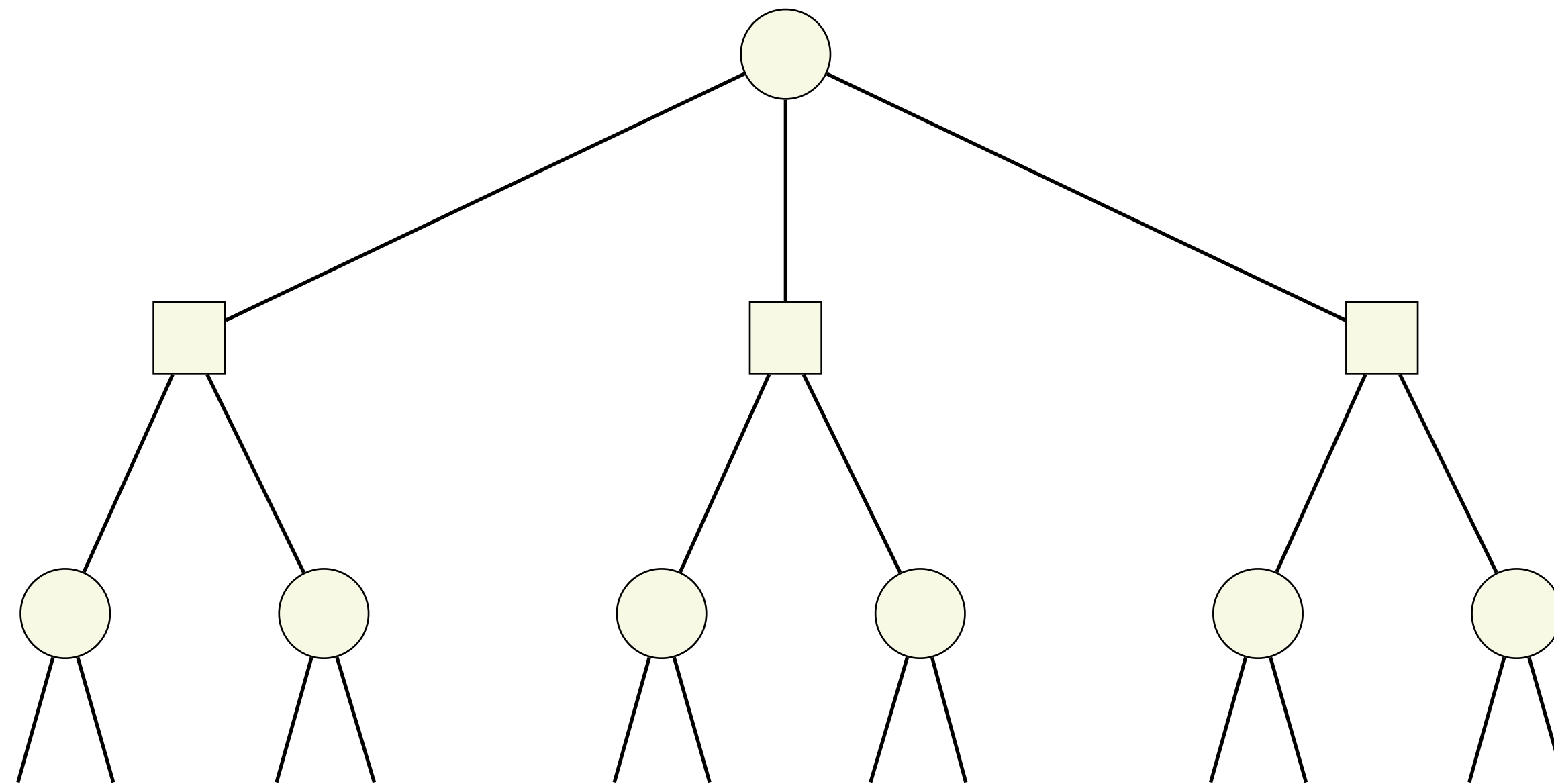
1. Why are we updating alpha if $ab > val$ instead of $ab > alpha$?
2. How does this line tell us we're a MAX node?
3. Why $alpha \geq beta$ instead of $alpha > beta$?

```
    #else a MIN node
    val = float('inf')
    for c in T[v]:
        ab = alphabeta(d+1, T, V, c, alpha, beta)
        if ab < val:
            beta, val = ab, ab
        if alpha >= beta:
            break
```

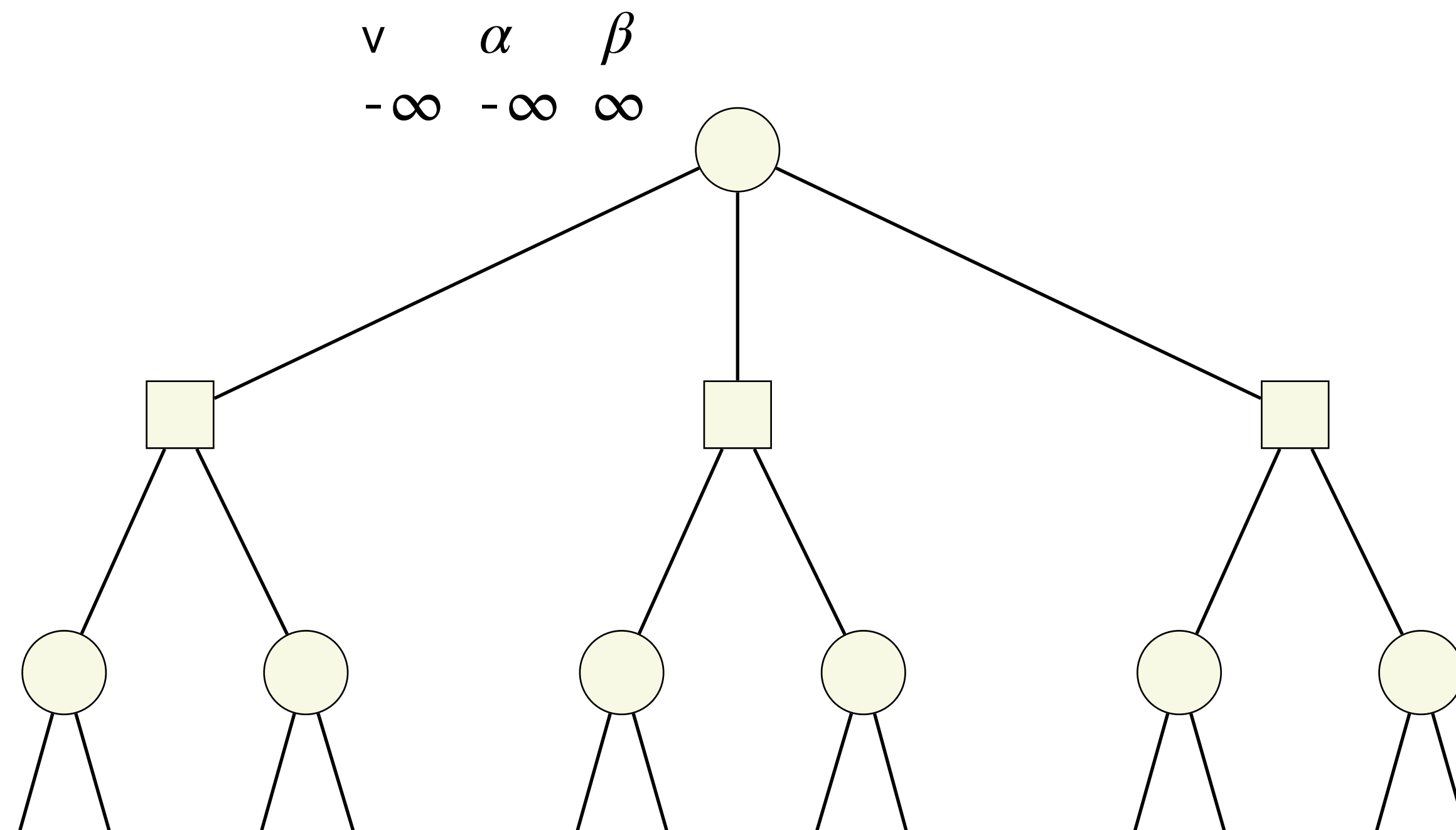
Alpha-Beta Search Example #2



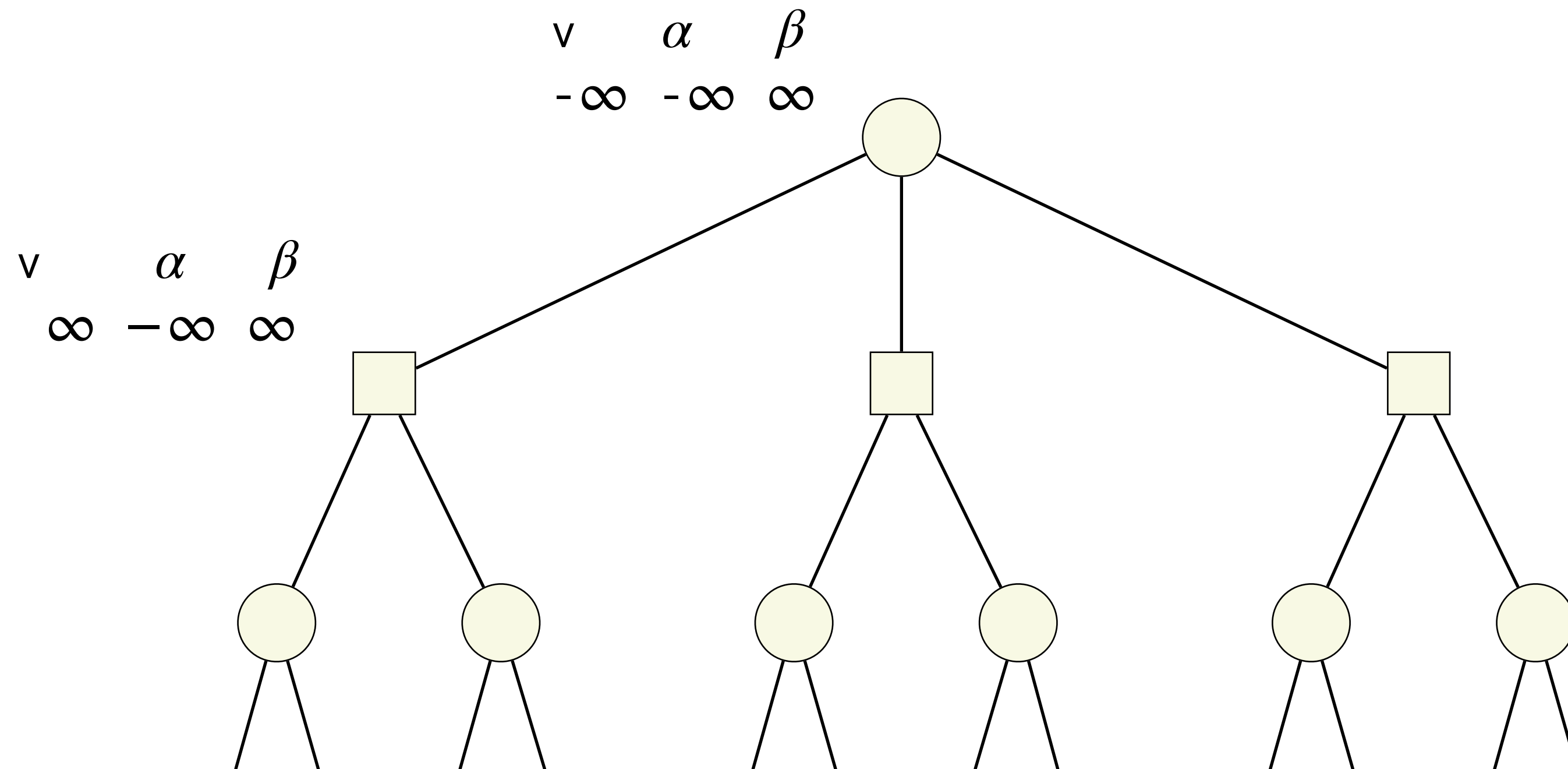
Alpha-Beta Search Example #2



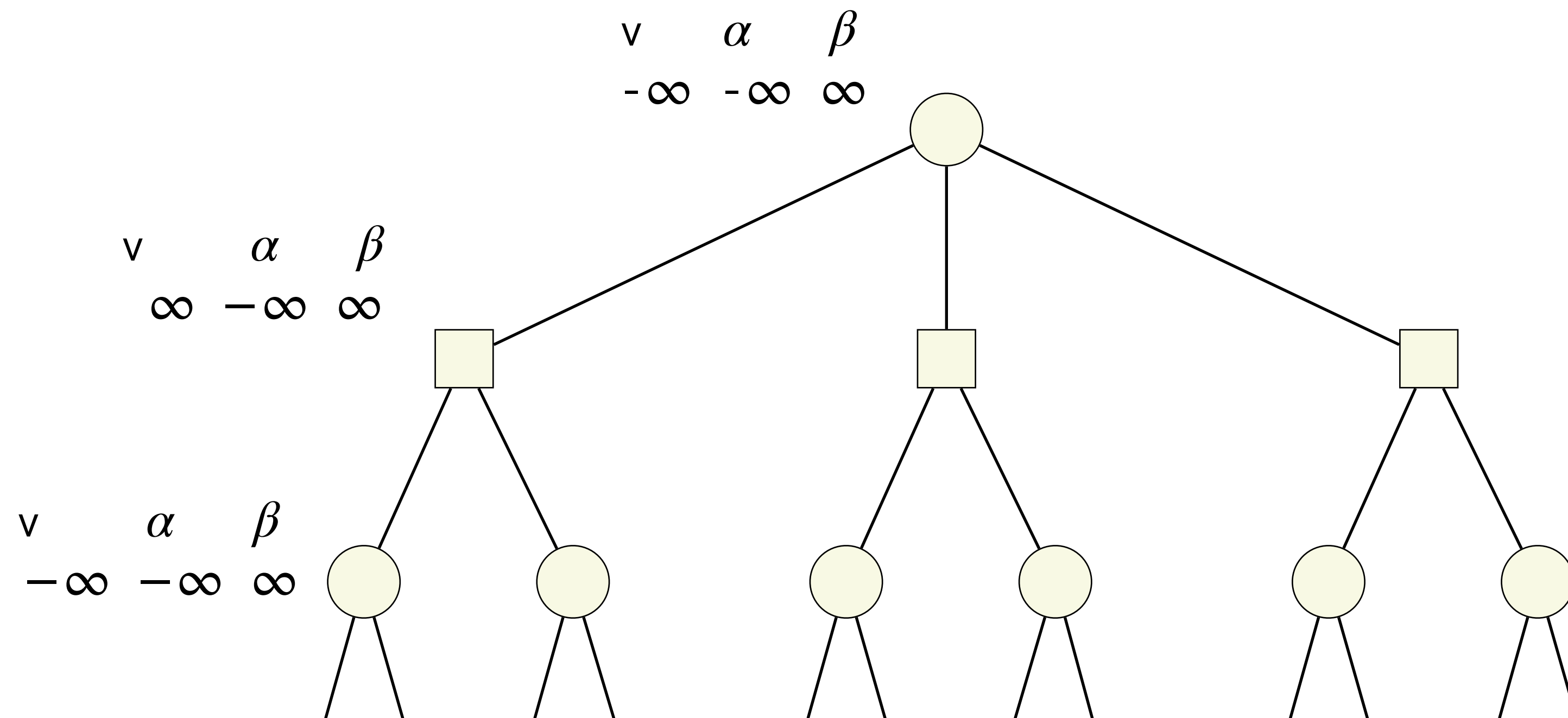
Alpha-Beta Search Example #2



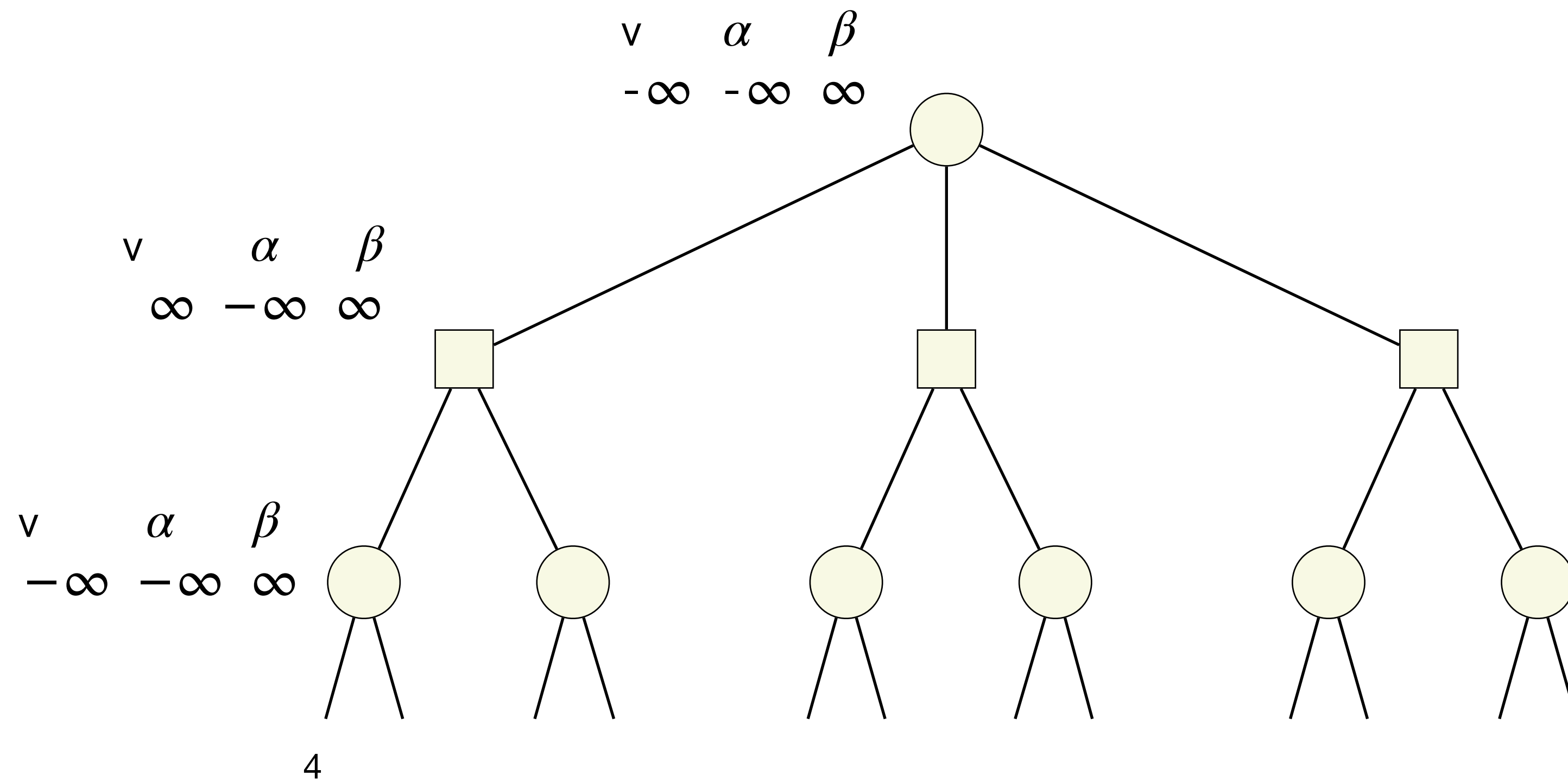
Alpha-Beta Search Example #2



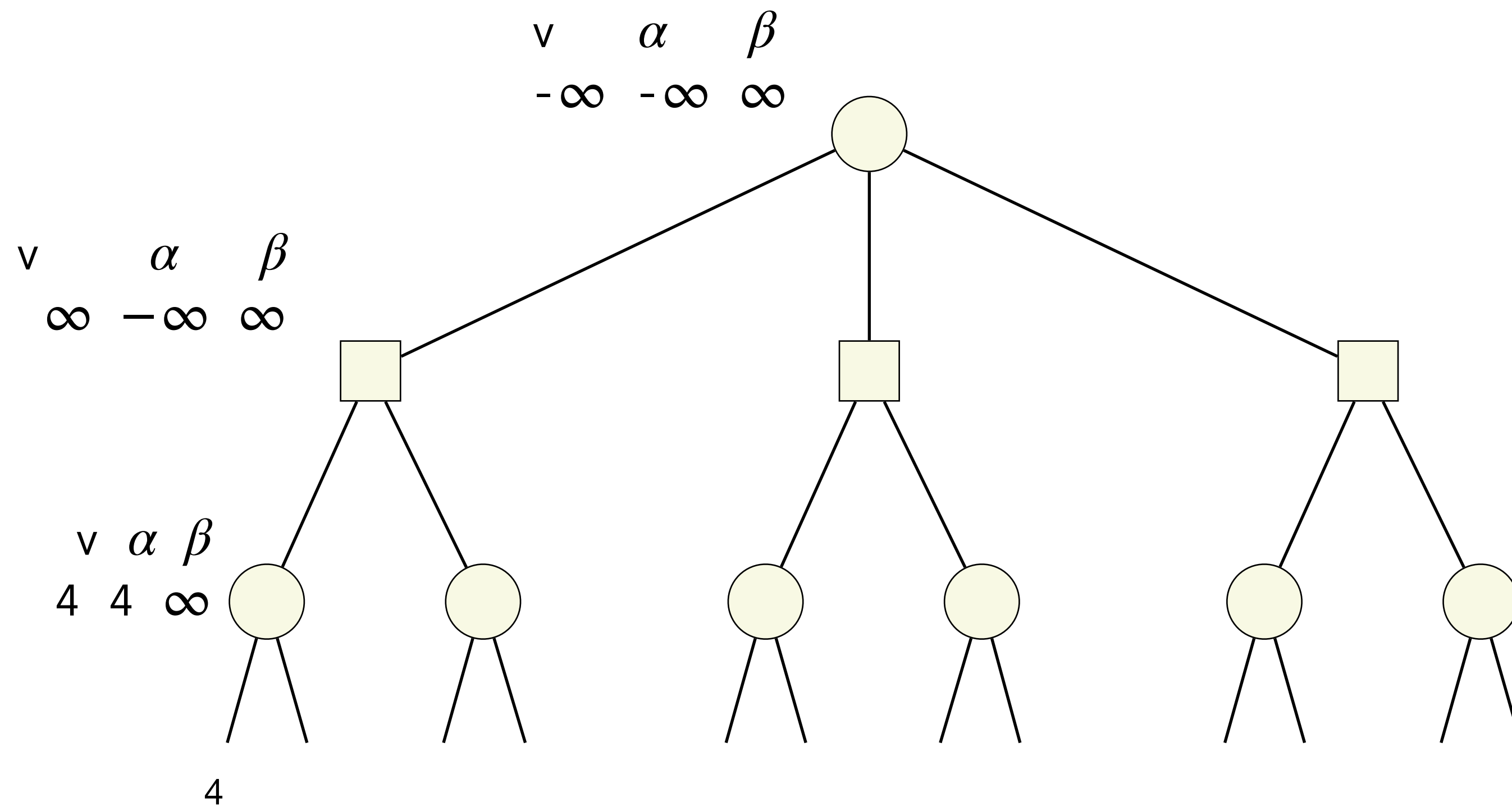
Alpha-Beta Search Example #2



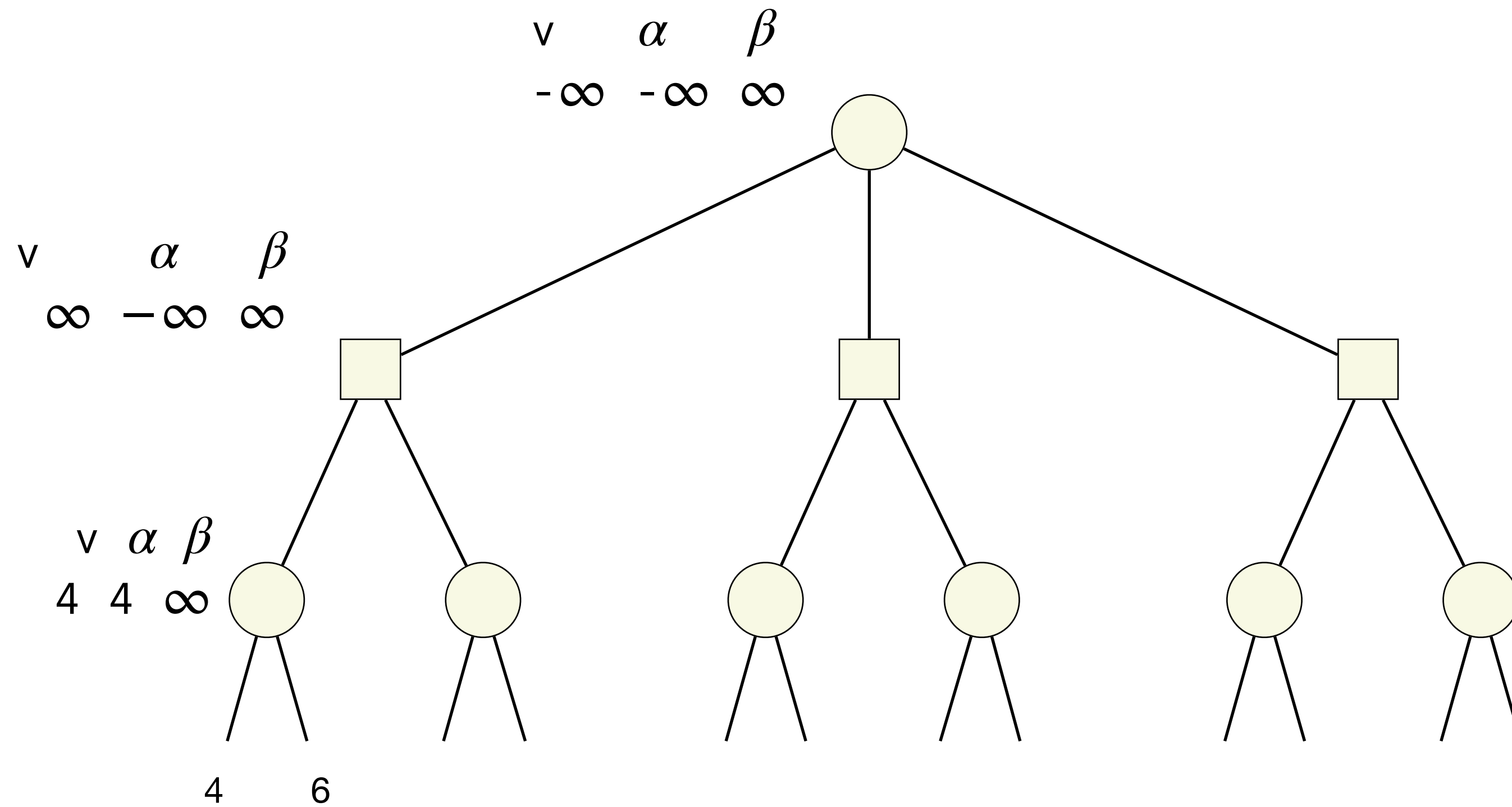
Alpha-Beta Search Example #2



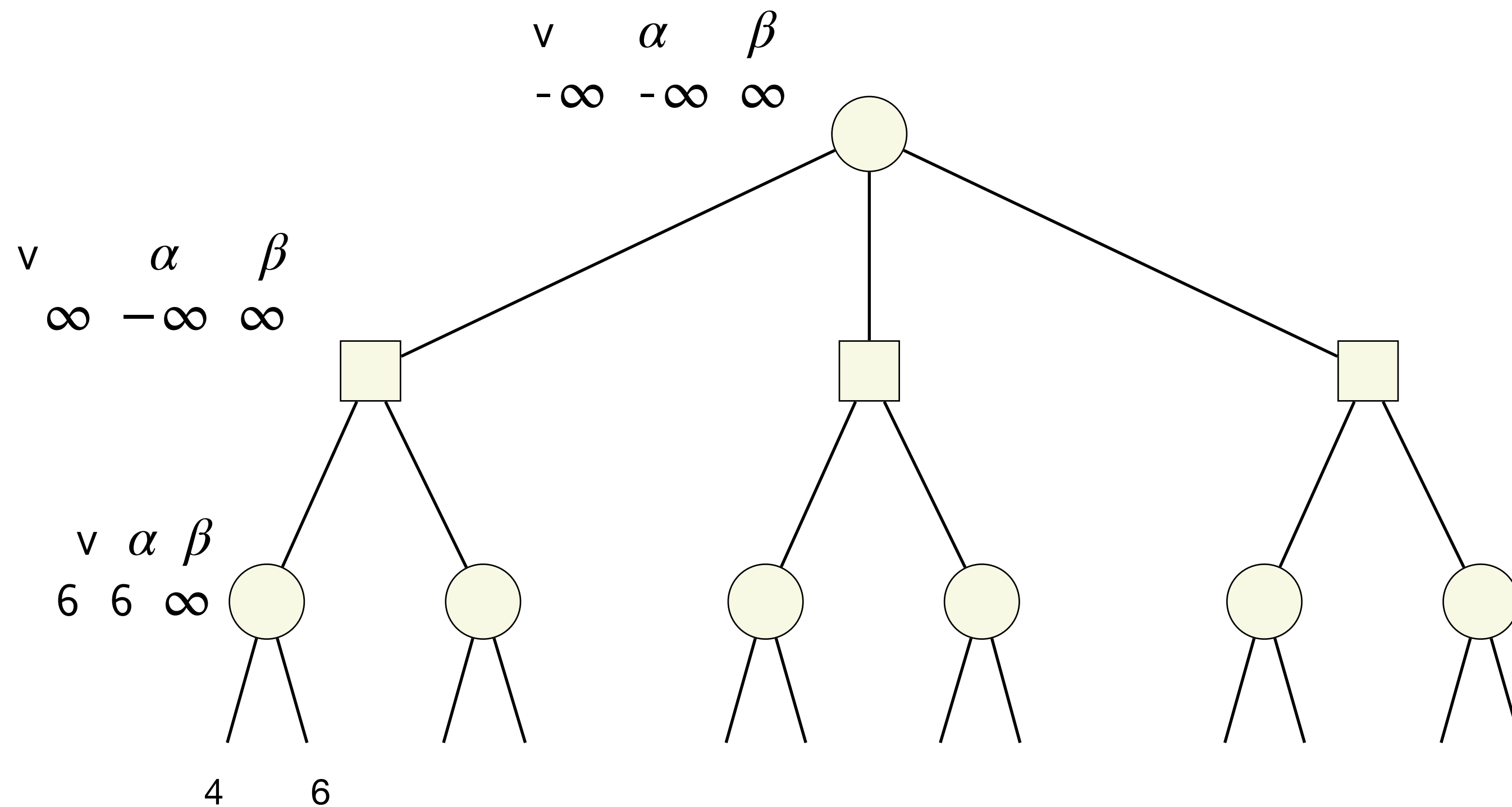
Alpha-Beta Search Example #2



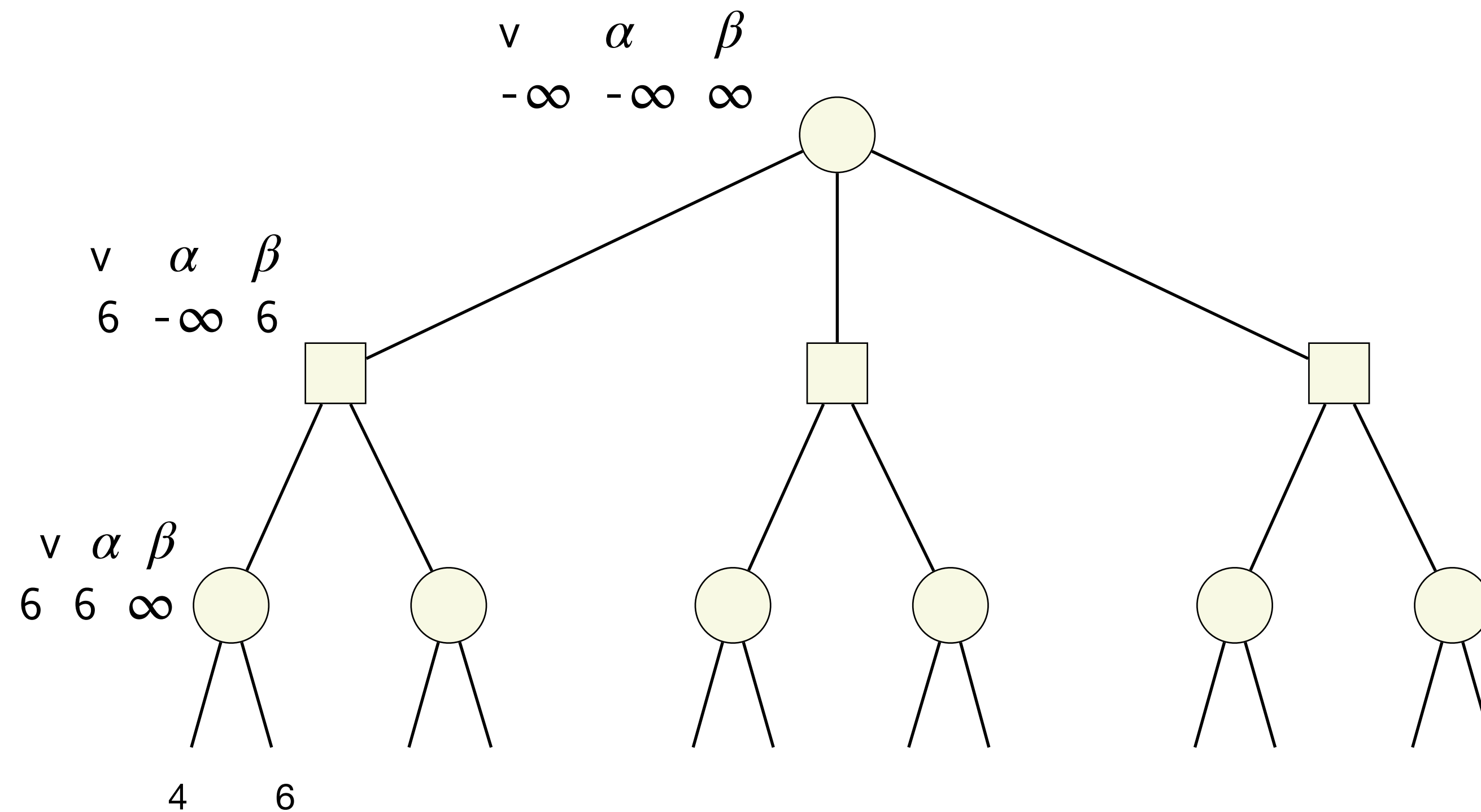
Alpha-Beta Search Example #2



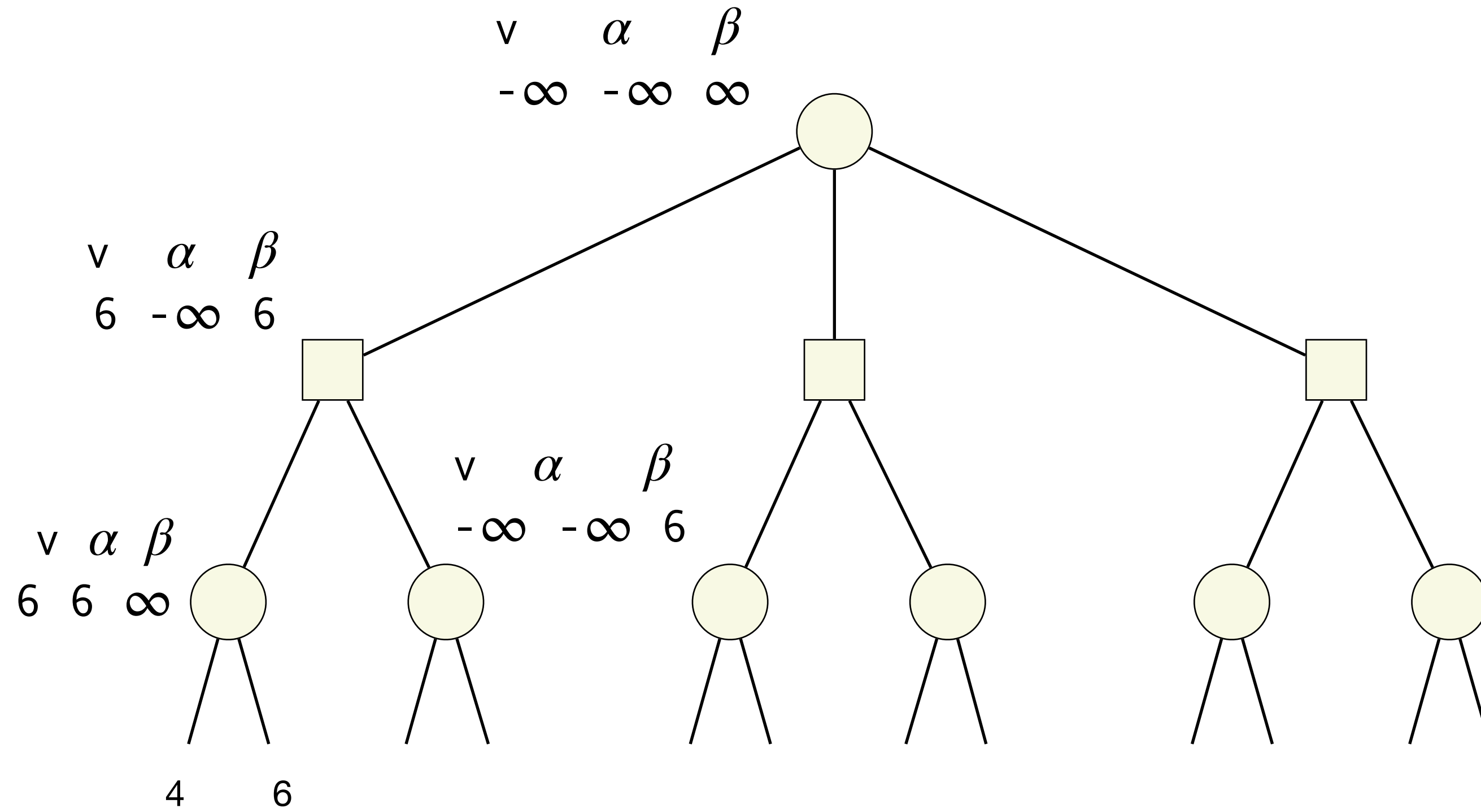
Alpha-Beta Search Example #2



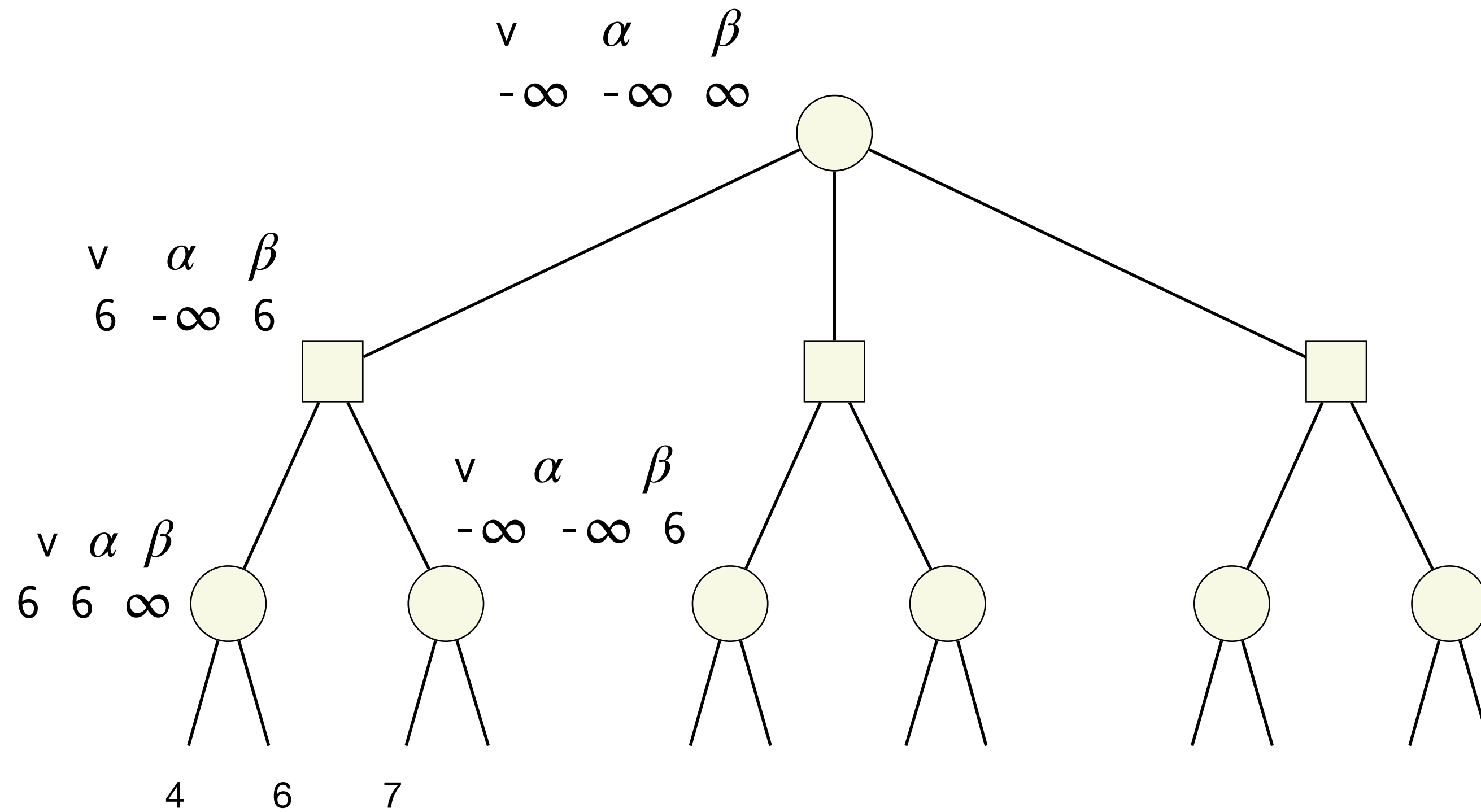
Alpha-Beta Search Example #2



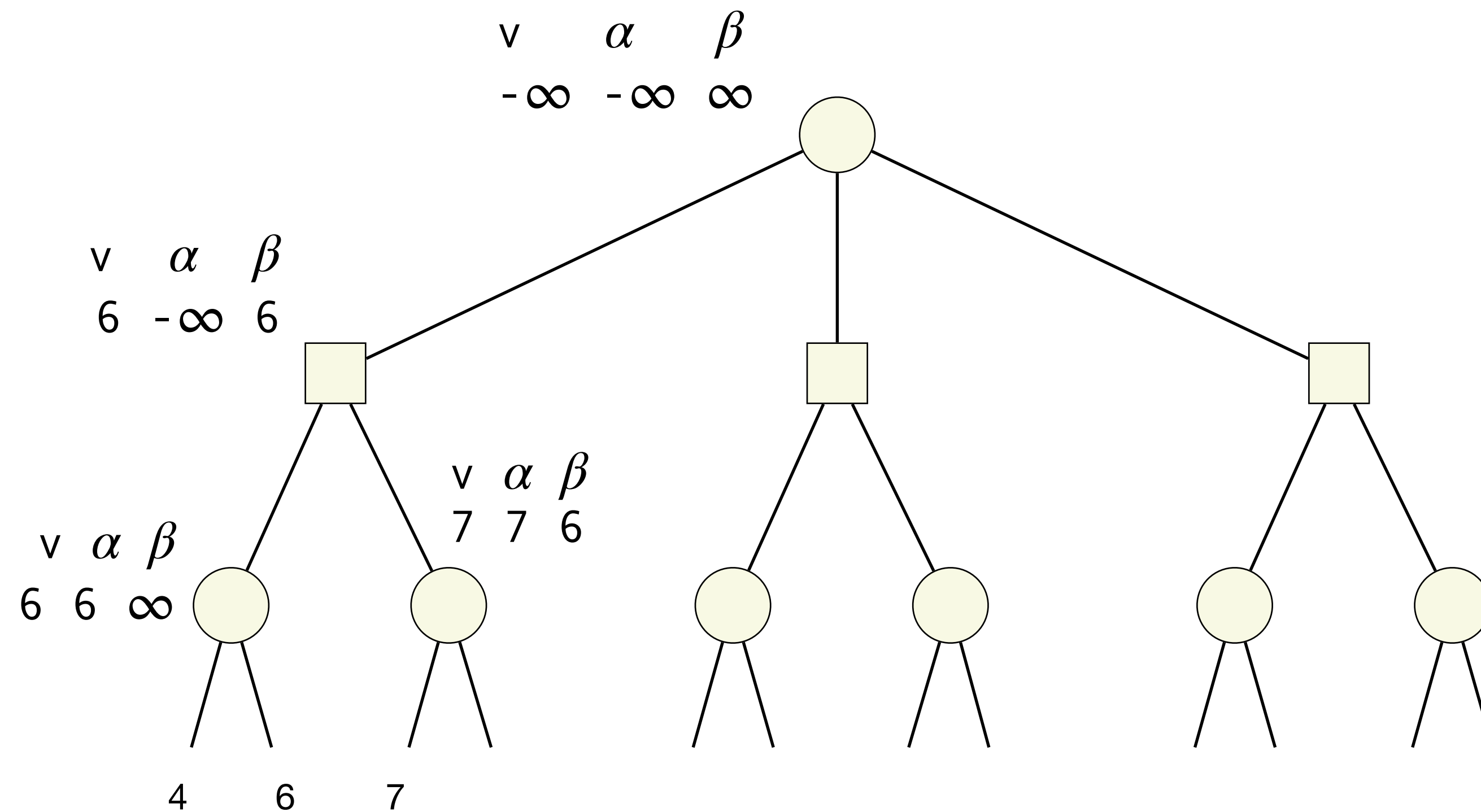
Alpha-Beta Search Example #2



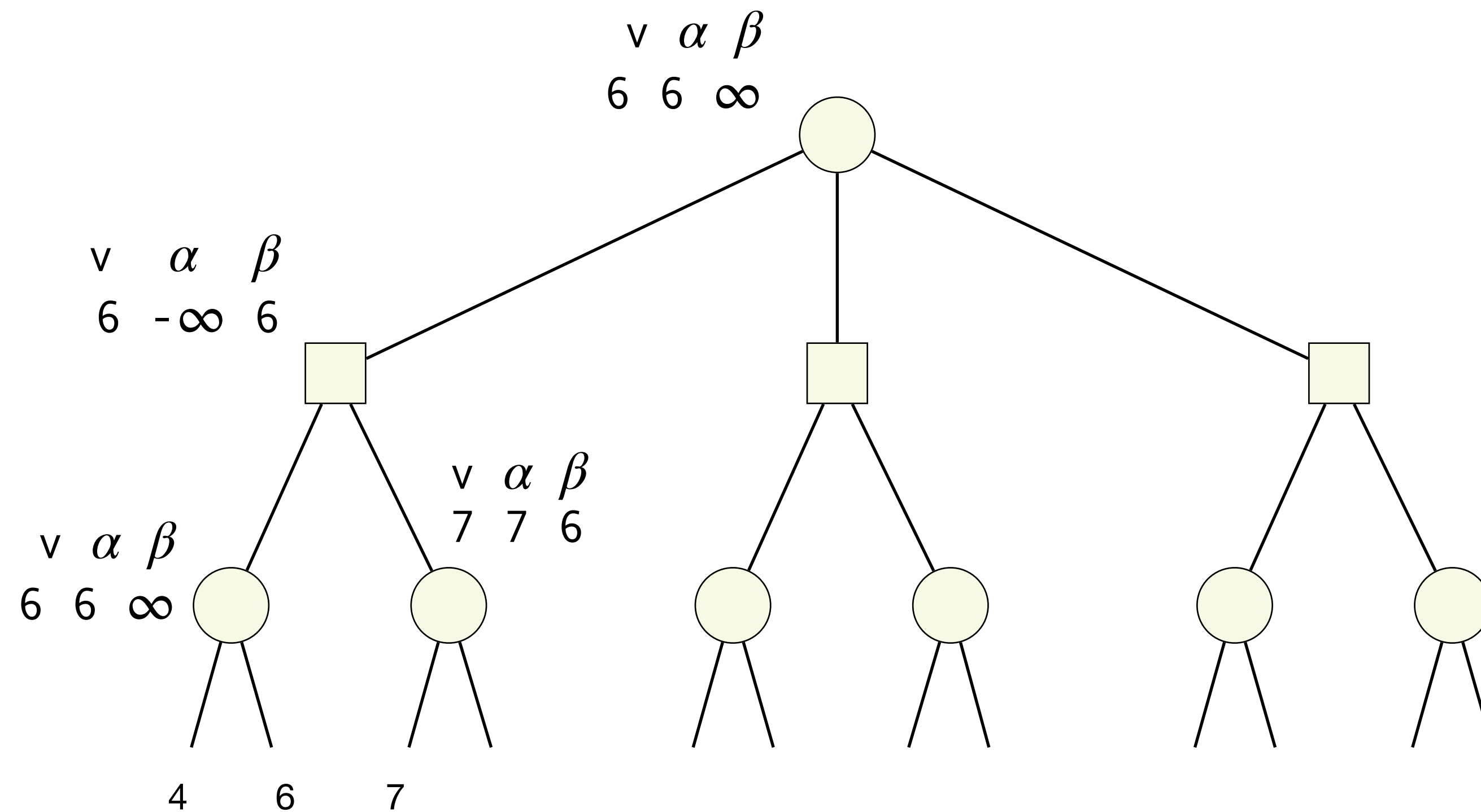
Alpha-Beta Search Example #2



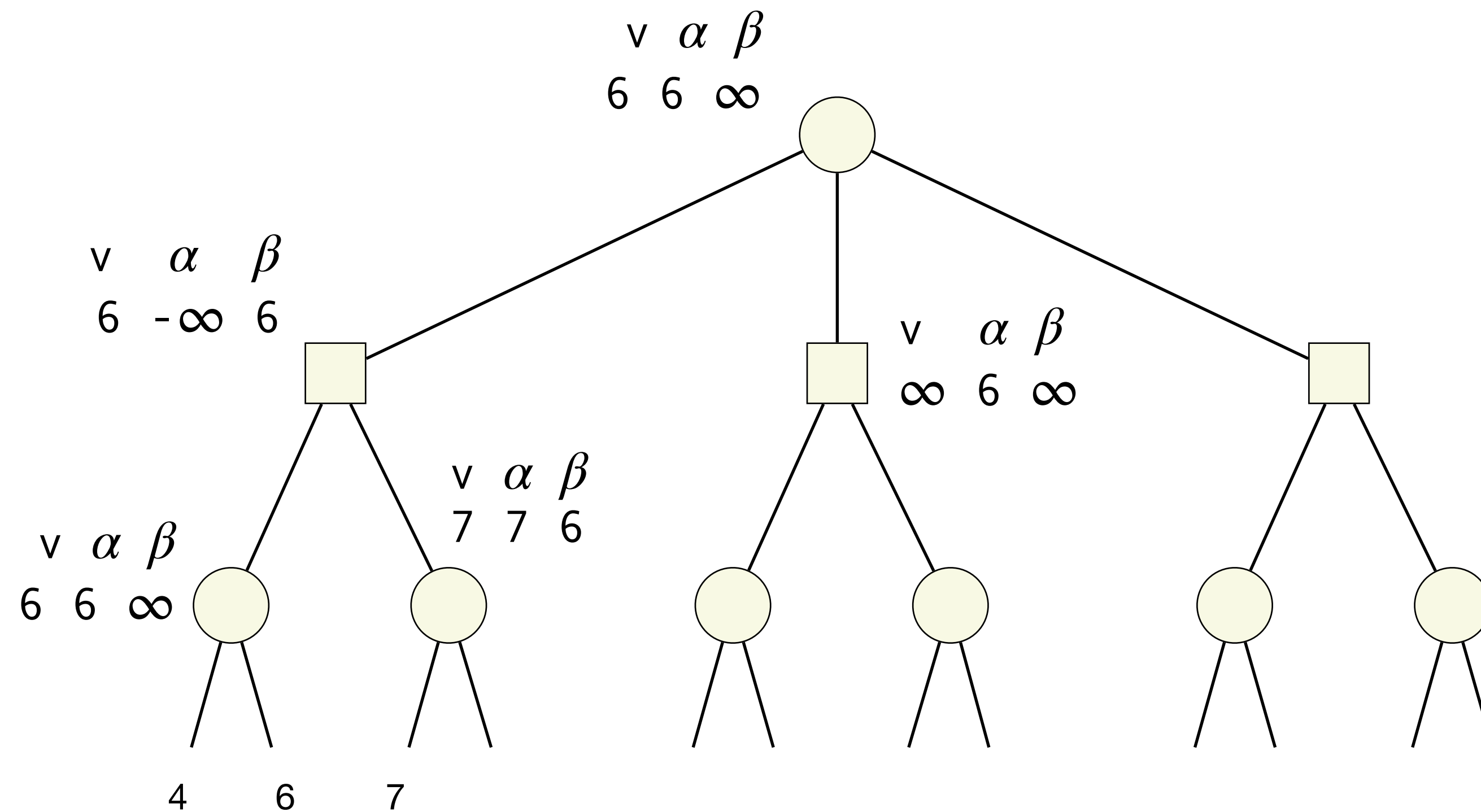
Alpha-Beta Search Example #2



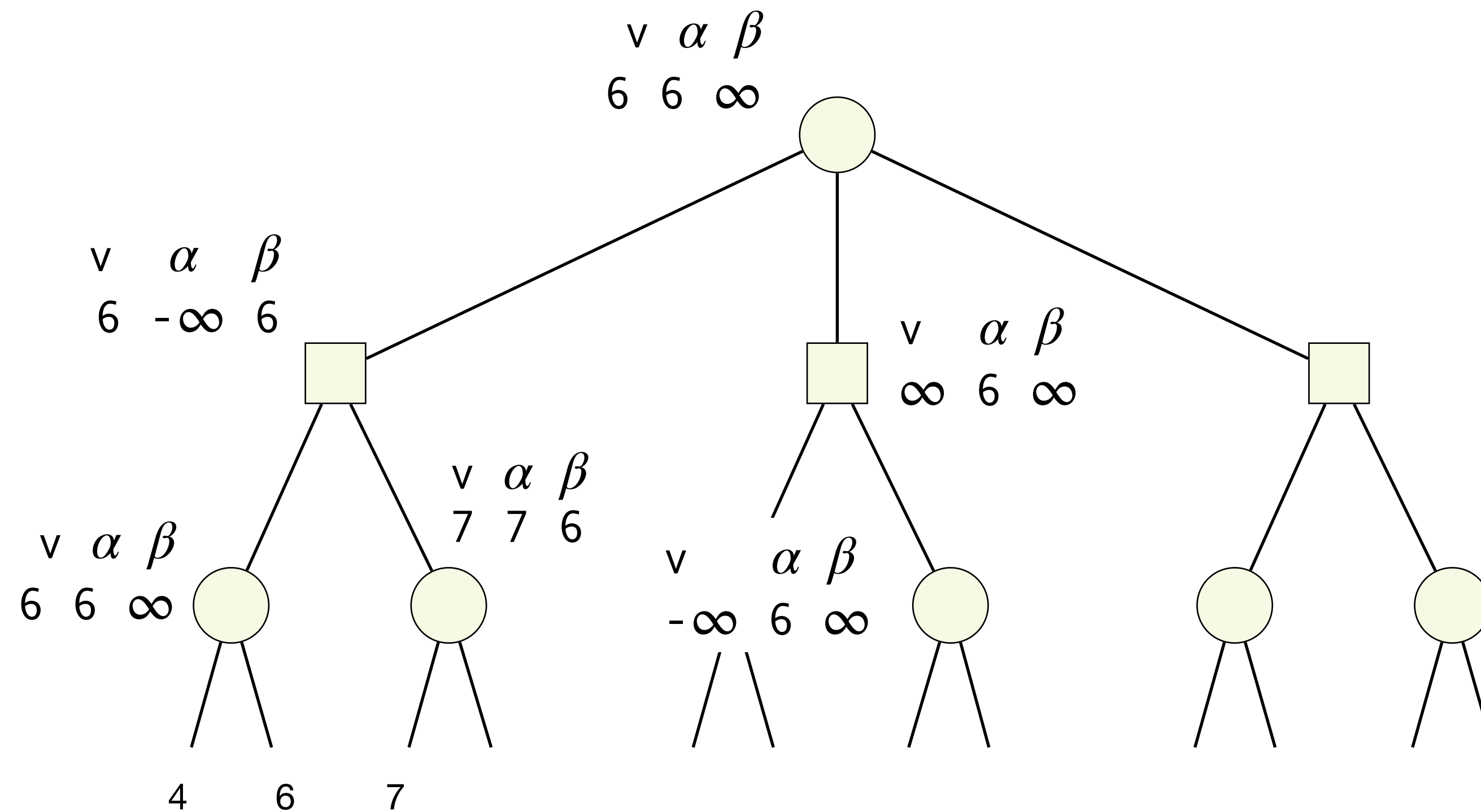
Alpha-Beta Search Example #2



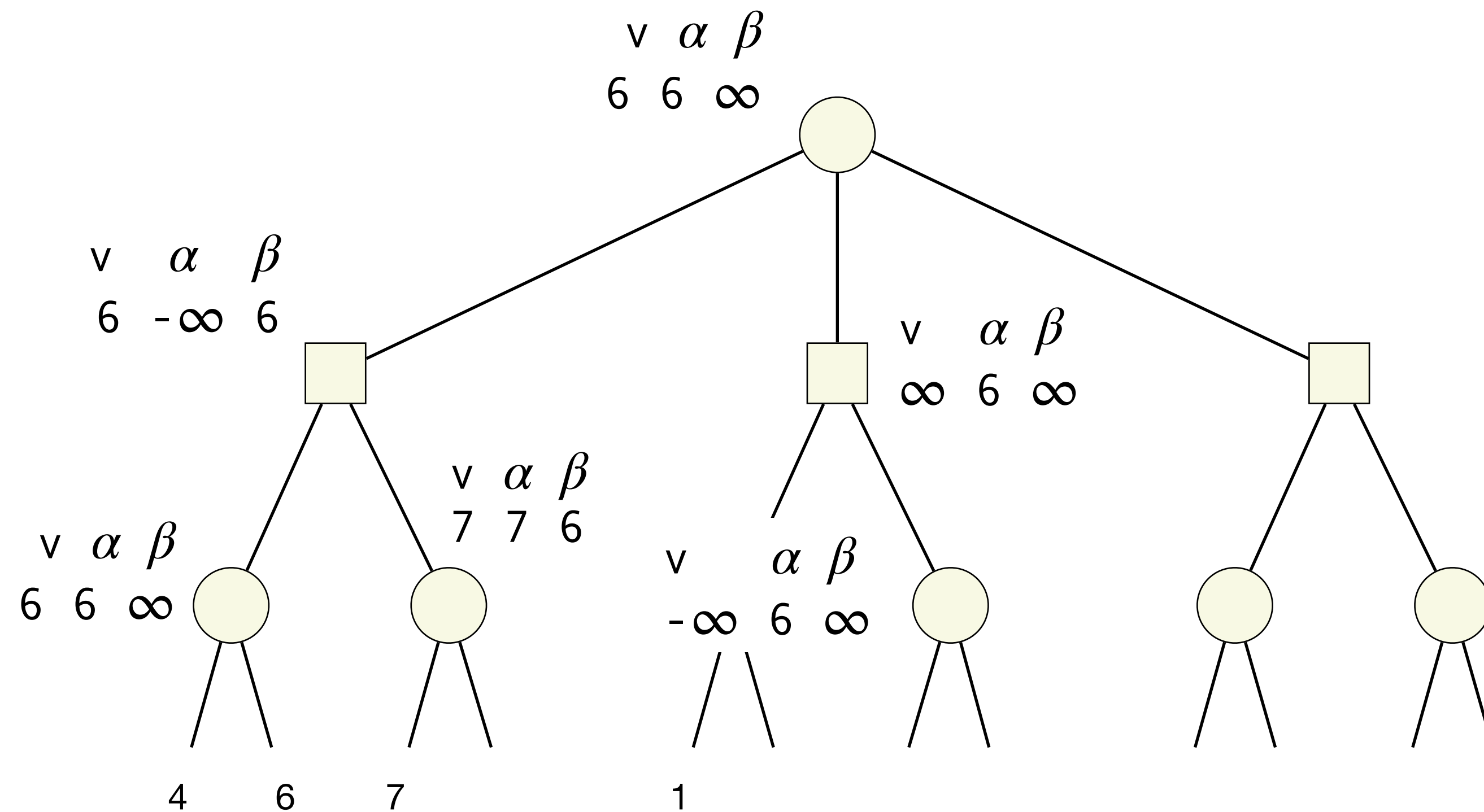
Alpha-Beta Search Example #2



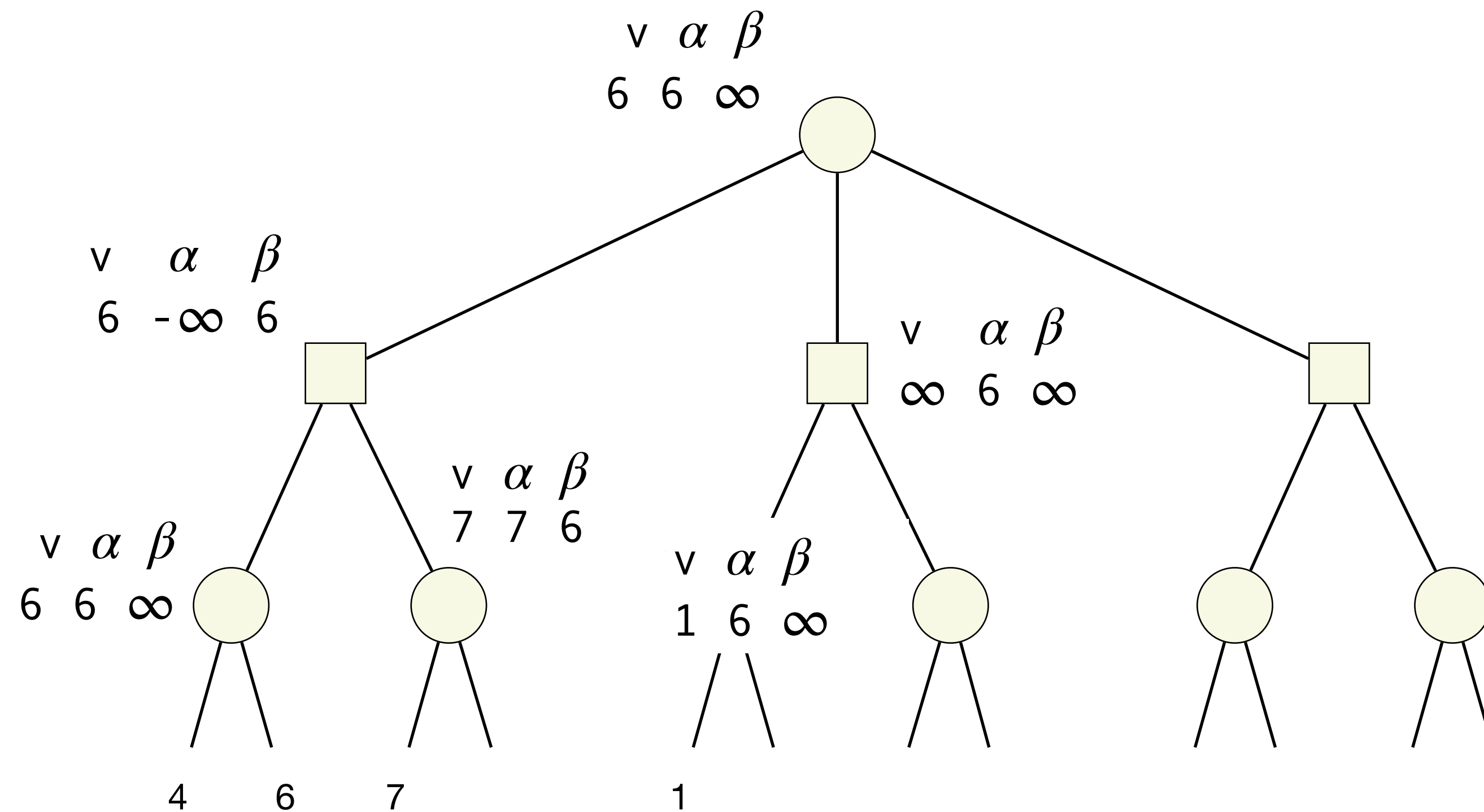
Alpha-Beta Search Example #2



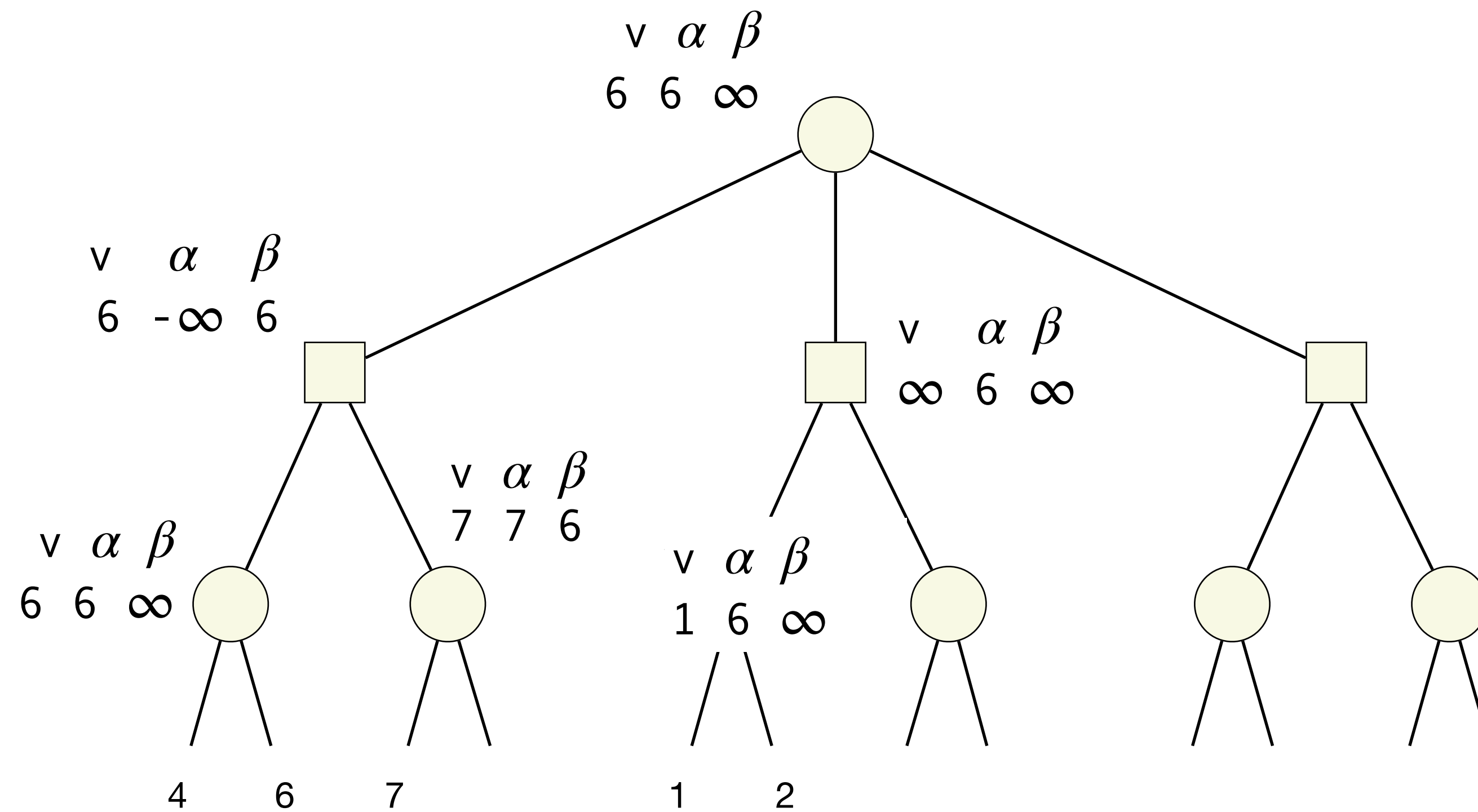
Alpha-Beta Search Example #2



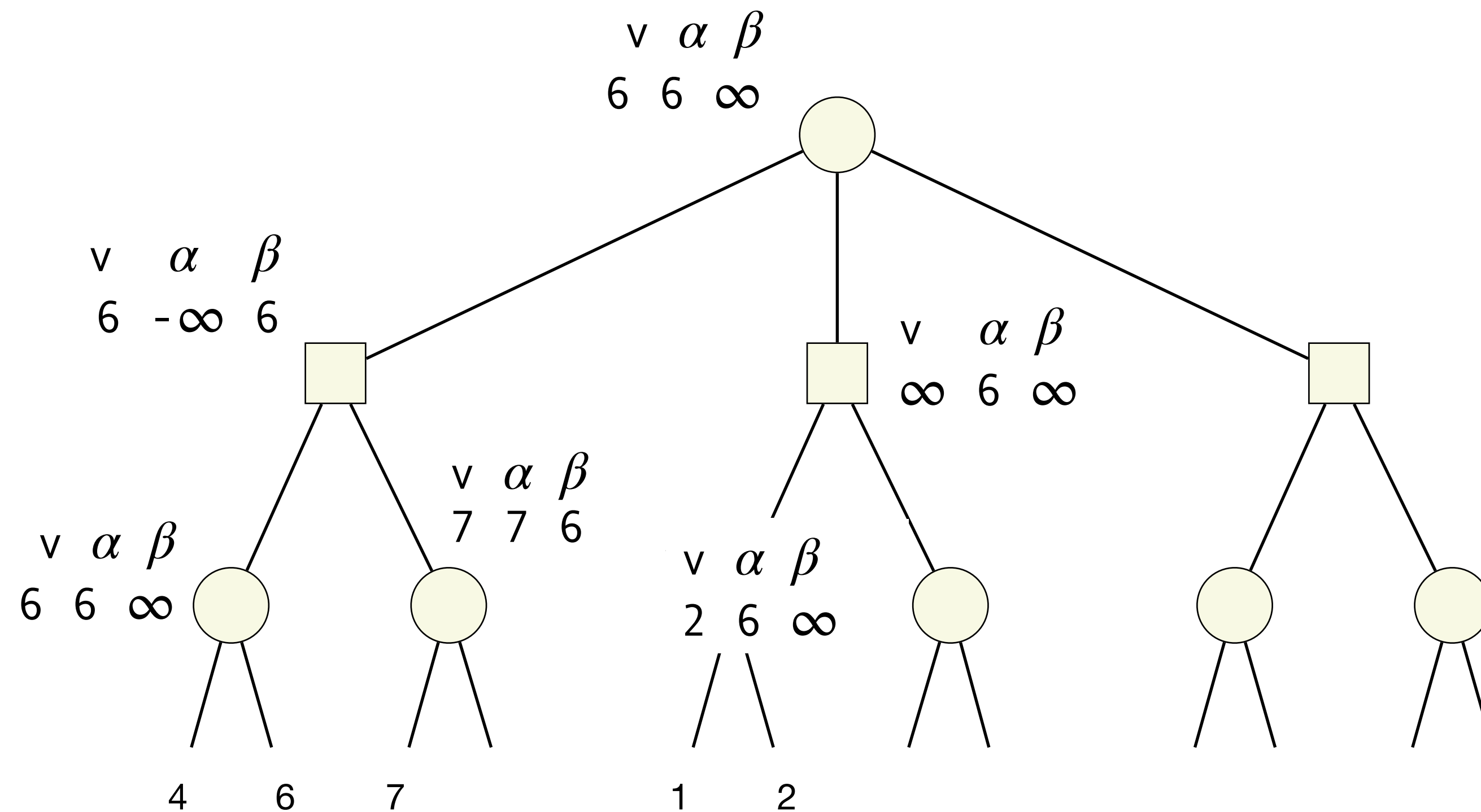
Alpha-Beta Search Example #2



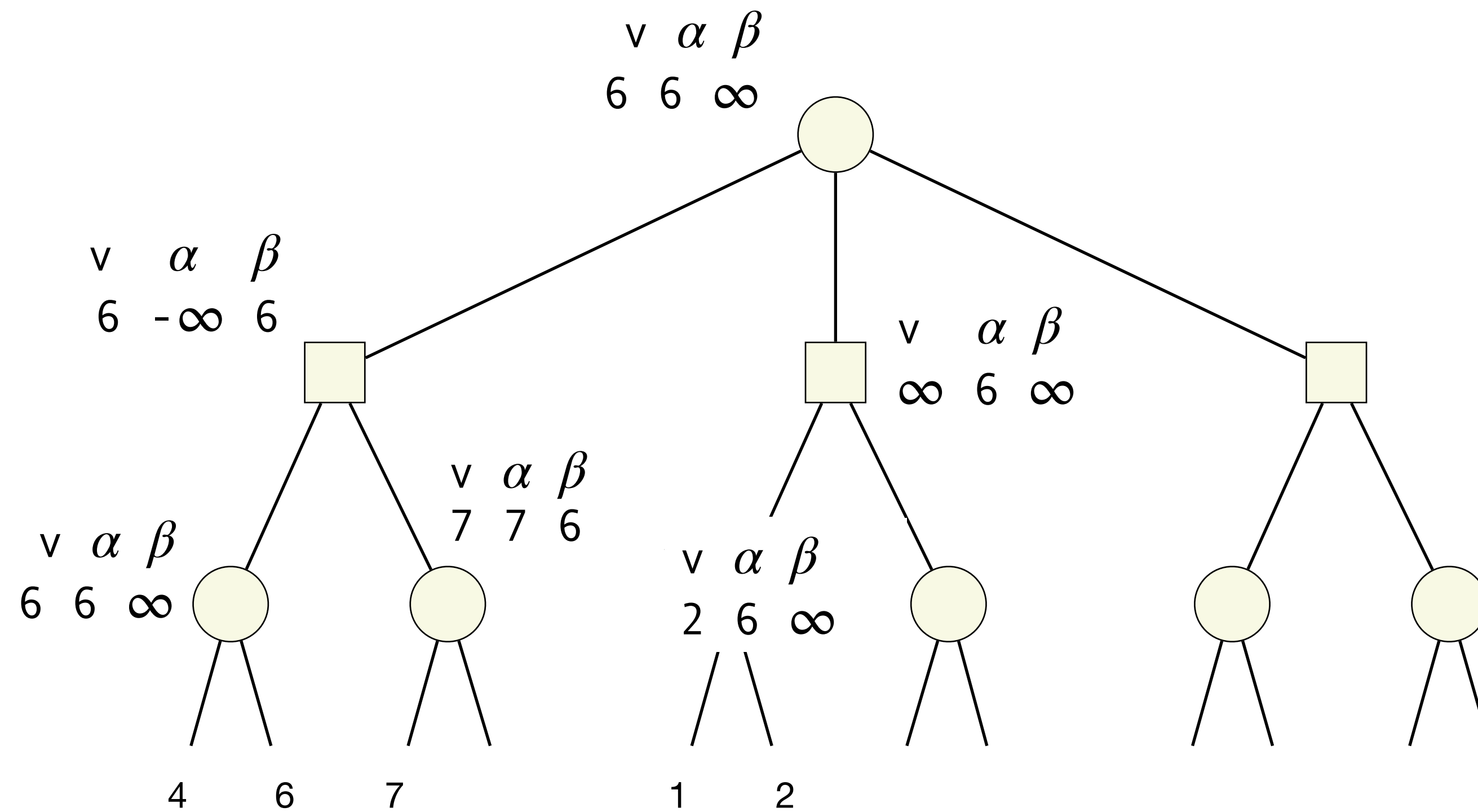
Alpha-Beta Search Example #2



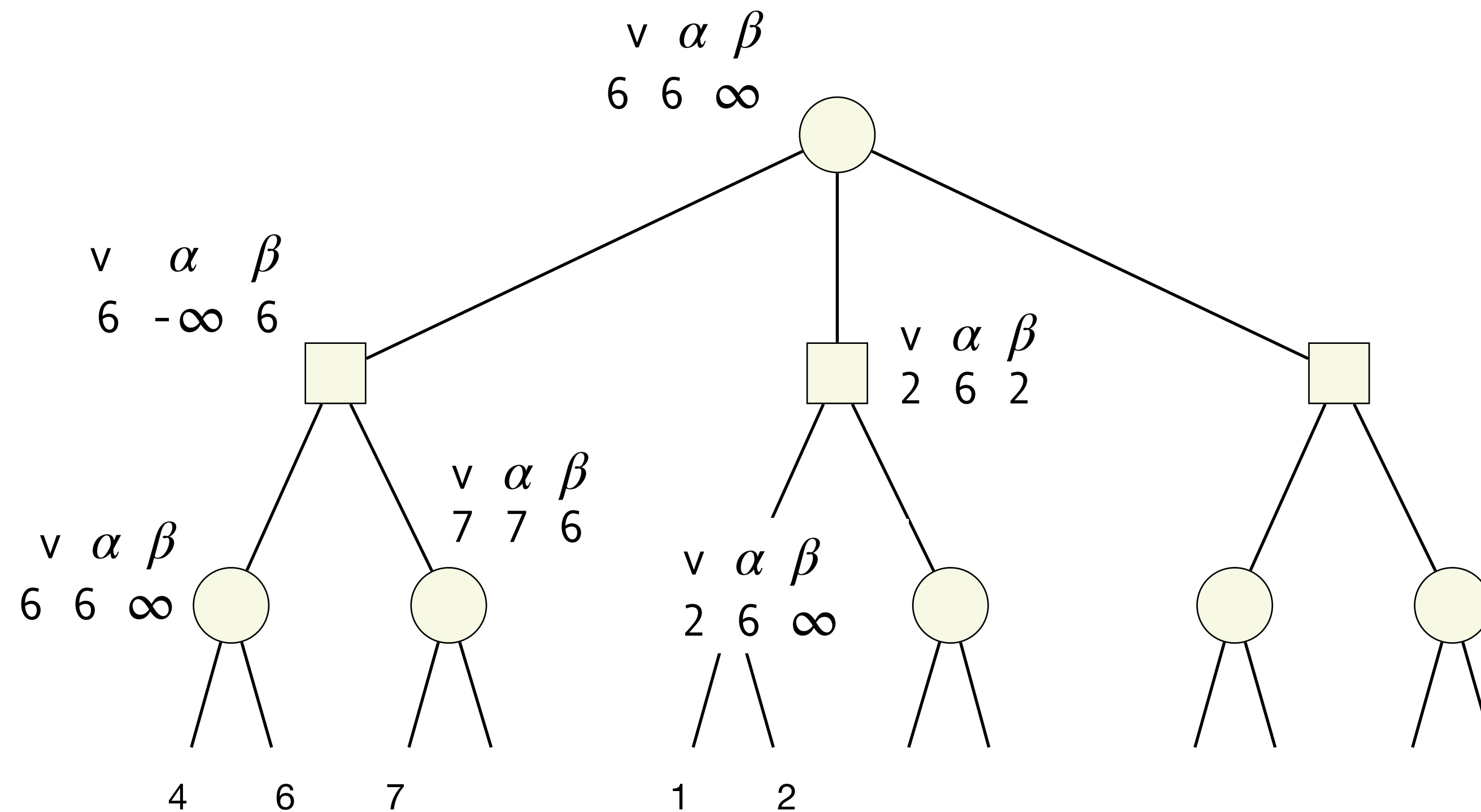
Alpha-Beta Search Example #2



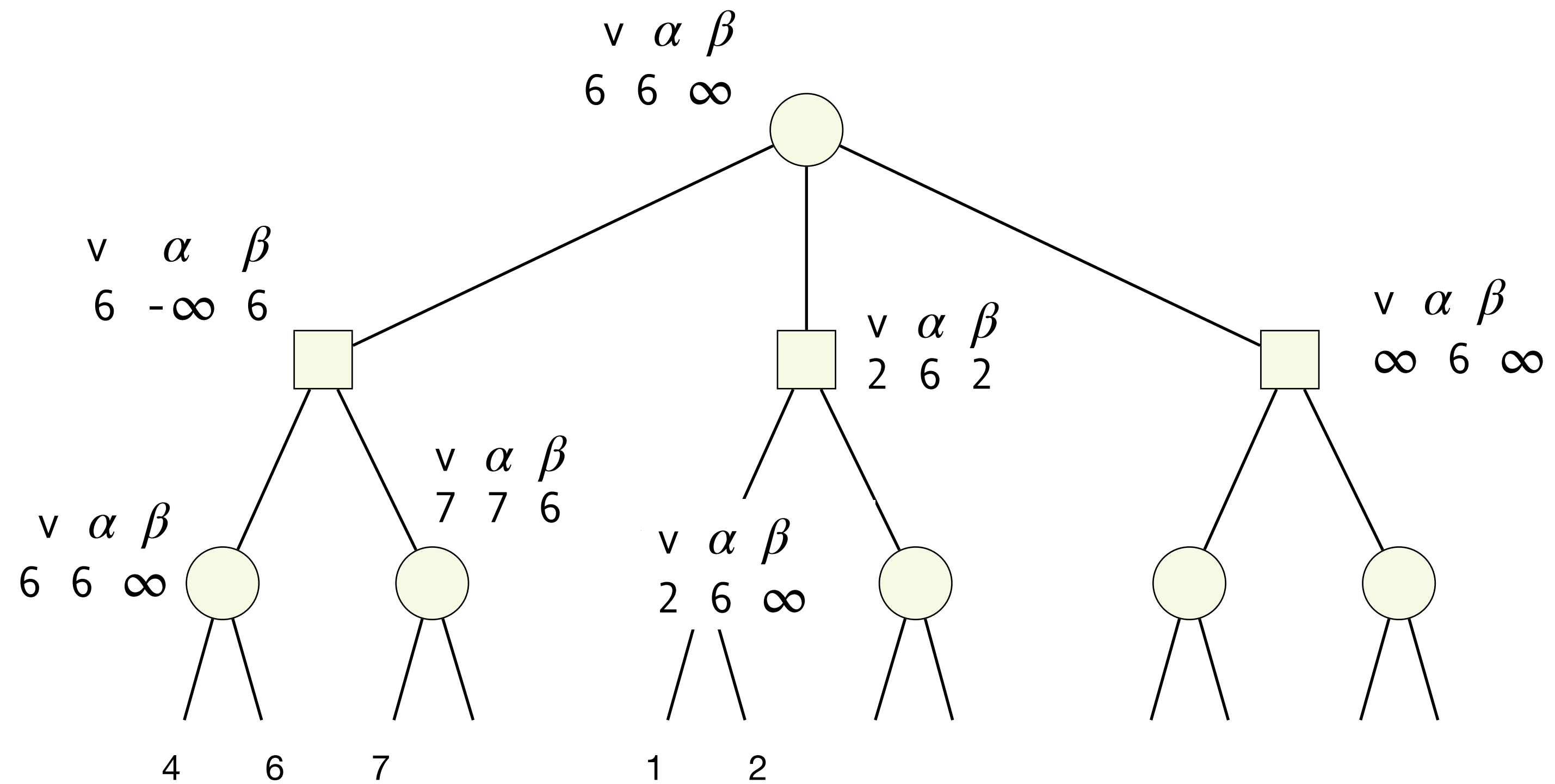
Alpha-Beta Search Example #2



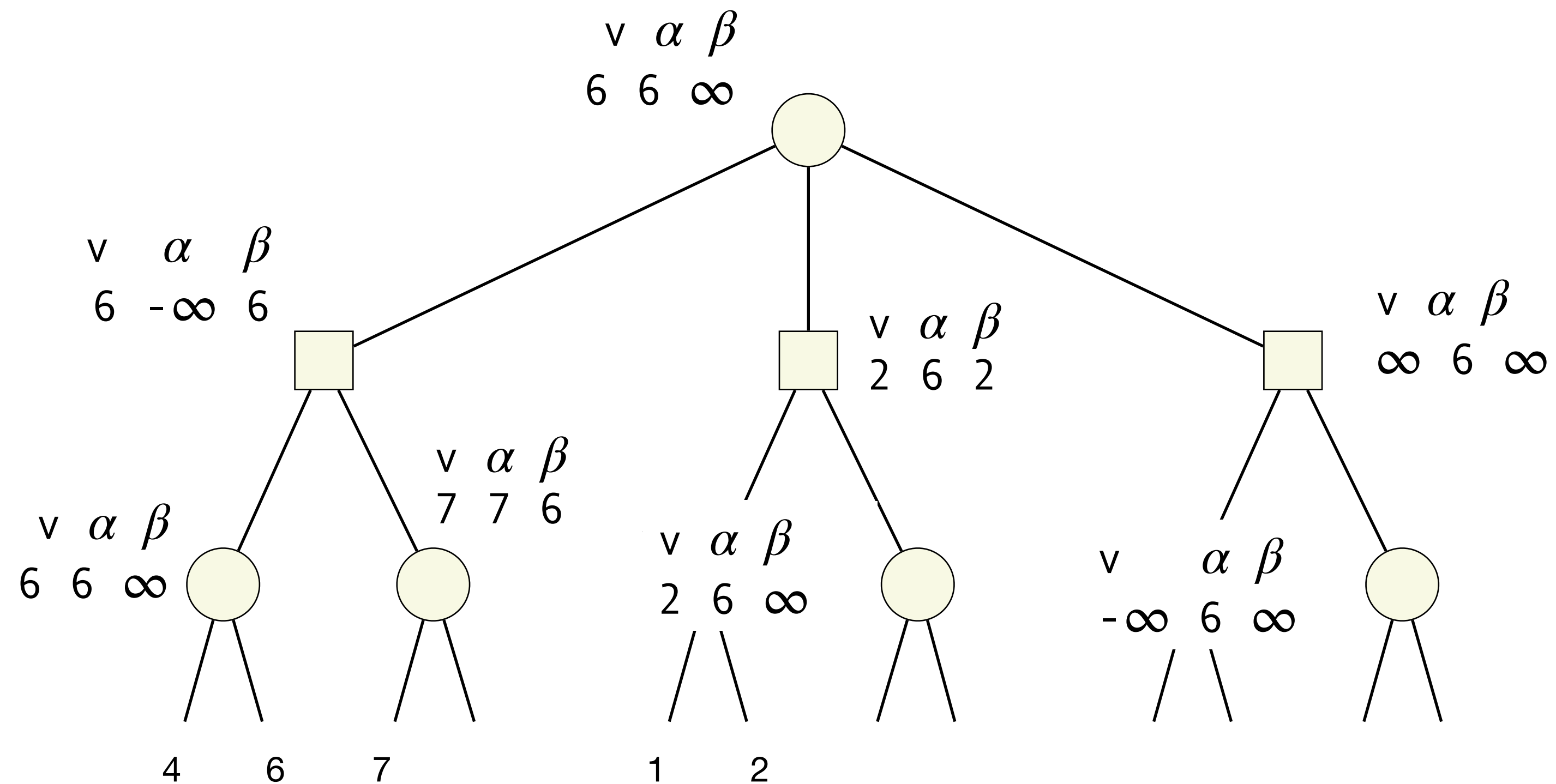
Alpha-Beta Search Example #2



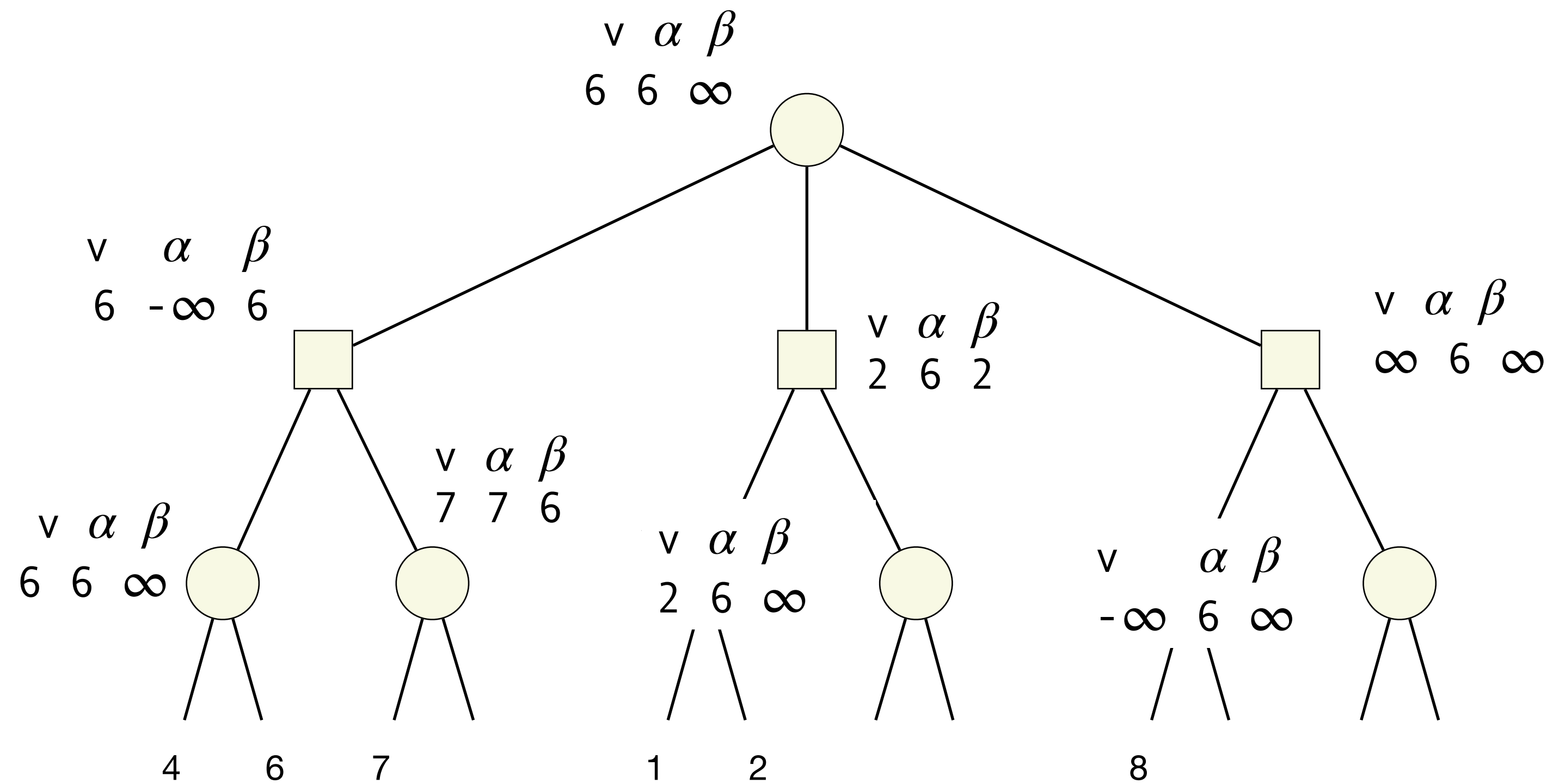
Alpha-Beta Search Example #2



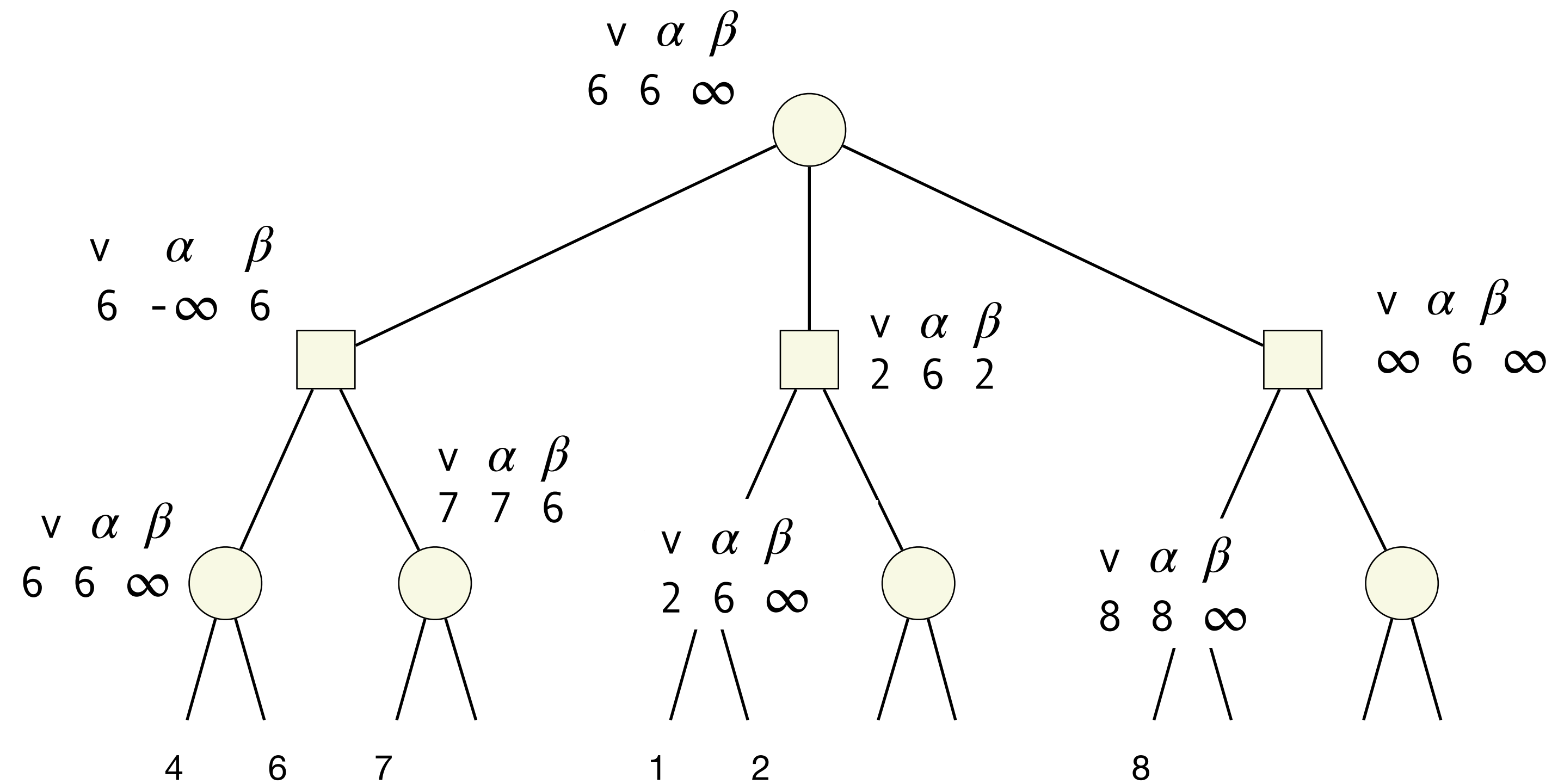
Alpha-Beta Search Example #2



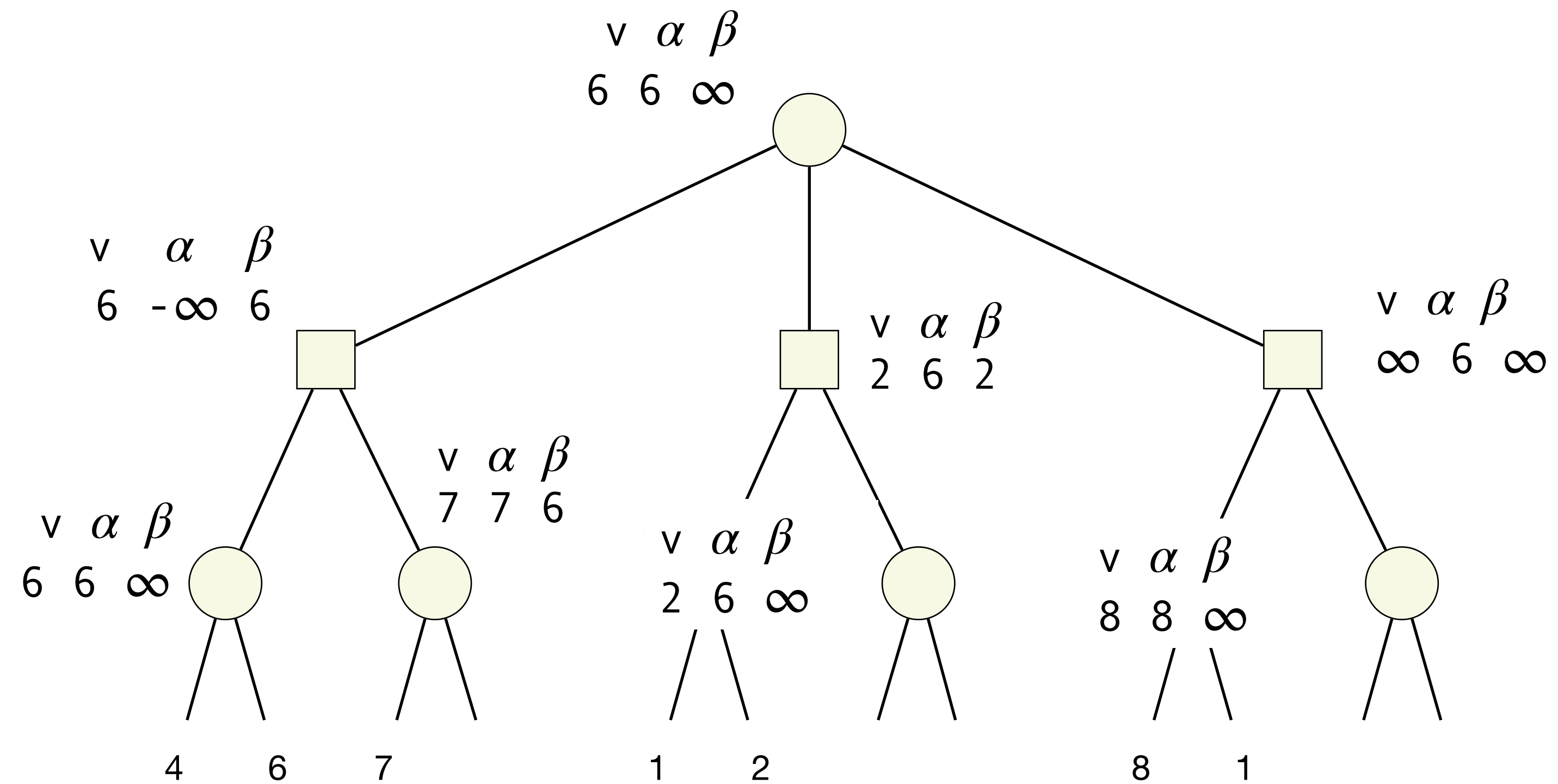
Alpha-Beta Search Example #2



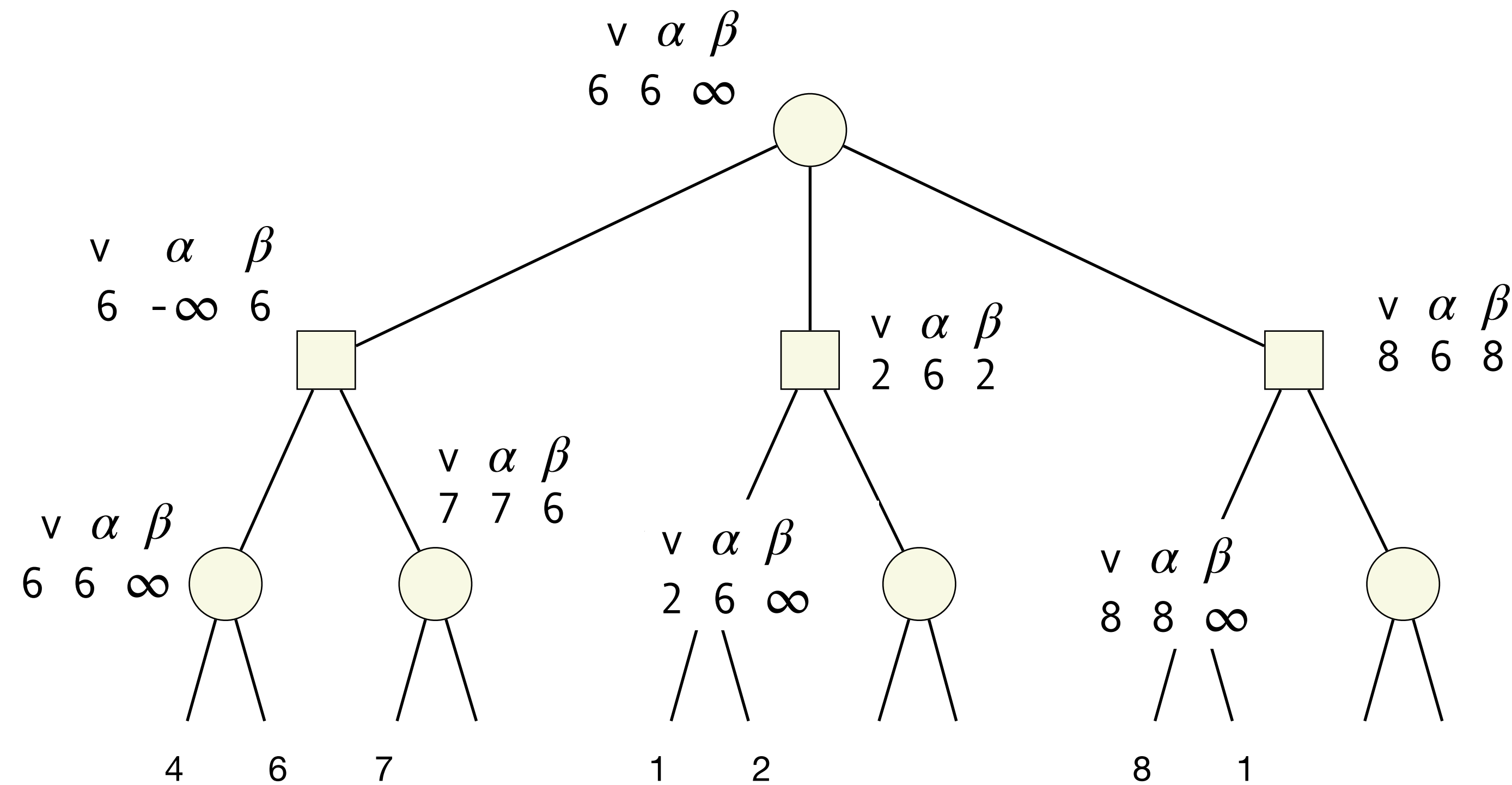
Alpha-Beta Search Example #2



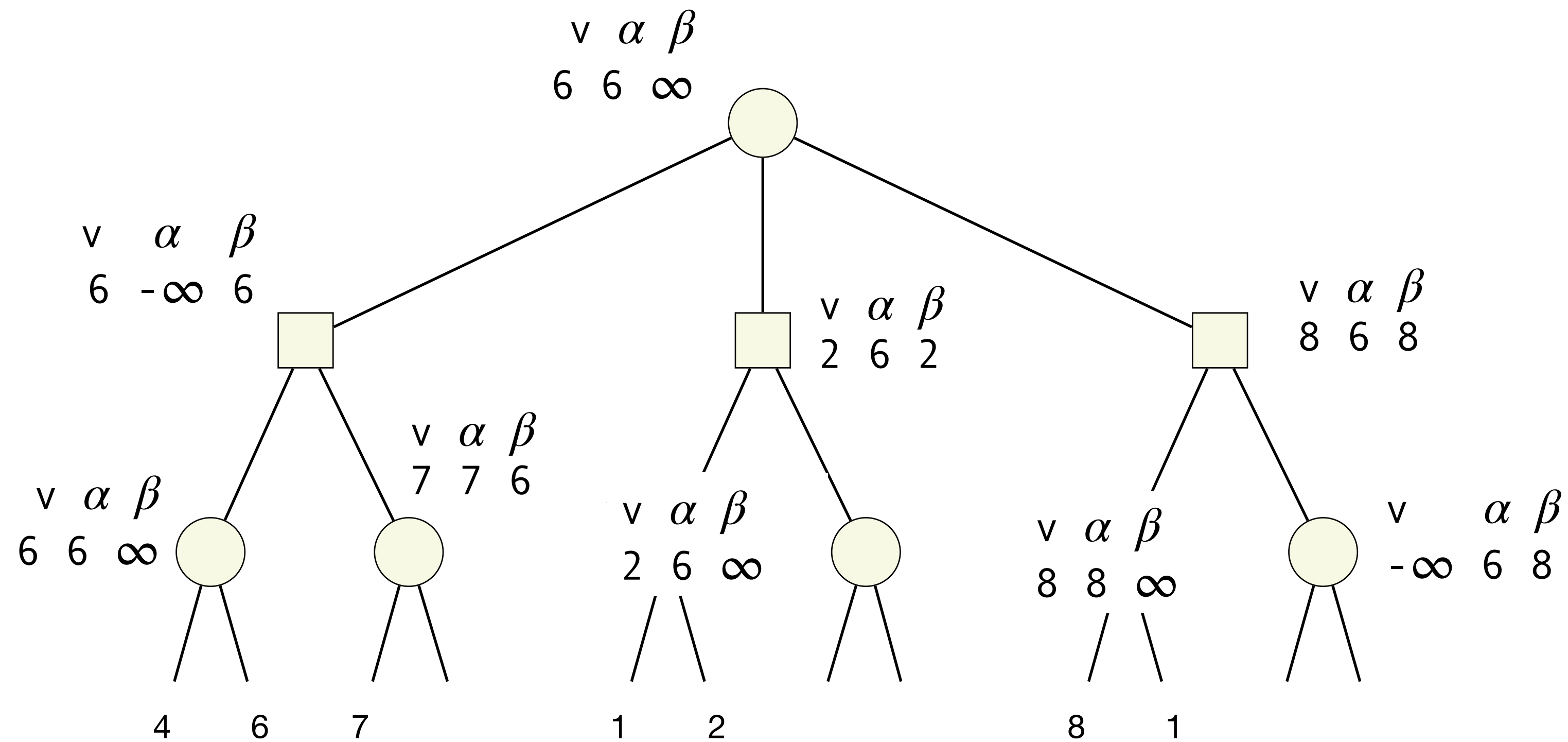
Alpha-Beta Search Example #2



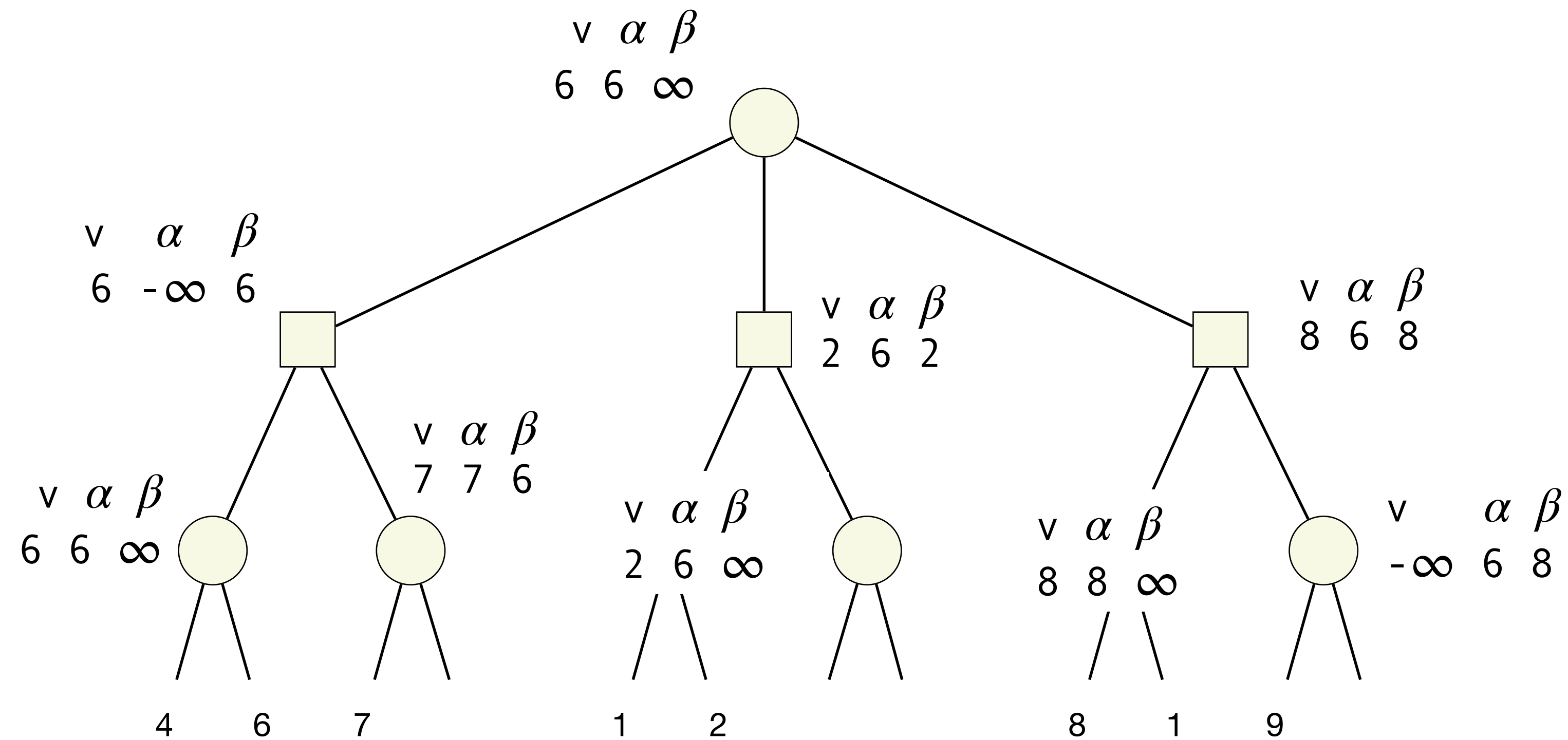
Alpha-Beta Search Example #2



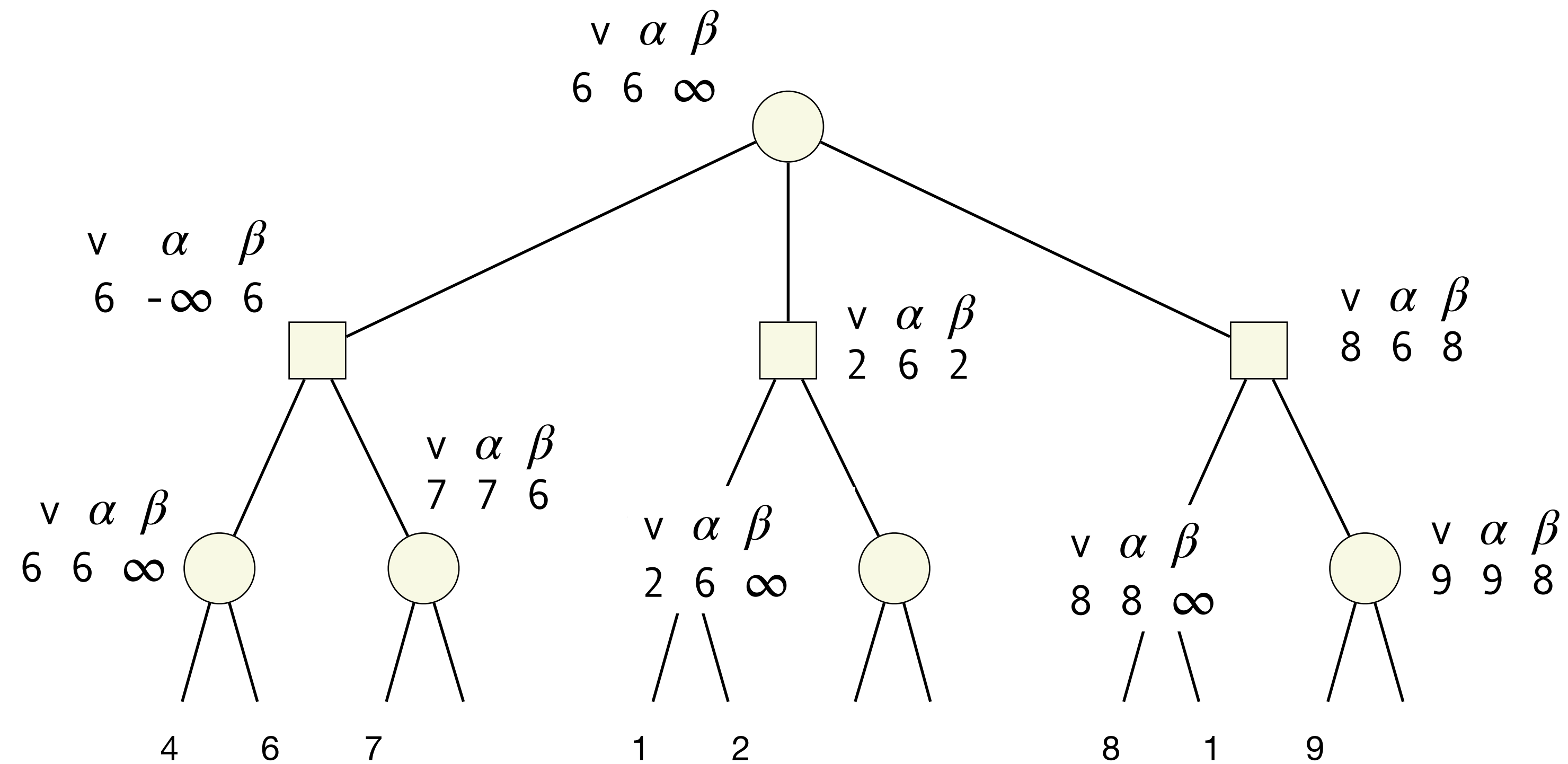
Alpha-Beta Search Example #2



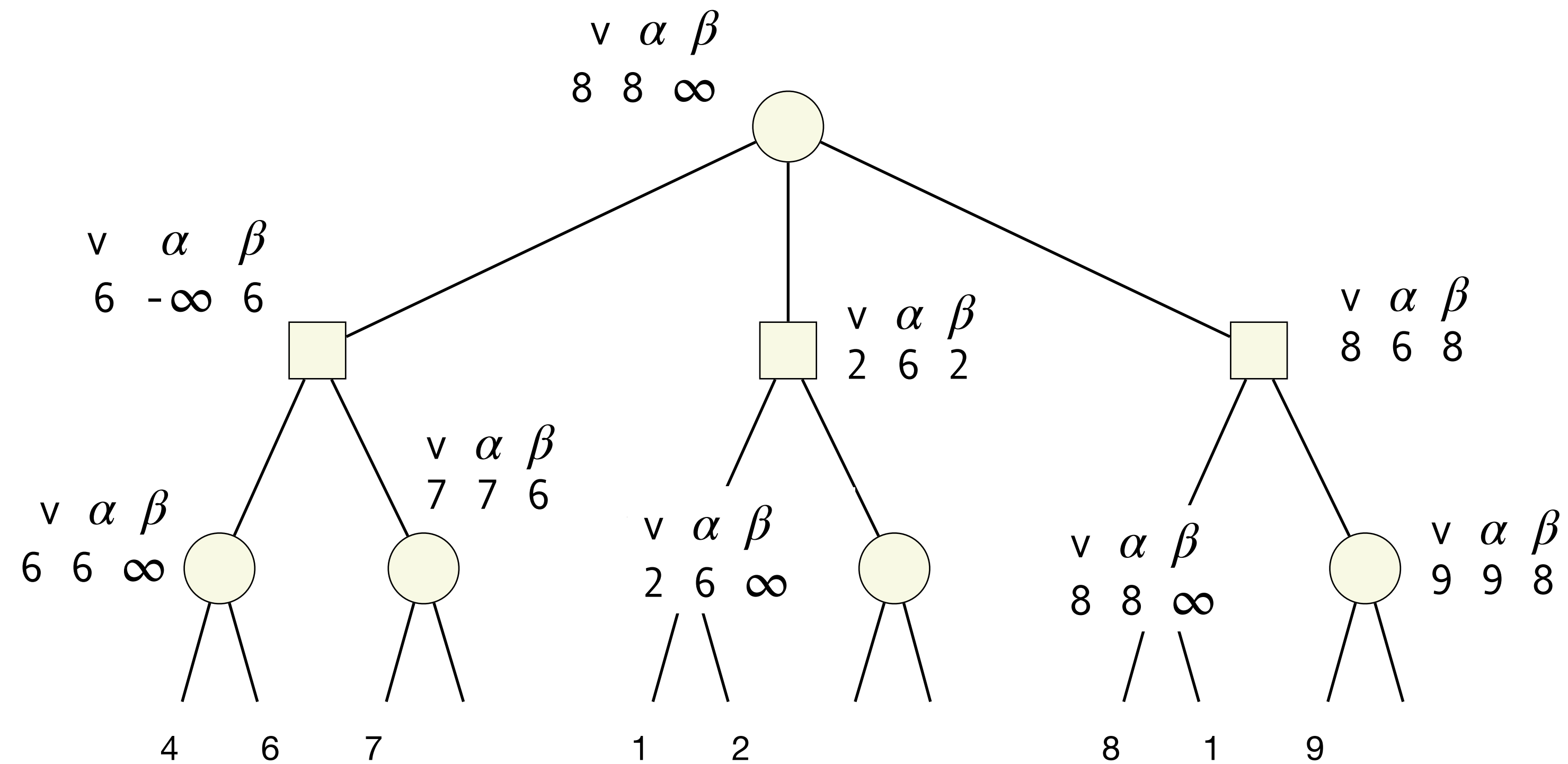
Alpha-Beta Search Example #2



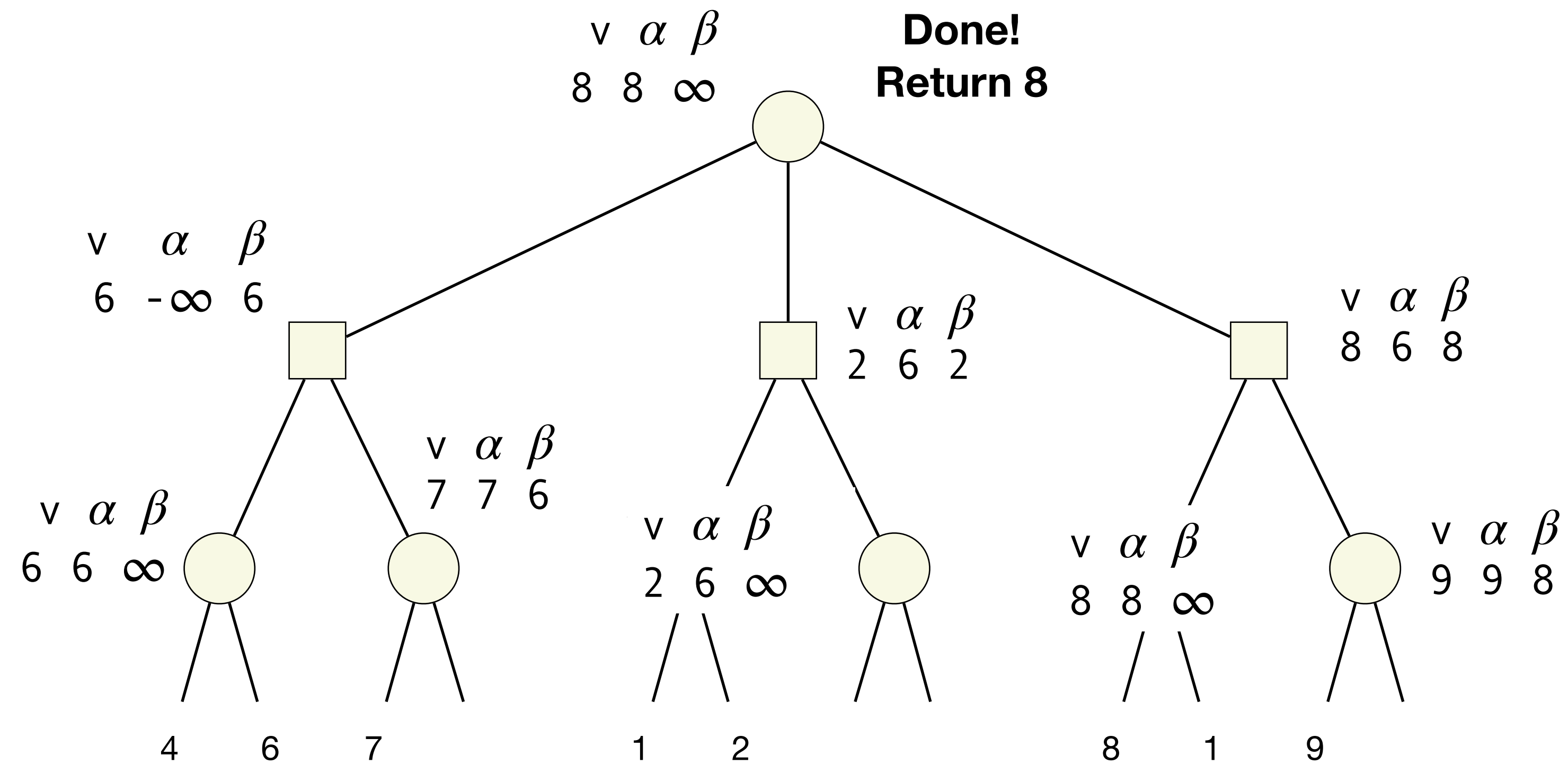
Alpha-Beta Search Example #2



Alpha-Beta Search Example #2



Alpha-Beta Search Example #2



Summary

- Minimax search examines every node in the search graph
- But when an ancestor **max** node has an option that is **higher** than the current subtree's value, we'll never reach this subtree (in optimal play)
- Similarly, when an ancestor **min** node has an option that has **lower** than the current subtree's value, then we'll never reach this subtree
- So, can stop exploring subtree once you prove that it has:
 - **lower value** than **ancestor max** node's best option, *or*
 - **higher value** than **ancestor min** node's best option
- **Alpha-beta search** checks before each recursive call whether subtree is optimally reachable
 - **Alpha: max value** option for any **ancestor max** node (possibly including self)
 - **Beta: min value** option for any **ancestor min** node (possibly including self)