



LECTURE 18
**Speculative Parallelism &
Computer Chess**

Charles E. Leiserson

November 17, 2022



THE HOME STRETCH

6.106 Endgame

Tuesday	Thursday	Friday
	November 17 Speculative parallelism & computer chess	November 18 Recitation
November 22 Structured and unstructured data	November 24 Thanksgiving (no class)	November 25 Thanksgiving (no class)
November 29 What compilers can and cannot do	December 1 Graphics processing units	December 2 Recitation
December 6 Jon Bentley!	December 8 Quiz 2	December 9 Recitation
December 13 Student presentations	December 15 Leiserchess exhibition tournament	

SPECULATIVE PARALLELISM



Searching an Unsorted Set

```
bool find(int *S, int n, int key) {  
    // S[] is an unsorted set of ints  
    assert(n > 0);  
  
    bool found = false;  
    for (int i = 0; i < n; ++i) {  
        if (S[i] == key) found = true;  
    }  
    return found;  
}
```

Short-Circuiting

Short-circuit optimization (Bentley rule)

- Quit the loop early if you find the element.

```
bool find(int *S, int n, int key) {  
    // S[] is an unsorted set of ints  
    assert(n > 0);  
  
    for (int i = 0; i < n; ++i) {  
        if (S[i] == key) return true;  
    }  
    return false;  
}
```

Question

- How can we parallelize this loop?

Short-Circuit Using D&C

```
void p_find_helper(int *S, int n, int key, bool* found) {
    if (*found) return;
    if (n == 1) {
        if (S[n] == key) *found = true;
    } else cilk_scope {
        cilk_spawn p_find_helper(S, n/2, key, found);
        if (*found) return;
        p_find_helper(S + n/2, n - n/2, key, found);
    }
    return;
}



bool p_find(int *S, int n, int key) {
    bool found = false;
    p_find_helper(S, n, key, &found);
    return found;
}
```

Short-Circuit Using D&C

```
void p_find_helper(int *S, int n, int key, bool* found) {
    if (*found) return;
    if (n == 1) {
        if (S[n] == key) *found = true;
    } else cilk_scope {
        cilk_spawn p_find_helper(S, n/2, key, found);
        if (*found) return;
        p_find_helper(S + n/2, n/2, key, found);
    }
    return;
}

bool p_find(int *S, int n, int key) {
    bool found = false;
    p_find_helper(S, n, key, &found);
    return found;
}
```

Notes

-  Nondeterministic code! 
- The benign race on `*found` is dangerous and technically undefined behavior in C.
 - Should use `int* found` (atomic).
- Is a memory fence necessary?
- Cilksan:
 - `__cilksan_disable_checking()`
 - `__cilksan_enable_checking()`

Speculative Parallelism

Definition

Speculative parallelism occurs when a program spawns a parallel task that would not be performed by the serial projection.

Rule of Thumb

Don't spawn speculative work unless there is little other opportunity for parallelism and there's a good chance you need it.



ALPHA-BETA SEARCH

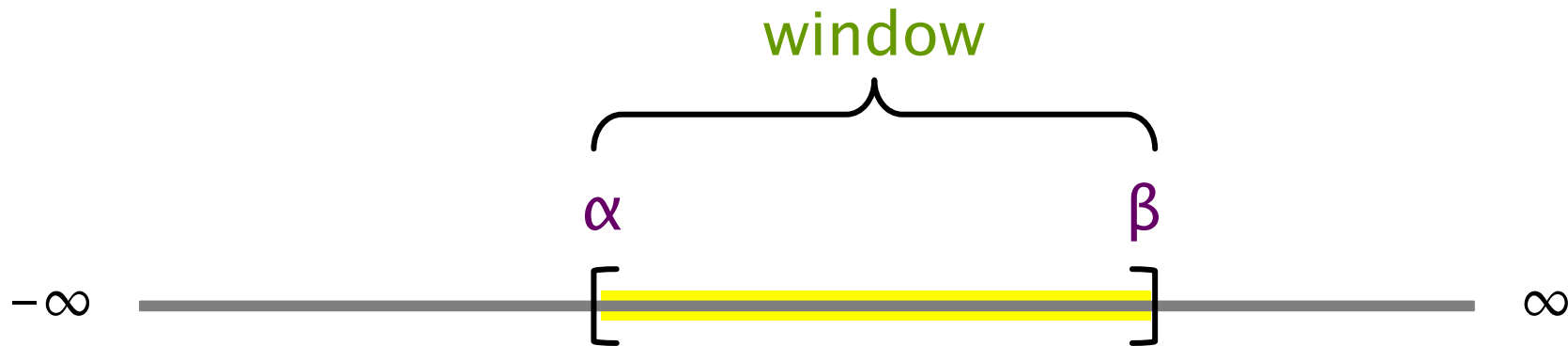


Alpha-Beta

```
int search(pos x, int alpha, int beta, int d) {
    if (d == 0 || is_leaf(x)) return static_eval(x);
    pos c[MAX_CHILDREN];
    int nc;
    gen_moves(x, c, &nc); // generate children
    for (int i = 0; i < nc; ++i) {
        int s = -search(c[i], -beta, -alpha, d-1); // negamax
        if (s > alpha) alpha = s;
        if (alpha >= beta) return alpha; // beta cutoff
    }
    return alpha;
}
```

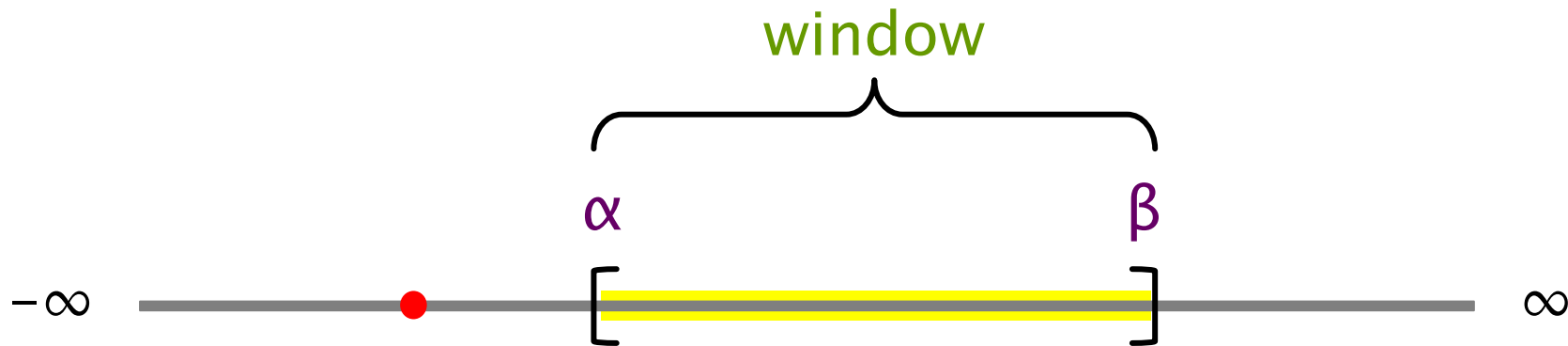
Alpha-Beta

```
int search(pos x, int alpha, int beta, int d) {
    if (d == 0 || is_leaf(x)) return static_eval(x);
    pos c[MAX_CHILDREN];
    int nc;
    gen_moves(x, c, &nc); // generate children
    for (int i = 0; i < nc; ++i) {
        int s = -search(c[i], -beta, -alpha, d-1); // negamax
        if (s > alpha) alpha = s;
        if (alpha >= beta) return alpha; // beta cutoff
    }
    return alpha;
}
```



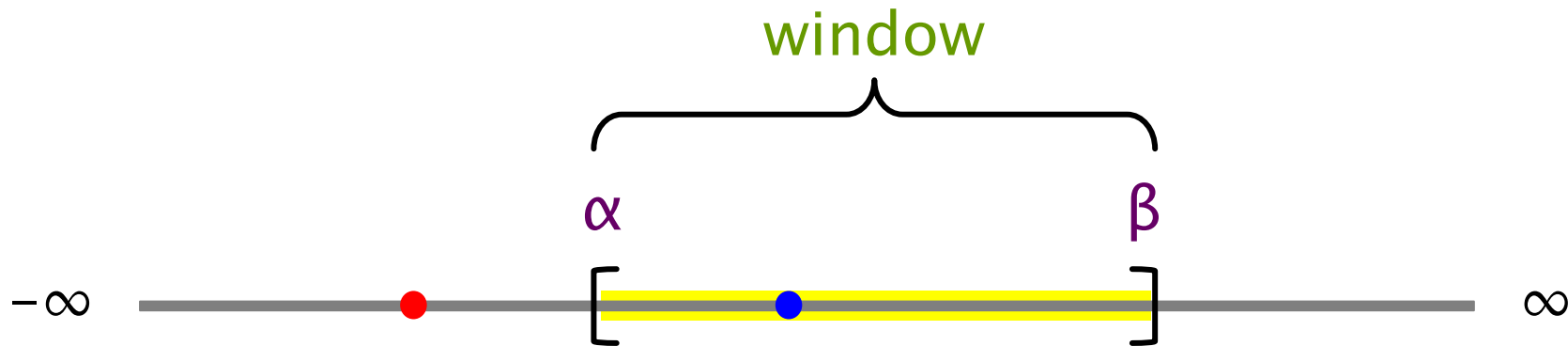
Alpha-Beta

```
int search(pos x, int alpha, int beta, int d) {
    if (d == 0 || is_leaf(x)) return static_eval(x);
    pos c[MAX_CHILDREN];
    int nc;
    gen_moves(x, c, &nc);           // generate children
    for (int i = 0; i < nc; ++i) {
        int s = -search(c[i], -beta, -alpha, d-1); // negamax
        if (s > alpha) alpha = s;
        if (alpha >= beta) return alpha;         // beta cutoff
    }
    return alpha;
}
```



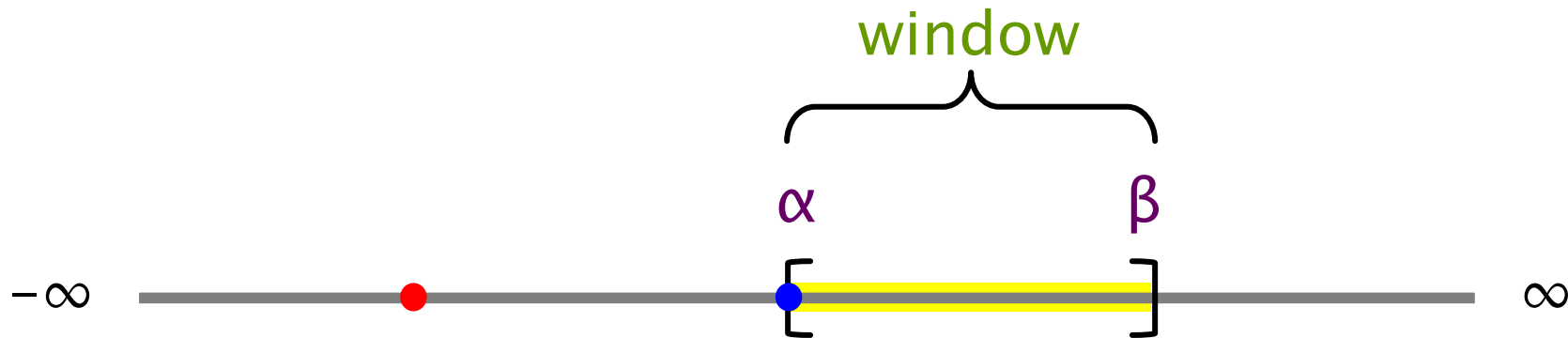
Alpha-Beta

```
int search(pos x, int alpha, int beta, int d) {
    if (d == 0 || is_leaf(x)) return static_eval(x);
    pos c[MAX_CHILDREN];
    int nc;
    gen_moves(x, c, &nc); // generate children
    for (int i = 0; i < nc; ++i) {
        int s = -search(c[i], -beta, -alpha, d-1); // negamax
        if (s > alpha) alpha = s;
        if (alpha >= beta) return alpha; // beta cutoff
    }
    return alpha;
}
```



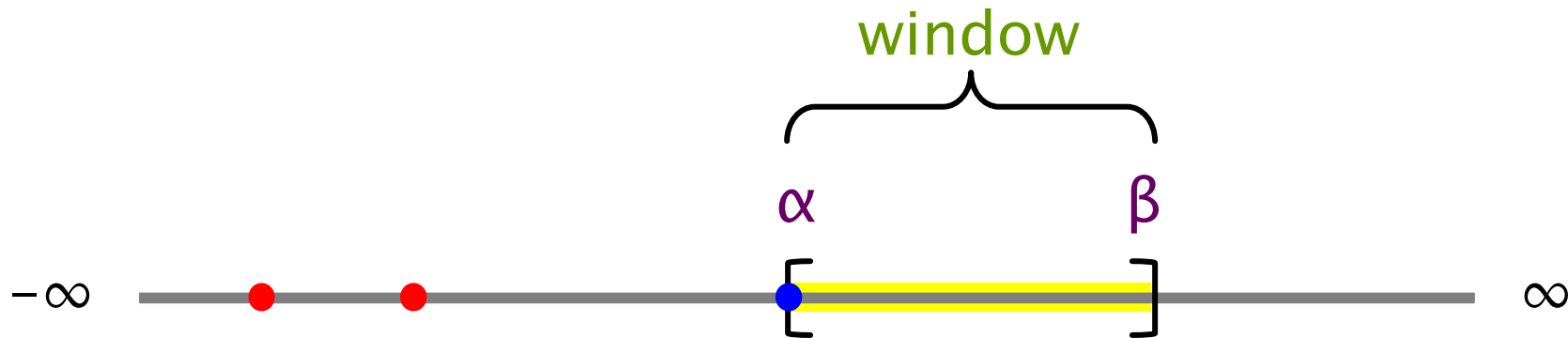
Alpha-Beta

```
int search(pos x, int alpha, int beta, int d) {
    if (d == 0 || is_leaf(x)) return static_eval(x);
    pos c[MAX_CHILDREN];
    int nc;
    gen_moves(x, c, &nc); // generate children
    for (int i = 0; i < nc; ++i) {
        int s = -search(c[i], -beta, -alpha, d-1); // negamax
        if (s > alpha) alpha = s;
        if (alpha >= beta) return alpha; // beta cutoff
    }
    return alpha;
}
```



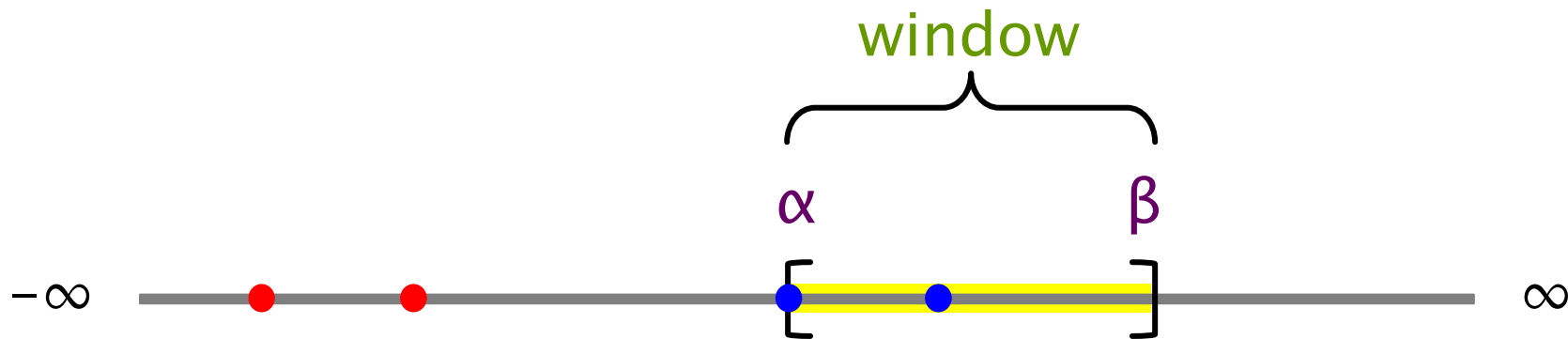
Alpha-Beta

```
int search(pos x, int alpha, int beta, int d) {
    if (d == 0 || is_leaf(x)) return static_eval(x);
    pos c[MAX_CHILDREN];
    int nc;
    gen_moves(x, c, &nc);           // generate children
    for (int i = 0; i < nc; ++i) {
        int s = -search(c[i], -beta, -alpha, d-1); // negamax
        if (s > alpha) alpha = s;
        if (alpha >= beta) return alpha;         // beta cutoff
    }
    return alpha;
}
```



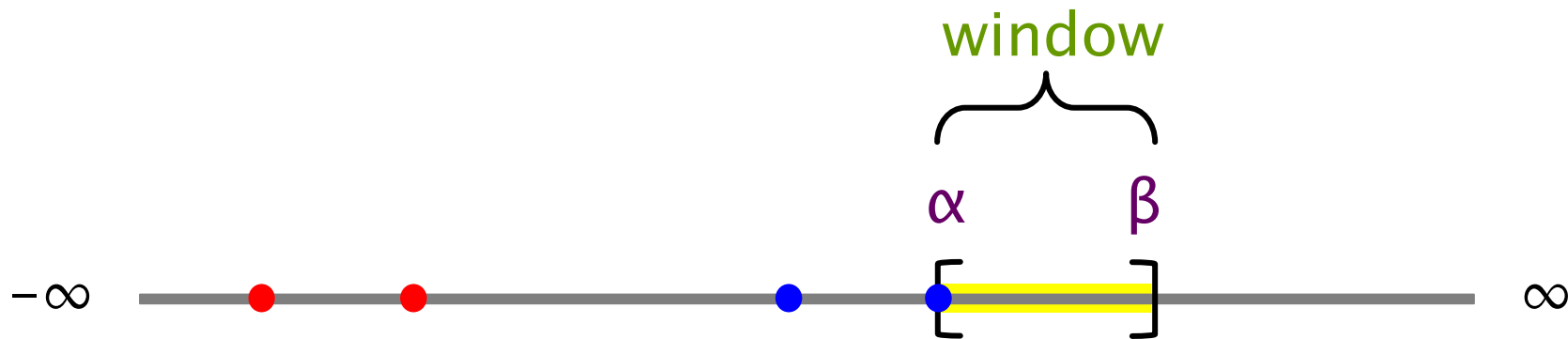
Alpha-Beta

```
int search(pos x, int alpha, int beta, int d) {
    if (d == 0 || is_leaf(x)) return static_eval(x);
    pos c[MAX_CHILDREN];
    int nc;
    gen_moves(x, c, &nc); // generate children
    for (int i = 0; i < nc; ++i) {
        int s = -search(c[i], -beta, -alpha, d-1); // negamax
        if (s > alpha) alpha = s;
        if (alpha >= beta) return alpha; // beta cutoff
    }
    return alpha;
}
```



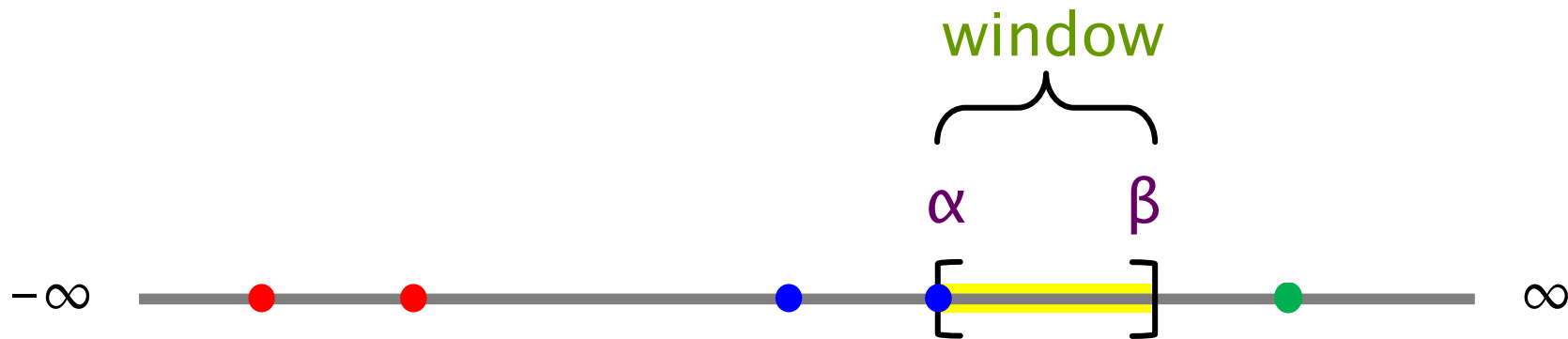
Alpha-Beta

```
int search(pos x, int alpha, int beta, int d) {
    if (d == 0 || is_leaf(x)) return static_eval(x);
    pos c[MAX_CHILDREN];
    int nc;
    gen_moves(x, c, &nc); // generate children
    for (int i = 0; i < nc; ++i) {
        int s = -search(c[i], -beta, -alpha, d-1); // negamax
        if (s > alpha) alpha = s;
        if (alpha >= beta) return alpha; // beta cutoff
    }
    return alpha;
}
```



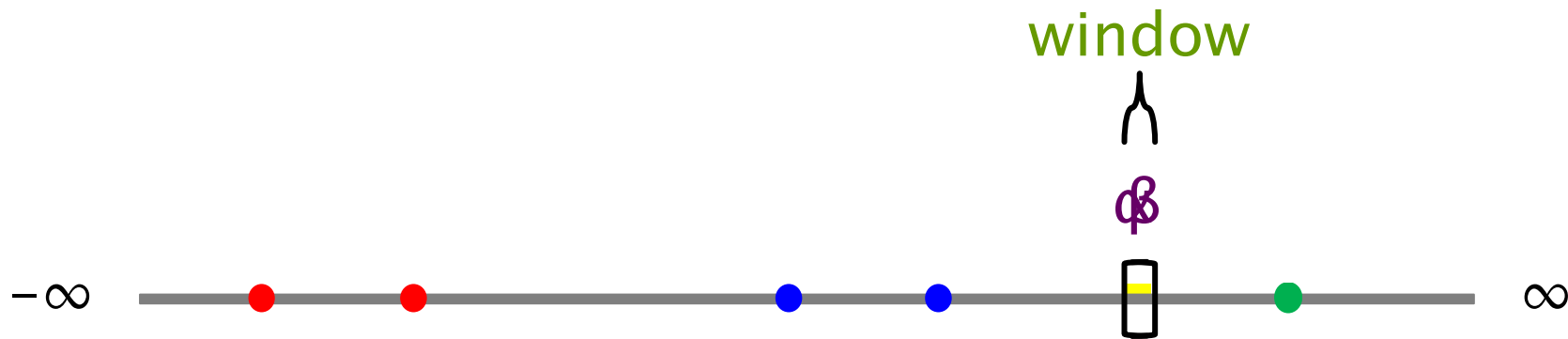
Alpha-Beta

```
int search(pos x, int alpha, int beta, int d) {
    if (d == 0 || is_leaf(x)) return static_eval(x);
    pos c[MAX_CHILDREN];
    int nc;
    gen_moves(x, c, &nc); // generate children
    for (int i = 0; i < nc; ++i) {
        int s = -search(c[i], -beta, -alpha, d-1); // negamax
        if (s > alpha) alpha = s;
        if (alpha >= beta) return alpha; // beta cutoff
    }
    return alpha;
}
```



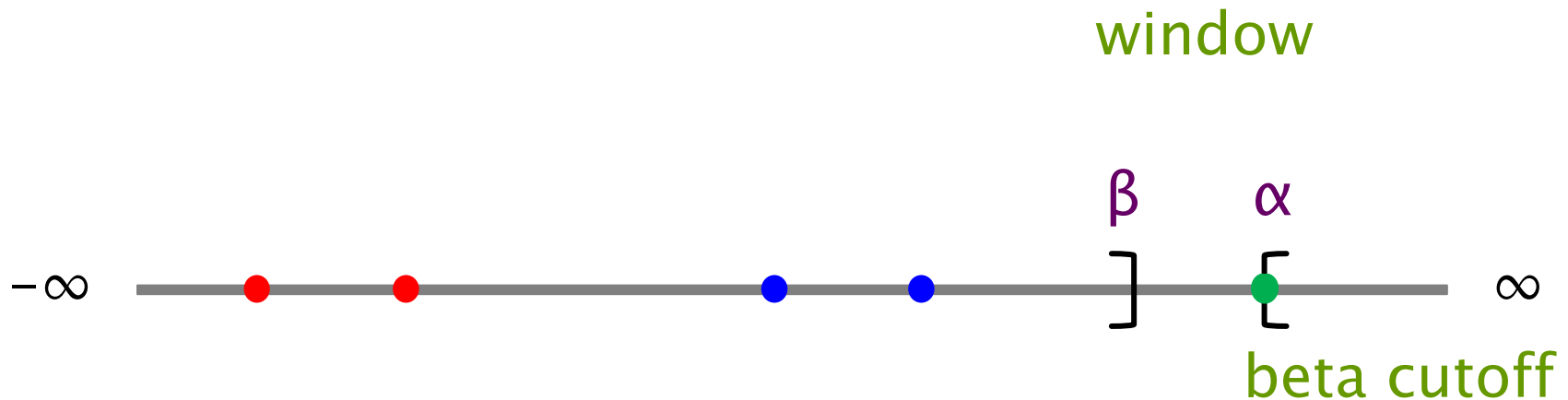
Alpha-Beta

```
int search(pos x, int alpha, int beta, int d) {
    if (d == 0 || is_leaf(x)) return static_eval(x);
    pos c[MAX_CHILDREN];
    int nc;
    gen_moves(x, c, &nc); // generate children
    for (int i = 0; i < nc; ++i) {
        int s = -search(c[i], -beta, -alpha, d-1); // negamax
        if (s > alpha) alpha = s;
        if (alpha >= beta) return alpha; // beta cutoff
    }
    return alpha;
}
```



Alpha-Beta

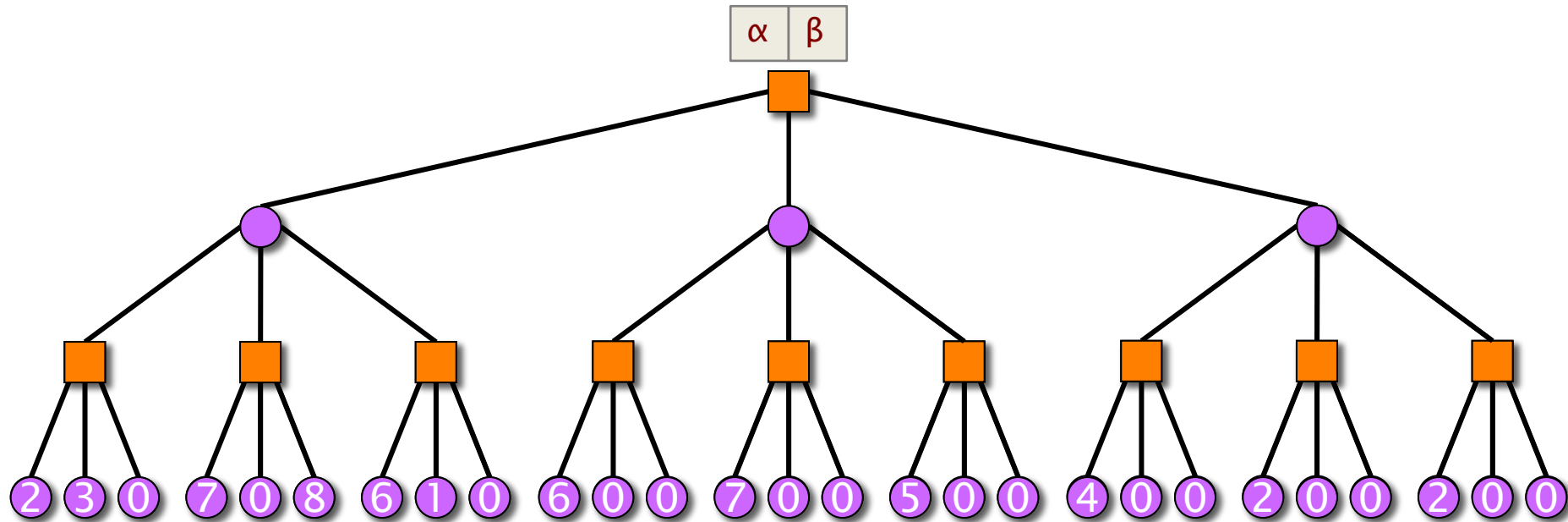
```
int search(pos x, int alpha, int beta, int d) {
    if (d == 0 || is_leaf(x)) return static_eval(x);
    pos c[MAX_CHILDREN];
    int nc;
    gen_moves(x, c, &nc);           // generate children
    for (int i = 0; i < nc; ++i) {
        int s = -search(c[i], -beta, -alpha, d-1); // negamax
        if (s > alpha) alpha = s;
        if (alpha >= beta) return alpha;         // beta cutoff
    }
    return alpha;
}
```



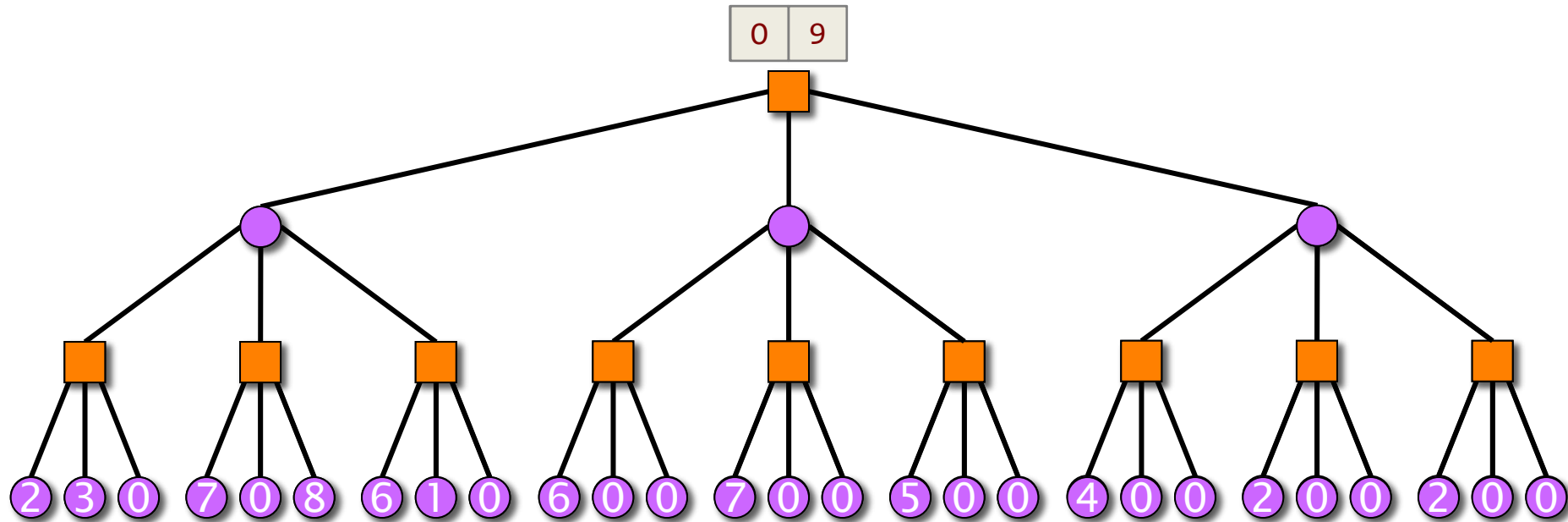
A WORKED EXAMPLE



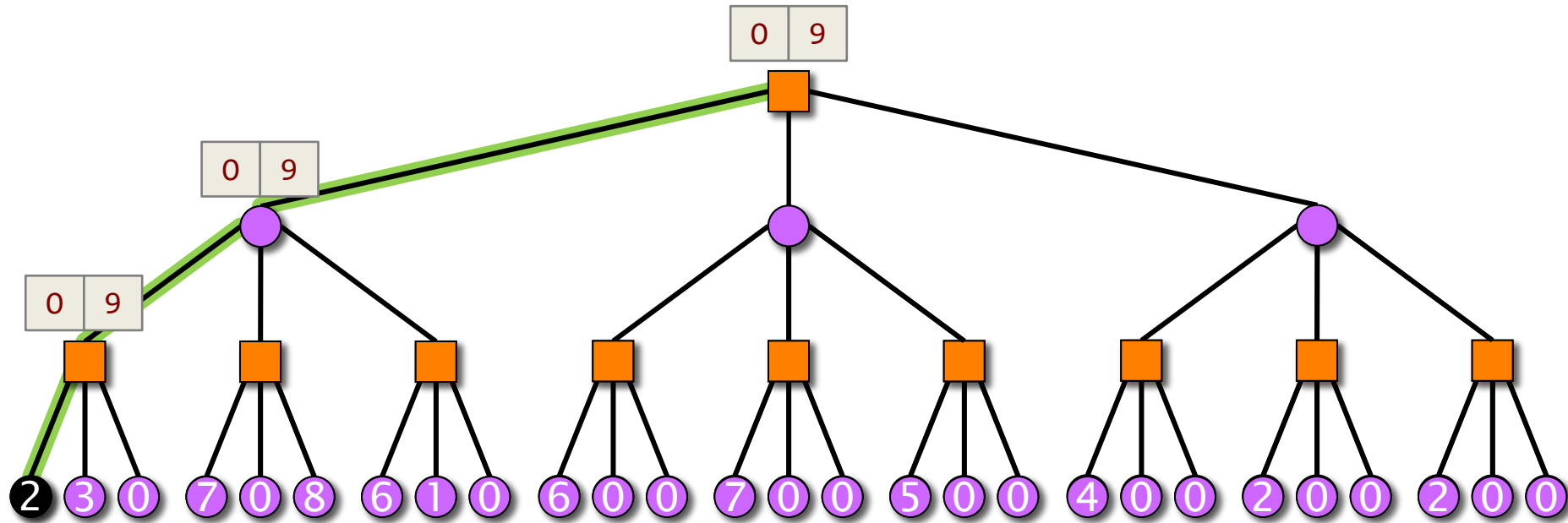
Alpha-Beta Search: Example



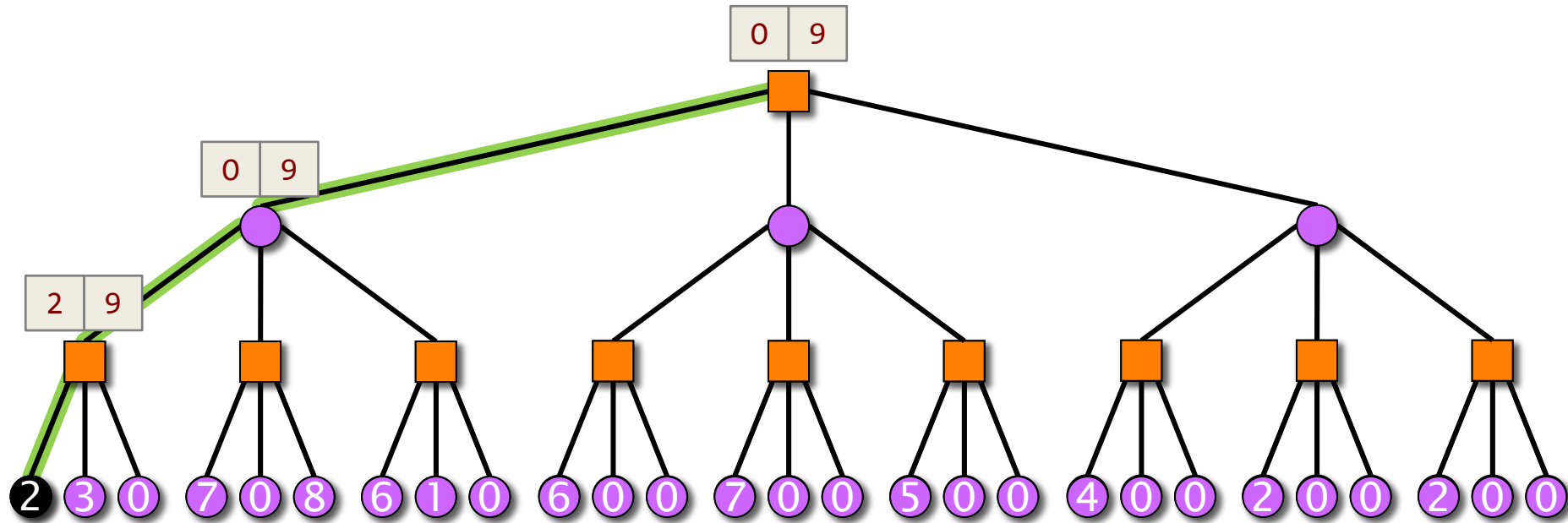
Alpha-Beta Search: Example



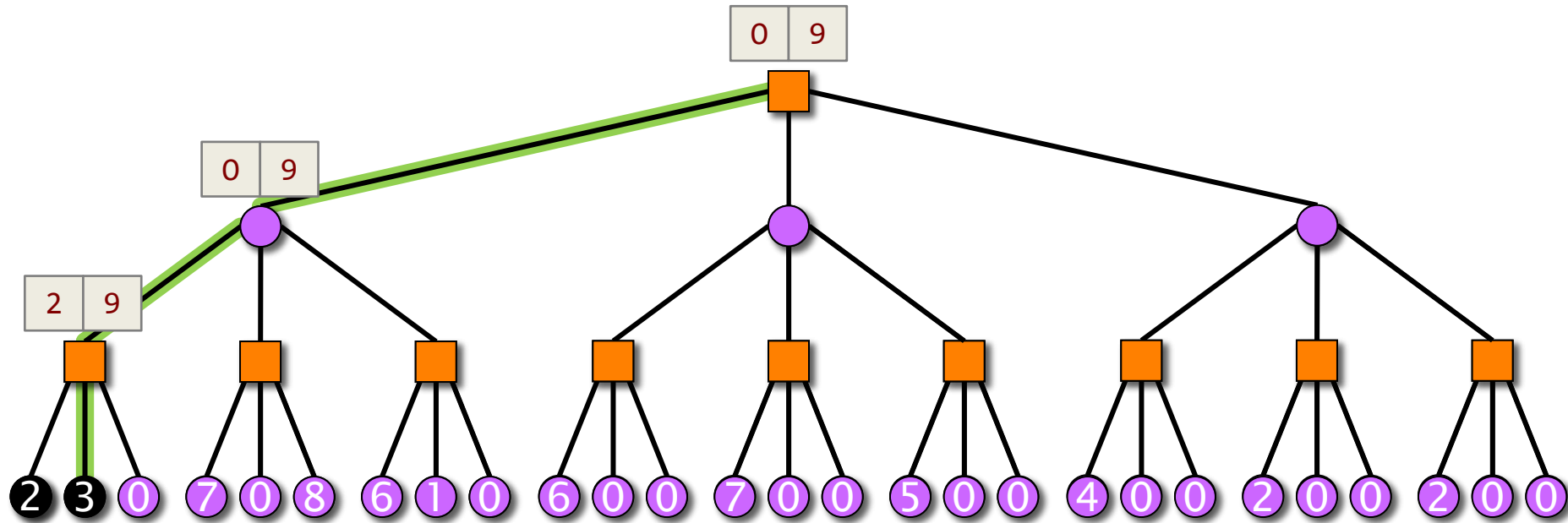
Alpha-Beta Search: Example



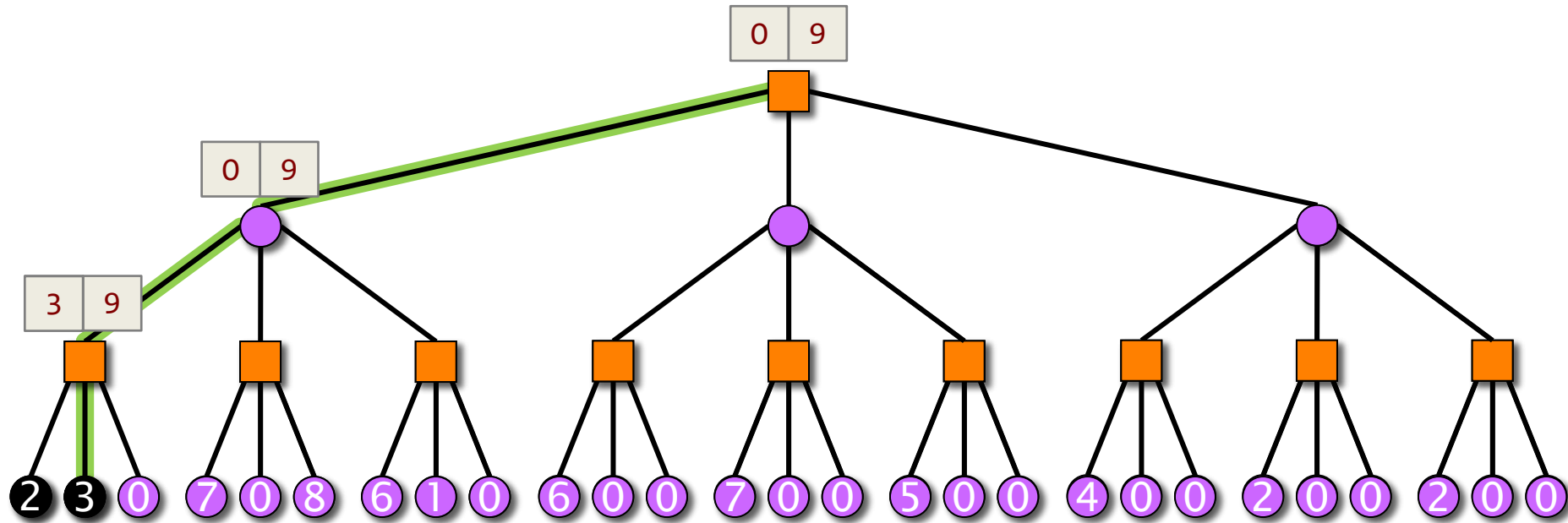
Alpha-Beta Search: Example



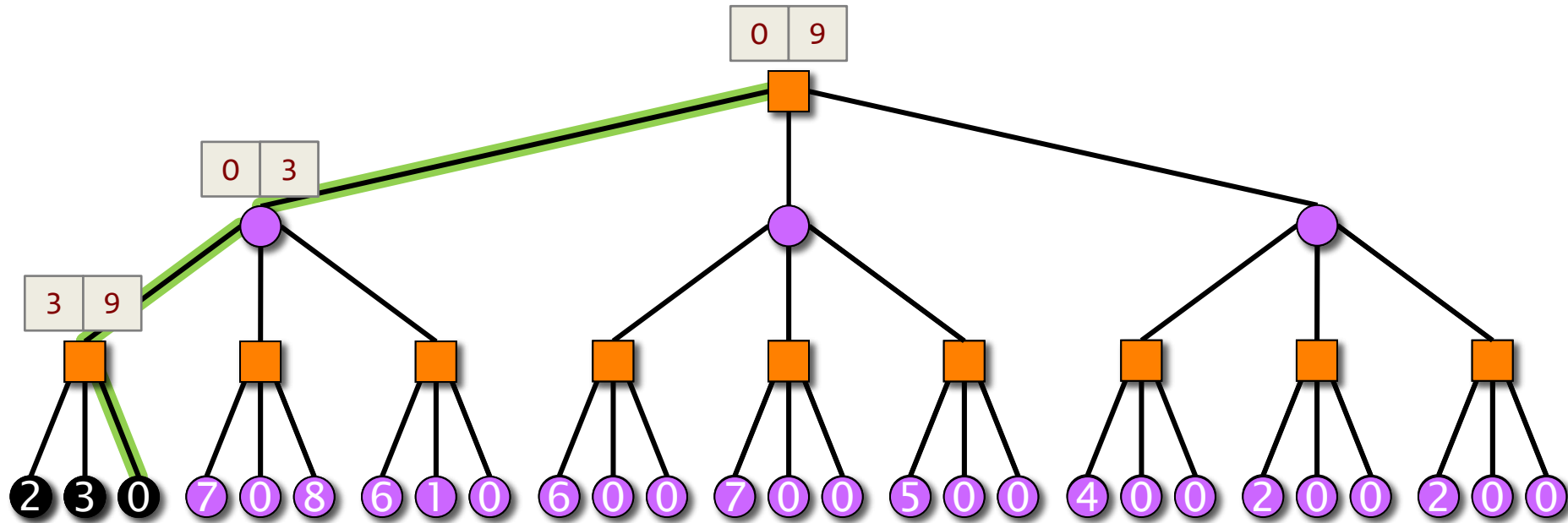
Alpha-Beta Search: Example



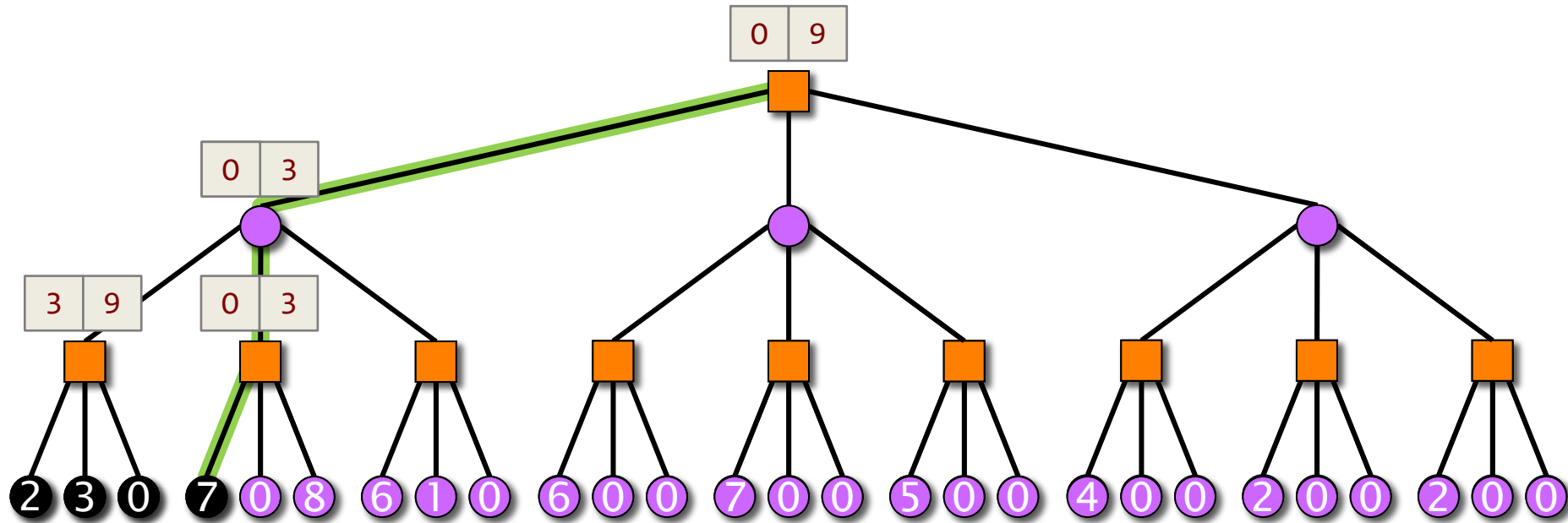
Alpha-Beta Search: Example



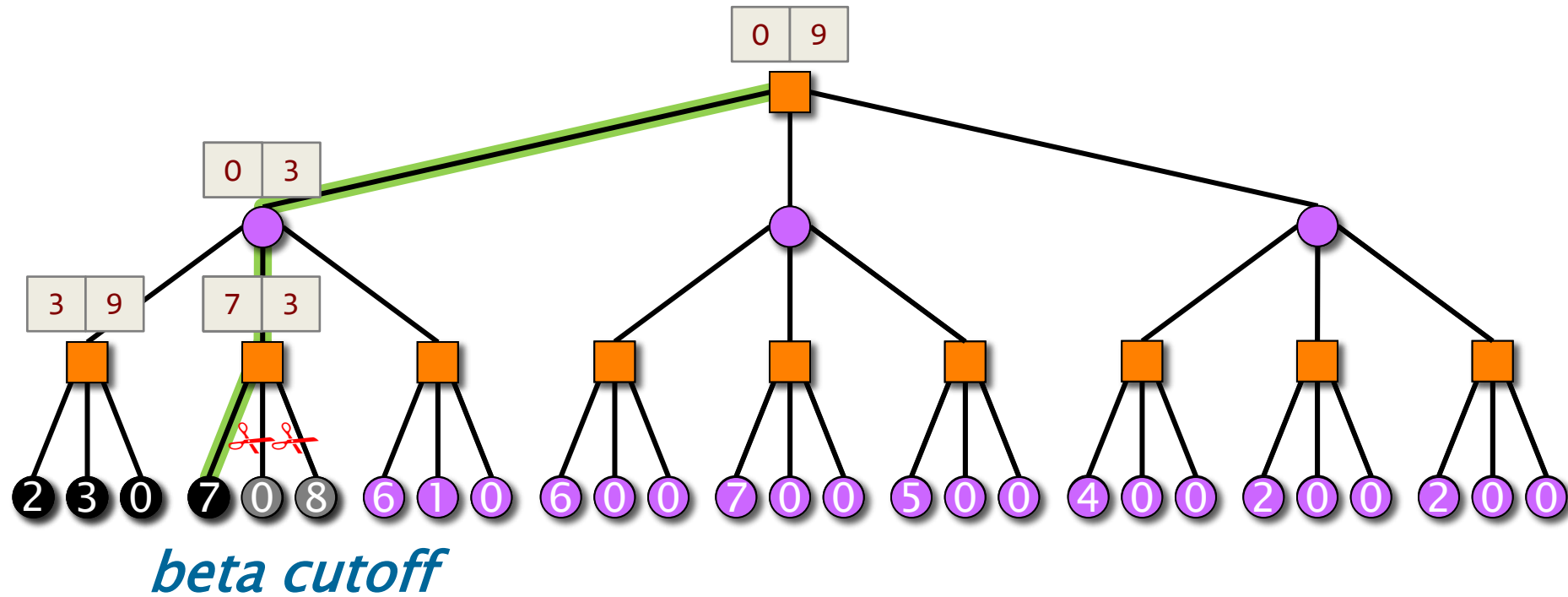
Alpha-Beta Search: Example



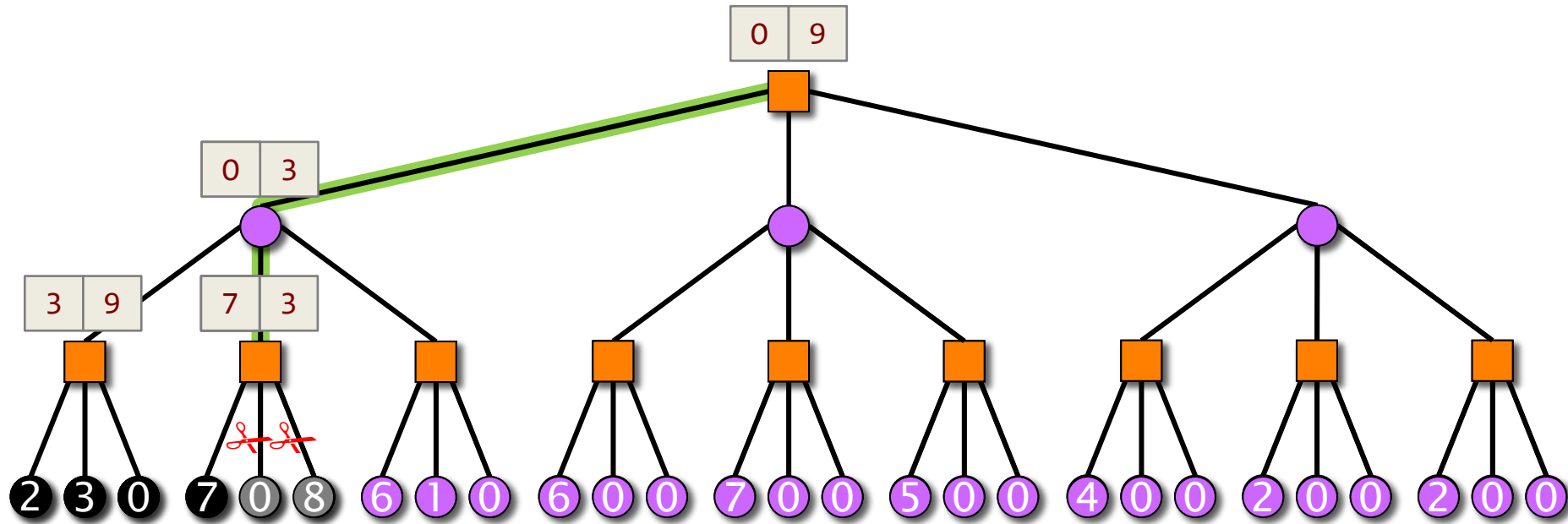
Alpha-Beta Search: Example



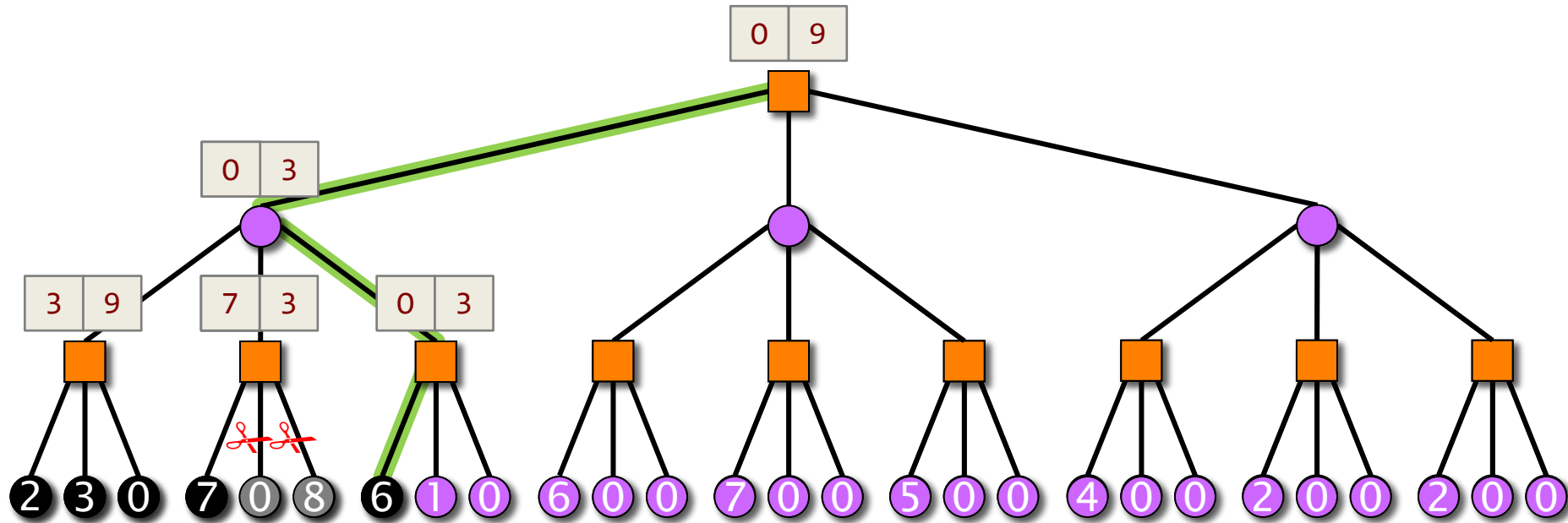
Alpha-Beta Search: Example



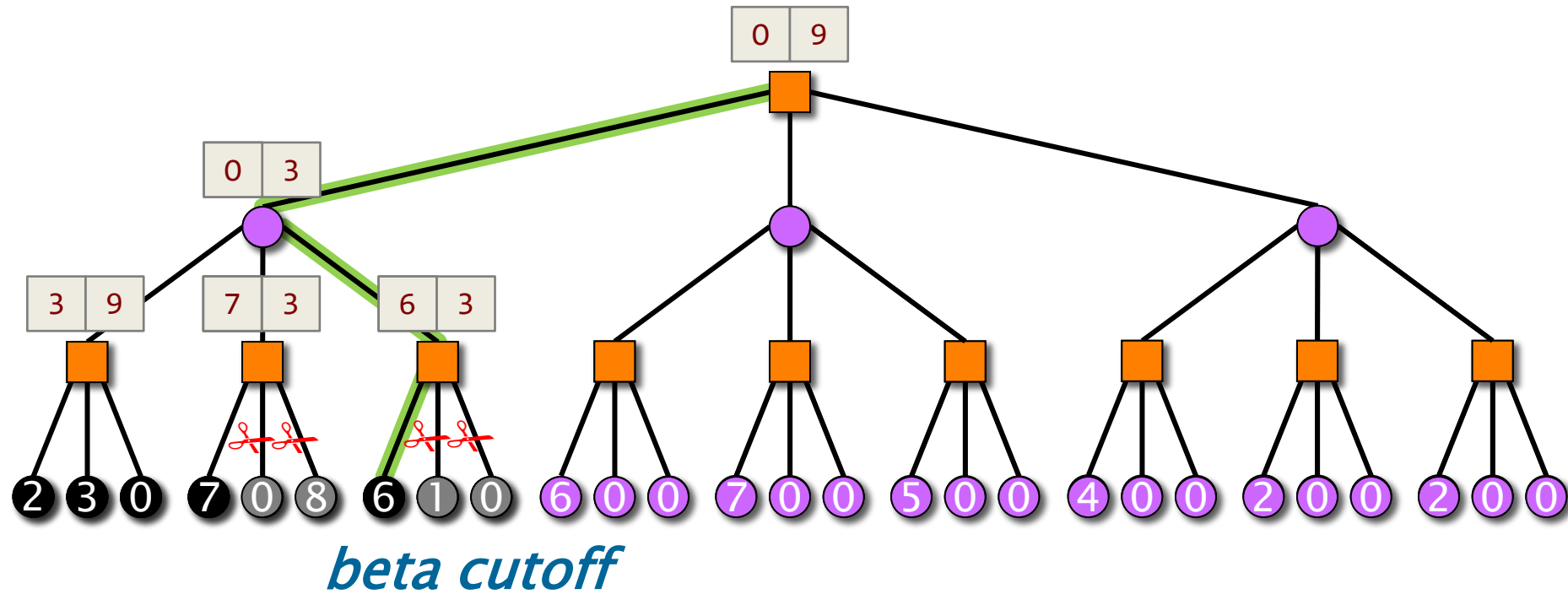
Alpha-Beta Search: Example



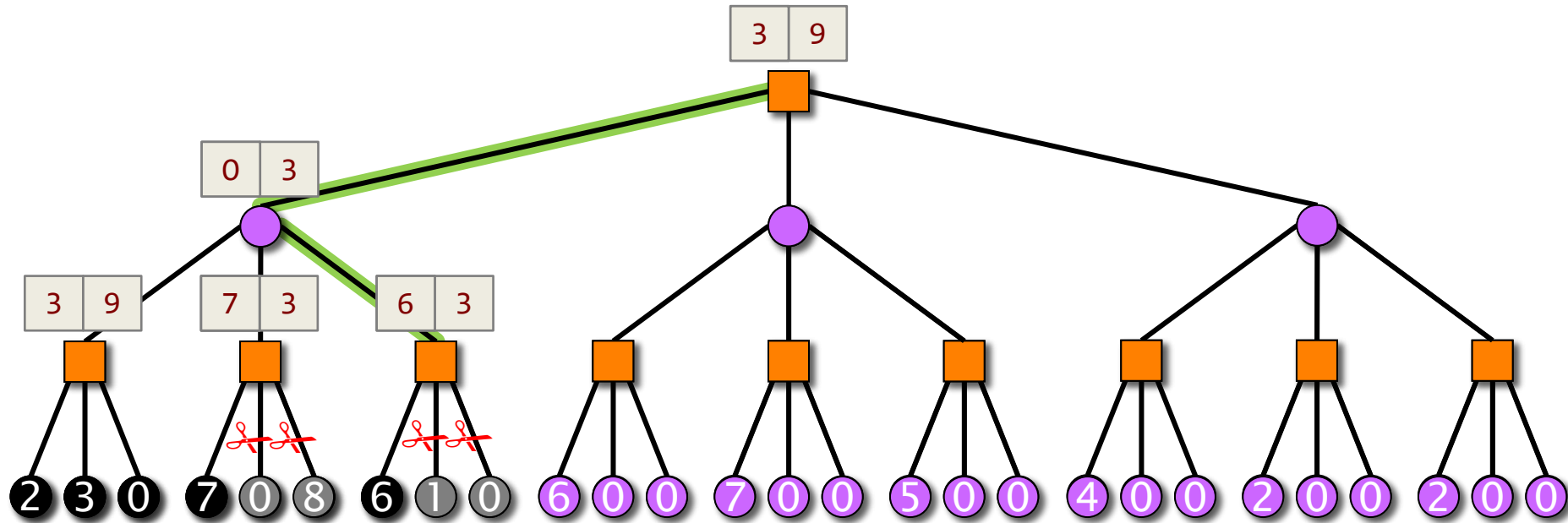
Alpha-Beta Search: Example



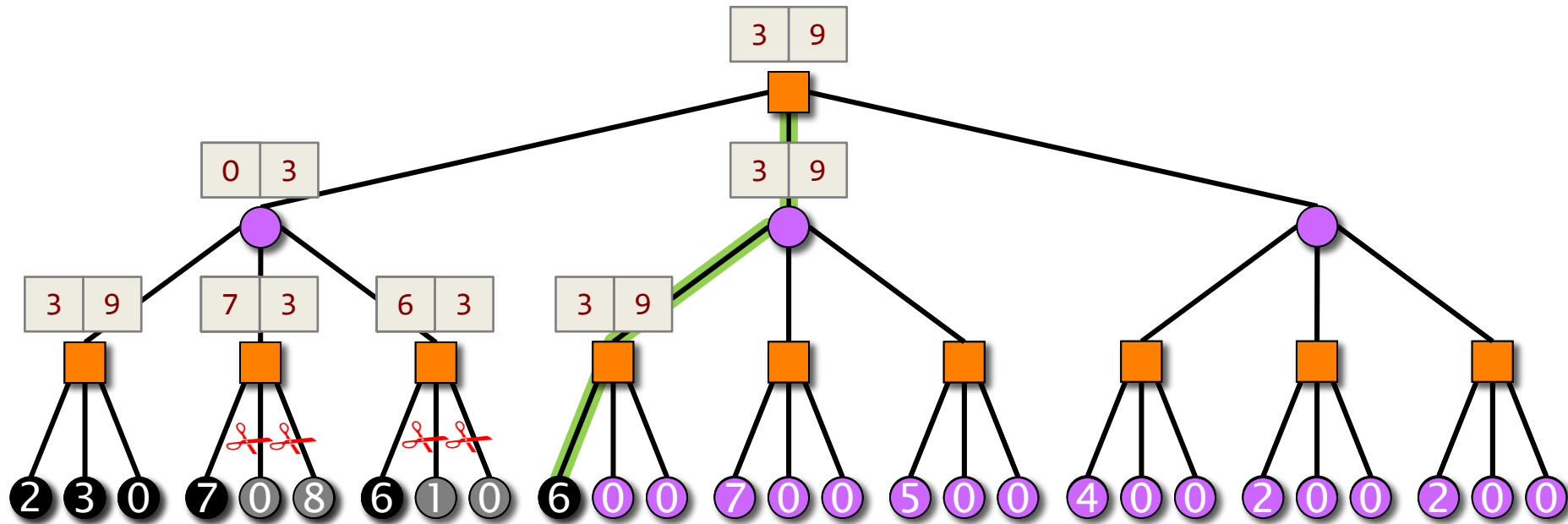
Alpha-Beta Search: Example



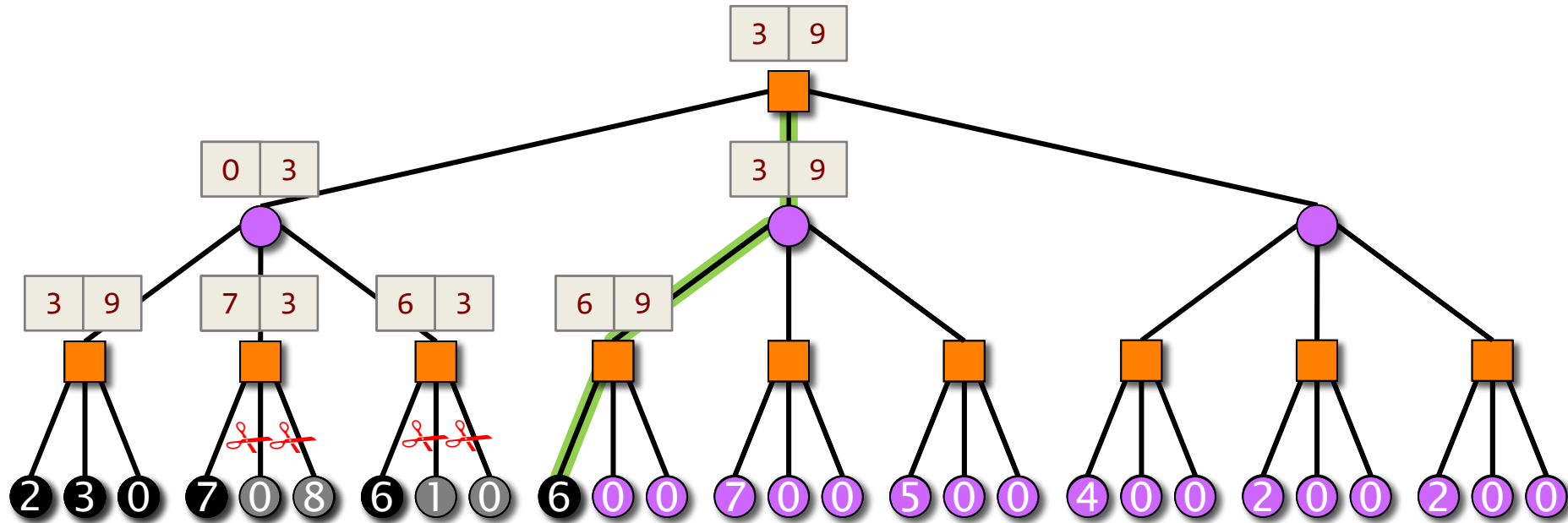
Alpha-Beta Search: Example



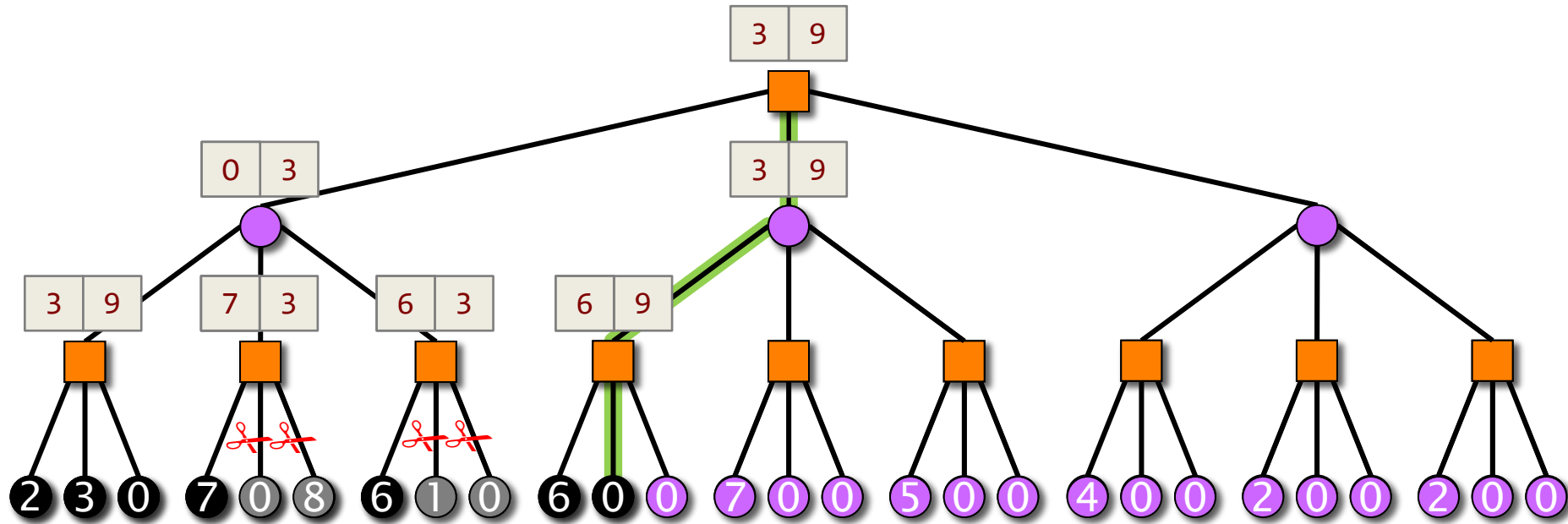
Alpha-Beta Search: Example



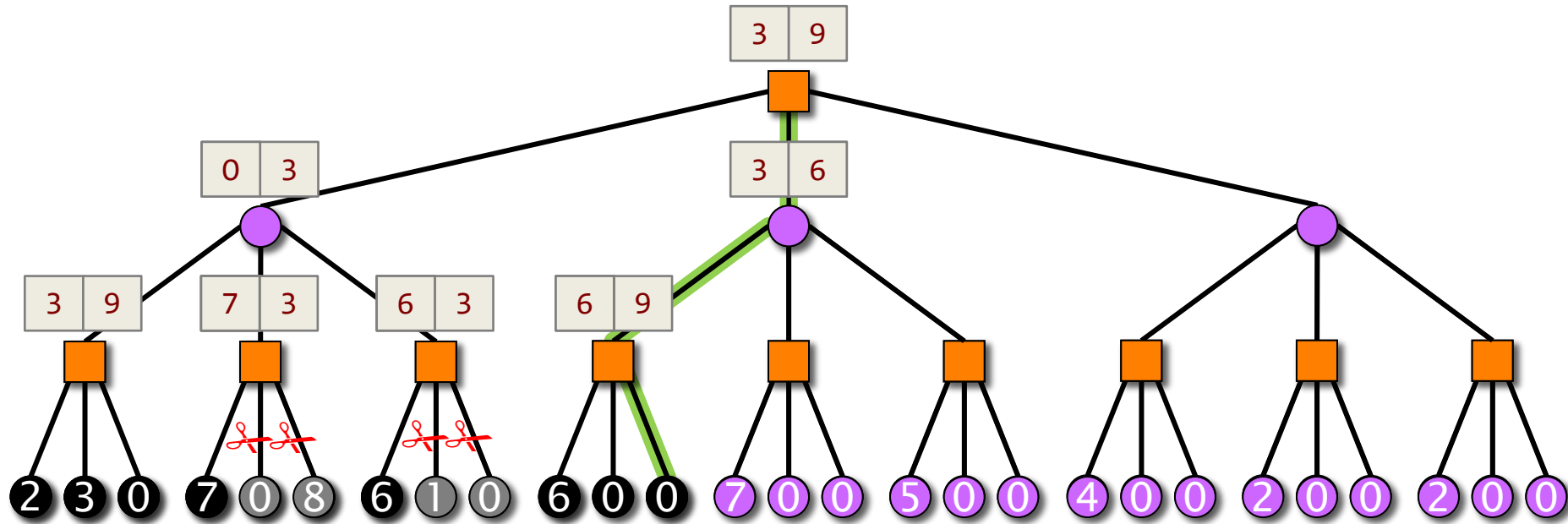
Alpha-Beta Search: Example



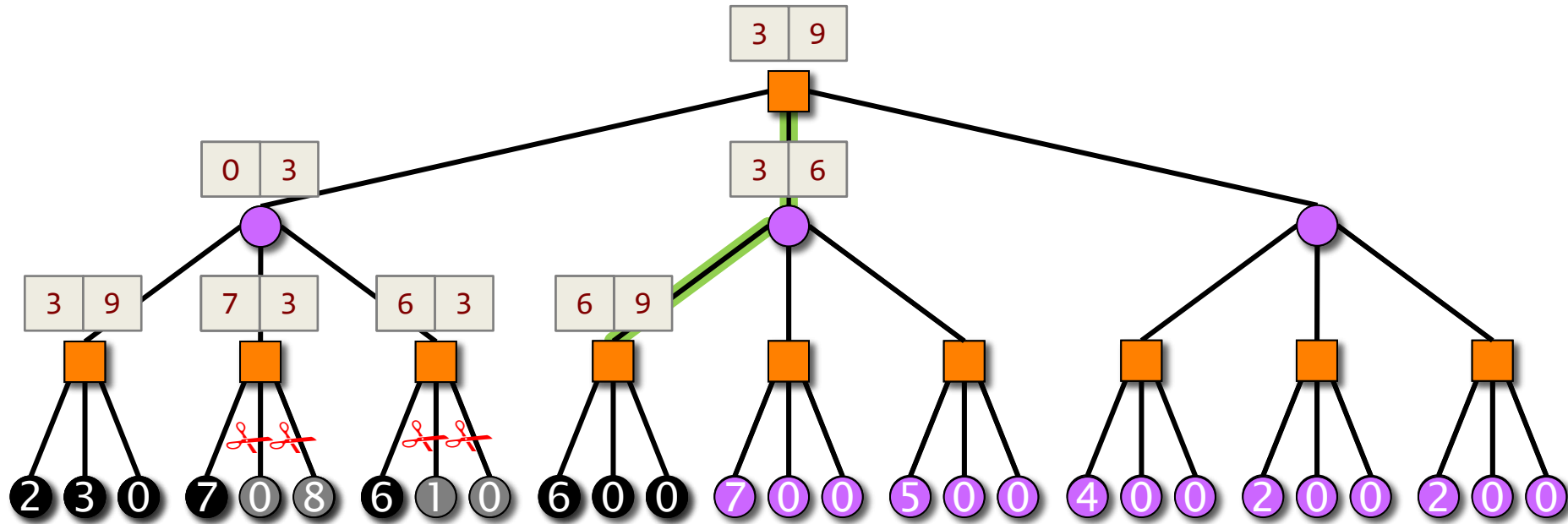
Alpha-Beta Search: Example



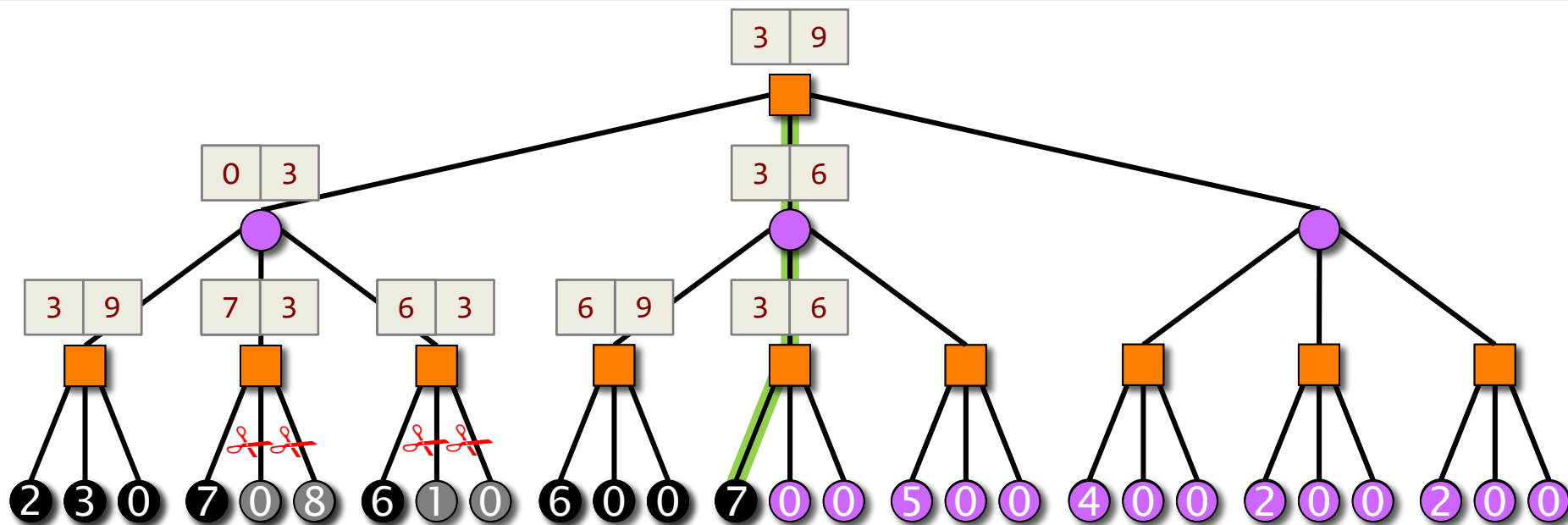
Alpha-Beta Search: Example



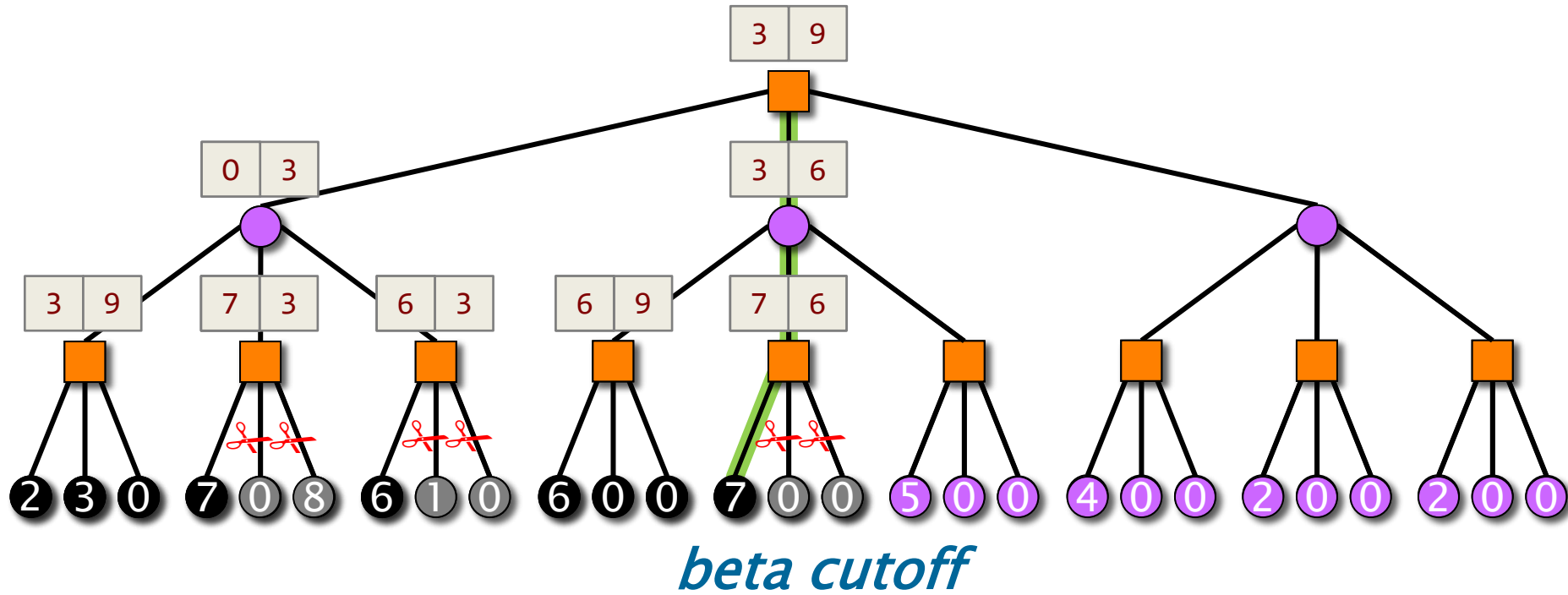
Alpha-Beta Search: Example



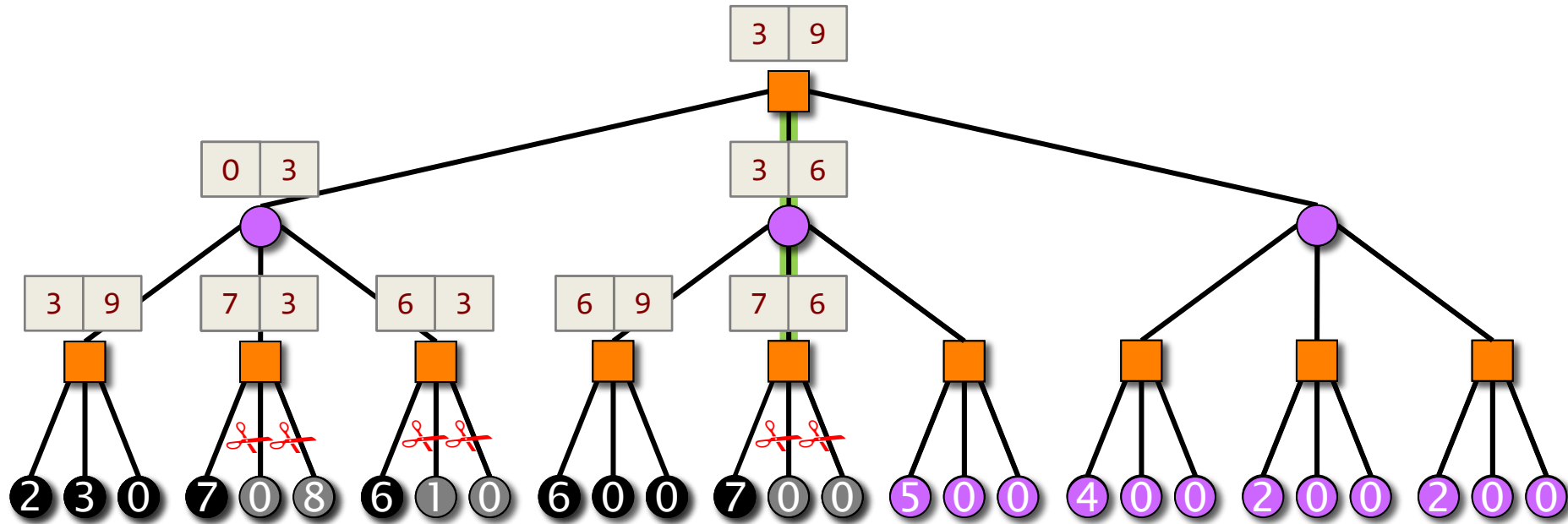
Alpha-Beta Search: Example



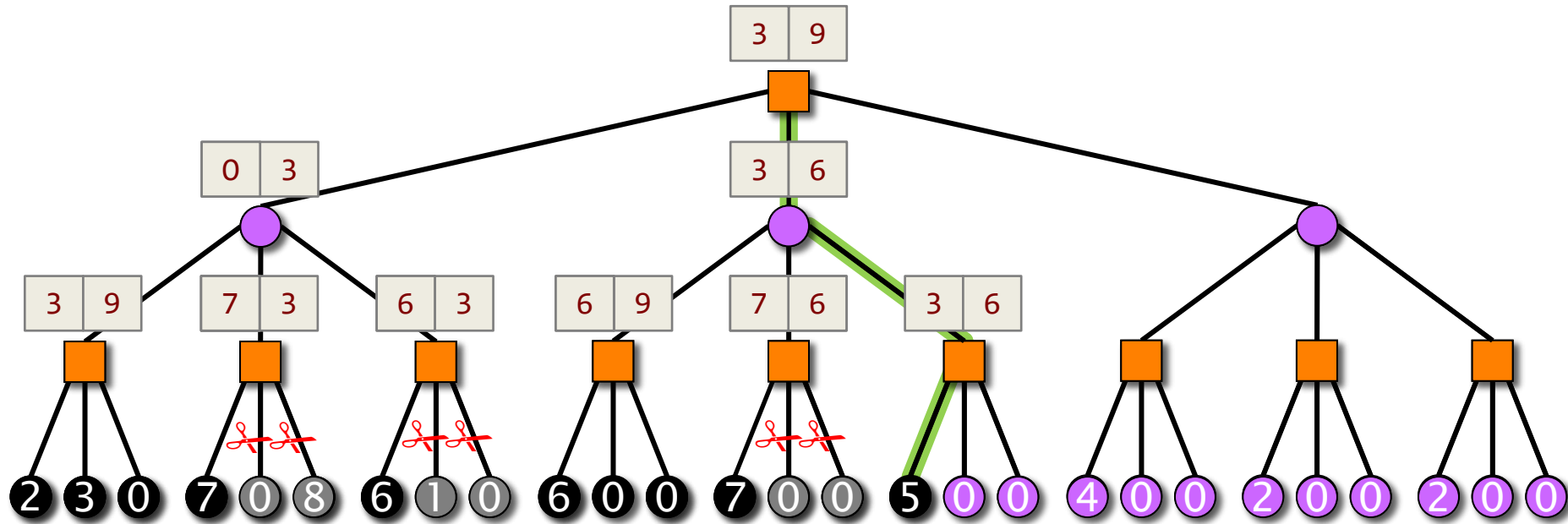
Alpha-Beta Search: Example



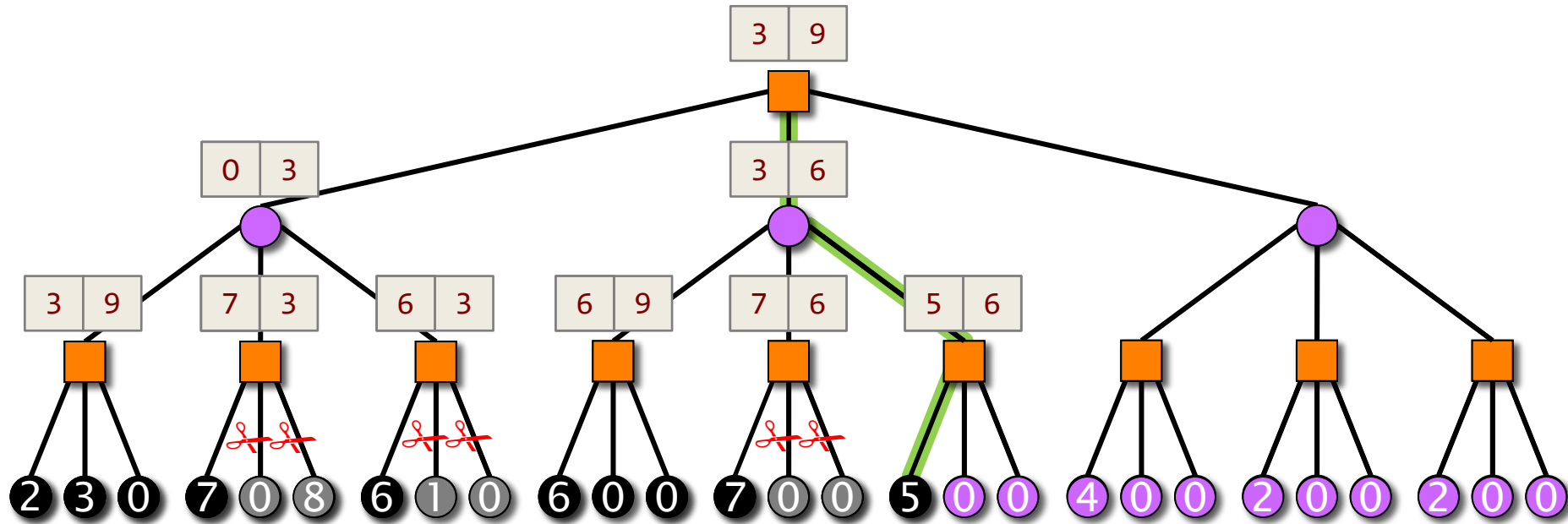
Alpha-Beta Search: Example



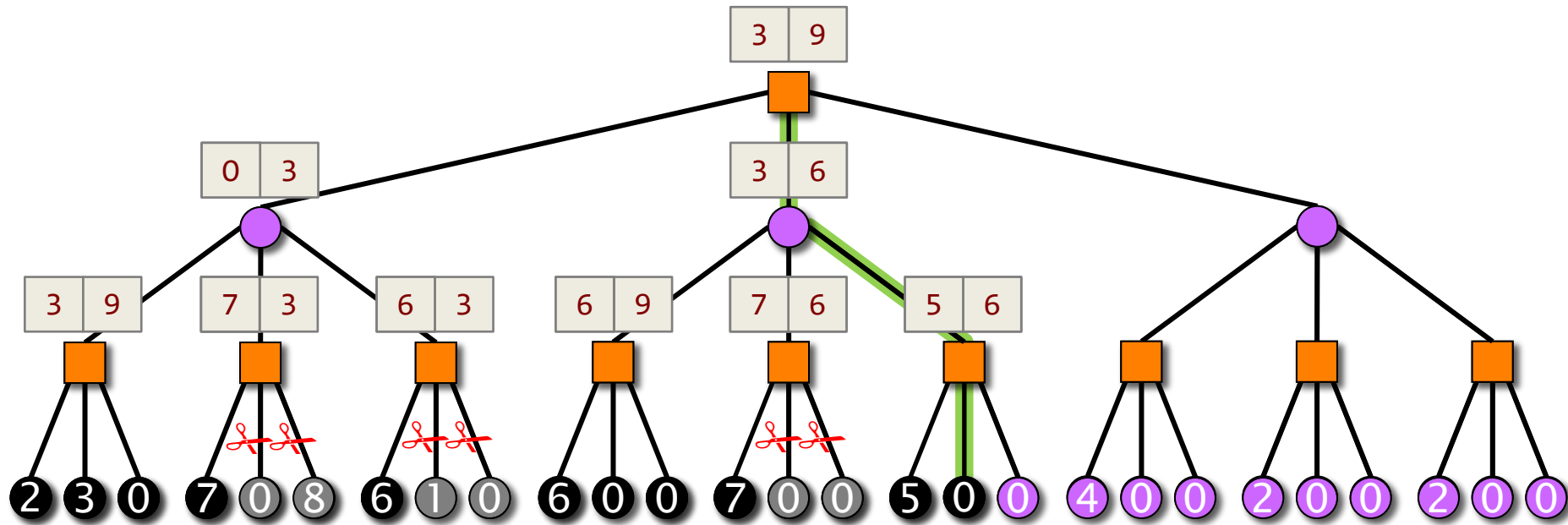
Alpha-Beta Search: Example



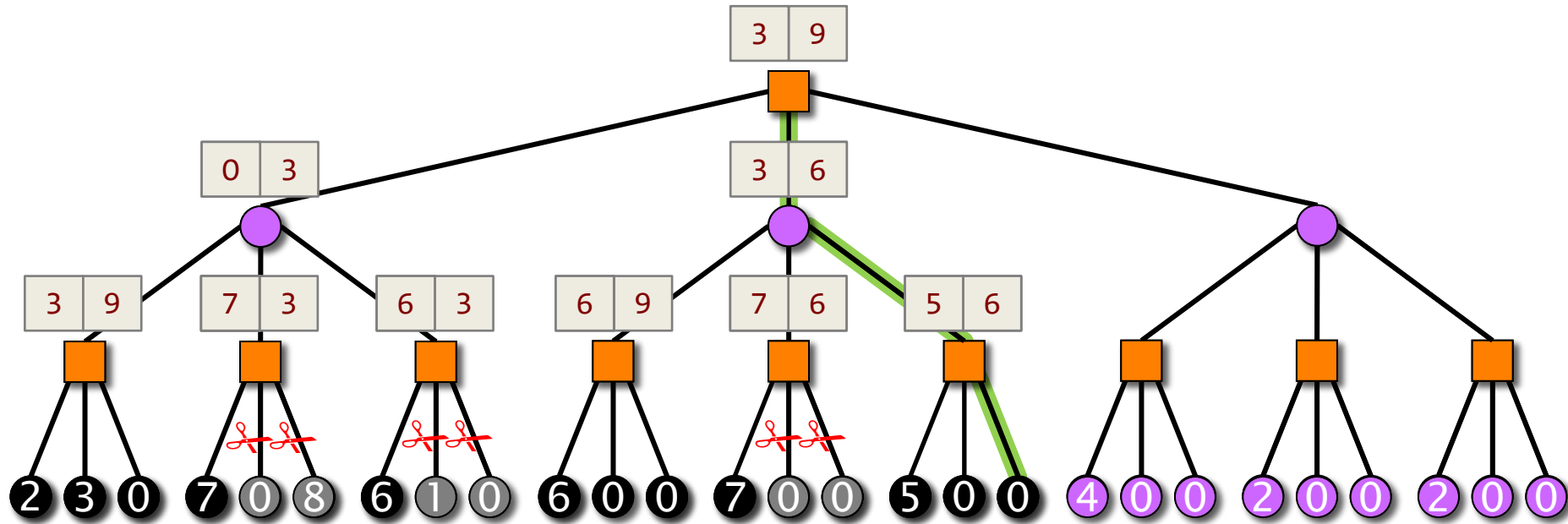
Alpha-Beta Search: Example



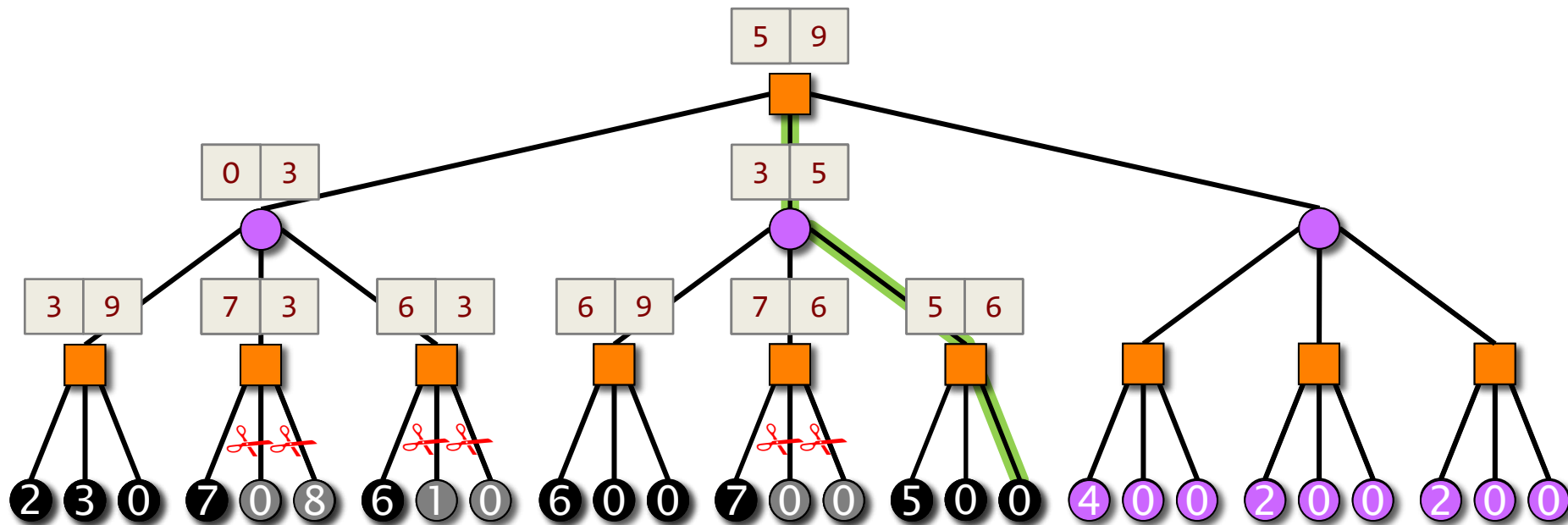
Alpha-Beta Search: Example



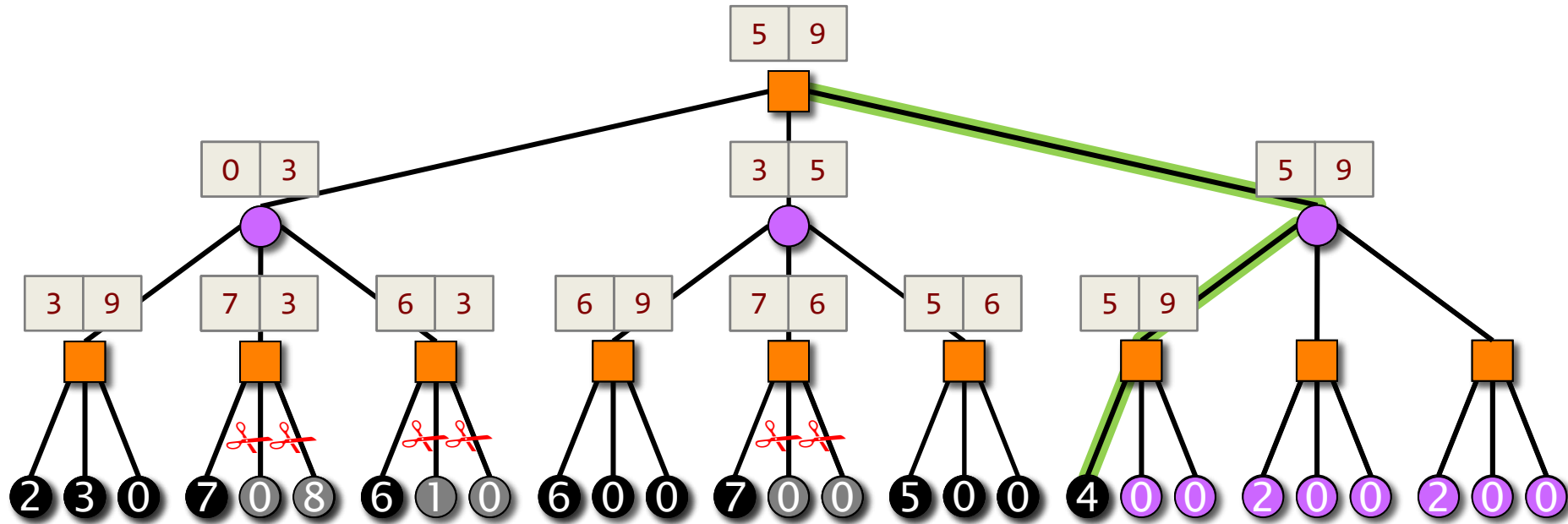
Alpha-Beta Search: Example



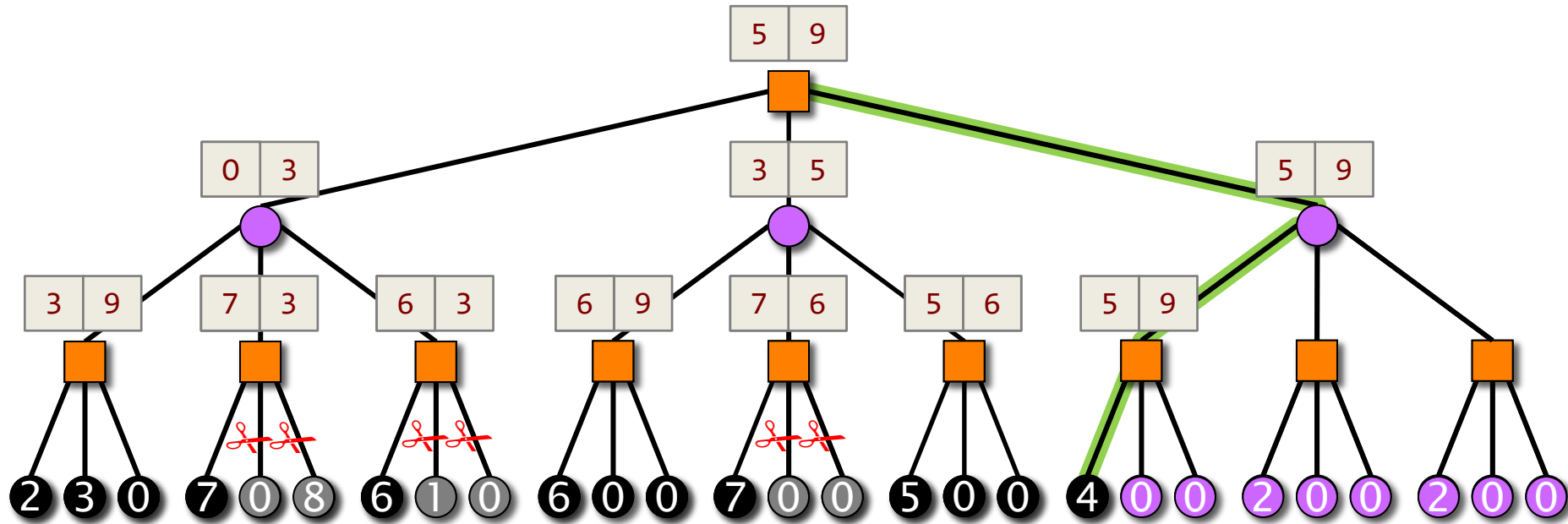
Alpha-Beta Search: Example



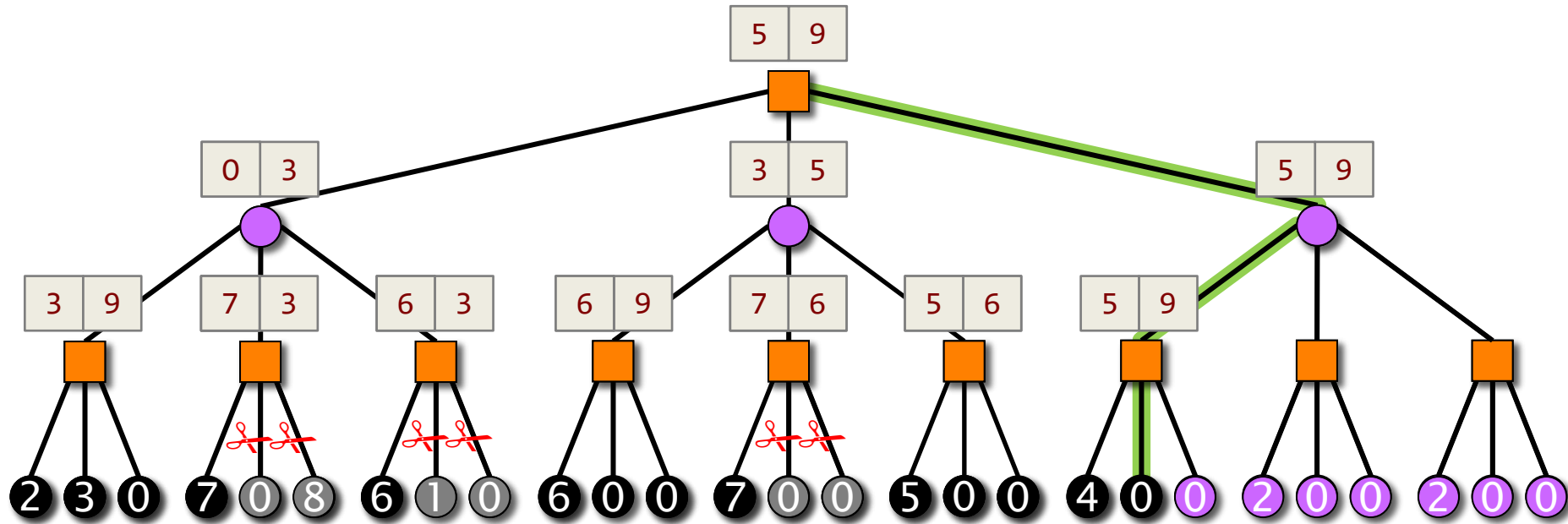
Alpha-Beta Search: Example



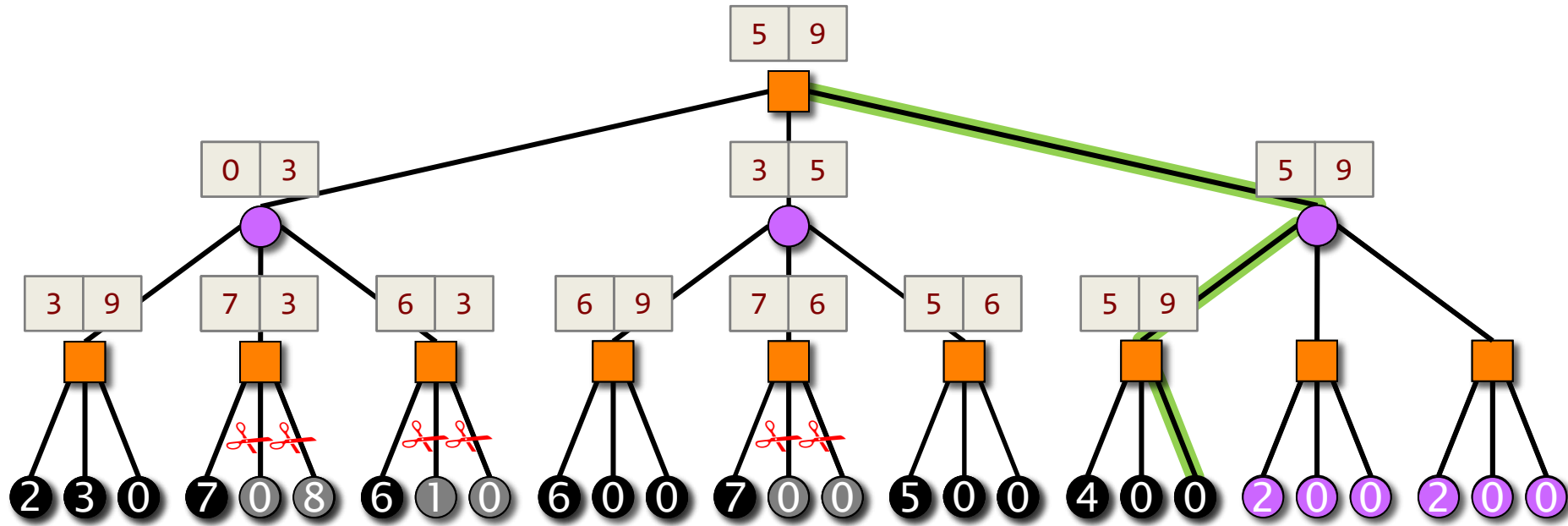
Alpha-Beta Search: Example



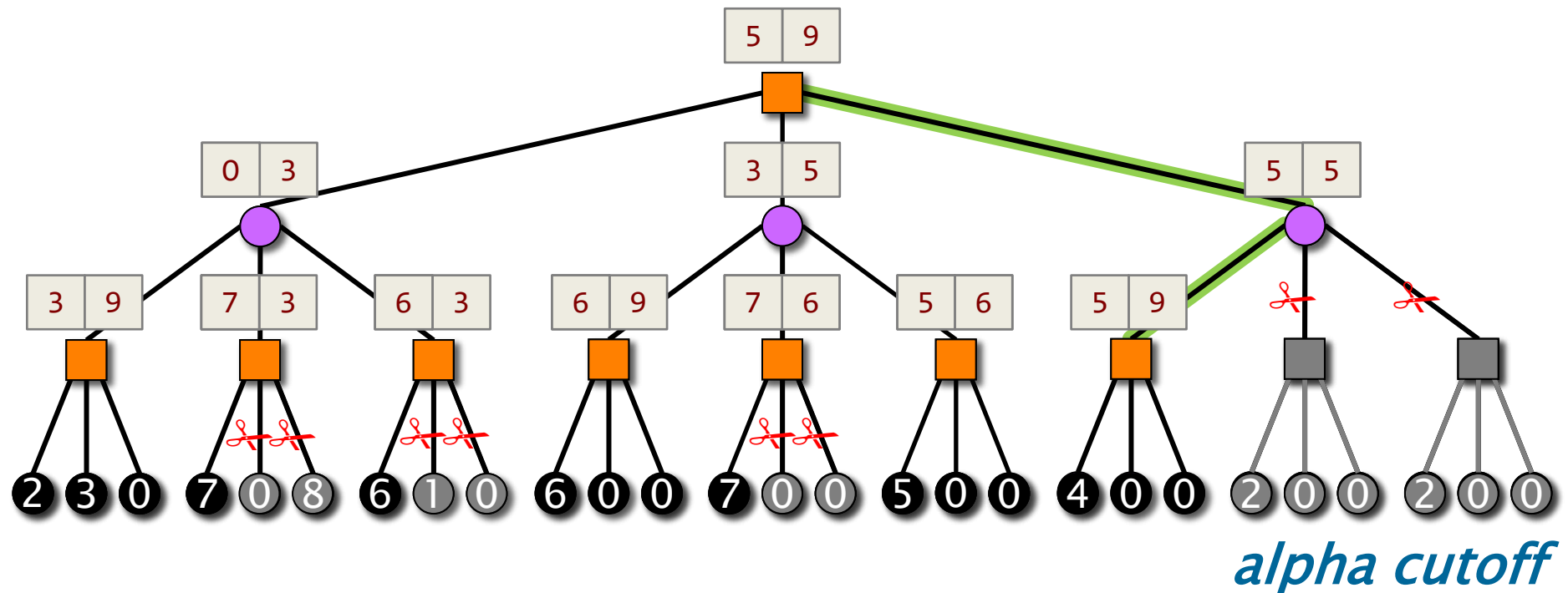
Alpha-Beta Search: Example



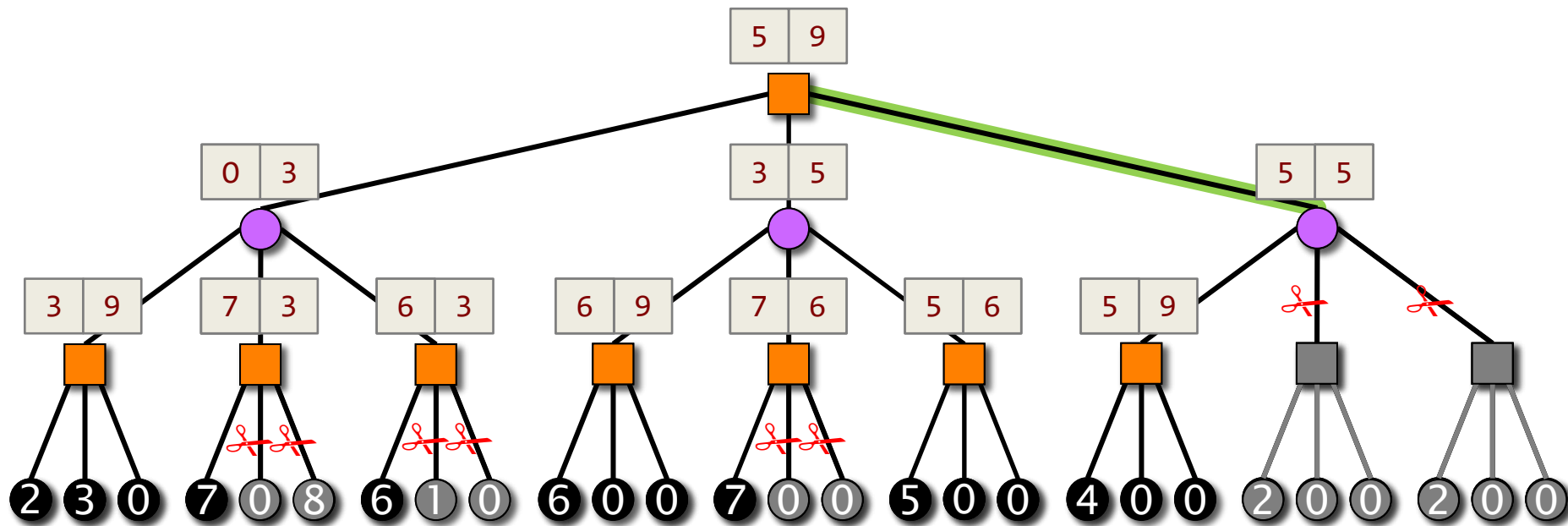
Alpha-Beta Search: Example



Alpha-Beta Search: Example



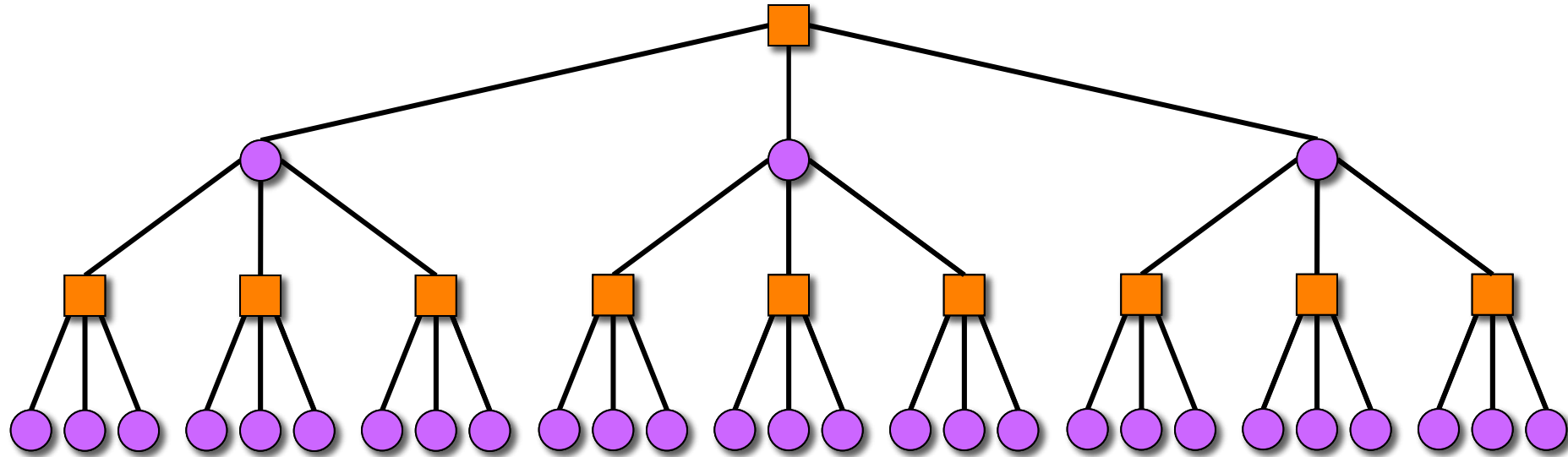
Alpha-Beta Search: Example



ANALYSIS OF ALPHA-BETA



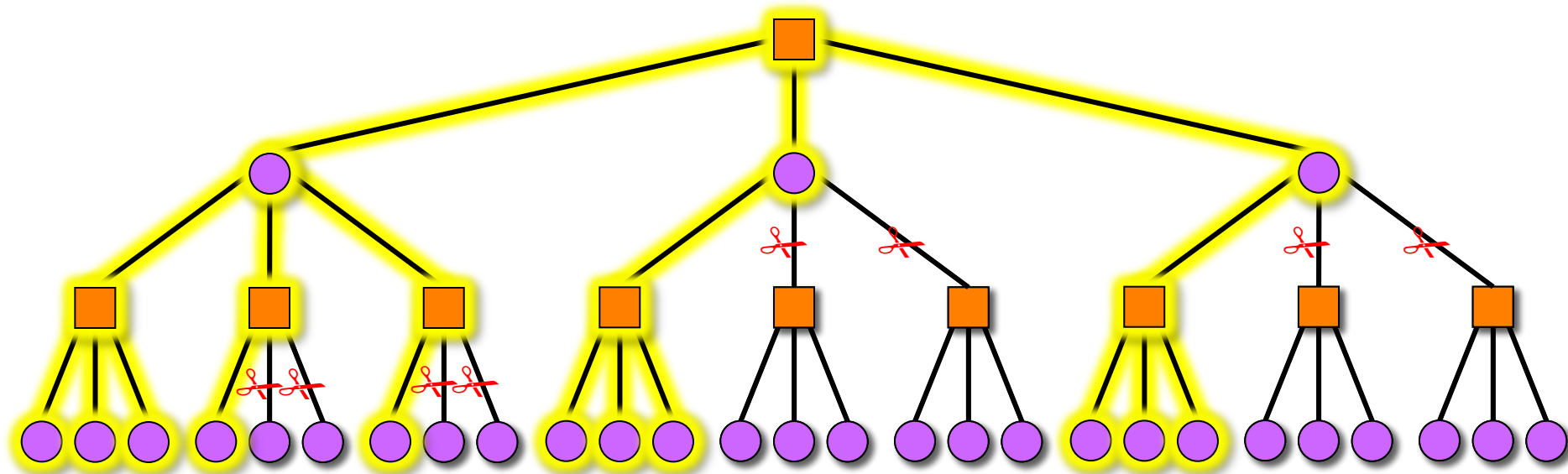
Min-Max Game Tree



The number of nodes in a game tree with branching factor $b > 1$ and depth $d \geq 0$ is

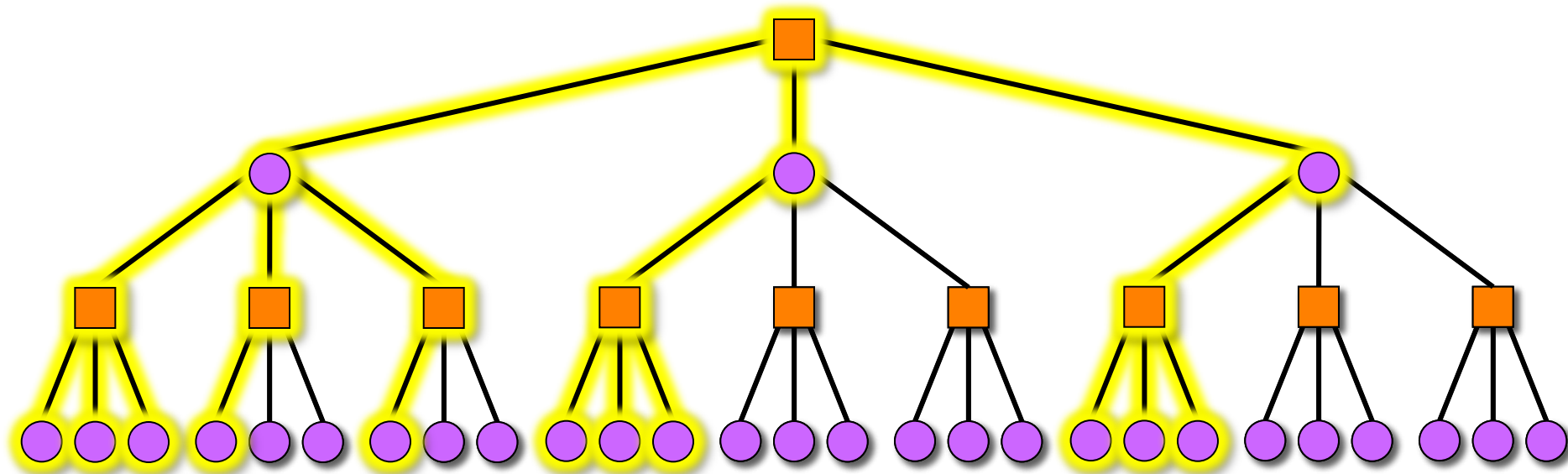
$$1 + b + b^2 + \dots + b^d = \frac{b^{d+1} - 1}{b - 1} \\ \approx bd$$

Best-Case Alpha-Beta Analysis



Theorem [KM75]. For a game tree with branching factor $b > 1$ and depth $d \geq 0$, an alpha-beta search with moves searched in **best-first order** examines exactly $b^{\lceil k/2 \rceil} + b^{\lfloor k/2 \rfloor} - 1$ nodes at ply k , which is approximately $2b^{d/2}$ nodes overall. ■

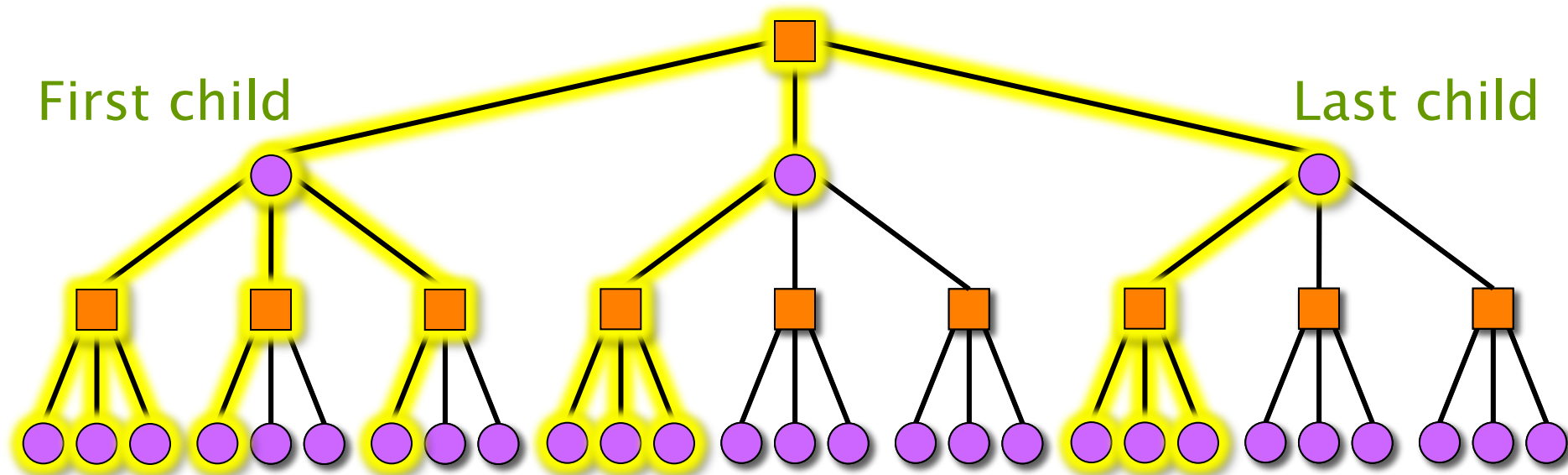
Best-Case Alpha-Beta Analysis



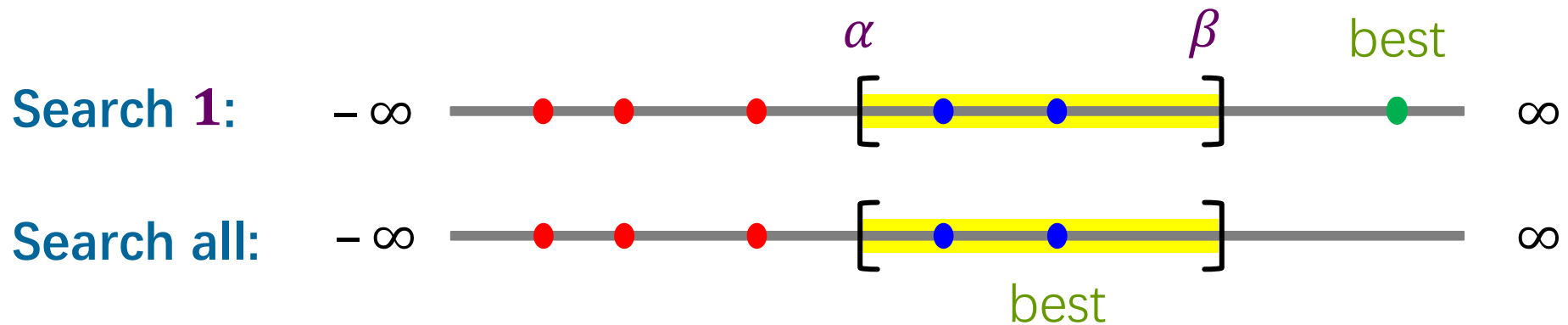
What that means: Since the naive algorithm examines about b^d nodes and alpha-beta examines only about $2b^{d/2} = 2\sqrt{b^d} = 2(\sqrt{b})^d$ nodes, alpha-beta effectively

- doubles the search depth,
- square-roots the work,
- square-roots the branching factor.

Best-Ordered Trees



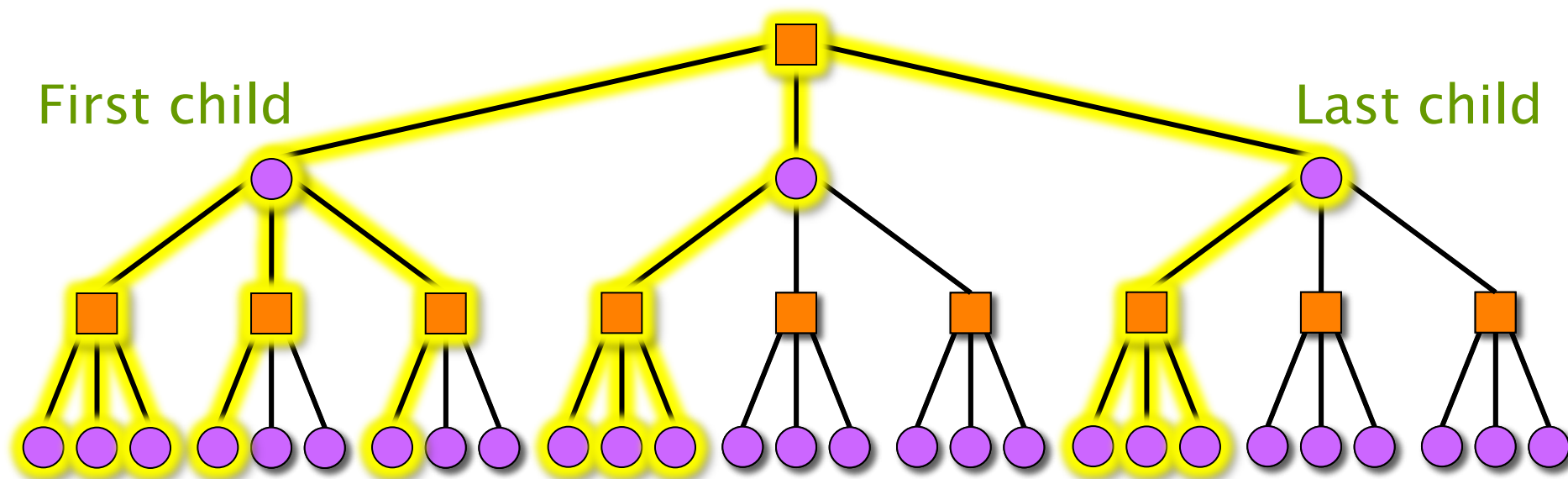
Observation: In a **best-ordered tree** (first child is best), the degree of every node is either **1** or **b** (maximal).



PARALLEL ALPHA-BETA SEARCH



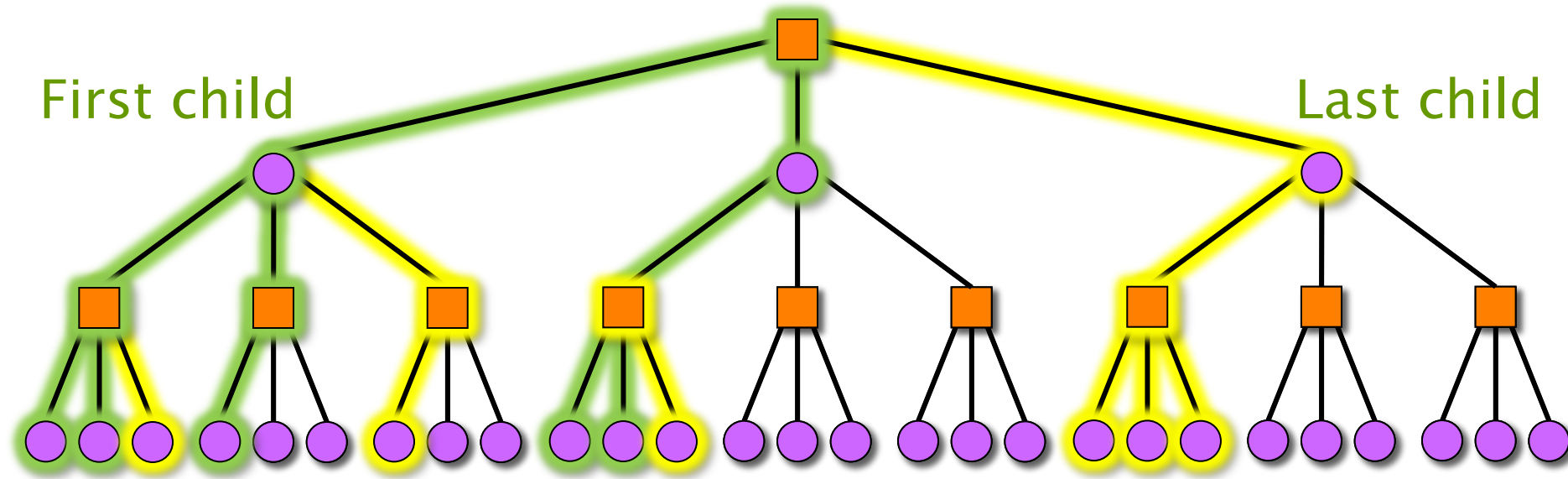
Parallel Alpha-Beta



Observation: In a **best-ordered tree** (first child is best), the degree of every node is either **1** or **b** (maximal).

Idea [FMMV89]: **Young siblings wait** — If the first (hope-fully best) child fails to generate a beta-cutoff, speculate that the node is maximal, and search the remaining children in parallel. Bet that you won't waste any work.

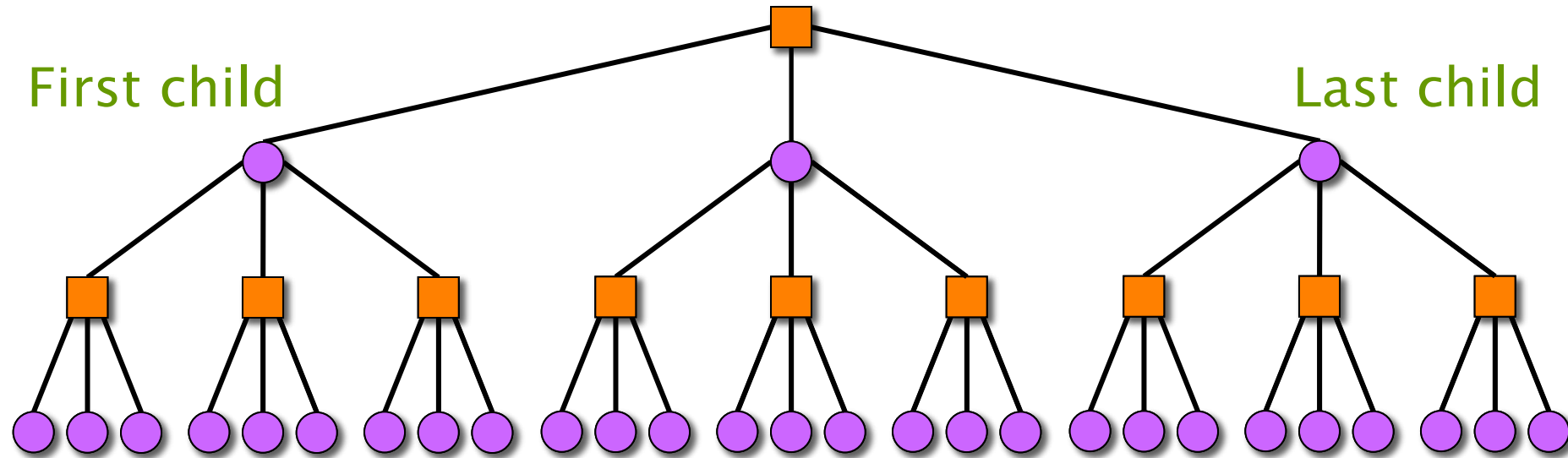
Parallelism of Young Siblings Wait



Analysis. If we search all maximal nodes' children 2 through b in parallel, the span is just like the work of searching a tree with branching factor 2 . Thus, $T_\infty = 2 \cdot 2^{d/2}$, and parallelism $= 2 \cdot b^{d/2} / (2 \cdot 2^{d/2}) = (b/2)^{d/2}$.

Example. If we search 1 billion nodes in a depth- 10 search, we might hope to achieve a parallelism of $10^9/64 \approx 16M$, which is more than ample.

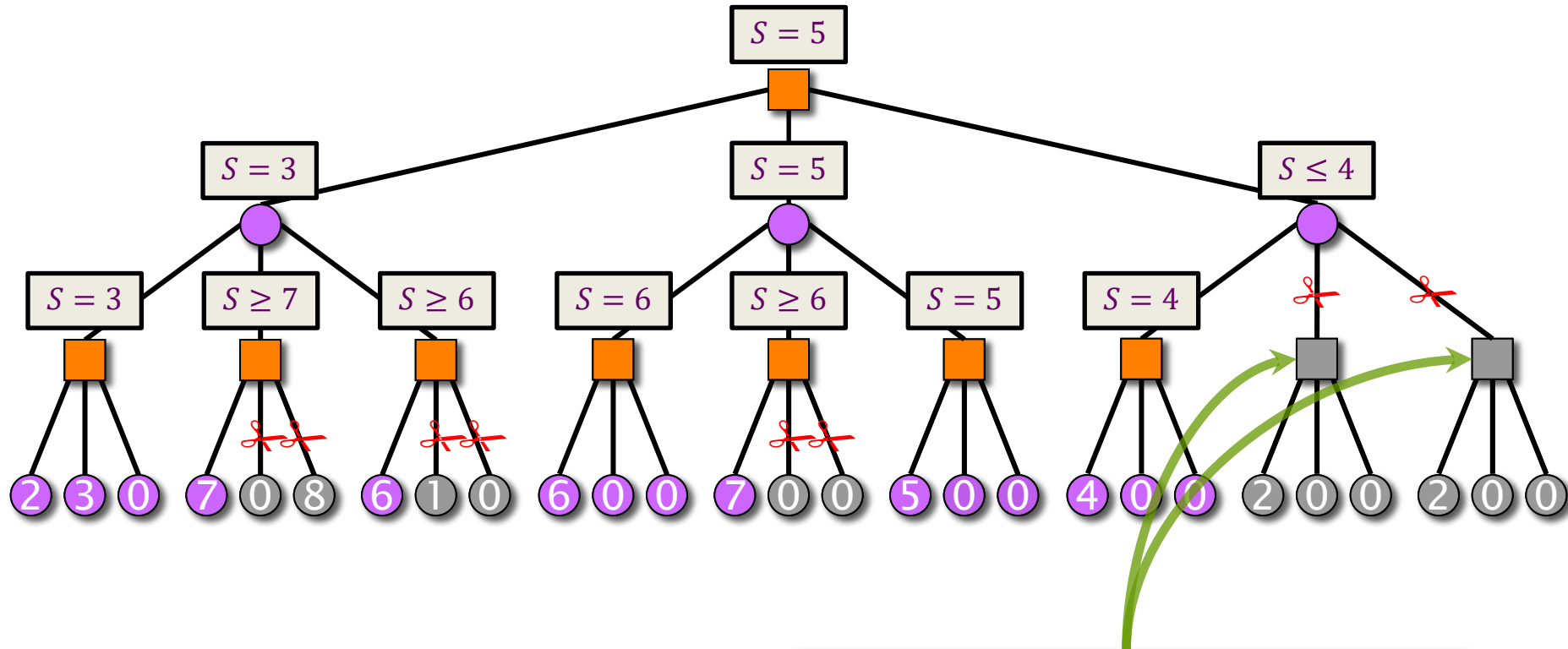
Back to Reality



But what if our speculation is wrong?

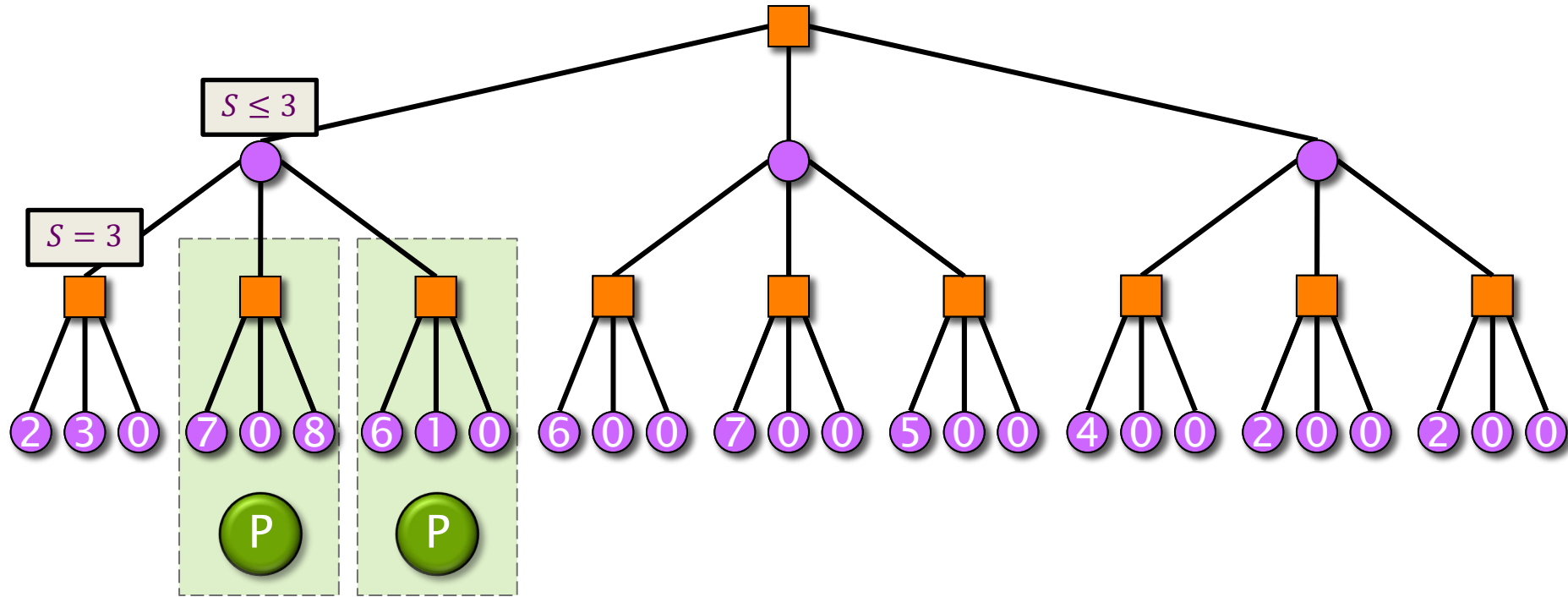
- In general, game trees are *not* best-ordered.
- Parallel alpha-beta search with young-siblings-wait wastes work if the best child is not first.

Alpha-Beta Search: Example



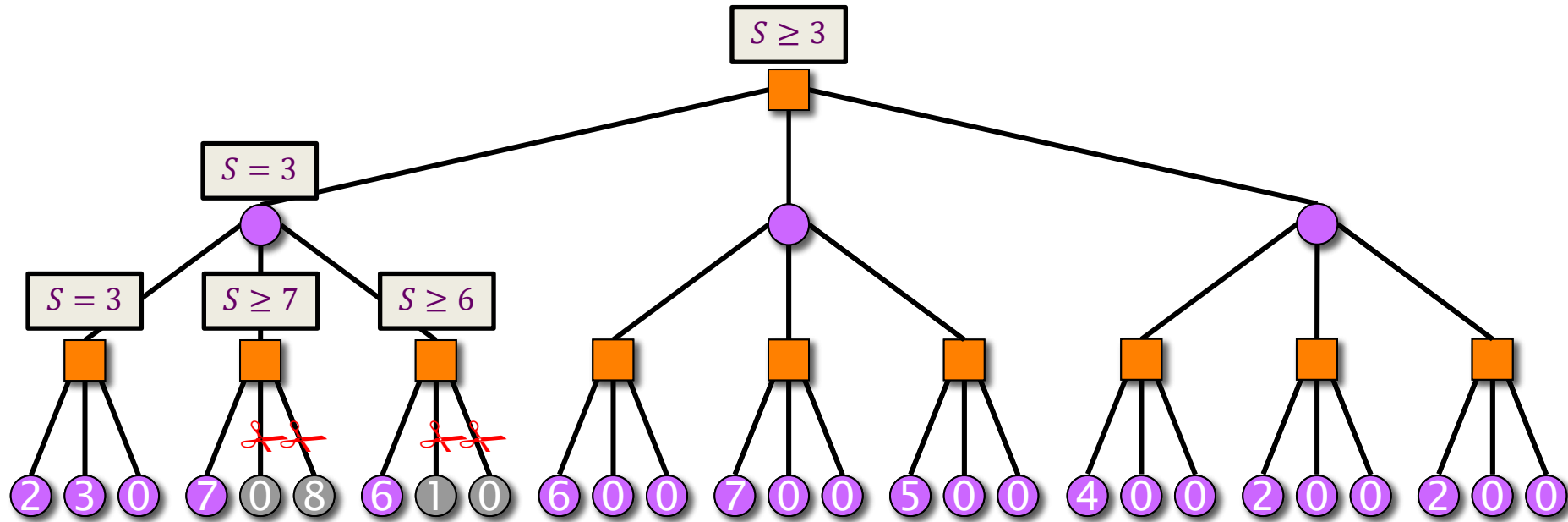
The second sibling of the root provides the cutoff.

Young Siblings Wait: Example

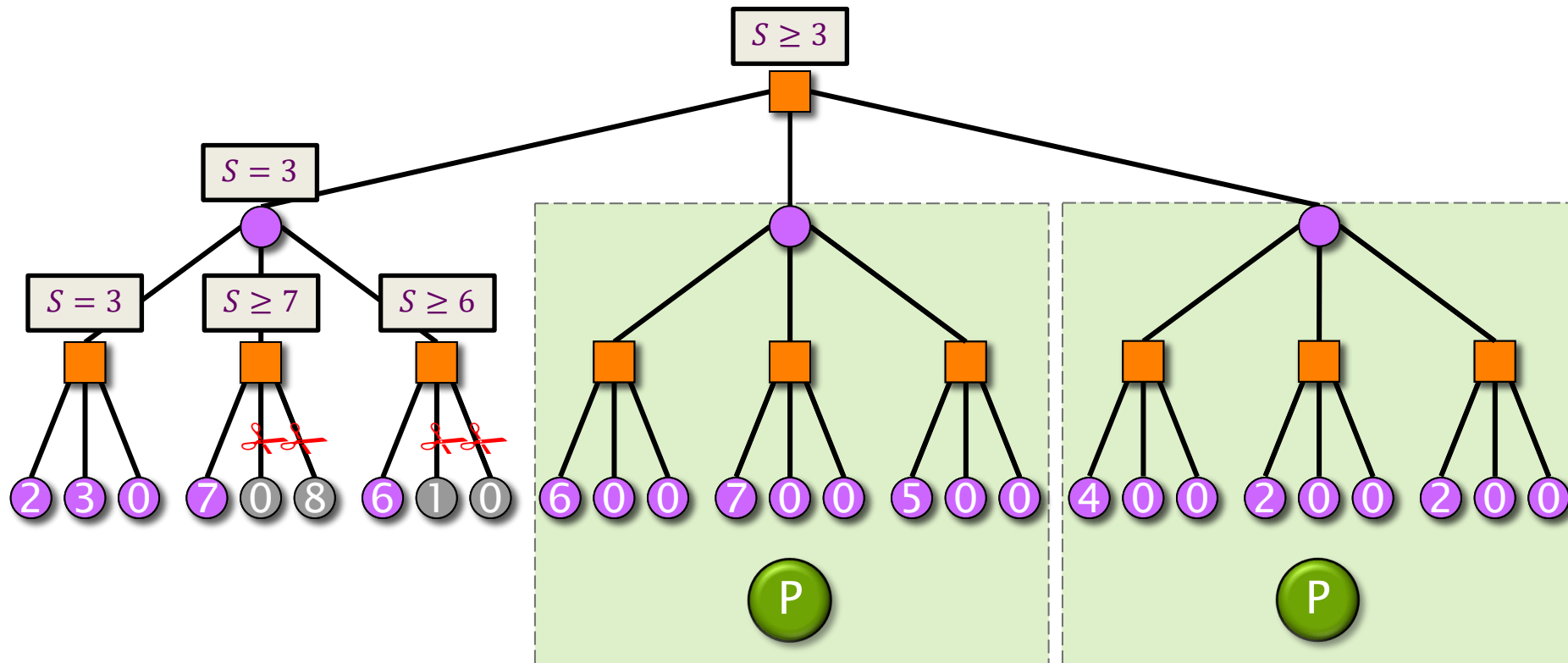


Parallel recursive
searches

Young Siblings Wait: Example

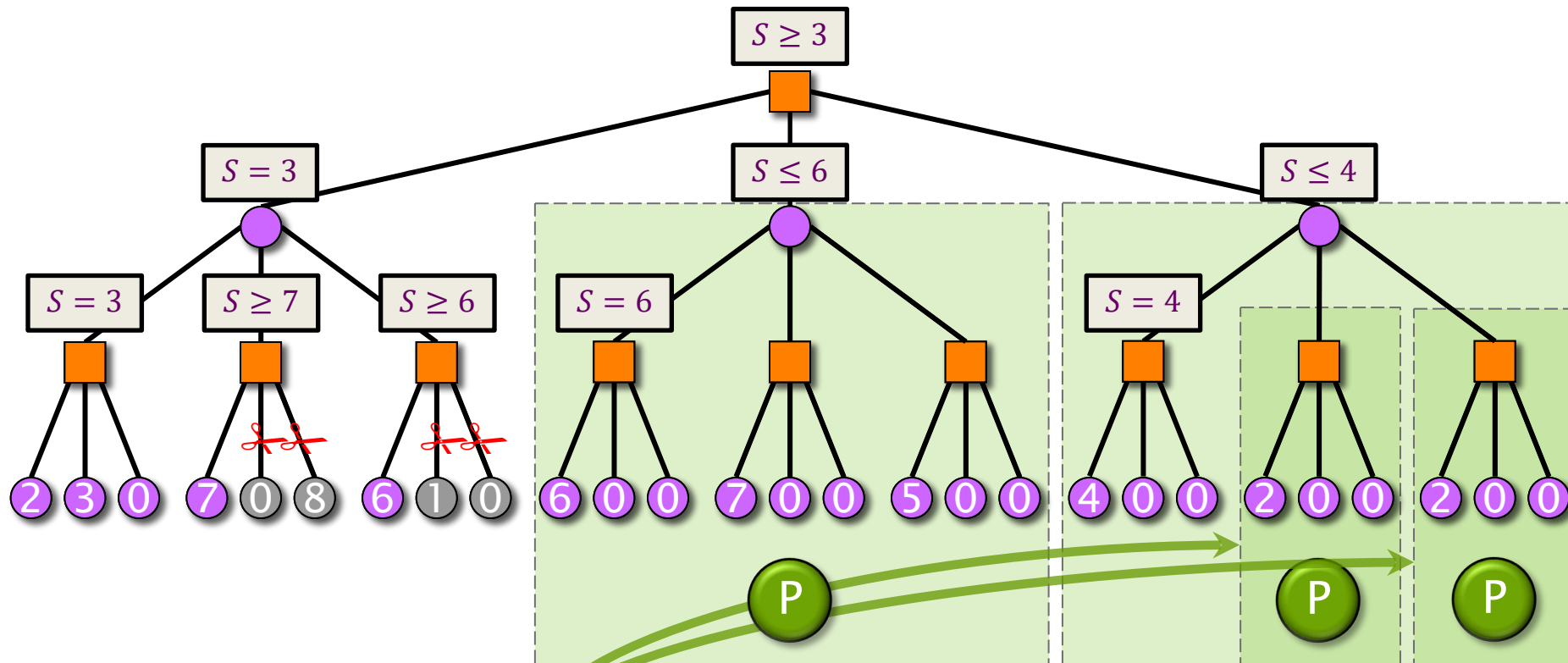


Young Siblings Wait: Example



Parallel recursive
searches

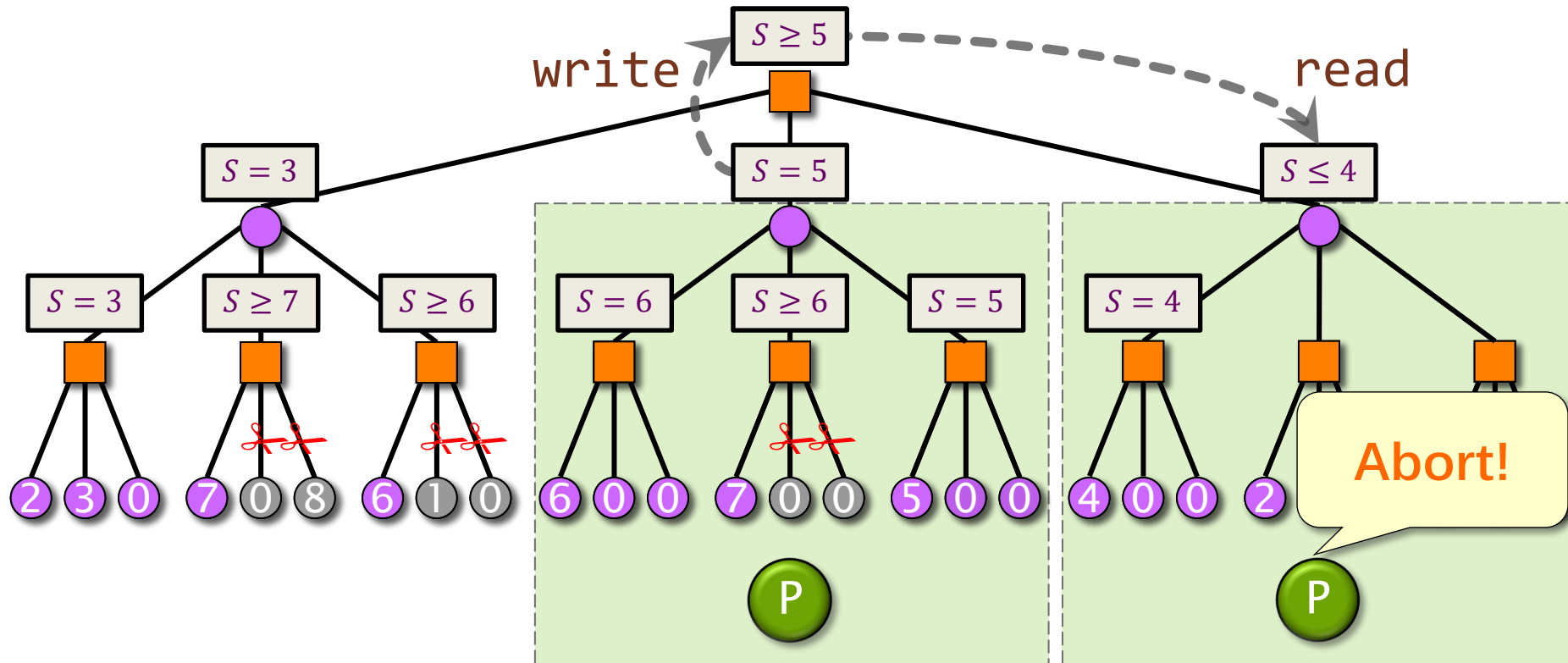
Young Siblings Wait: Example



A value from the second child of the root is not yet available to prune these searches.

Parallel recursive searches

Aborting Unnecessary Work



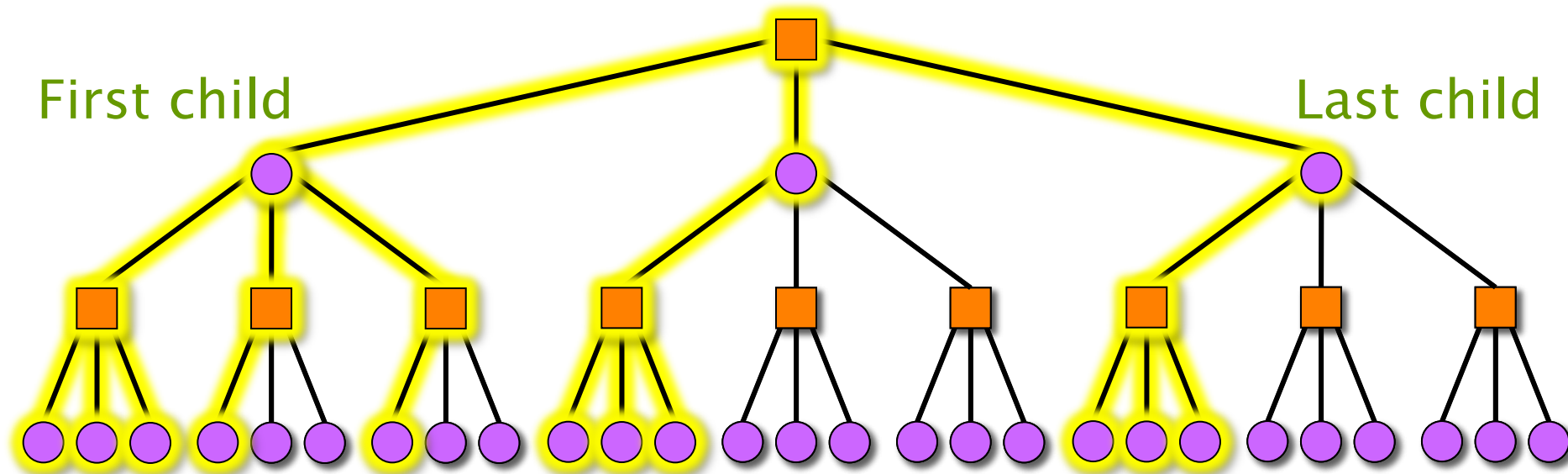
Possible idea: Nodes propagate their scores up the tree to keep alpha/beta values current.

- Poll up the tree and abort if possible.

Problem: Difficult to implement correctly.

Problem: Efficiency relies on lucky scheduling!

Wasted Work in Parallel Alpha-Beta



In practice, speculation in parallel alpha-beta search always wastes some work. Balance two conflicting goals:

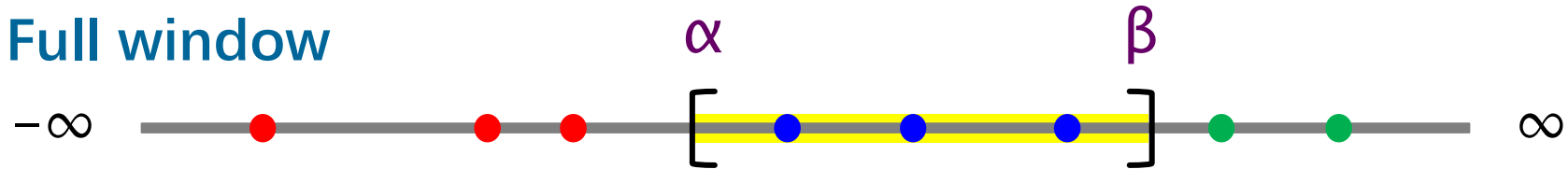
- Avoid unnecessary work — **work-first principle**.
- Generate sufficient parallelism to guarantee a good parallel speedup — **small span**.

JAMBOREE SEARCH

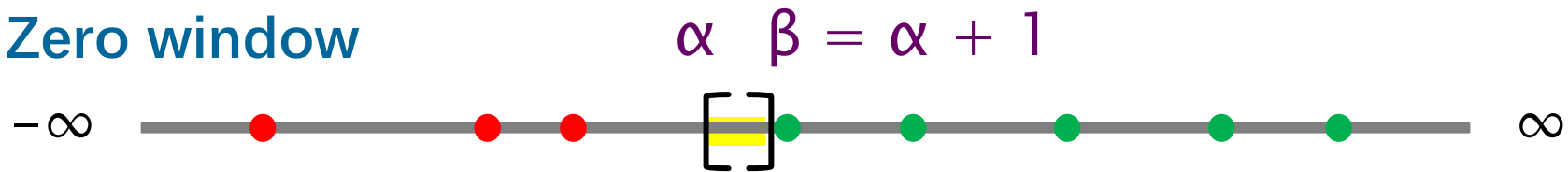


Zero- versus Full-Window Search

Full window



Zero window

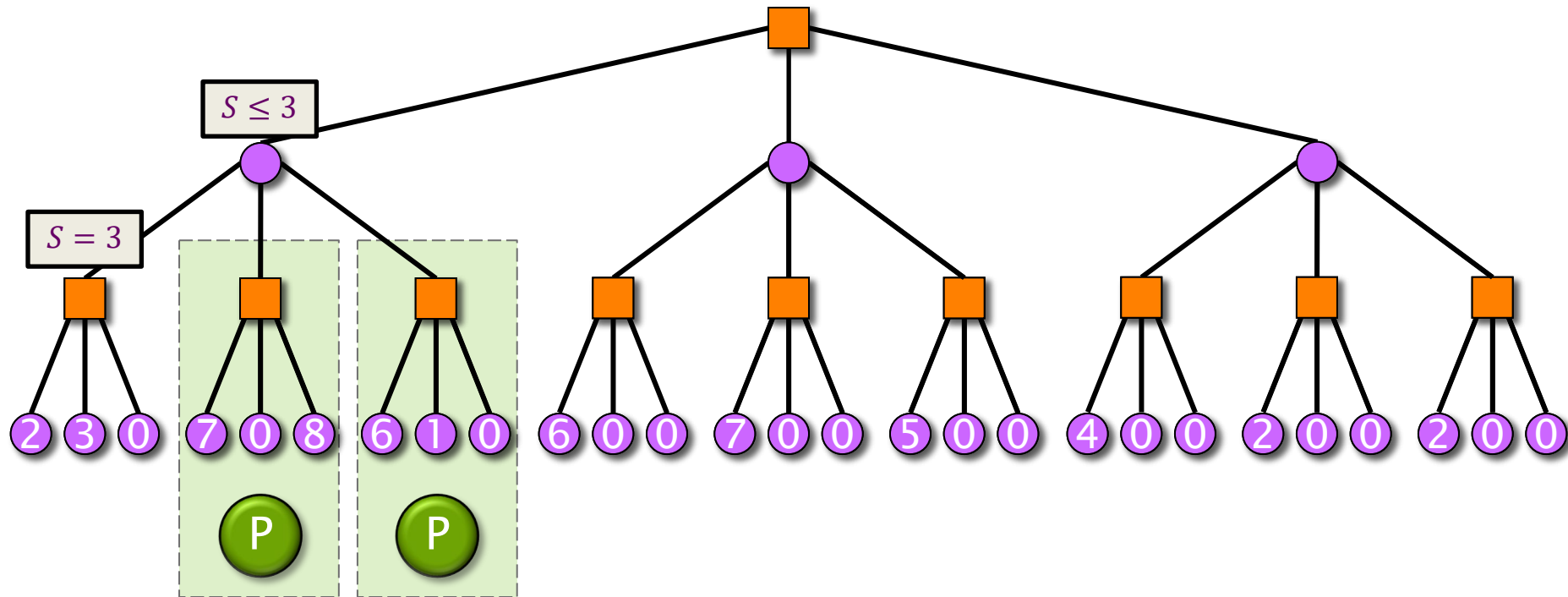


- A zero-window (misnomer) search essentially tests whether a score is less than or greater than α .
- It is more efficient than a full-window search, because it prunes more aggressively.

Jamboree Search [K94]

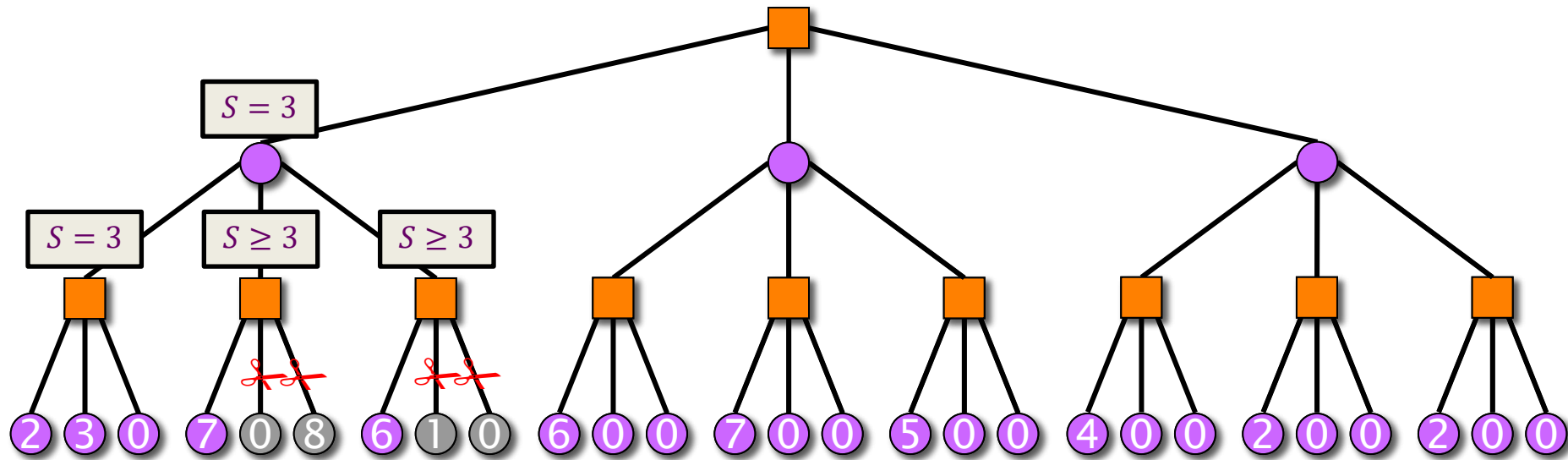
```
int jamboree(pos x, int alpha, int beta, int d) {
    if (d == 0 || is_leaf(x)) return static_eval(x);
    pos c[MAX_CHILDREN];
    int nc;
    gen_moves(x, c, &nc);
    int b = -jamboree(c[0], -beta, -alpha, d-1);
    if (b >= beta) return b;
    if (b > alpha) alpha = b;
    cilk_for (i = 1; i < nc; ++i) {
        int s = -jamboree(c[i], -alpha-1, -alpha, d-1);
        if (s > b) b = s;
        if (s >= beta) abort_and_return s;
        if (s > alpha) {
            // wait for completion of all previous iterations
            s = -jamboree(c[i], -beta, -alpha, d-1);
            if (s >= beta) abort_and_return s;
            if (s > alpha) alpha = s;
            if (s > b) b = s;
        }
        // mark the completion of the ith iteration
    }
    return b;
}
```

Jamboree Search: Example

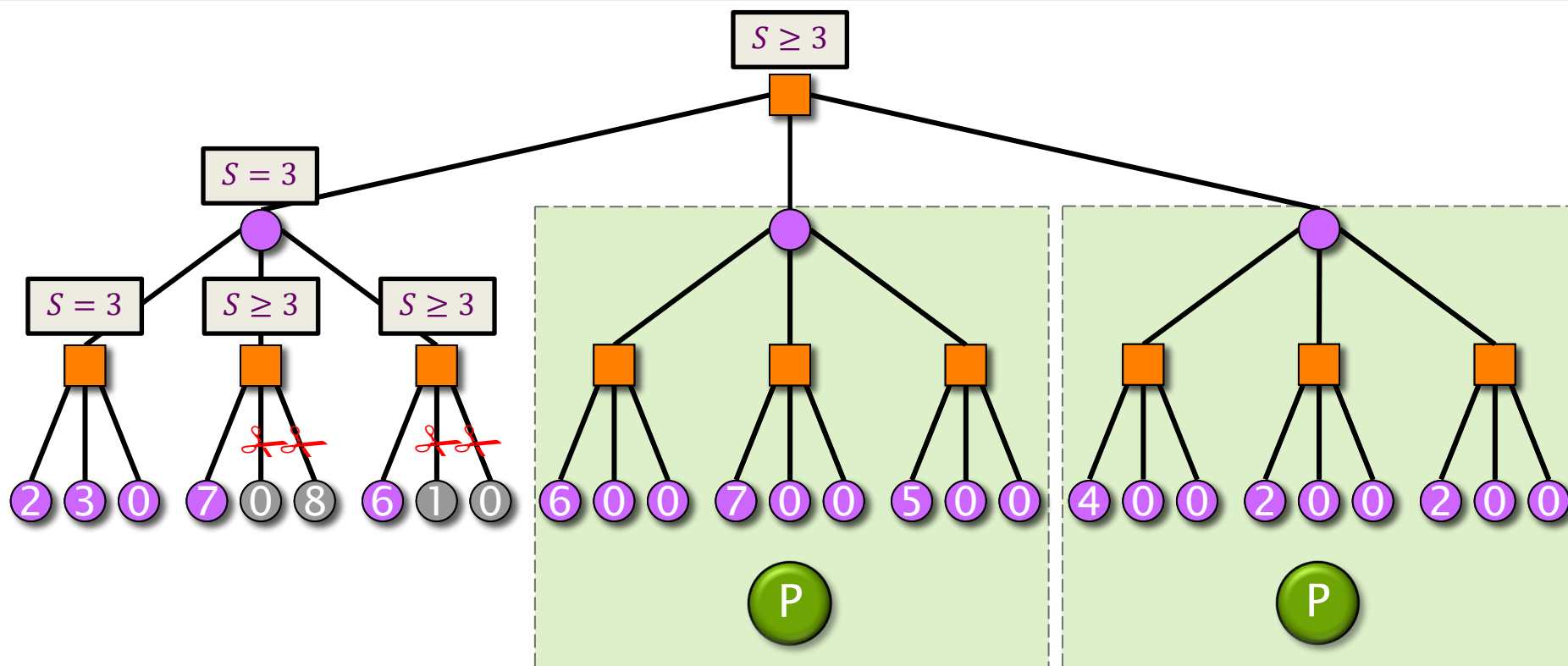


Recursive zero-window
search for $S \geq 3$.

Jamboree Search: Example

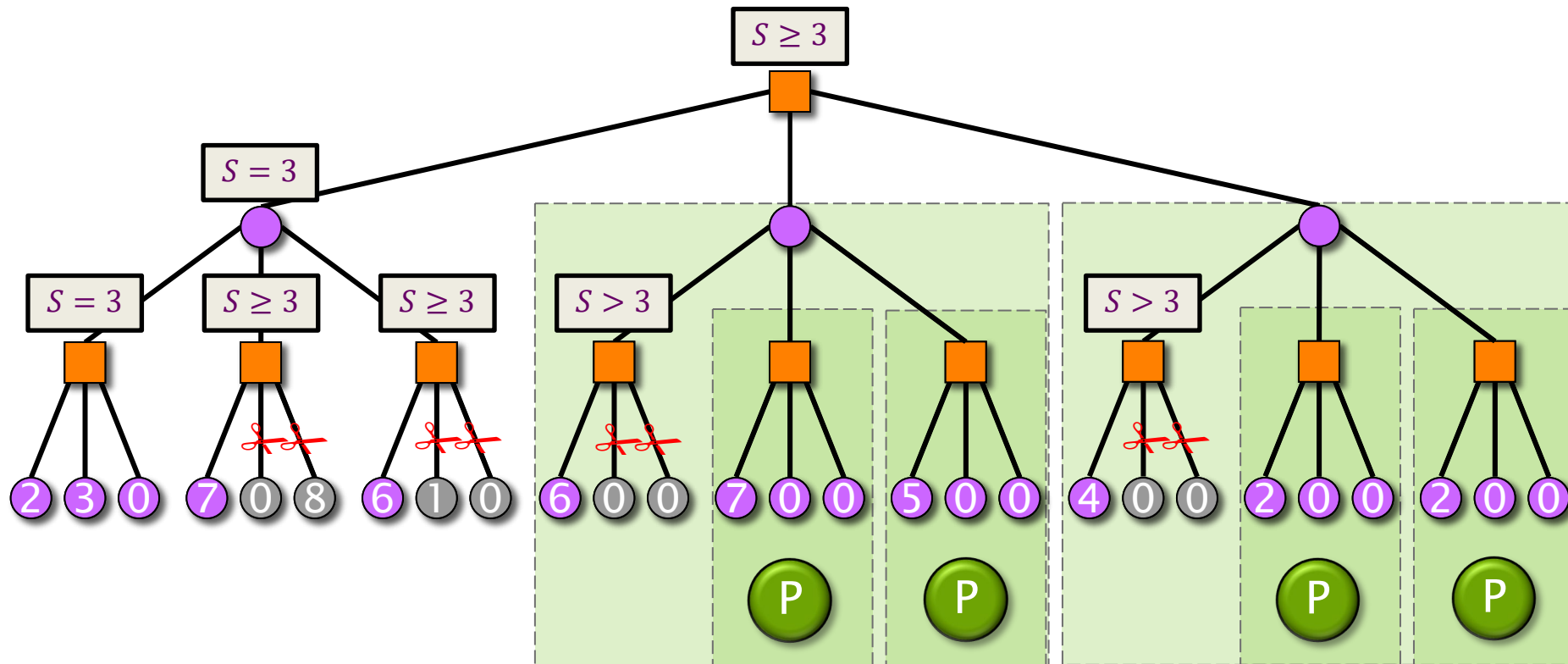


Jamboree Search: Example



Recursive zero-window
search for $S \leq 3$.

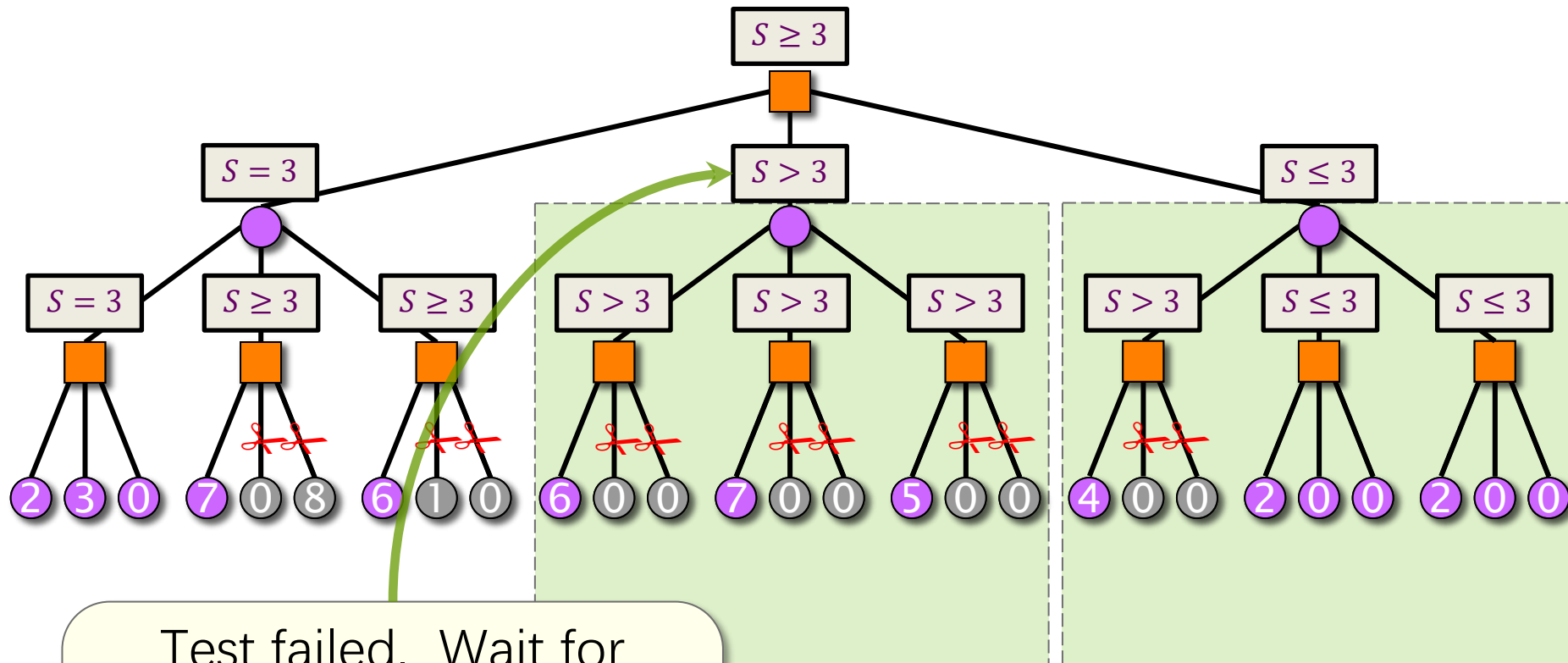
Jamboree Search: Example



Recursive zero-
window search
for $S \leq 3$.

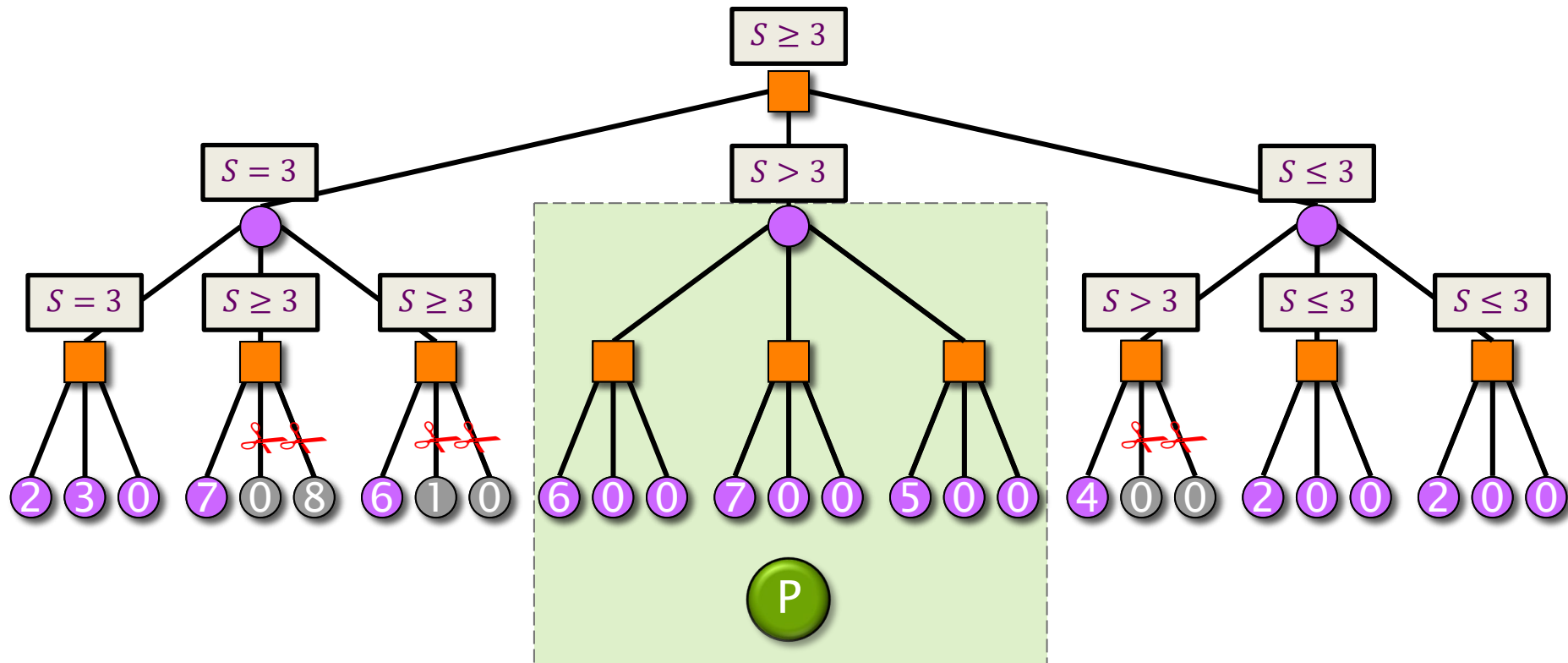
Recursive zero-
window search
for $S \leq 3$.

Jamboree Search: Example

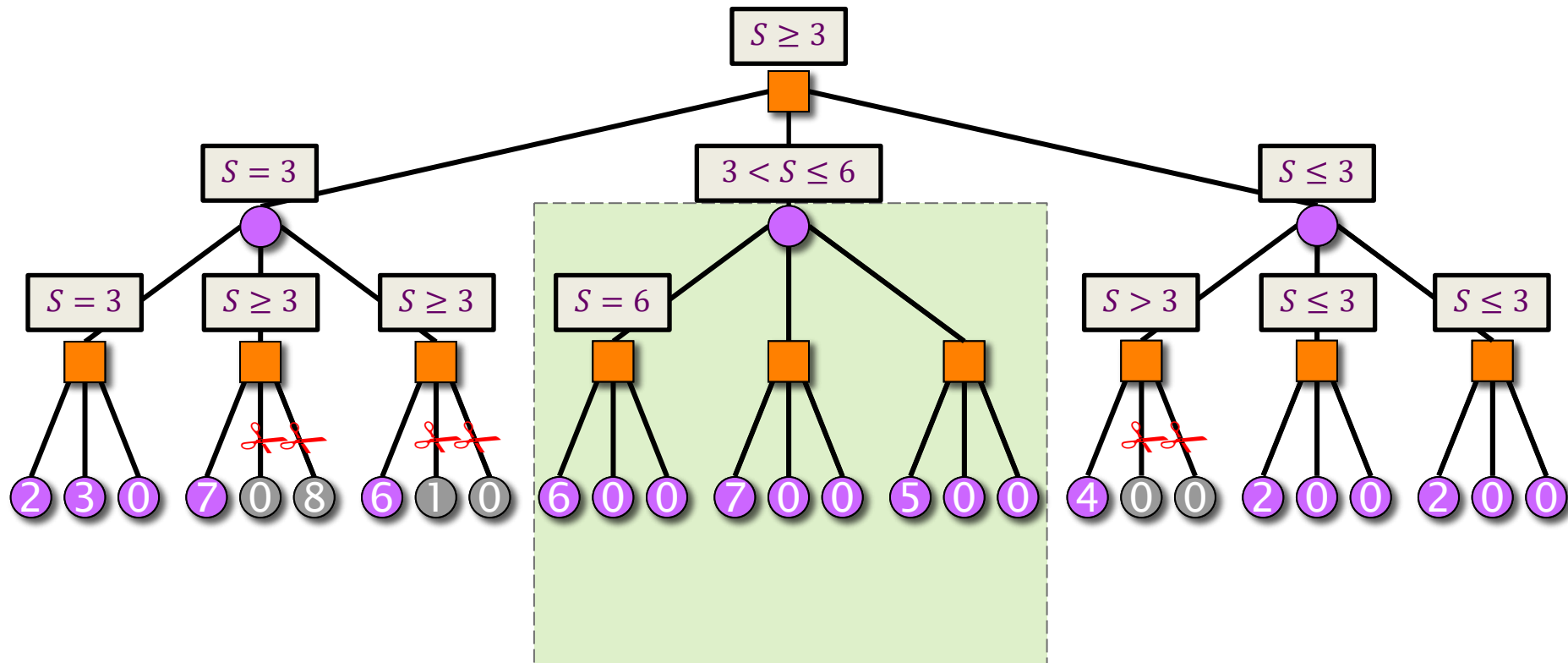


Test failed. Wait for preceding children to finish, then recursively evaluate this subtree.

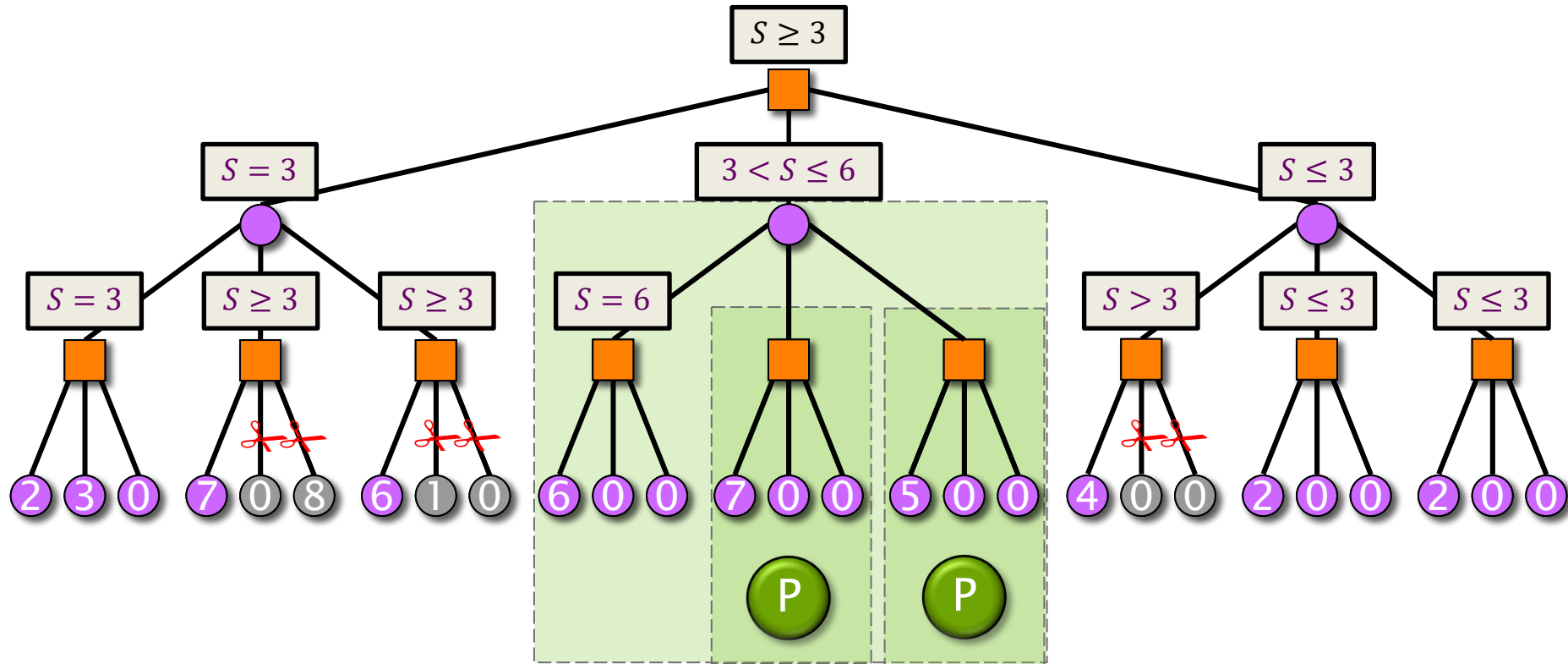
Jamboree Search: Example



Jamboree Search: Example

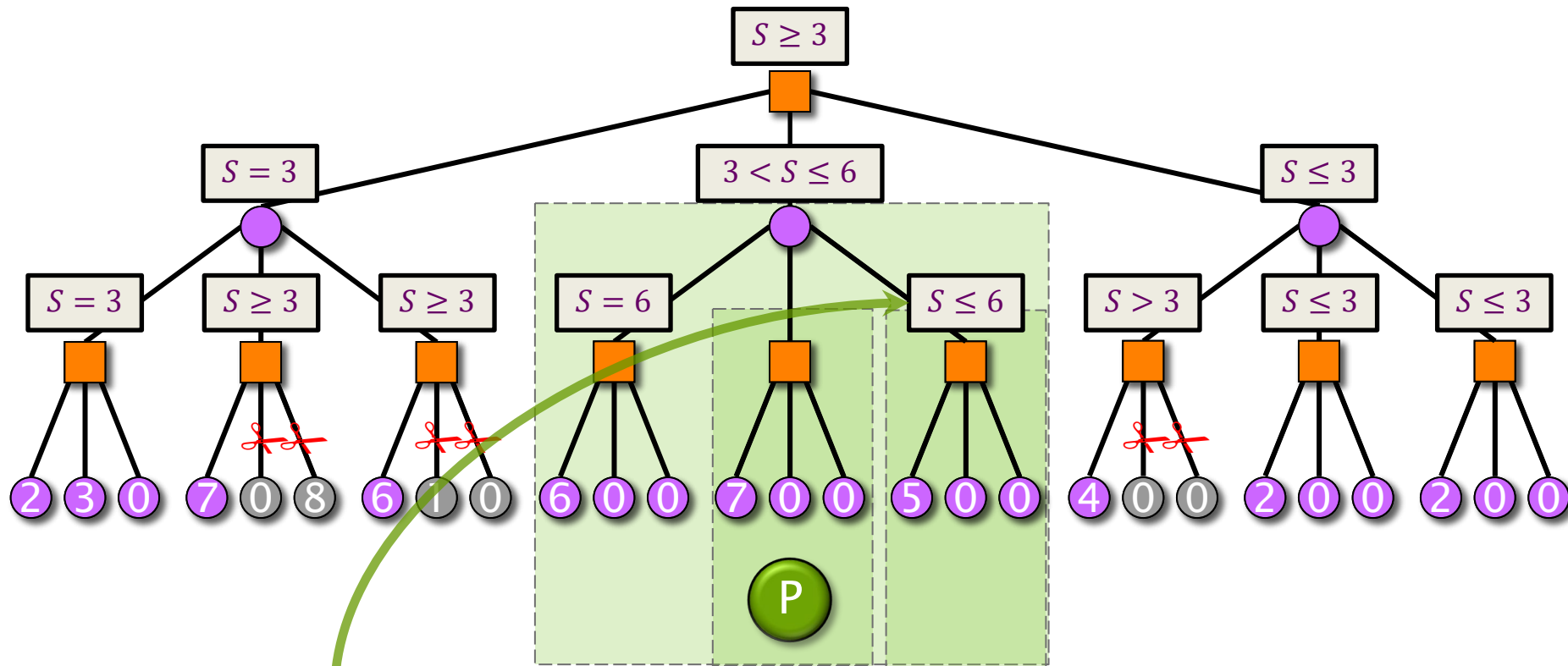


Jamboree Search: Example



Recursive zero-
window search
for $S \geq 6$.

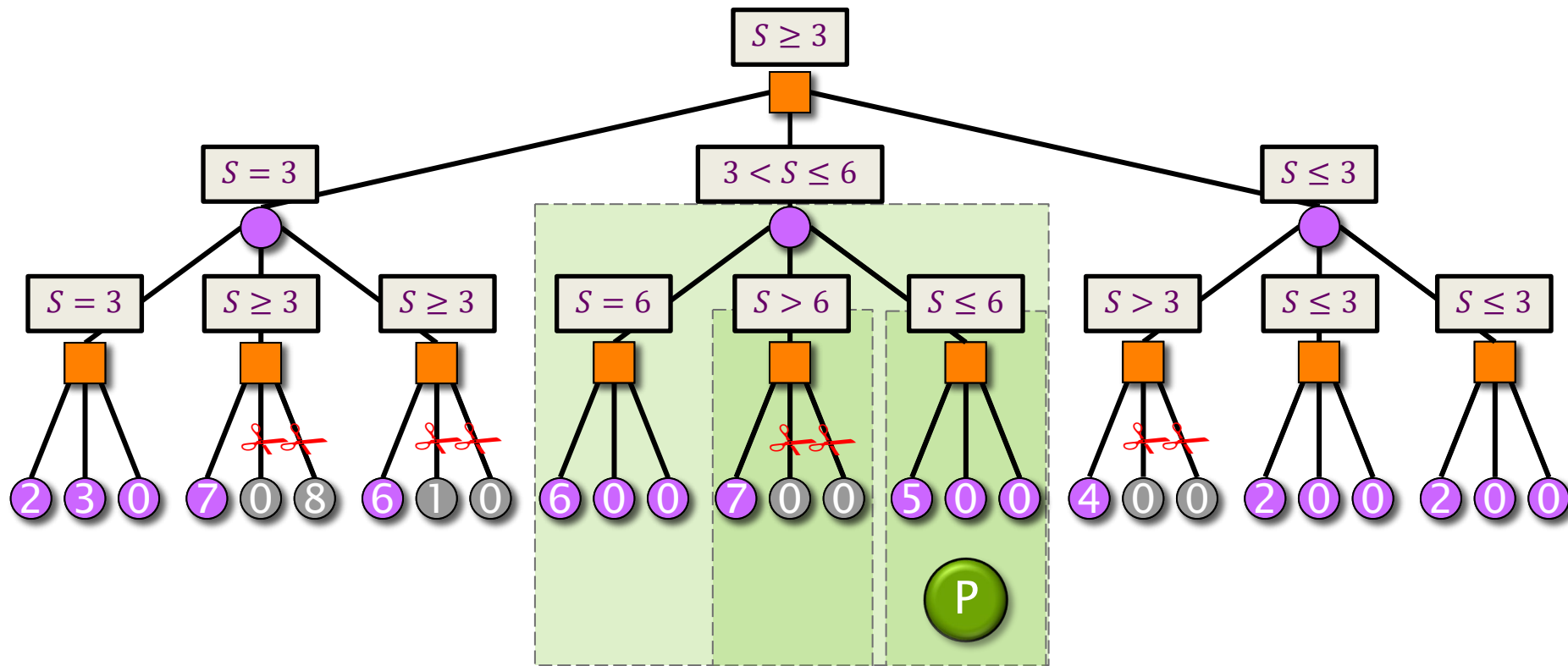
Jamboree Search: Example



Test failed. Wait for preceding children to finish.

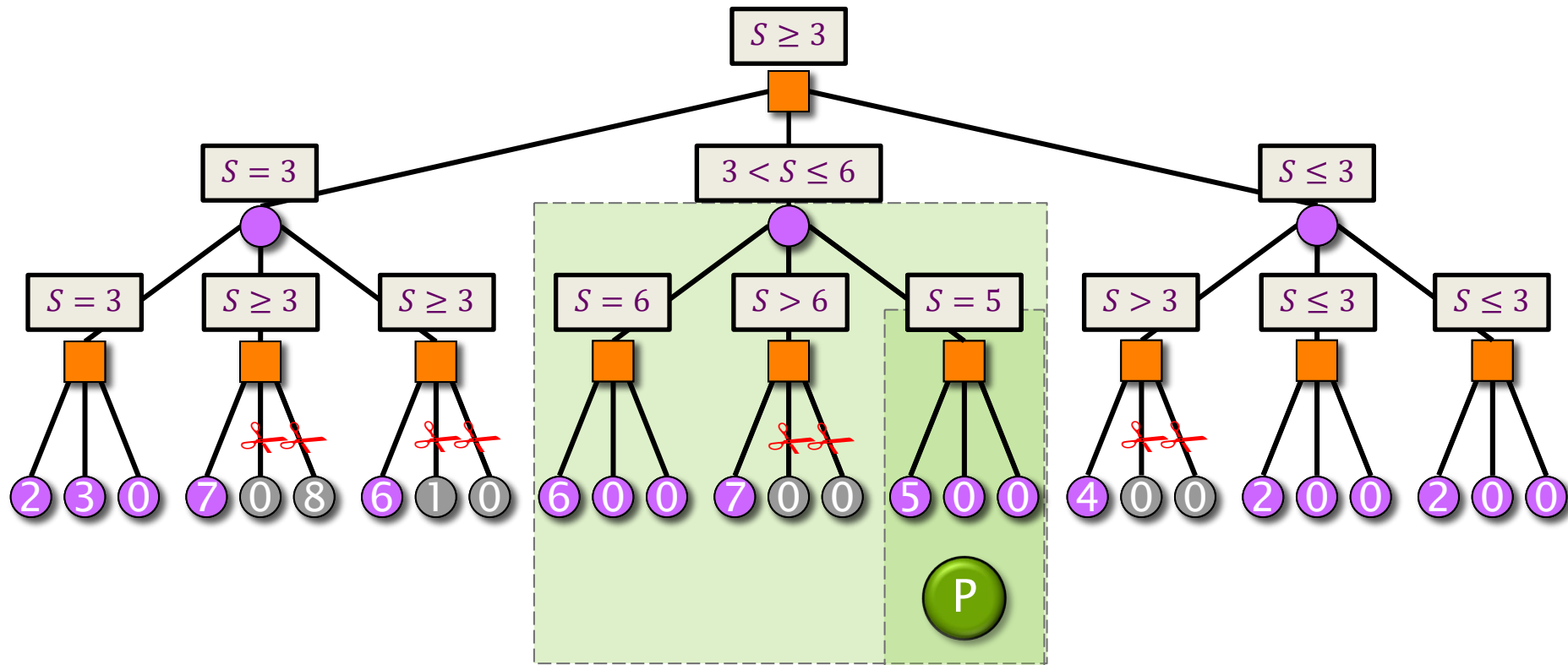
Recursive zero-window search for $S \geq 6$.

Jamboree Search: Example



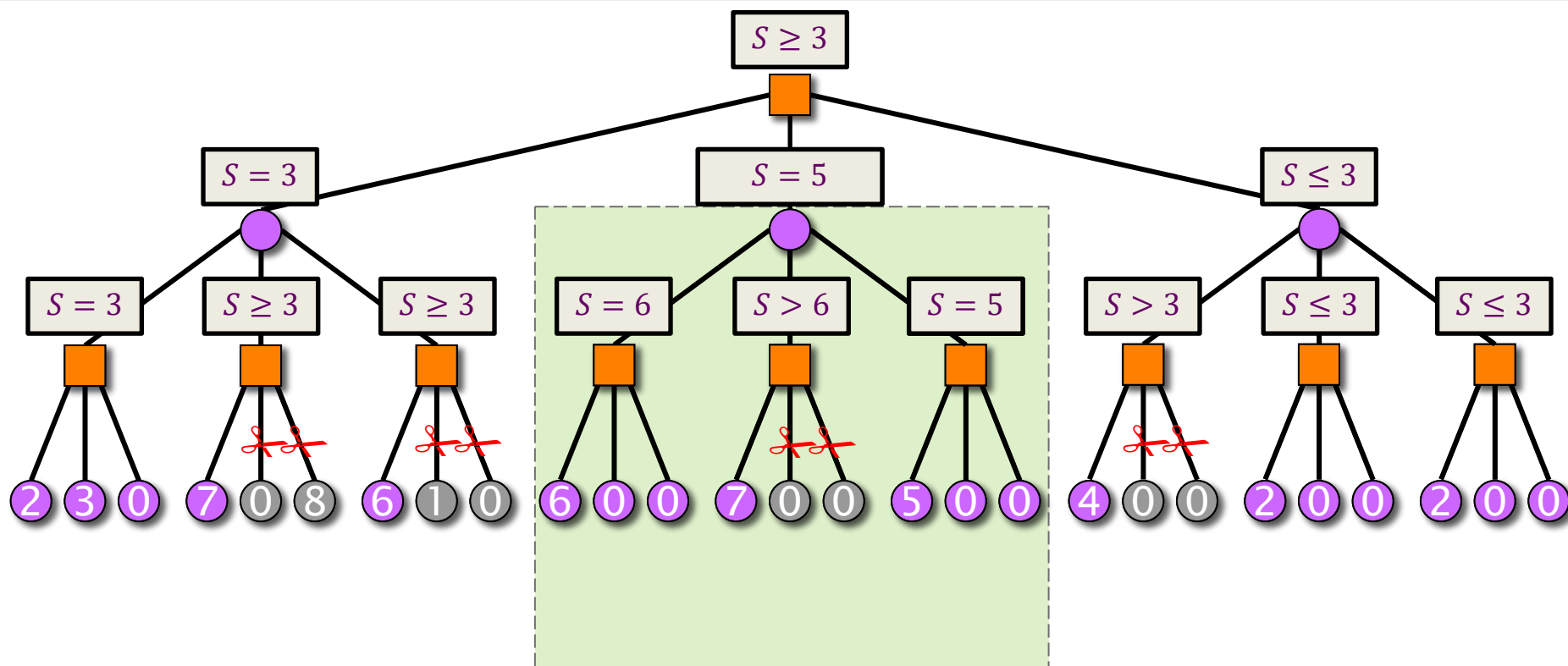
Recursive full-
window search.

Jamboree Search: Example



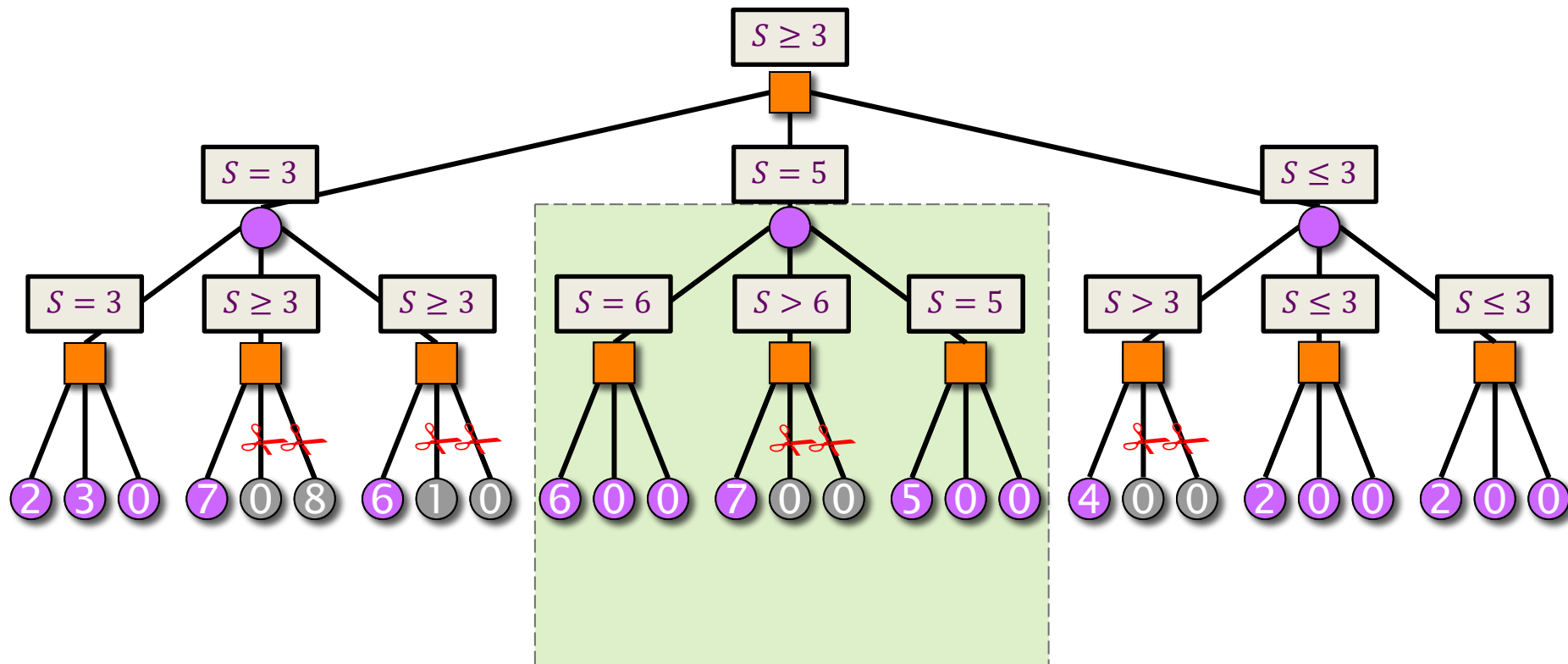
Recursive full-
window search.

Jamboree Search: Example



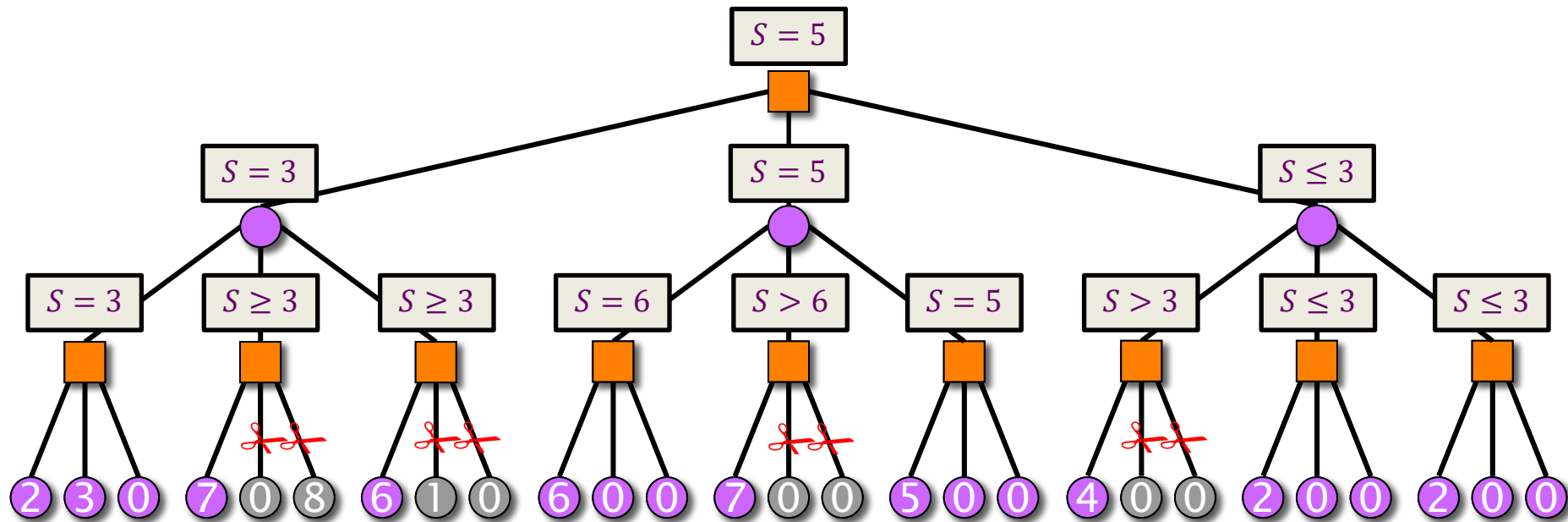
Recursive full-
window search.

Jamboree Search: Example



Recursive full-
window search.

Jamboree Search: Example



Jamboree Pseudocode

```
int jamboree(pos x, int alpha, int beta, int d) {
    if (d == 0 || is_leaf(x)) return static_eval(x);
    pos c[MAX_CHILDREN];
    int nc;
    gen_moves(x, c, &nc);
    int b = -jamboree(c[0], -beta, -alpha, d-1);
    if (b >= beta) return b;
    if (b > alpha) alpha = b;
    cilk_for (i = 1; i < nc; ++i) {
        int s = -jamboree(c[i], -alpha-1, -alpha, d-1);
        if (s > b) b = s;
        if (s >= beta) abort_and_return s;
        if (s > alpha) {
            // wait for completion of all previous iterations
            s = -jamboree(c[i], -beta, -alpha, d-1);
            if (s >= beta) abort_and_return s;
            if (s > alpha) alpha = s;
            if (s > b) b = s;
        }
        // mark the completion of the ith iteration
    }
    return b;
}
```


Abort Mechanism

```
typedef struct searchNode {  
    struct searchNode *parent;  
    position_t position;  
    bool abort_flag;  
} searchNode;
```

- If a returning child would cause a beta cutoff, it sets its parent's **abort_flag**.
- During the search, each node polls up the search tree to see whether any ancestor nodes are signaling an abort.
 - If so, it quits the search.
- After the last child returns, the parent can return.

Getting Started with Parallel Leiserchess

The Leiserchess codebase is already structured to support a simple parallelization of `scout_search()`.

`scout_search.c`

```
static score_t scout_search(searchNode* node, int depth,
                             uint64_t* node_count_serial) {
    ...
    cilk_for (int mv_index = 0; mv_index < num_of_moves;
              mv_index++) {
        // Get the next move from the move list.
        int local_index = number_of_moves_evaluated++;
        move_t mv = get_move(move_list[local_index]);
        ...
    }
    ...
}
```

The resulting search is not the same as Jamboree search, but it should help you to get started.

Races in Parallel Leisearchess

Simply parallelizing the loop produces racy code:

- best-move history table,
- killer table,
- transposition table.

Consider how you might address them:

- synchronize, for example, by locking or using atomics;
- make local copies;
- use reducers;
- risk it!



**GOOD LUCK ON THE HOME
STRETCH!**