Othello AI Tournament!

CS2: Intro. to Programming Methods
Weeks 9-10
Wednesday, March 9, 2016

Administrative notes

- Next assignment is due Tuesday, March 15 at 17:00
- Tournament cutoff is Monday, March 14, at 23:59
  - TAs need some time to set things up
  - We may accept tournament submissions after that, but there's no guarantee

Administrative notes

- The tournament itself will be held in Annenberg 104 (the lab)
- Tournament starts Tuesday March 17 in the afternoon
- The games themselves will be displayed on the lab computers as they are played
- Come by and watch!

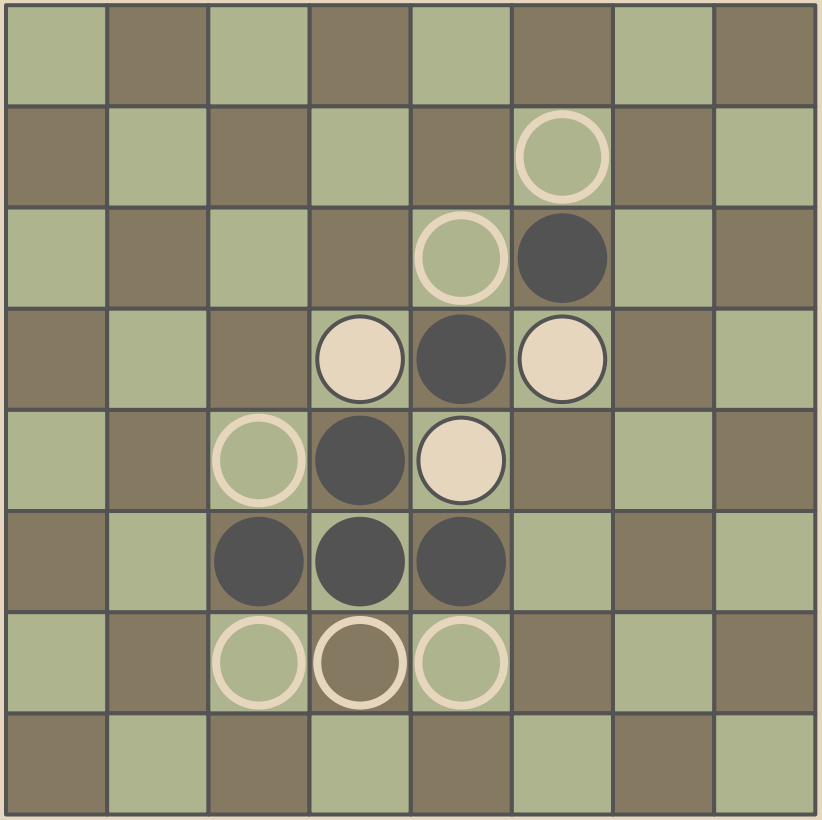
Othello++: additional considerations

- Board
  - Bitboards
  - You’re on your own for this one...
- Evaluation
- Search

Othello++: additional considerations

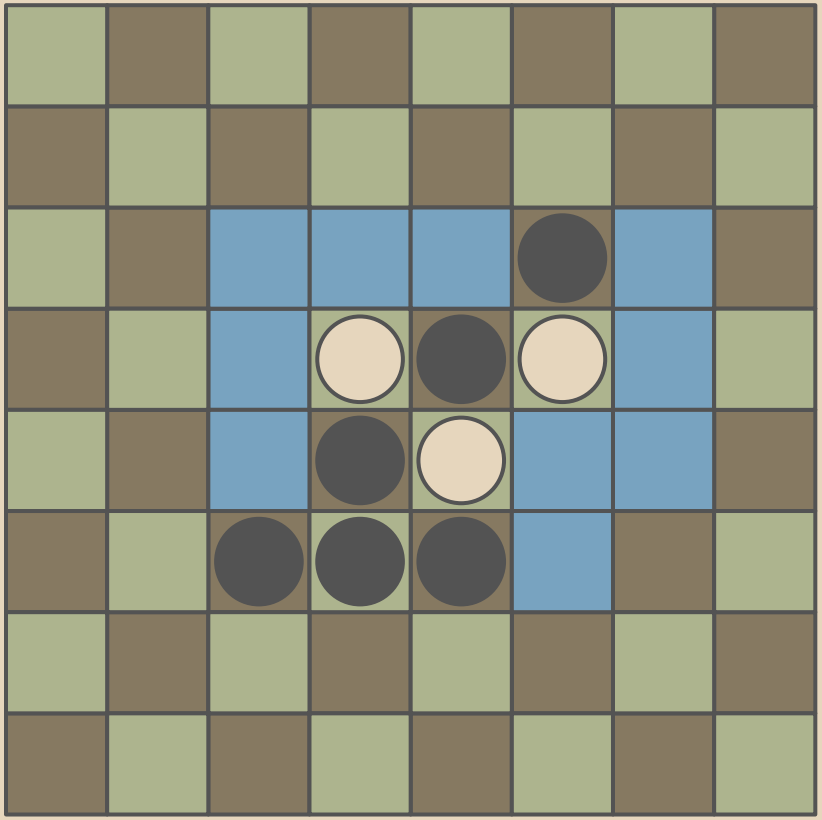
- Mobility
- Frontier squares
- Alpha-beta pruning
  - Move ordering
- Transposition tables
- Iterative deepening
- Opening books
- Endgame solver

Mobility



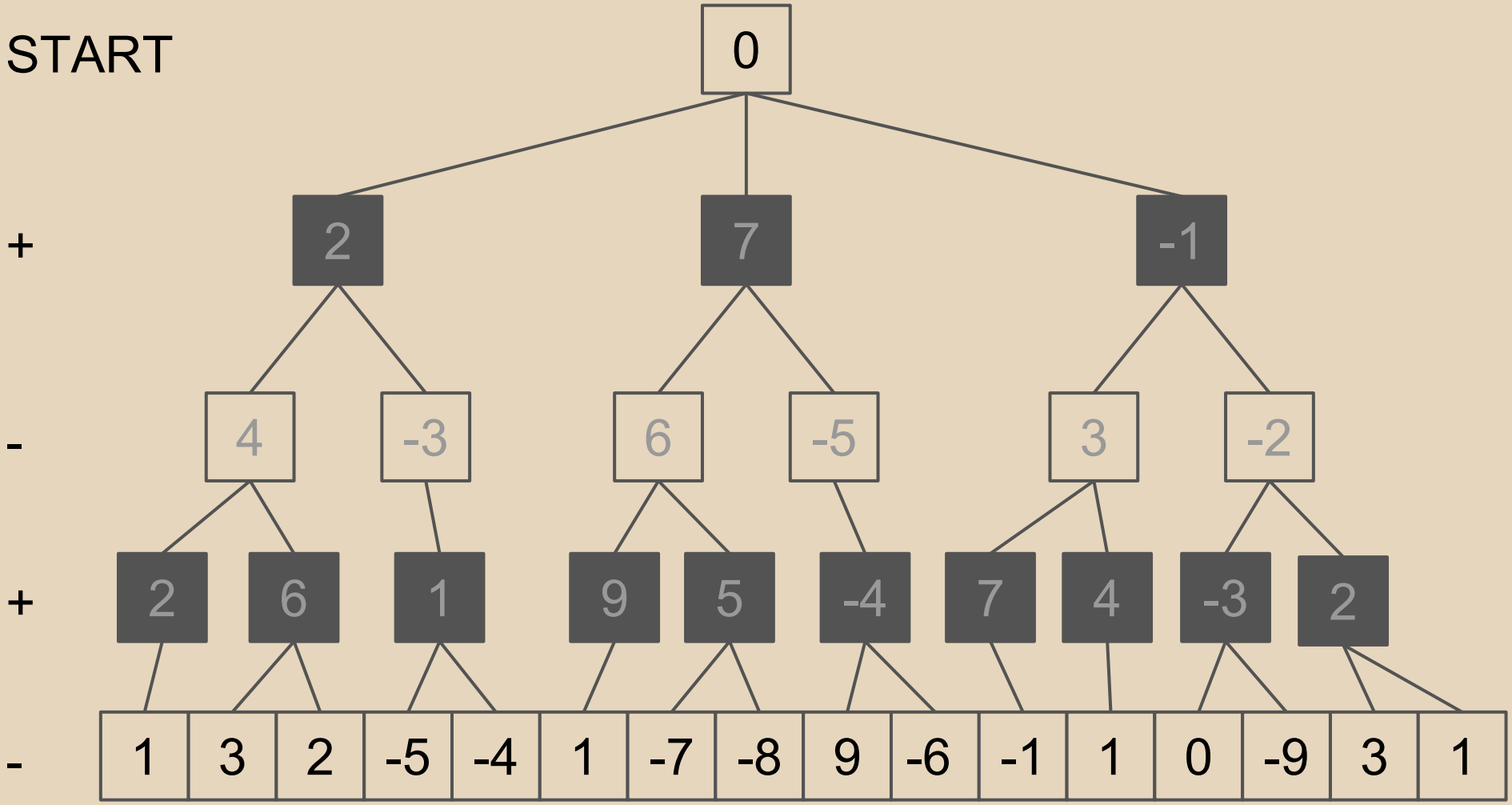
- The number of moves available
- Generally a good thing
  - more choices = better chance of a good choice
  - maximize for self
  - minimize for the other player
- Slow to calculate, but very important!

Frontier squares



- The number of open squares adjacent to our own pieces
- Generally bad for us
  - more frontier squares = more potential points for attack
  - minimize for self
  - maximize for opponent

Minimax again!



Alpha-beta pruning

- We can do better than exhaustive search.
- Store two numbers (alpha and beta) for each node as we're doing our depth-first search
  - Alpha: the maximum score we are assured of so far
  - Beta: the minimum score our opponent is assured of so far
- Note: minimax is alpha-beta with no beta

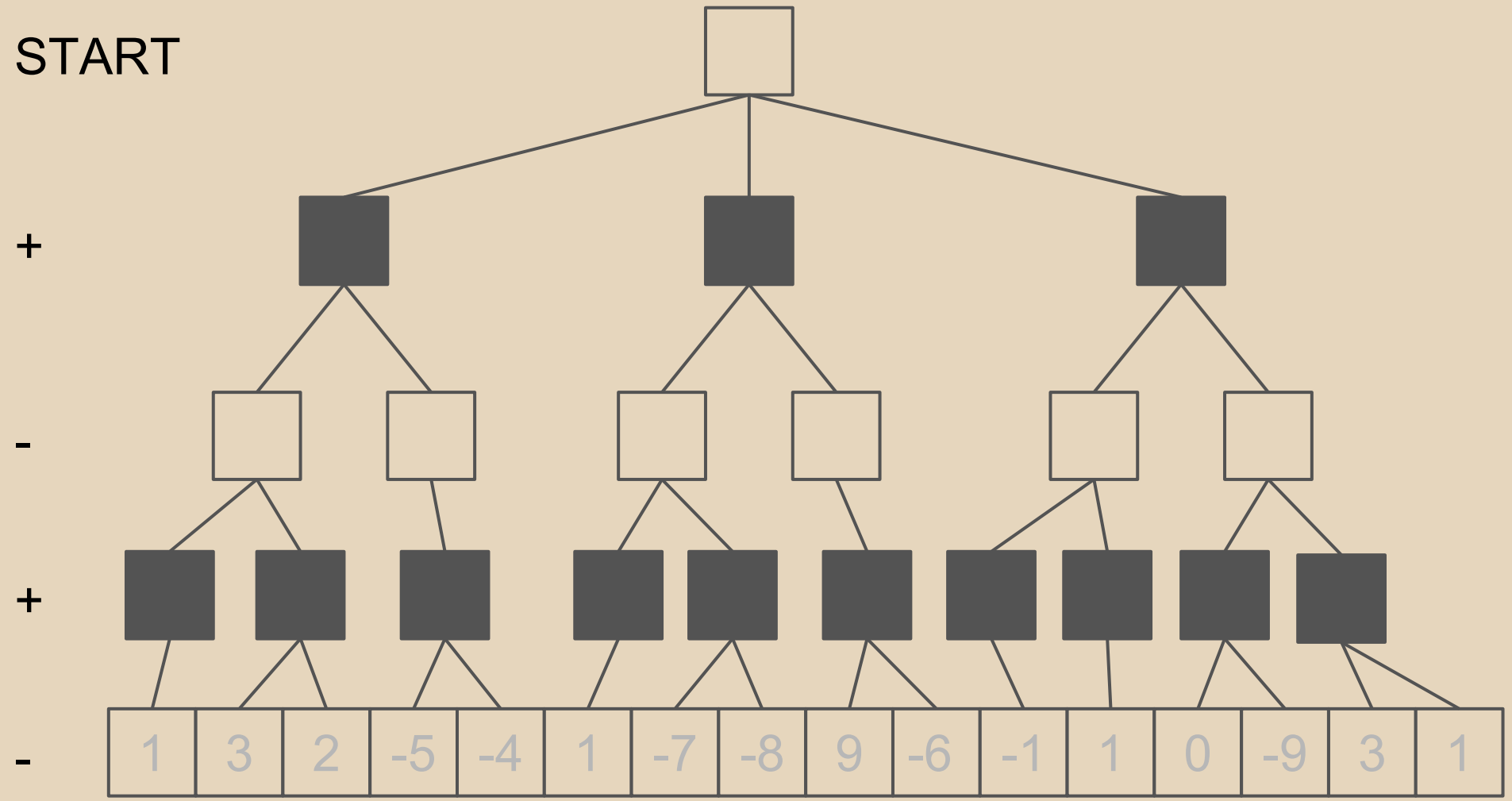
Alpha-beta pruning

- What if we can make a move such that alpha > beta?
  - Then the opponent should never have let us get here in the first place
- What if opponent can make a move such that beta < alpha?
  - Then we should never have let the opponent get there in the first place
- In either case, we can stop evaluating the current subtree

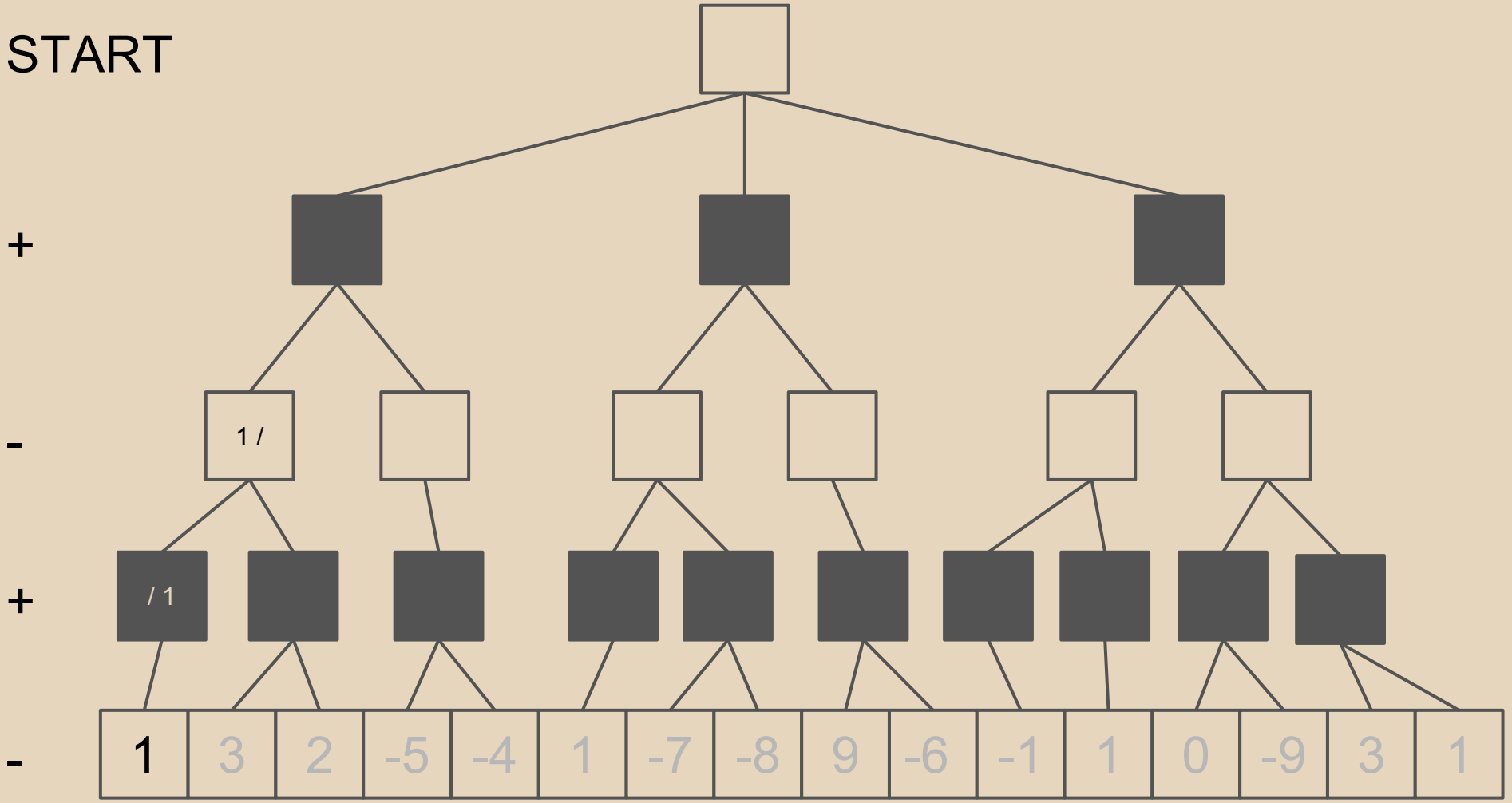
Computing alpha-beta in practice

- My turn?
  - I'm trying to maximize
  - Return alpha = max(alpha, beta $\forall$ children)
- Opponent's turn?
  - Opponent wants to minimize
  - Return beta = min(beta, alpha $\forall$ children)
- Recursive algorithm
  - If it's my turn for a given node, it's the opponent's turn for all the children
  - Inherit alpha/beta values from parent
  - Base case: bottom of tree (where value is given)
  - Initial call: alpha, beta = INFINITY

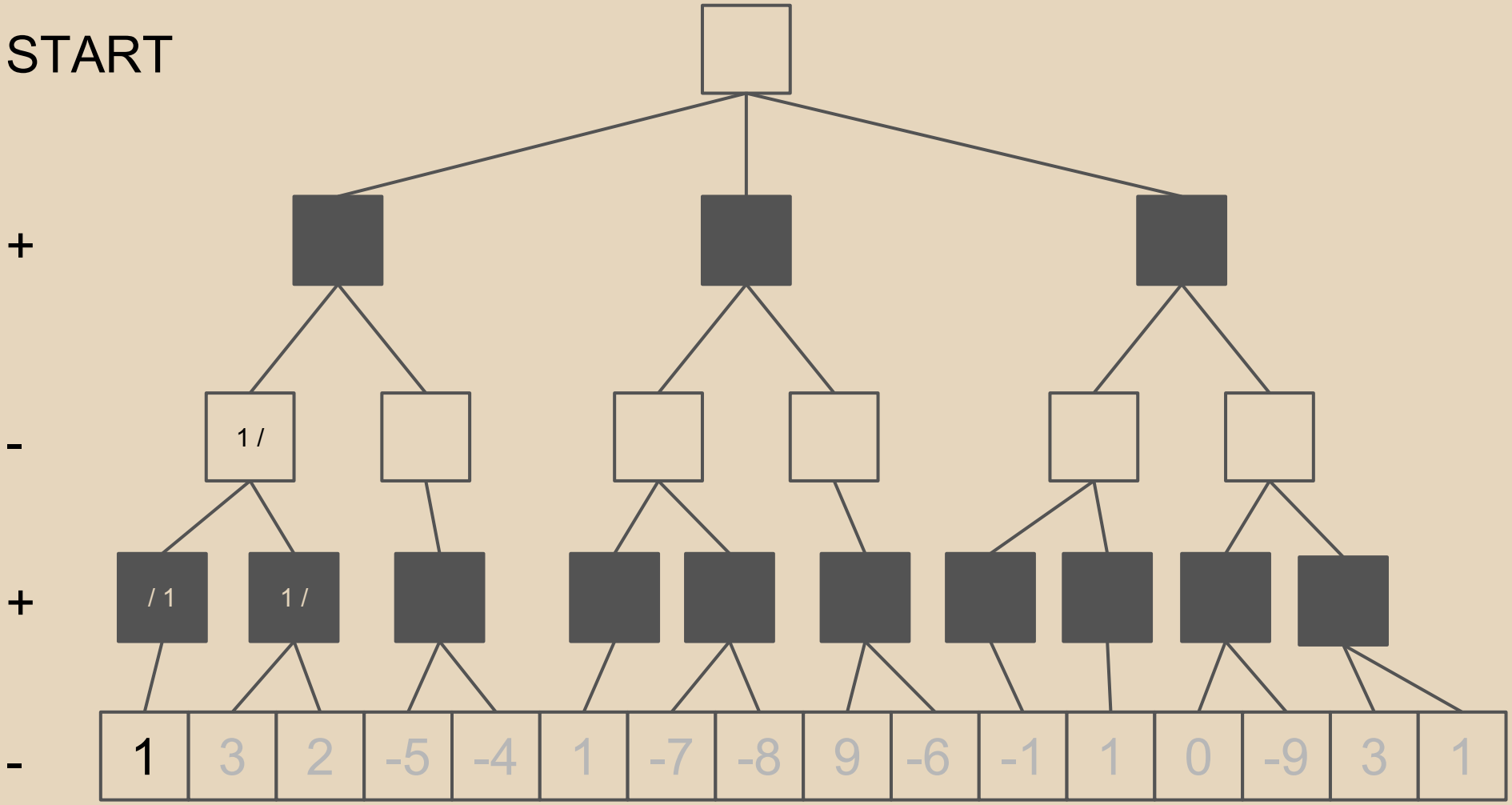
Alpha-beta example



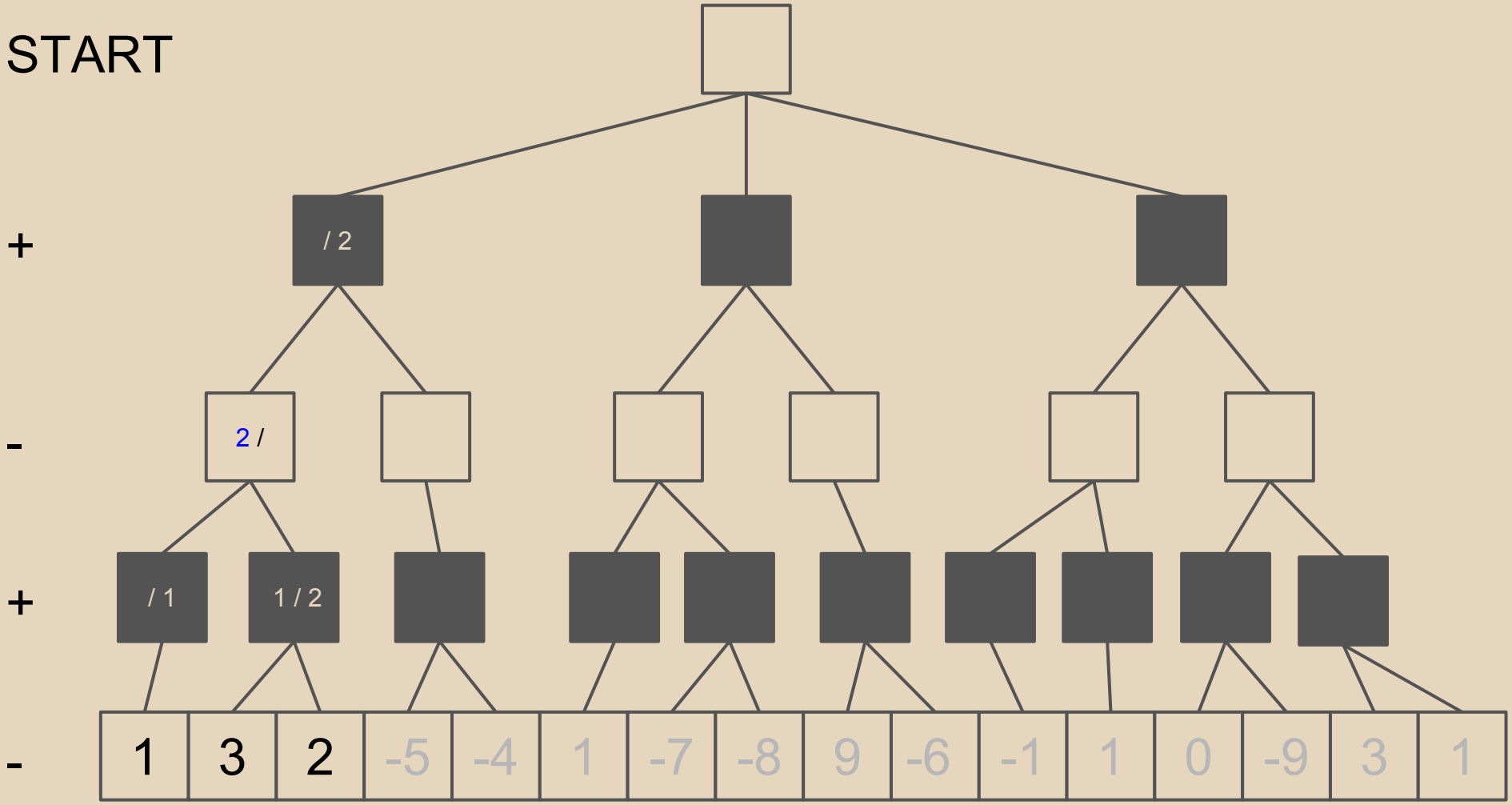
Now start searching...



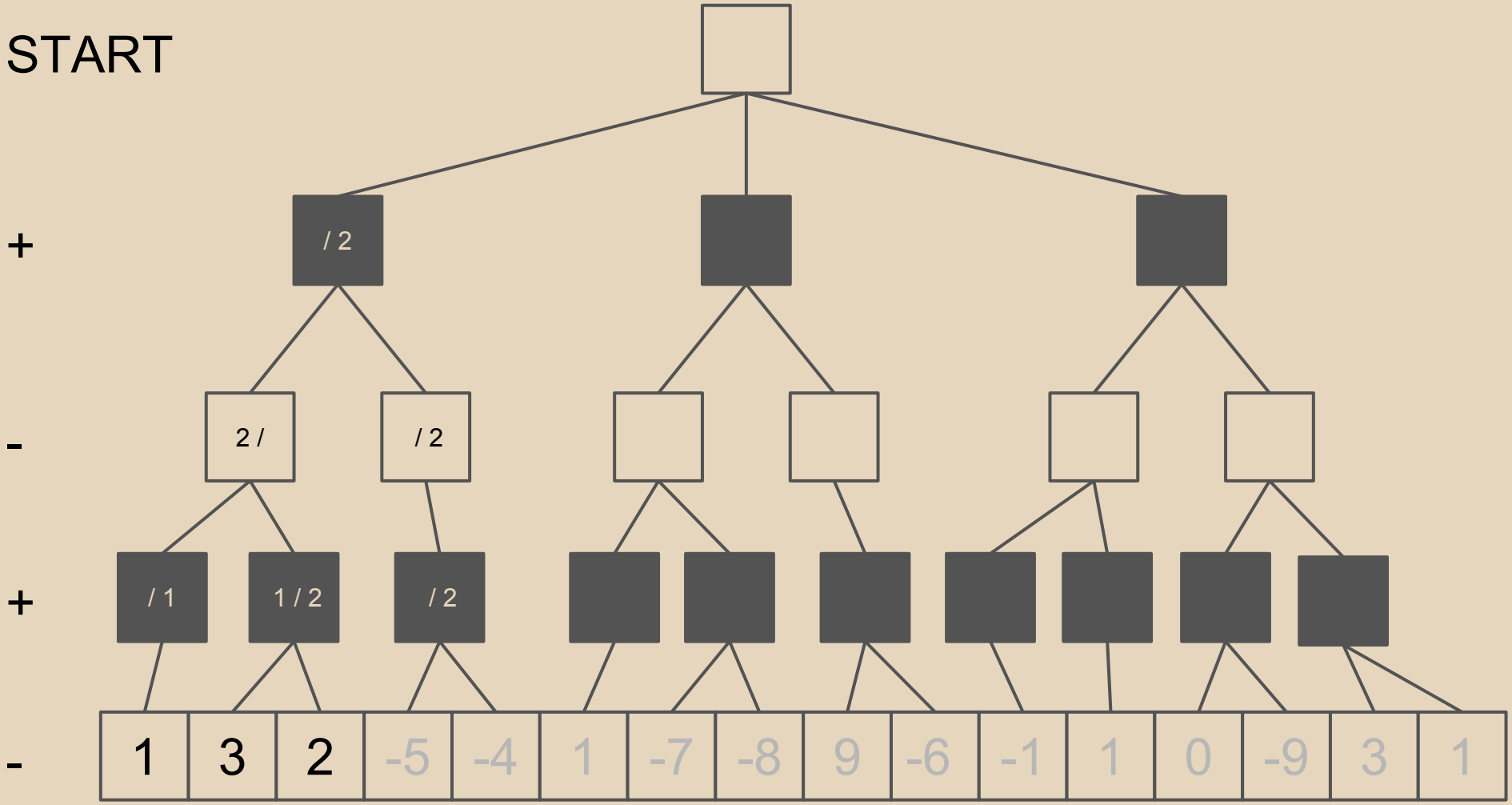
Now start searching...



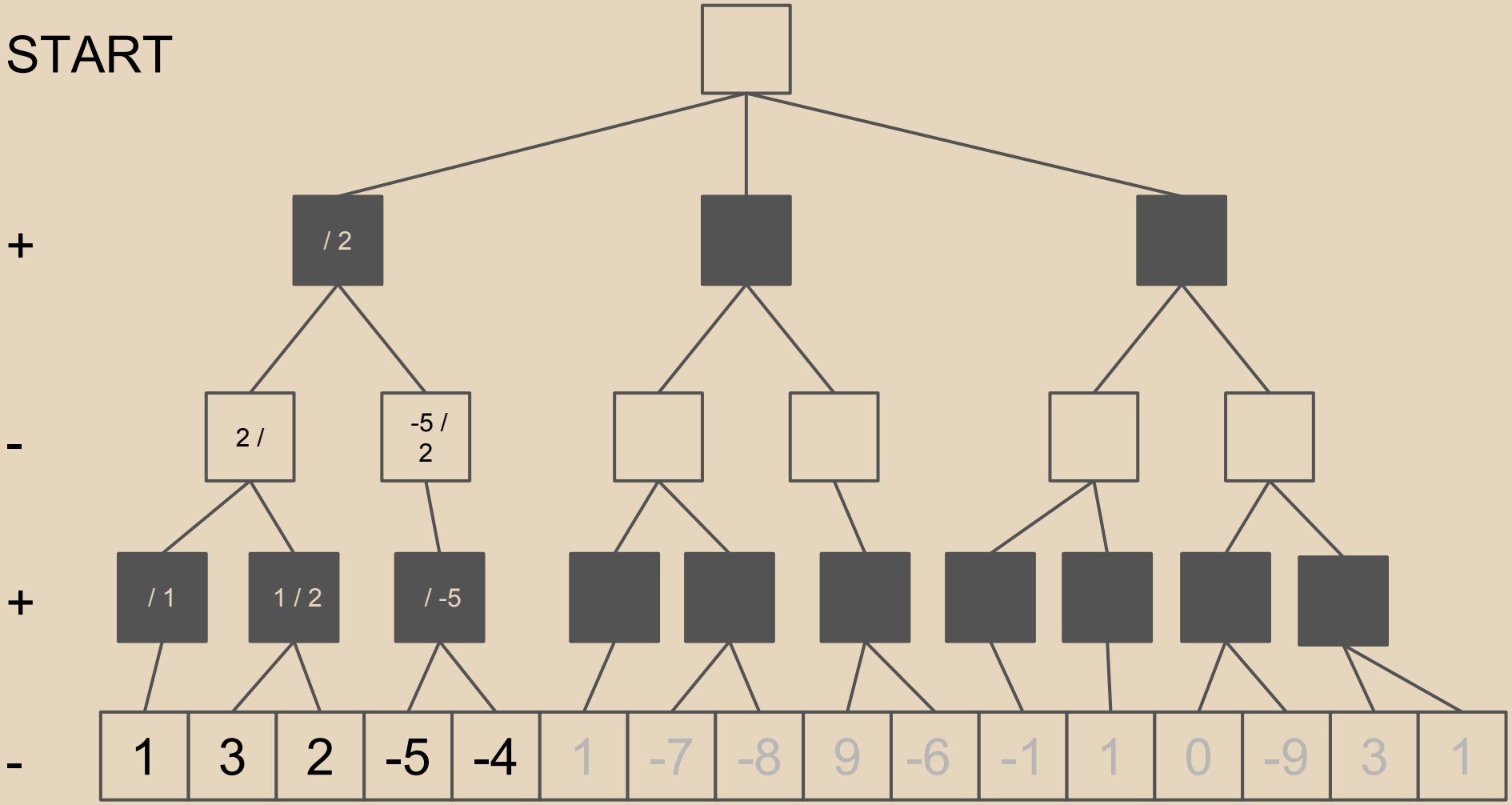
Now start searching...



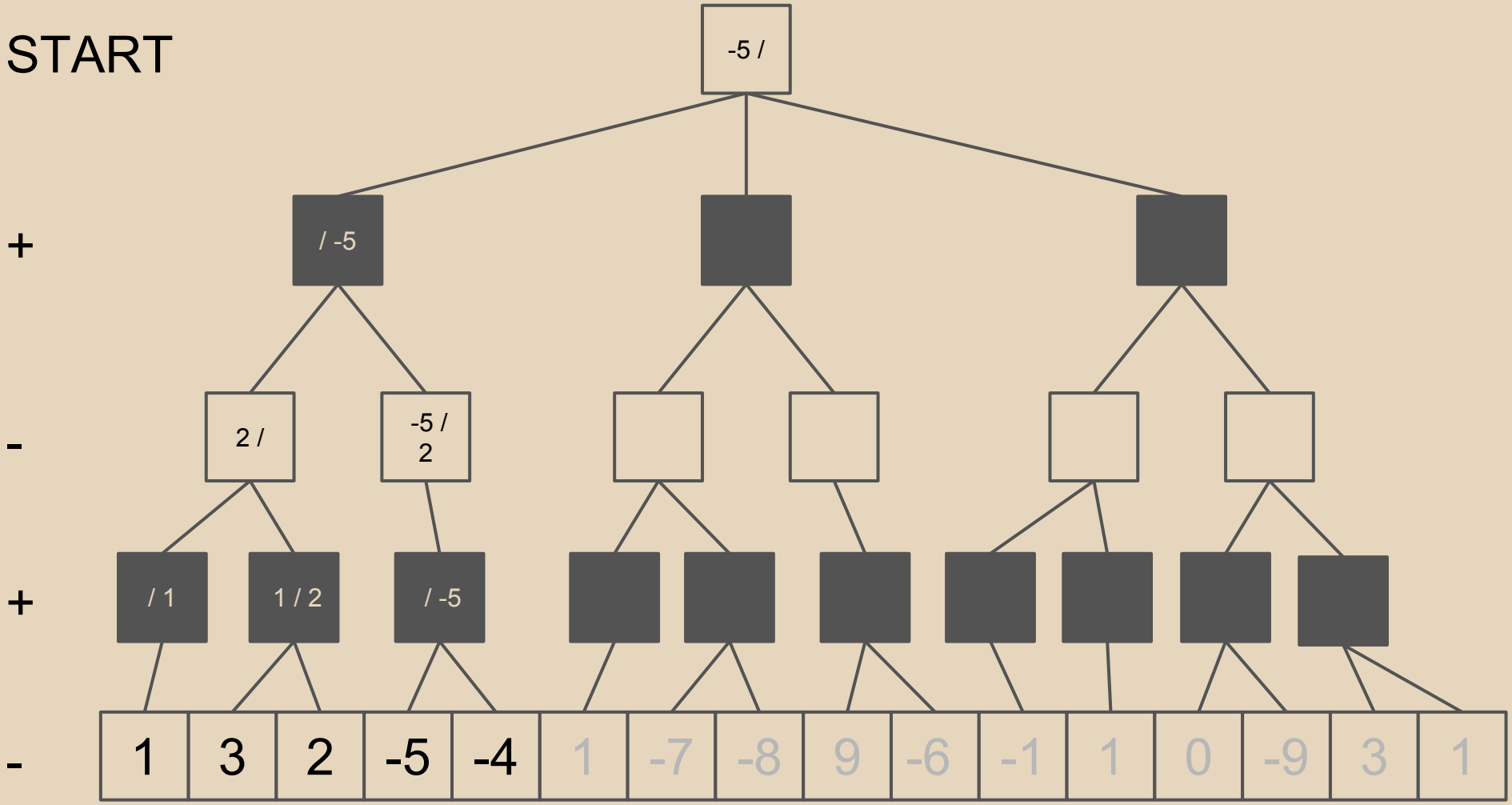
Now start searching...



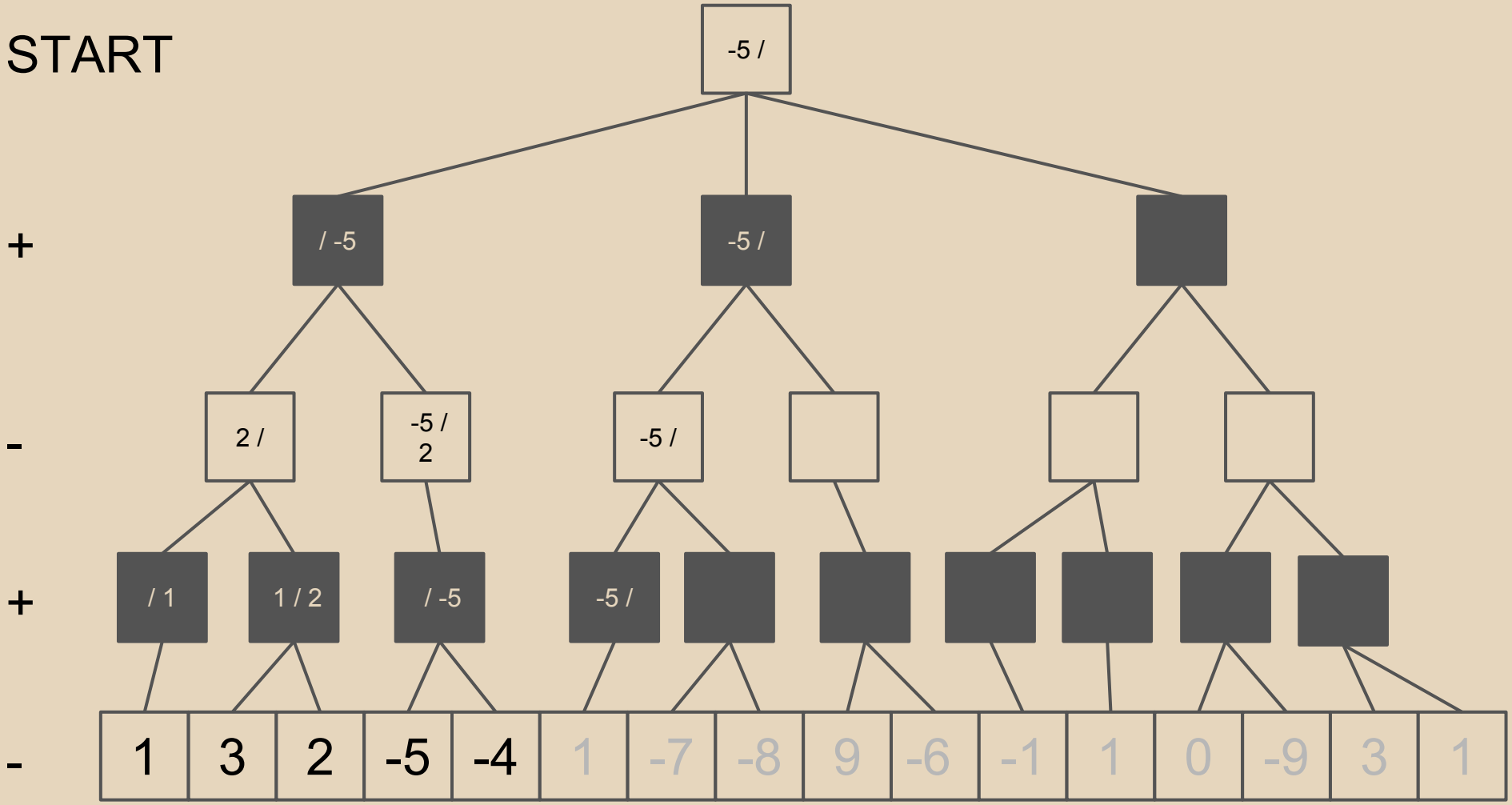
Now start searching...



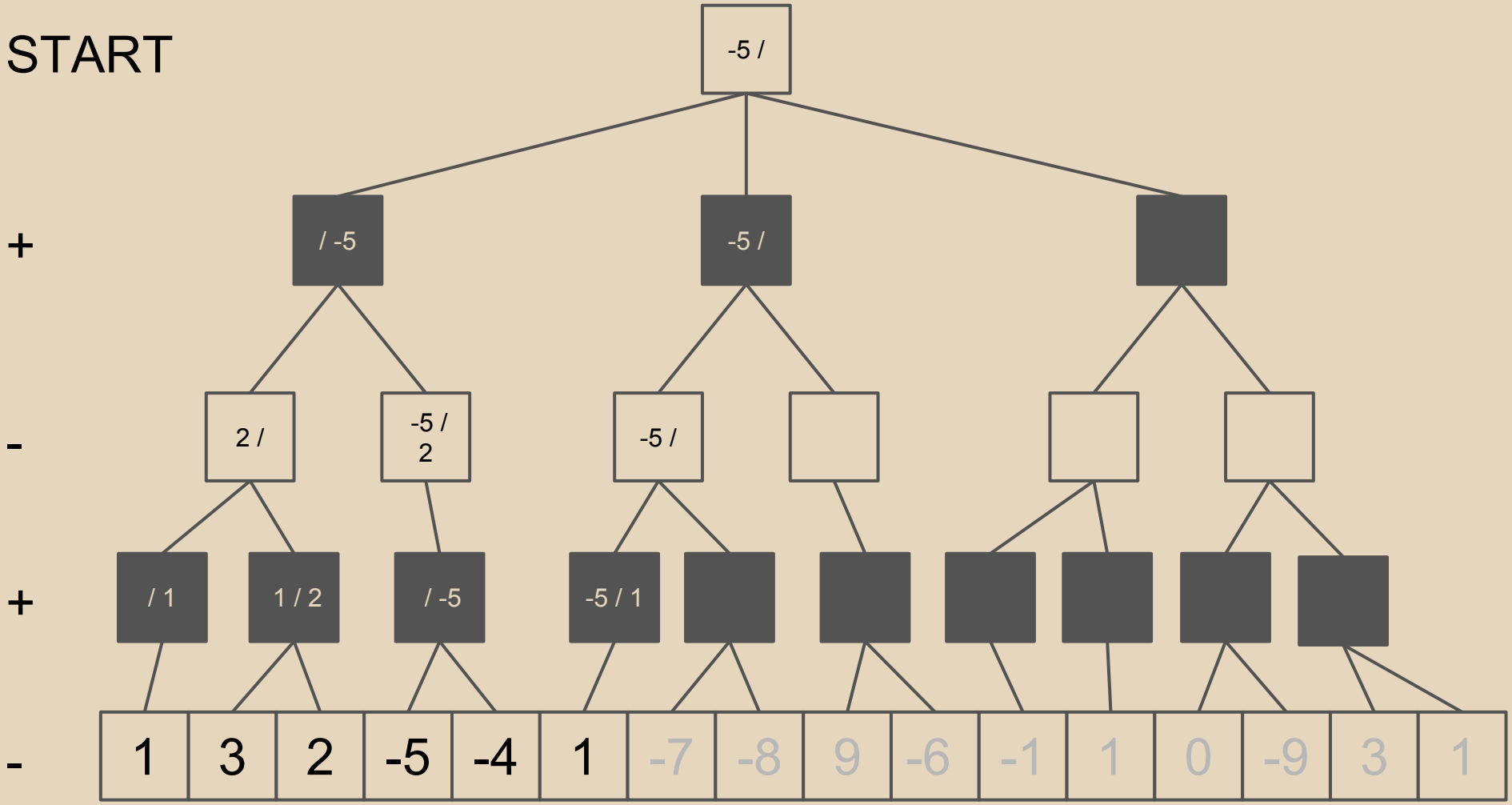
Now start searching...



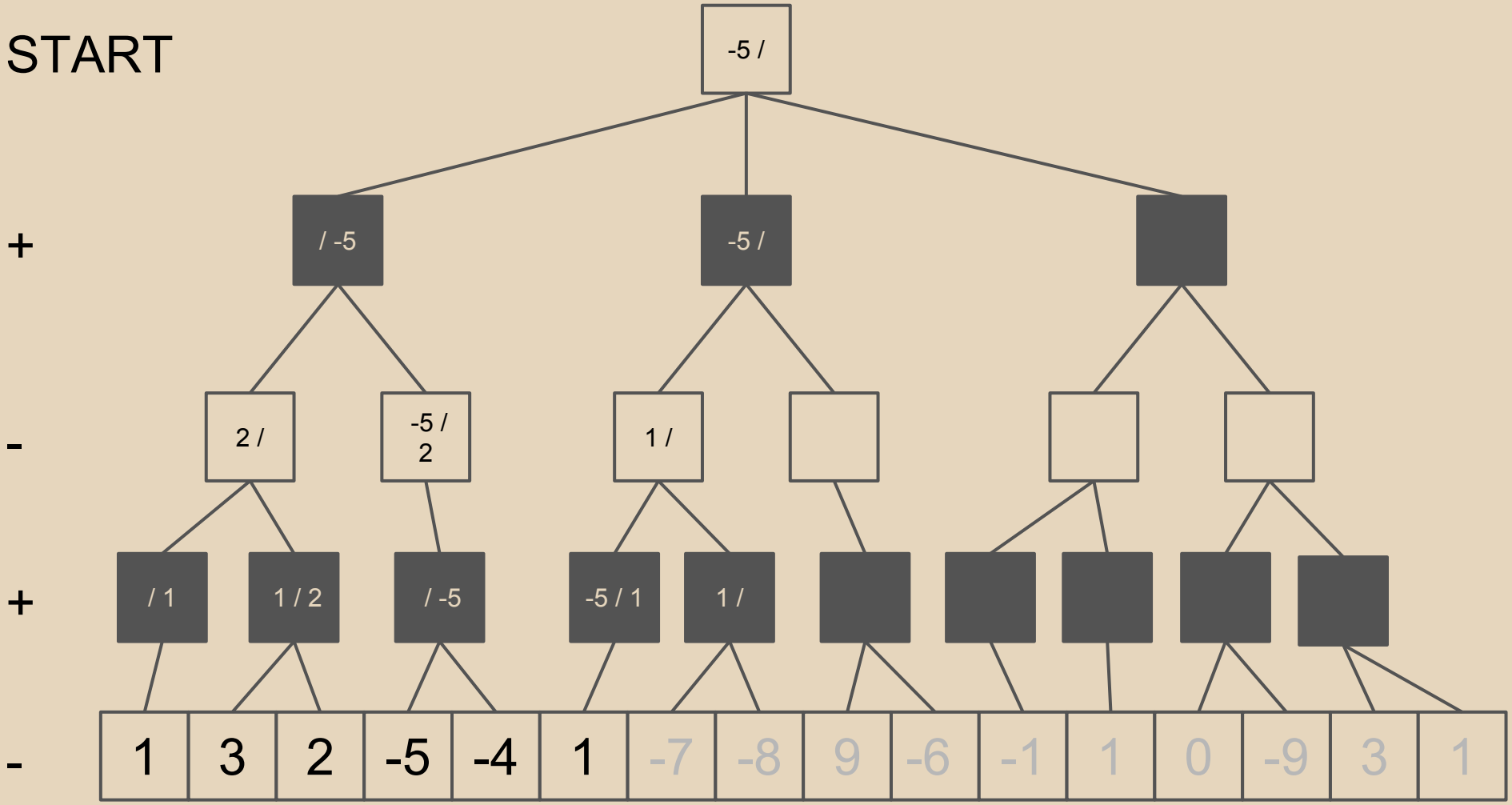
Now start searching...



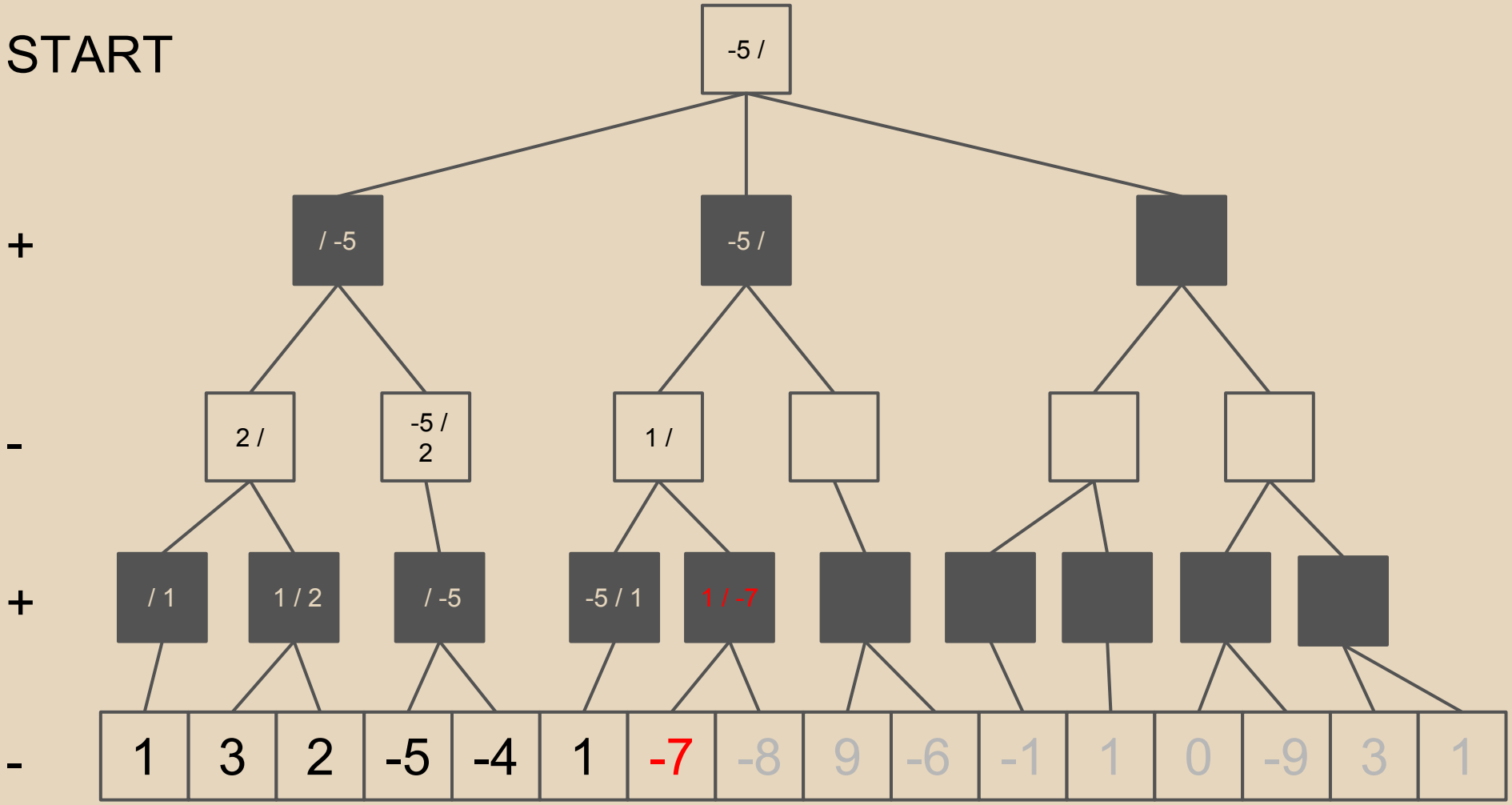
Now start searching...



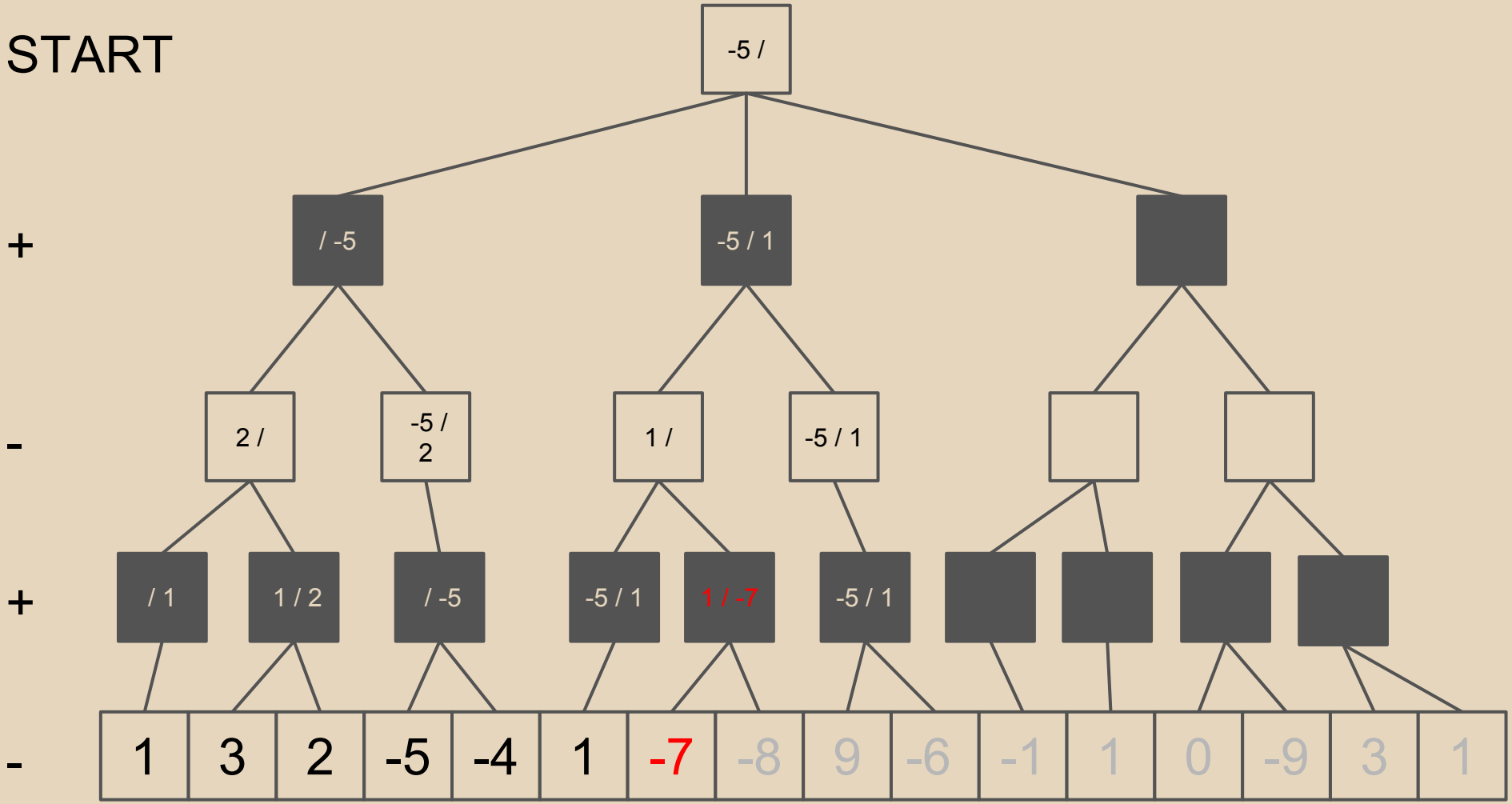
Now start searching...



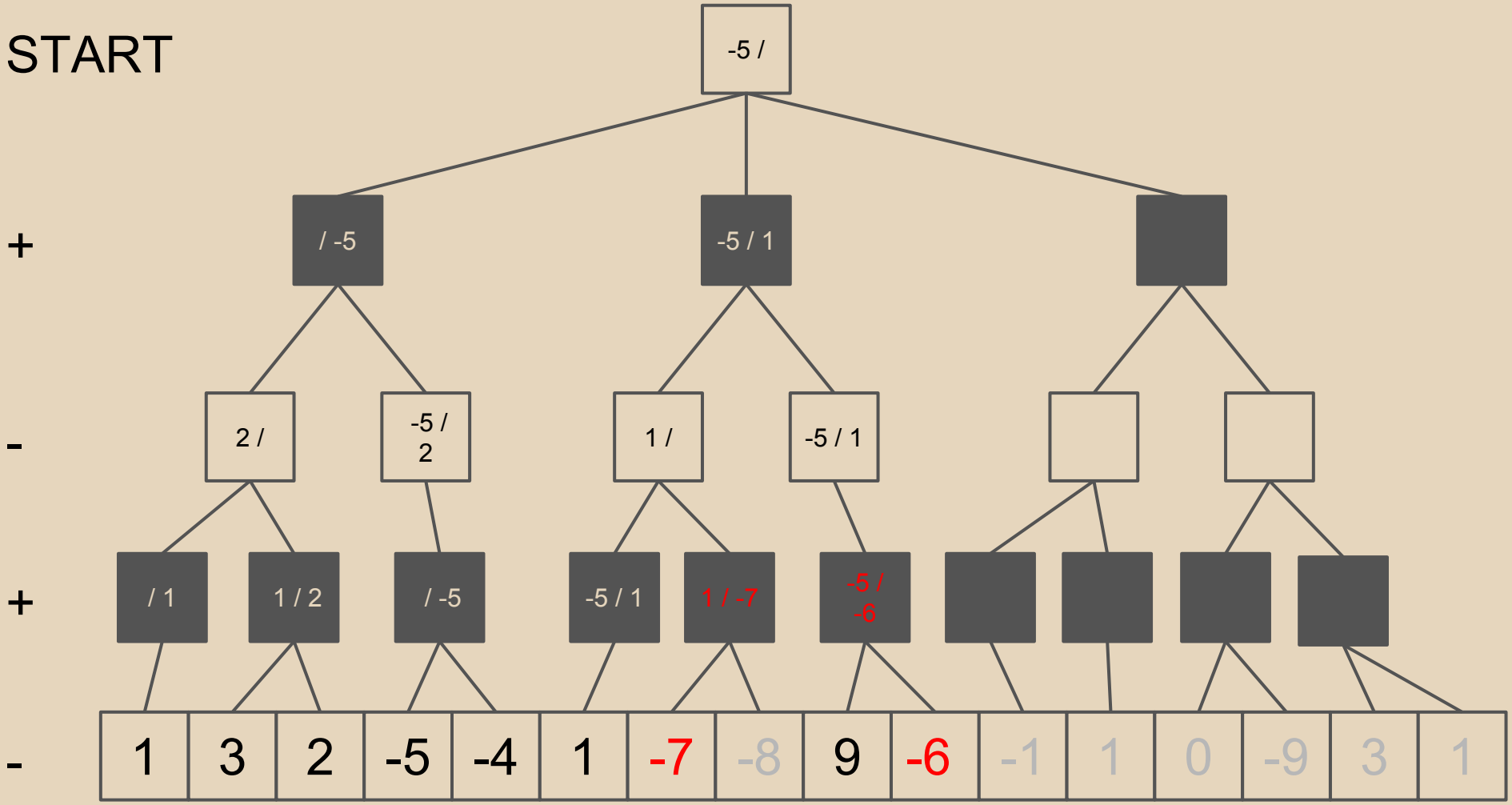
Cutoff!



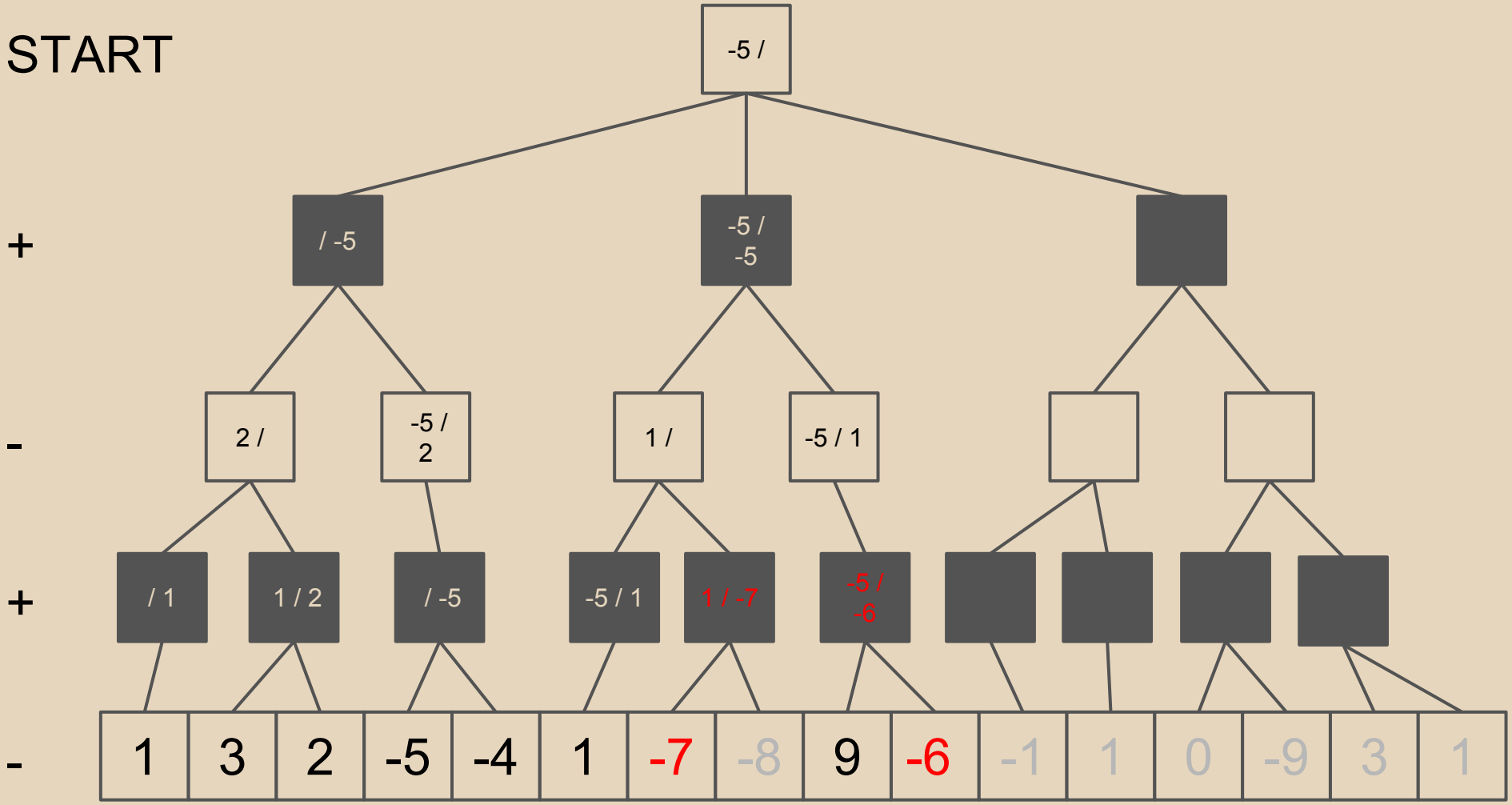
Continue searching



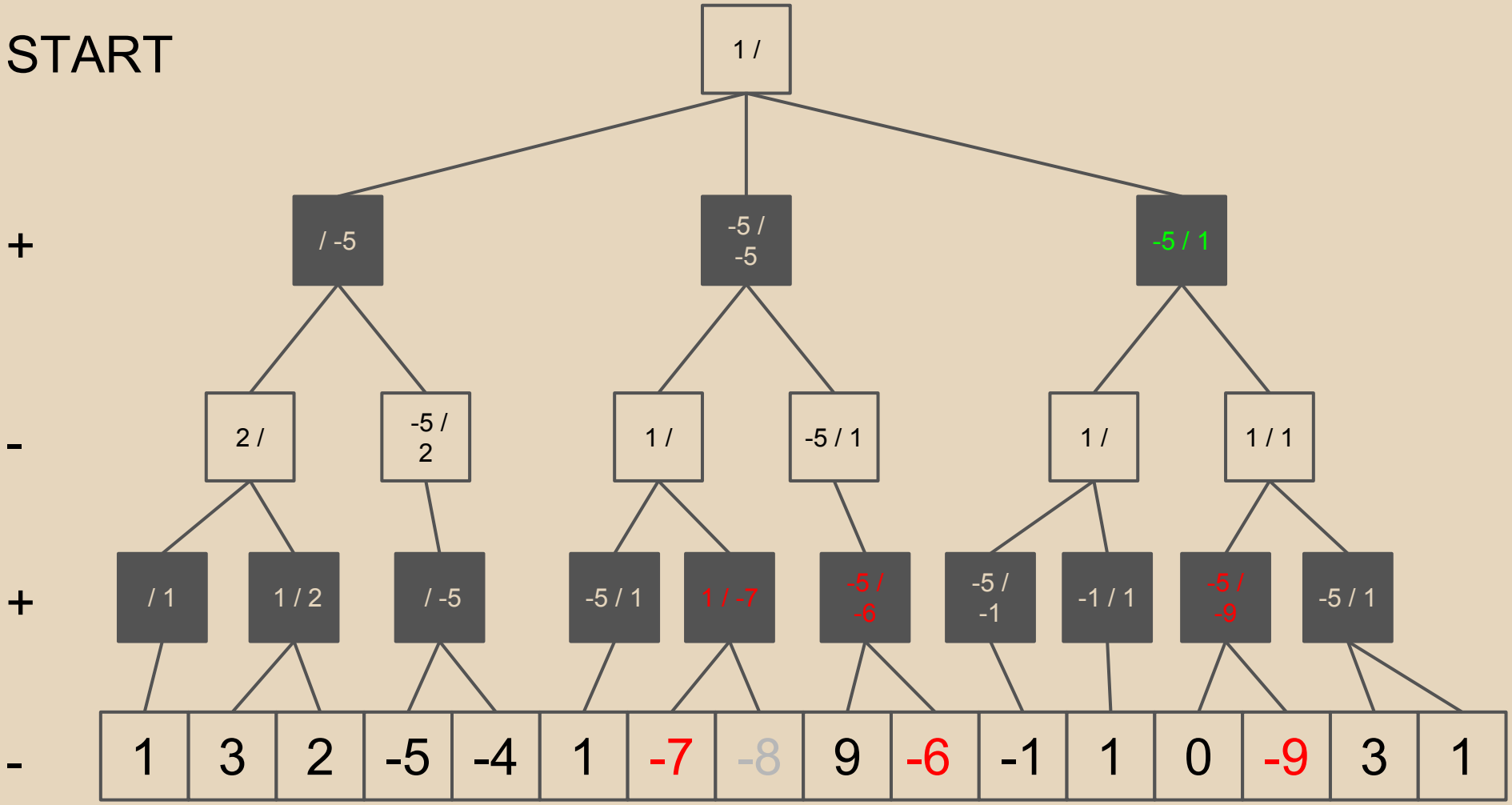
Continue searching



Continue searching



You get the idea...



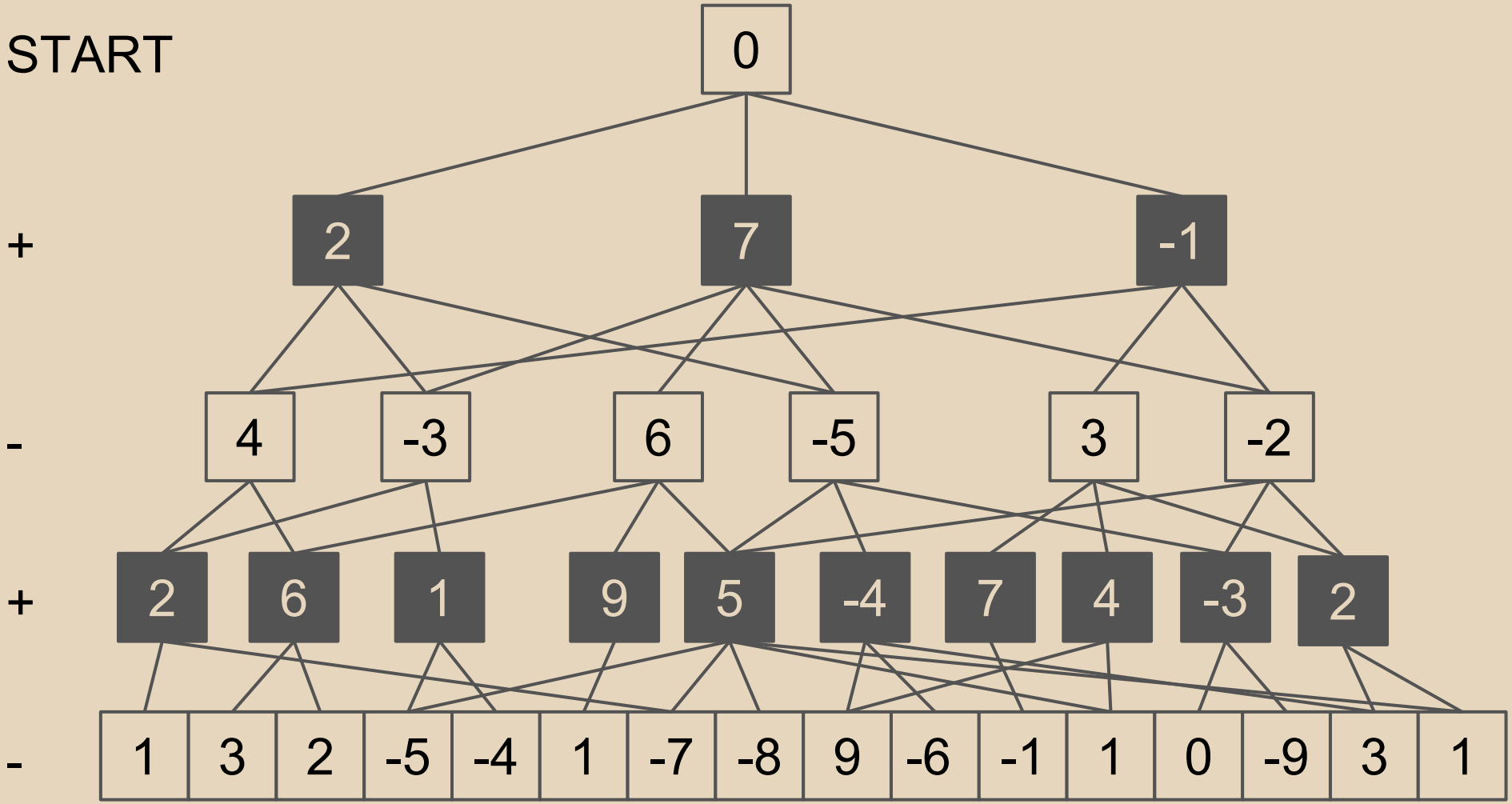
Negamax

- Don’t worry if that was confusing…
- Othello is a zero-sum game
  - Flip the scores and always maximize

Negamax alpha-beta

```
ab (depth, alpha, beta)
    if (depth == 0)
        evaluate()
    for each move:
        score = -ab(depth-1, -beta, -alpha)
        if (score > alpha)
            alpha = score
        if (score >= beta)
            cutoff!
return alpha
```

Game trees can contain duplicate positions...



Transposition tables

- Basically add memoization to our search
  - Remember dynamic programming?
  - If the same position comes up (with the same parameters) and we've already analyzed it, we don't want to reanalyze it
  - Better to look up the solution we already have
- Best way to do this is by hash table
  - Convert board position to a 'hash'
  - (almost) Constant-time lookup then

Transposition tables

- Caution: There are lots of possible positions
  - >> 10^20 legal positions
  - *Cannot* store all of them!
  - Store only the positions that are the most 'popular'
  - Could simply overwrite filled buckets, or keep some sort of popularity metric (for multi-slot buckets)
- Caution #2: Very difficult to implement correctly!
  - Things typically stored: best move (if available), bound and bound type, depth
  - Zobrist hashing

Iterative deepening

- Your time is limited!
  - Up to 16 minutes
  - Want to use our time as efficiently as possible
  - Follow paths that result in fast cutoffs
- Want to be able to do decently even if we run out of time or are interrupted

Iterative deepening

- Start search at a shallow depth
- Store some information about the results
  - Which moves might force alpha-beta cutoffs early?
  - Evaluate the best moves first.
- Repeat search at a deeper depth
- Continue in this vein for as long as needed
  - until we can't afford to spend any more time
- If interrupted, can use the result from an earlier depth rather than from an incomplete search

Move Ordering

- Essential to getting the most out of alpha-beta
- Use:
  - transposition table
  - iterative deepening results
  - shallow searches
  - other heuristics… (PV vs. non-PV nodes, piece-square tables, fastest-first, etc.)
  - You need a good understanding of alpha-beta conceptually to do this effectively

Opening books

- Precalculated responses to early moves
  - Records the best series of responses to particular moves in the early game (down to some small depth)
  - Reduces amount of calculation necessary in the early game
  - Again, dynamic programming / memoization is your friend!
- Often pre-generated
  - The best programs update their opening books after each game
  - We don't expect you to do that

Endgame solvers

- Counting the number of stones is bad in the midgame… (good players often try to minimize this in the early midgame)
- But once the game ends, stone count is the only thing that matters!
- Detect the end of the game and use a different evaluation function
- Can be optimized

For the adventurous…

- Negascout/PVS
  - Fail-soft alpha beta, aspiration windows
- Pattern-based evaluations / machine learning
- Multi-prob cut

Useful git commands: Stash

Suppose we want to save our current changes *without* committing (say, to pull from the remote repository)

- `git stash`: Stashes local changes without committing, reverts back to HEAD state
- `git stash apply`: Replays last stash change (does not commit)

Example: You've been working for hours and find out you need to fix a bug *NOW*. Don't want to lose your work -- so stash it!

Useful git commands: Branch

Sometimes we aren't quite sure what direction our code should take; or we want to try something out without destroying the main branch.

- `git branch`: Start a new branch
- `git checkout`: Switch to another branch
- `git merge`: Synchronize two branches
  - Can have "merge conflicts" to resolve before the merge is complete

Example: You fixed the bug in a branch, now sync with the main repo for your users.

:)

Have fun!