Course: Introduction to AI

Prof. Ahmed Guessoum
The National Higher School of AI

Chapter 5

Adversarial Search

Outline

- Game Theory
- Optimal Decisions in Games
- Heuristic Alpha--Beta Tree Search
- Stochastic Games
- Partially Observable Games

Games

- We study here **competitive** environments, in which the agents' goals are in conflict, leading to **adversarial search** problems (**games**).
- In mathematical **game theory**, a branch of economics, any multi-agent environment is a game, given the impact of each agent on the others is "significant," whether the agents are cooperative or competitive.
- Games in AI (e.g. chess) are what game theorists call deterministic, turn-taking, two-player, **zero-sum games** of **perfect information**.
- The utility values at the end of the game are always equal and opposite: a player wins; the other loses.

Games

- The state of a game is easy to represent, and agents are usually restricted to a small number of actions whose outcomes are defined by precise rules.
- Games are interesting *because* they are too hard to solve.
- Chess:
  - average branching factor: ~35
  - games often go to 50 moves by each player

  ➔ search tree has about 35^{100}, i.e. 10^{154} nodes (search graph has "only" about 10^{40} distinct nodes).
- Making *some* decision is required even when calculating the *optimal* decision is infeasible.
- Games also penalize inefficiency severely

Games

- Multi-player games are possible, but we first consider games with two players, called MAX and MIN.
  - MAX moves first; then they take turns moving until the game is over.
  - At the end of the game, points are awarded to the winning player and penalties are given to the loser.

Games

A game can be formally defined as a kind of search problem with the following elements:

- S_0: The **initial state** (configuration) of the game.
- PLAYER(s): which player has the move in a state.
- ACTIONS(s): Returns the set of legal moves in a state.
- The **transition model**: defines the result of a move. RESULT(s, a)
- TERMINAL-TEST(s): true when the game is over and false otherwise. States where the game has ended are called **terminal states**.

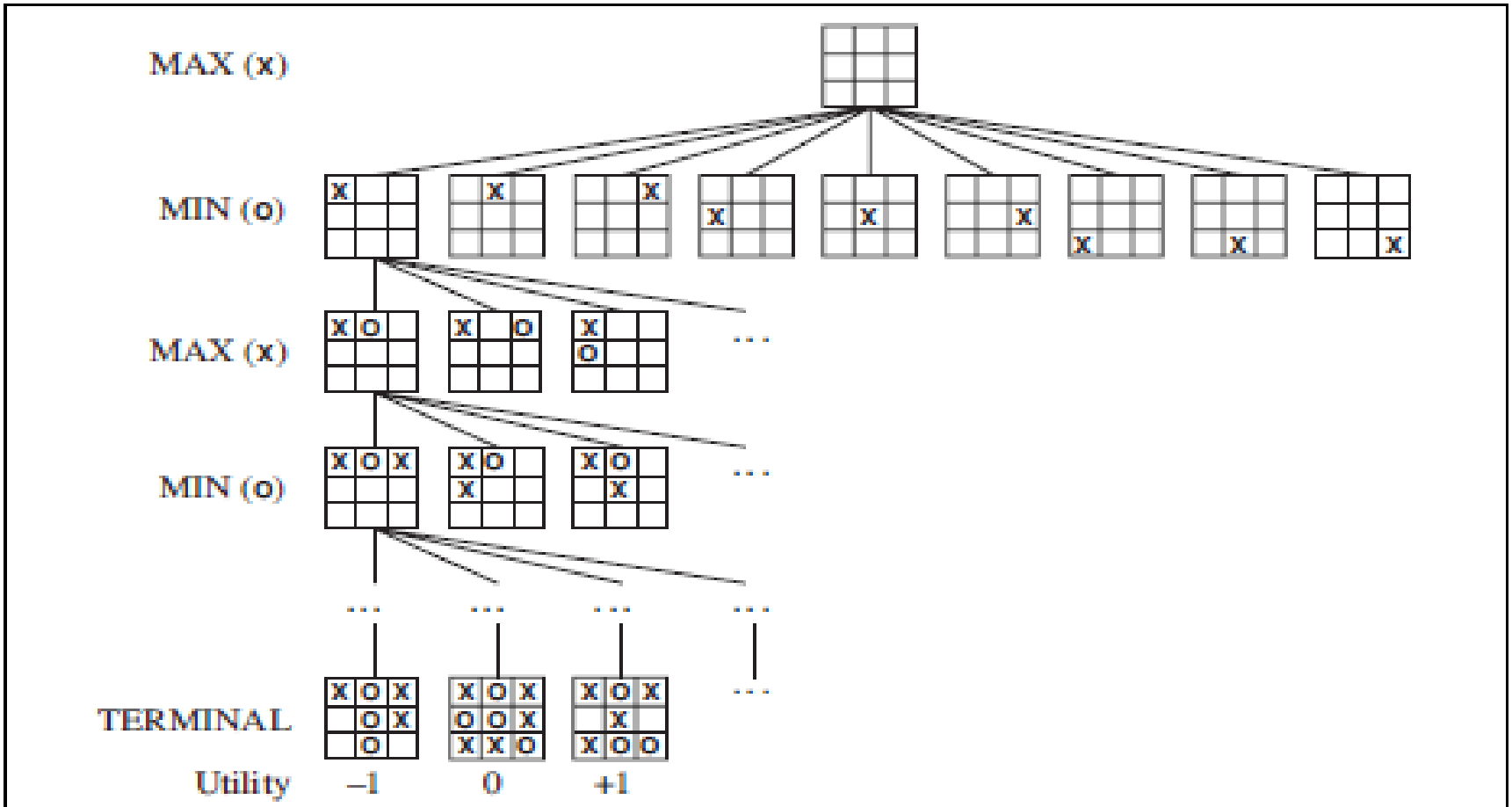
Games

- UTILITY(s,p): A **utility function** (or objective function or payoff function), defines the final numeric value for a game that ends in terminal state s for a player p.
  - In chess, the outcome is a win, loss, or draw, with values +1, 0, or ½. Other games have different possible outcomes; the payoffs in backgammon range from 0 to +192.
- A **zero-sum game** is (confusingly) defined as one where the total payoff to all players is the same for every instance of the game.
  - Chess is zero-sum because every game has payoff of either 0 + 1, 1 + 0 or ½ + ½.
  - "Constant-sum" would have been a better term.
- **Game tree:** a tree where the nodes are game states and the edges are moves.

Tic-Tac-Toe (partial) tree

For tic-tac-toe the **game tree** is relatively small— 9! = 362,880 terminal nodes.

Search tree: a tree that is superimposed on the full game tree; it examines enough nodes to allow a player to determine what move to make.



Outline

- Game Theory
- Optimal Decisions in Games
- Heuristic Alpha--Beta Tree Search
- Stochastic Games
- Partially Observable Games

Optimal Decisions in Games

- In a normal search problem: an optimal solution is a sequence of actions leading to a goal state.
- In adversarial search, MIN has a role also. MAX therefore must find a contingent **strategy**, which specifies MAX's move in the initial state, then MAX's moves in the states resulting from every possible response by MIN, etc.
- An **optimal strategy** leads to outcomes at least as good as any other strategy when one is playing an infallible opponent.

Optimal Decisions in Games

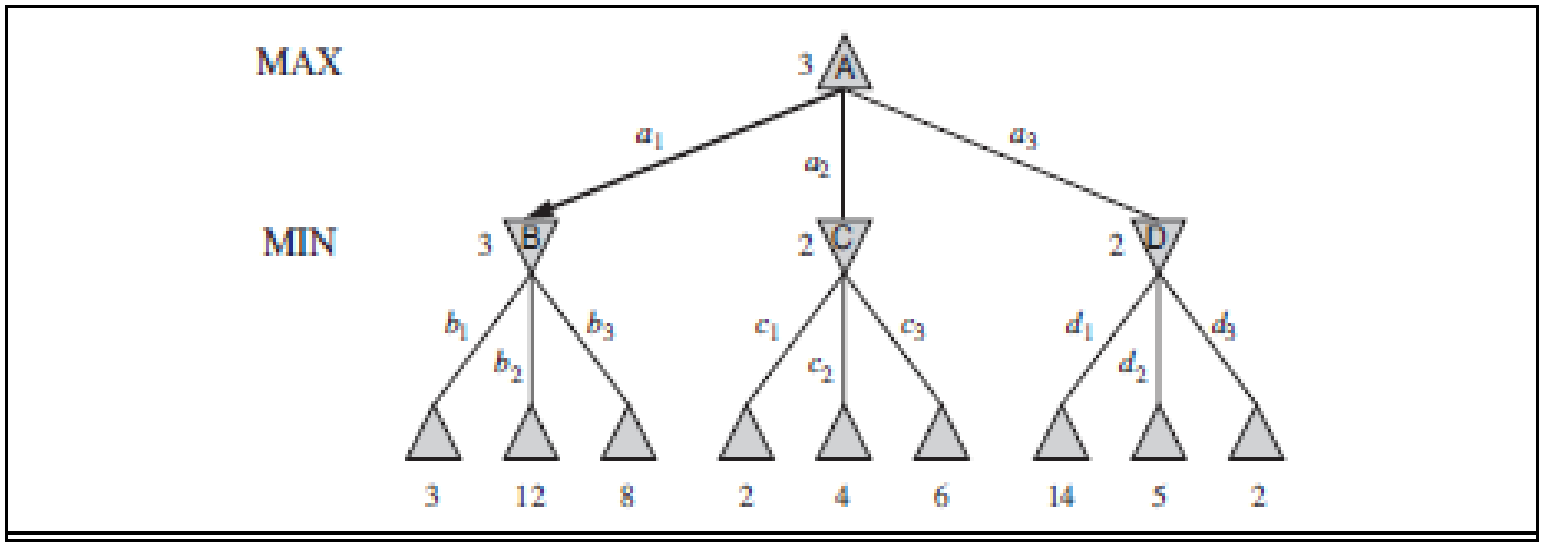


Figure 5.2 A two-ply game tree. The △ nodes are "MAX nodes," in which it is MAX's turn to move, and the ▽ nodes are "MIN nodes." The terminal nodes show the utility values for MAX; the other nodes are labeled with their minimax values. MAX's best move at the root is a_1, because it leads to the state with the highest minimax value, and MIN's best reply is b_1, because it leads to the state with the lowest minimax value.

- The tree is one move deep, consisting of two half-moves, each of which is called a **ply**.
- The utilities of the terminal states range from 2 to 14.

Minimax strategy

- The optimal strategy can be determined from the **minimax value** of each node (MINIMAX(n)).
- The minimax value of a node is the utility (for MAX) of being in the corresponding state, *assuming that both players play optimally* from there to the end of the game.

$$\text{MINIMAX}(s) = \begin{cases} \text{UTILITY}(s) & \text{if TERMINAL\_TEST}(s) \\ \max_{a \in \text{Actions}(s)} \text{MINIMAX}(\text{RESULT}(s,a)) & \text{if PLAYER}(s) = \text{MAX} \\ \min_{a \in \text{Actions}(s)} \text{MINIMAX}(\text{RESULT}(s,a)) & \text{if PLAYER}(s) = \text{MIN} \end{cases}$$

- In previous figure: The **minimax decision** at the root: action a1 is the optimal choice for MAX because it leads to the state with the highest minimax value.
- if MIN does not play optimally, then it is easy to show that MAX will do even better

Minimax Algorithm

```
function MINIMAX-DECISION(state) returns an action
  return arg max_{a ∈ ACTIONS(s)} MIN-VALUE(RESULT(state, a))

function MAX-VALUE(state) returns a utility value
  if TERMINAL-TEST(state) then return UTILITY(state)
  v ← −∞
  for each a in ACTIONS(state) do
    v ← MAX(v, MIN-VALUE(RESULT(s, a)))
  return v

function MIN-VALUE(state) returns a utility value
  if TERMINAL-TEST(state) then return UTILITY(state)
  v ← ∞
  for each a in ACTIONS(state) do
    v ← MIN(v, MAX-VALUE(RESULT(s, a)))
  return v
```

Figure 5.3 An algorithm for calculating minimax decisions. It returns the action corresponding to the best possible move, that is, the move that leads to the outcome with the best utility, under the assumption that the opponent plays to minimize utility. The functions MAX-VALUE and MIN-VALUE go through the whole game tree, all the way to the leaves, to determine the backed-up value of a state. The notation $\text{argmax}_{a \in S} f(a)$ computes the element a of set S that has the maximum value of $f(a)$.

Minimax Algorithm

- The minimax algorithm performs a **complete depth-first exploration** of the game tree.
- *m*: maximum depth of the tree; *b*: branching factor, then

time complexity of minimax : $O(b^m)$.

space complexity is $O(bm)$ if algorithm generates all actions at once, or $O(m)$ if it generates actions one at a time.

Optimal decisions in multiplayer games

- Some popular games allow more than two players.
- Extending the minimax idea to them is technically straightforward (though there are subtleties).
- The single value for each node is replaced with a *vector* of values.
  - E.g., in a 3-player game with players A, B, and C, a vector v_A, v_B, v_C is respectively associated with each node.
  - For terminal states, this vector gives the utility of the state from each player's viewpoint.
- The backed-up value of a node n is always the utility vector of the successor state with the highest value for the player choosing at n.

Example: minimax for multiplayer game

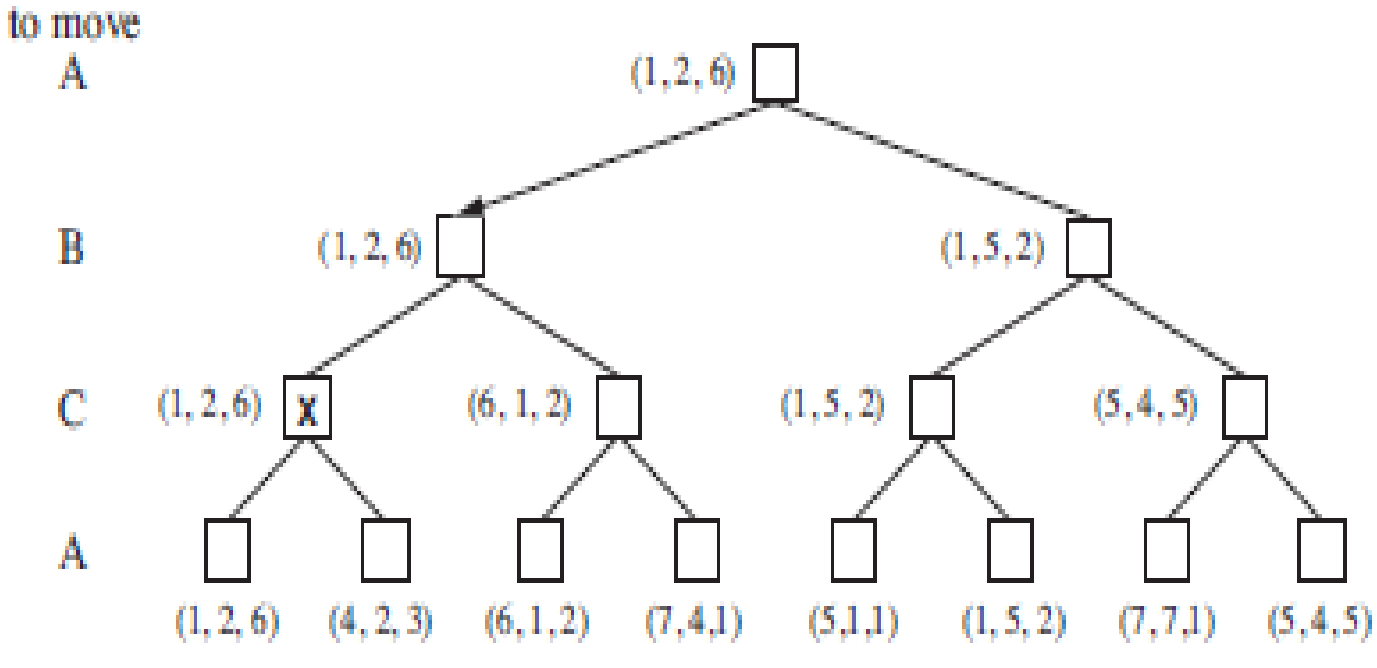


Figure 5.4 The first three plies of a game tree with three players (A, B, C). Each node is labeled with values from the viewpoint of each player. The best move is marked at the root.

Optimal decisions in multiplayer games

- Multiplayer games usually involve **alliances**, whether formal or informal, among the players.
- Alliances change as the game proceeds.
- Alliances can be a natural consequence of optimal strategies for each player in a multiplayer game.
  - Suppose A and B are in weak positions and C is in a stronger position. Then it is often optimal for both A and B to attack C rather than each other.
  - As soon as C weakens under the joint attacks, the alliance loses its value, and either A or B could change the approach.

Outline

- Game Theory
- Optimal Decisions in Games
- Heuristic Alpha--Beta Tree Search
- Stochastic Games
- Partially Observable Games

Alpha–Beta Pruning

- Problem with minimax search: the number of game states it has to examine is exponential in the depth of the tree.
- Idea: it is possible to compute the correct minimax decision without looking at every node in the game tree.

  ➔ Possible to **prune** non useful explorations
- **Alpha–Beta pruning**: when applied to a standard minimax tree, it returns the same move as minimax would, but prunes away branches that cannot possibly influence the final decision.
- Alpha–beta pruning can be applied to trees of any depth; it is often possible to prune entire subtrees.

Alpha–Beta Pruning

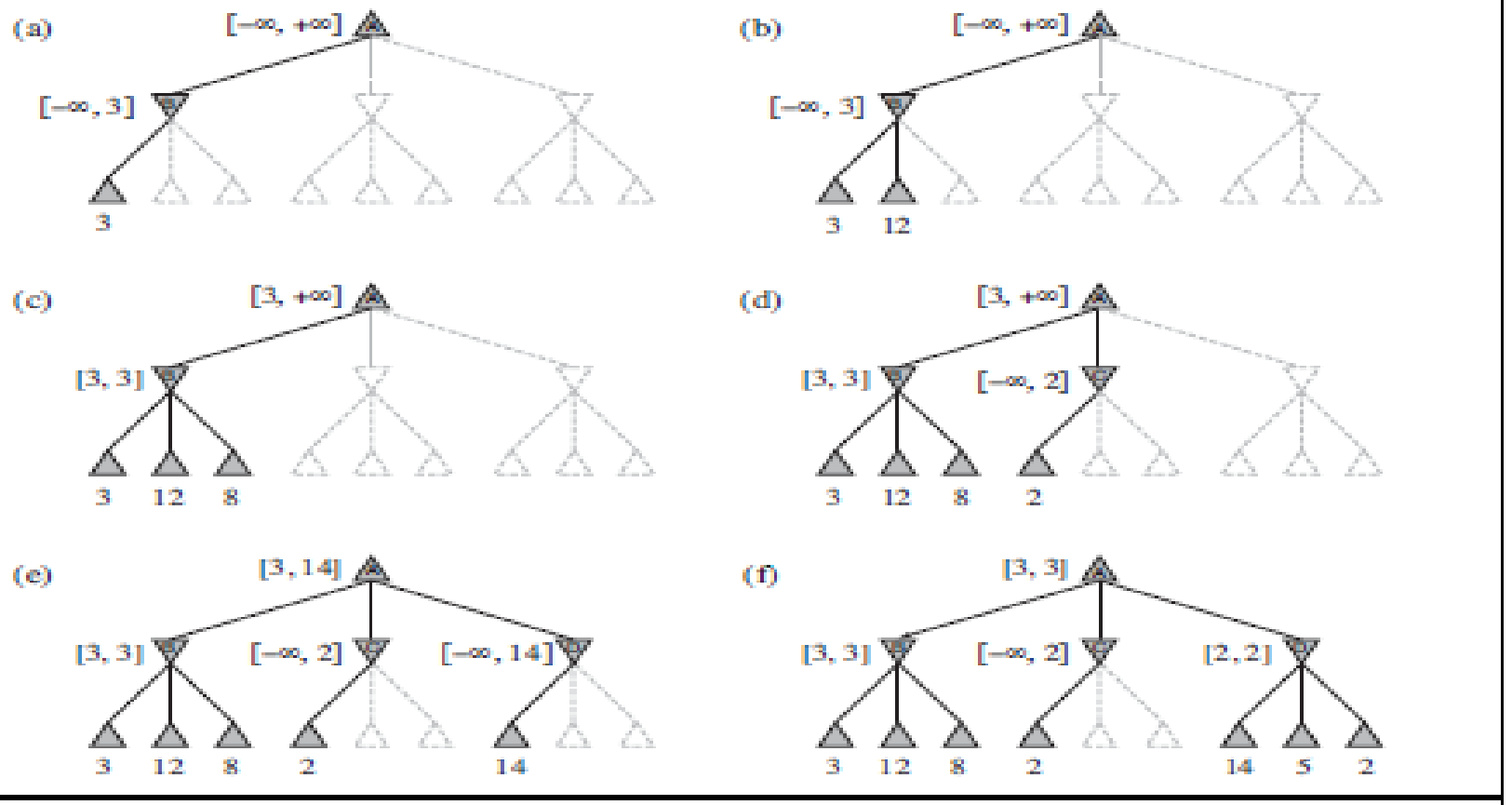
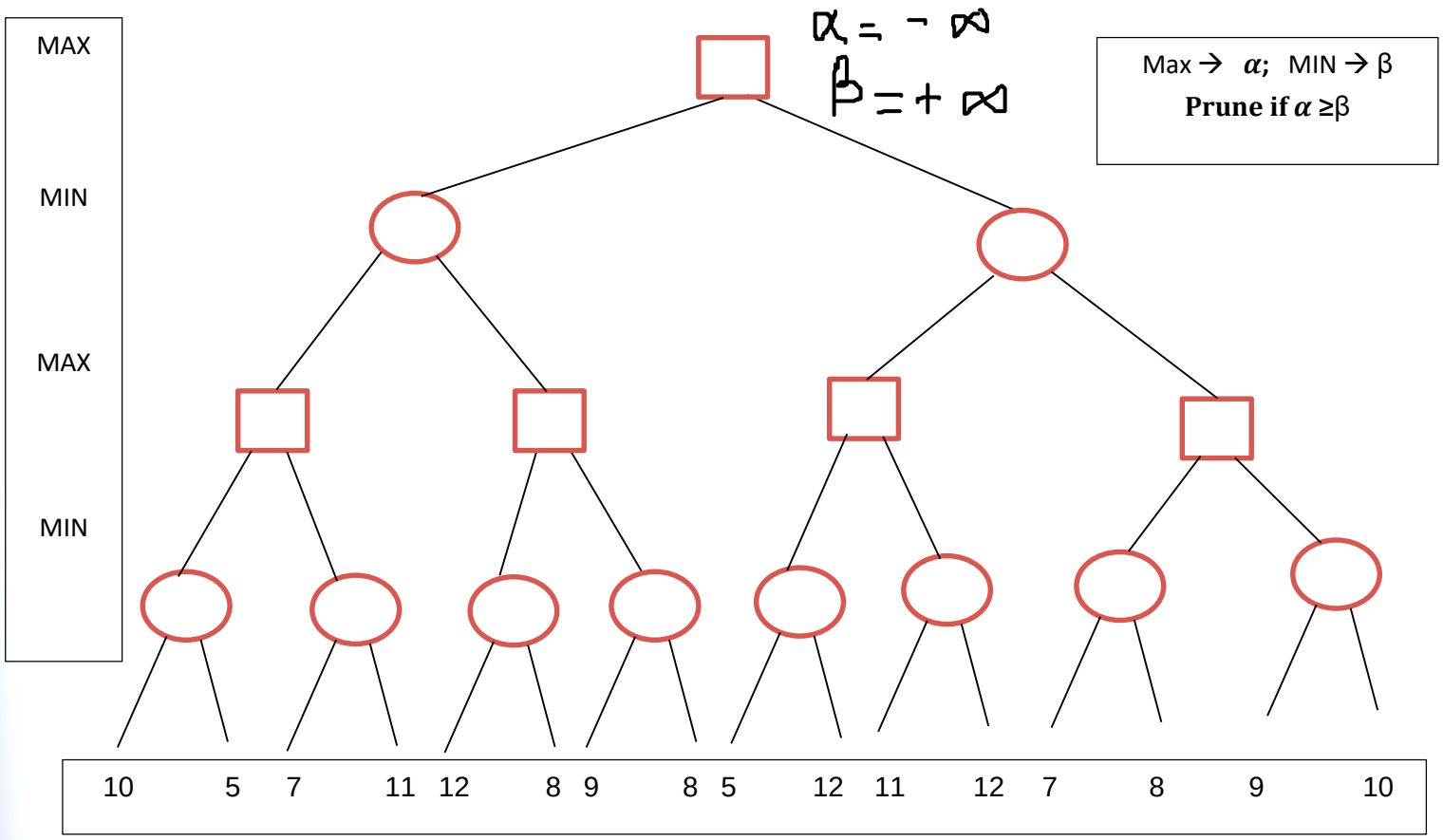
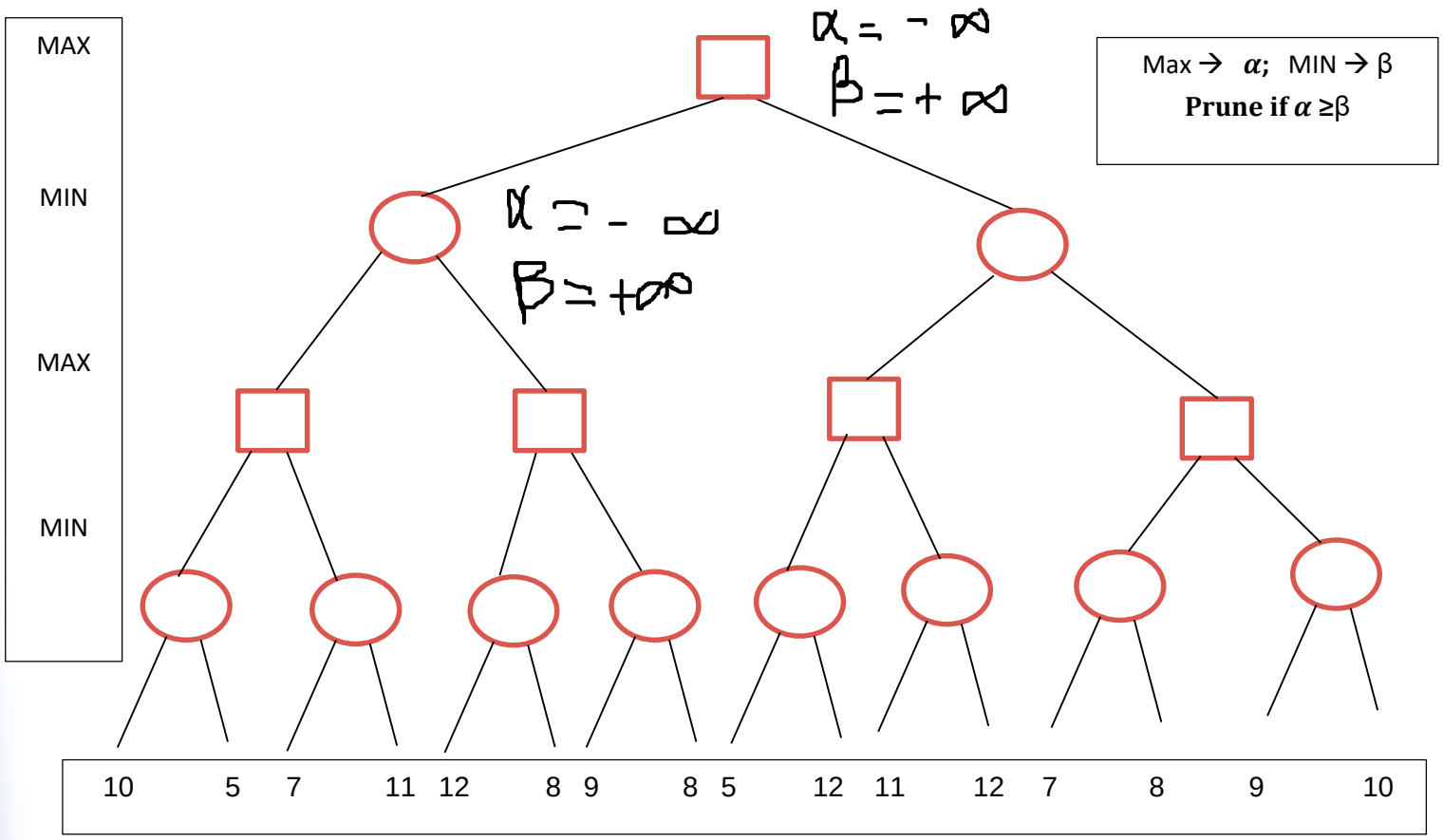


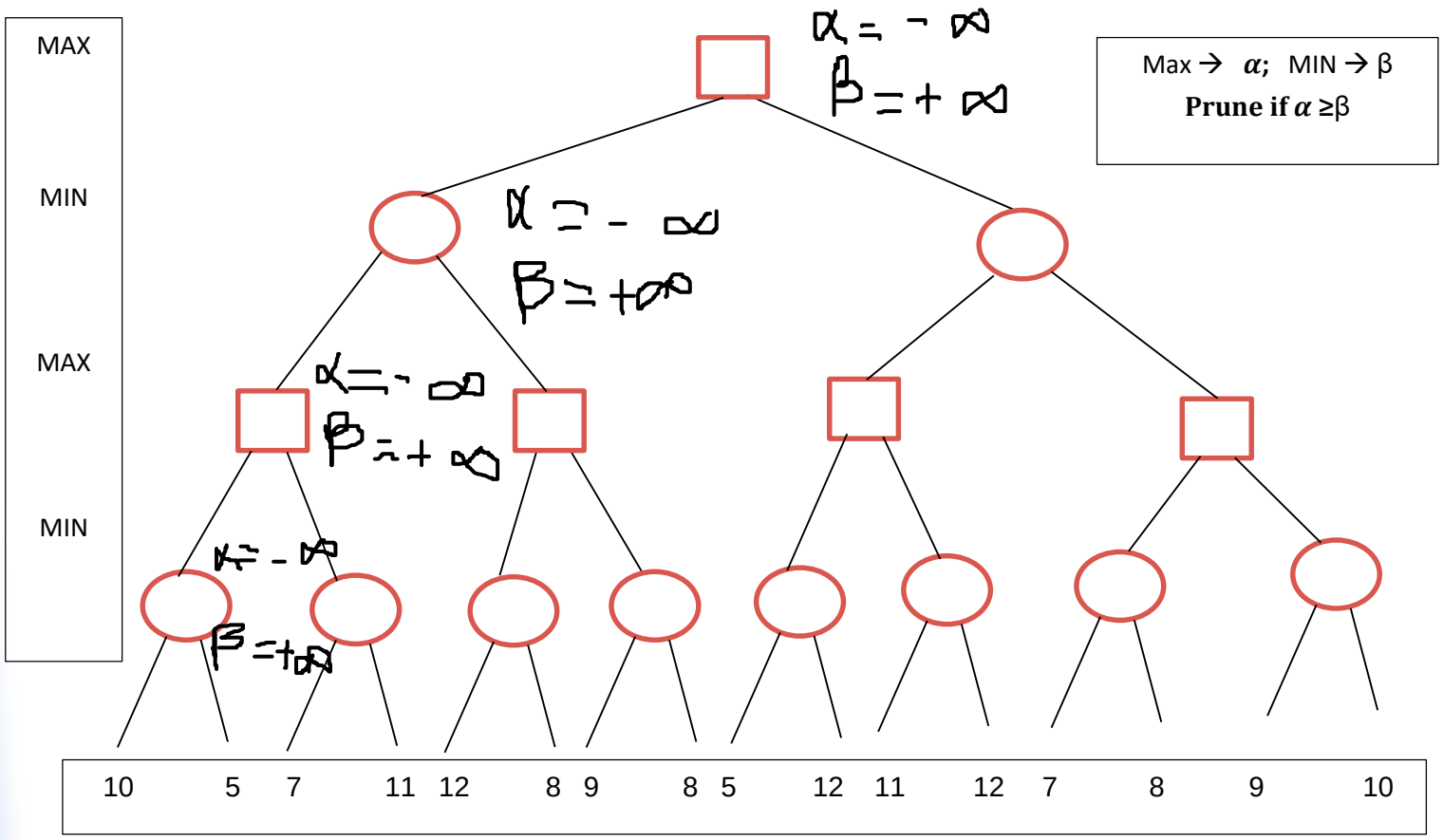
Figure 5.5 Stages in the calculation of the optimal decision for the game tree in Figure 5.2. At each point, we show the range of possible values for each node. (a) The first leaf below B has the value 3. Hence, B, which is a MIN node, has a value of *at most* 3. (b) The second leaf below B has a value of 12; MIN would avoid this move, so the value of B is still at most 3. (c) The third leaf below B has a value of 8; we have seen all B's successor states, so the value of B is exactly 3. Now, we can infer that the value of the root is *at least* 3, because MAX has a choice worth 3 at the root. (d) The first leaf below C has the value 2. Hence, C, which is a MIN node, has a value of *at most* 2. But we know that B is worth 3, so MAX would never choose C. Therefore, there is no point in looking at the other successor states

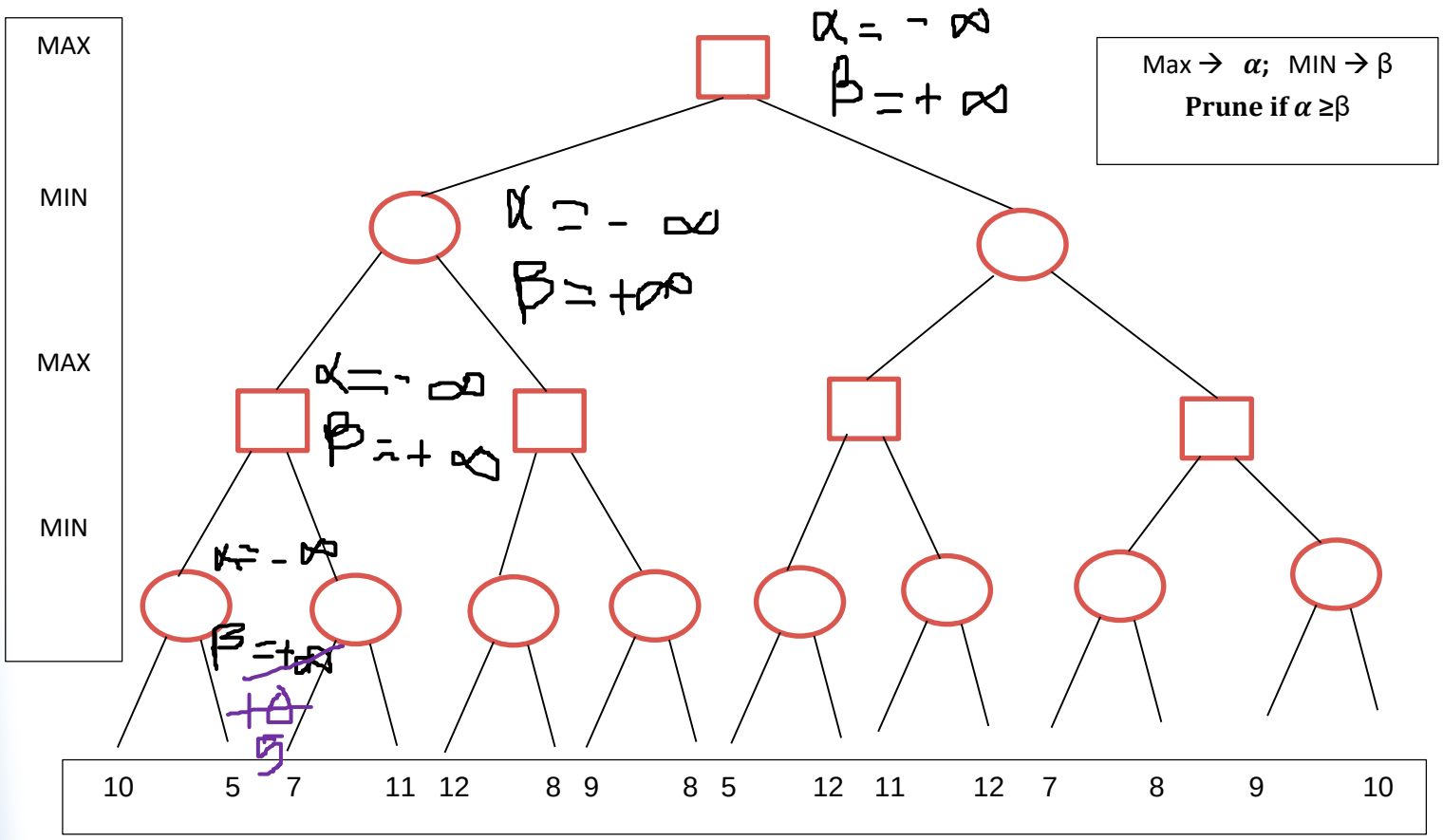
Alpha–Beta Pruning

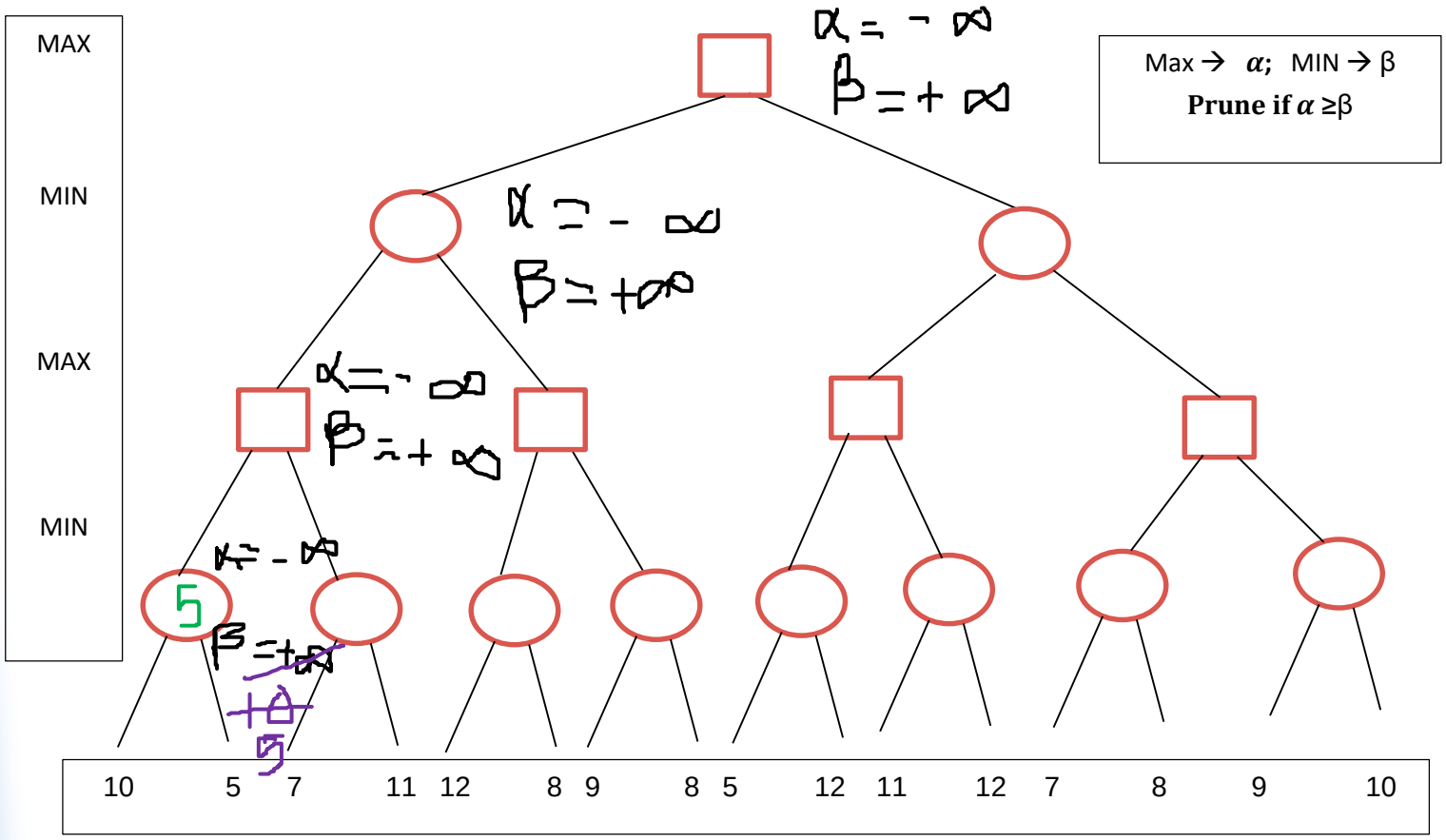
- Alpha–Beta pruning gets its name from the two parameters that describe bounds on the backed-up values that appear anywhere along the path:
  - α = value of the best (i.e., highest-value) choice we have found so far at any choice point along the path for MAX.
  - β = value of the best (i.e., lowest-value) choice we have found so far at any choice point along the path for MIN.
- Alpha–beta search updates the values of α (resp., β) as it goes along and prunes the remaining branches at a node (i.e., terminates the recursive call) as soon as the value of the current node is known to be worse than the current α (resp., β) value for MAX (resp., MIN).

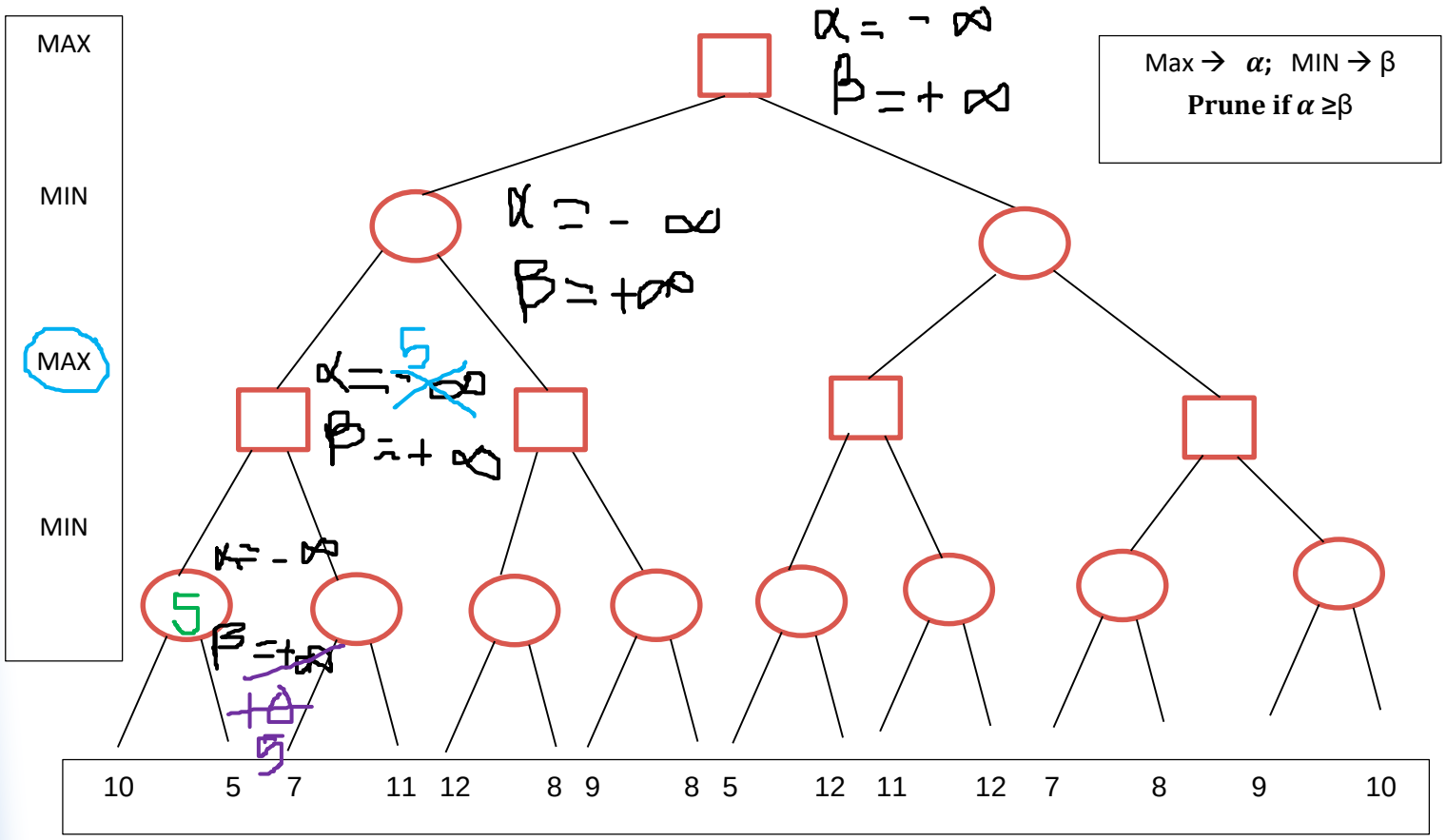

MAX
MIN
MAX
MIN
α = −∞
β = +∞
Max → α; MIN → β
Prune if α ≥β
10 5 7 11 12 8 9 8 5 12 11 12 7 8 9 10

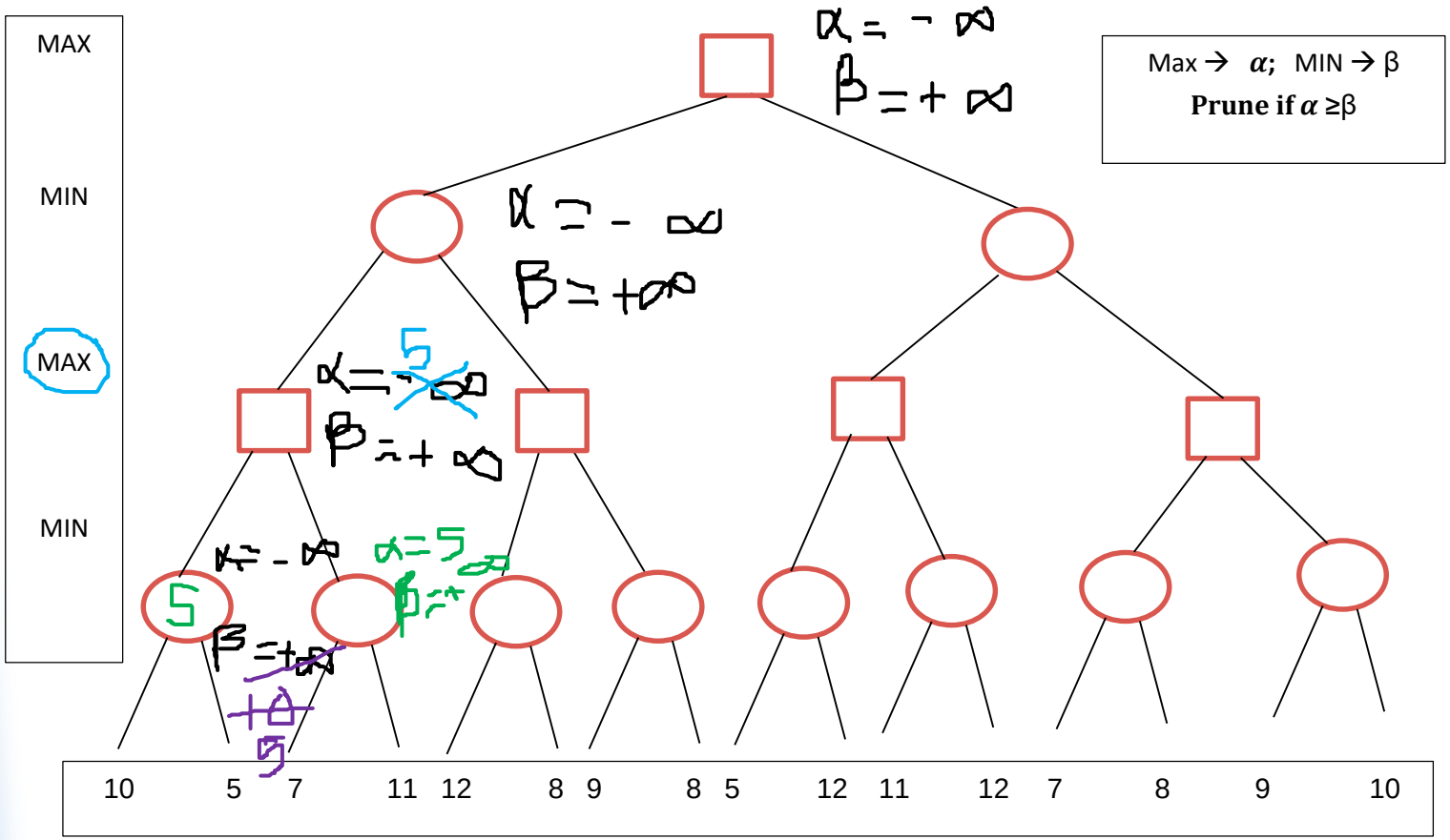

MAX
MIN
MAX
MIN
α = −∞
β = +∞
α = −∞
β = +∞
Max → α; MIN → β
Prune if α ≥ β
10 5 7 11 12 8 9 8 5 12 11 12 7 8 9 10

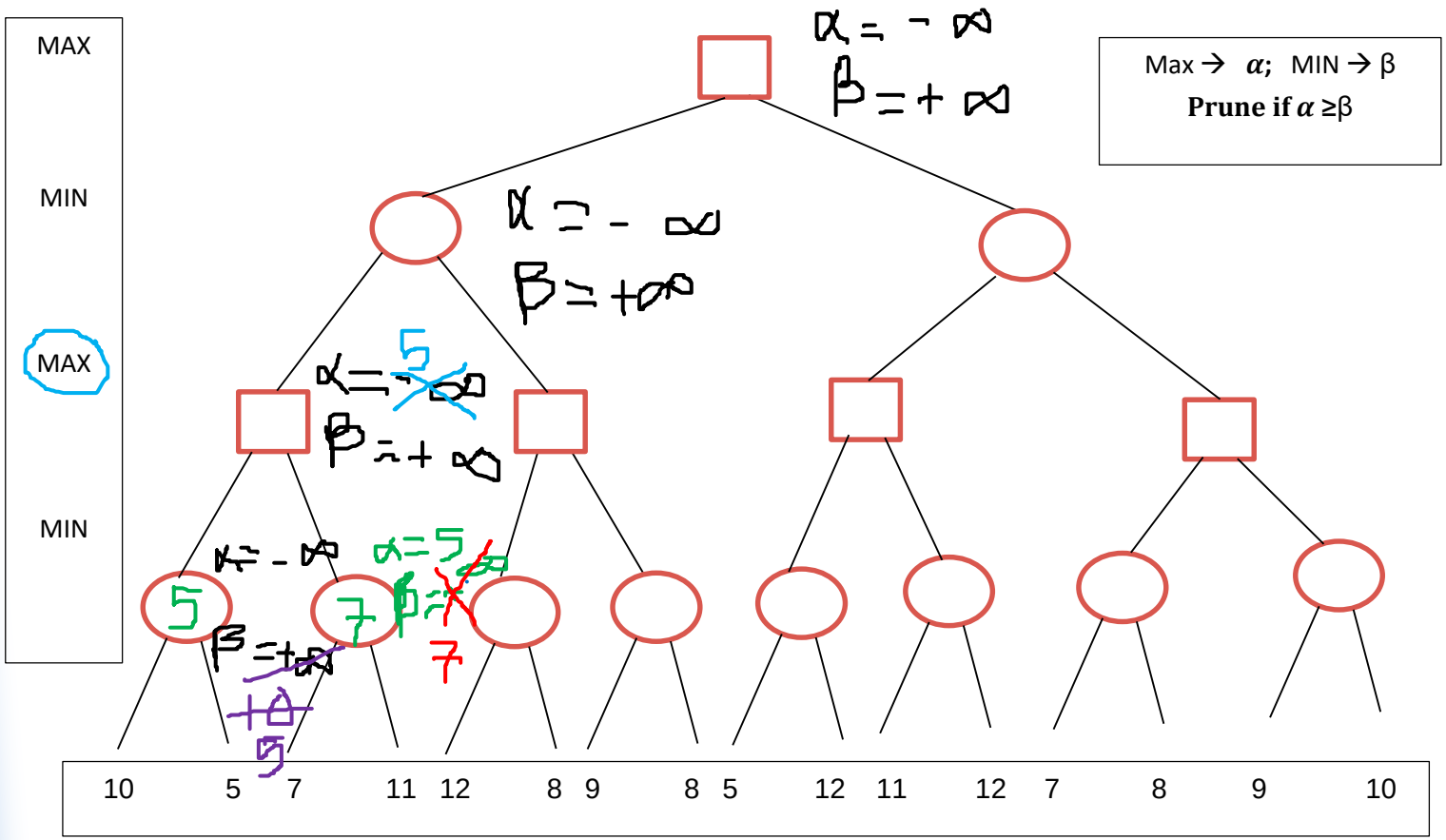

MAX
MIN
MAX
MIN
Max → α; MIN → β
Prune if α ≥β
10 5 7 11 12 8 9 8 5 12 11 12 7 8 9 10

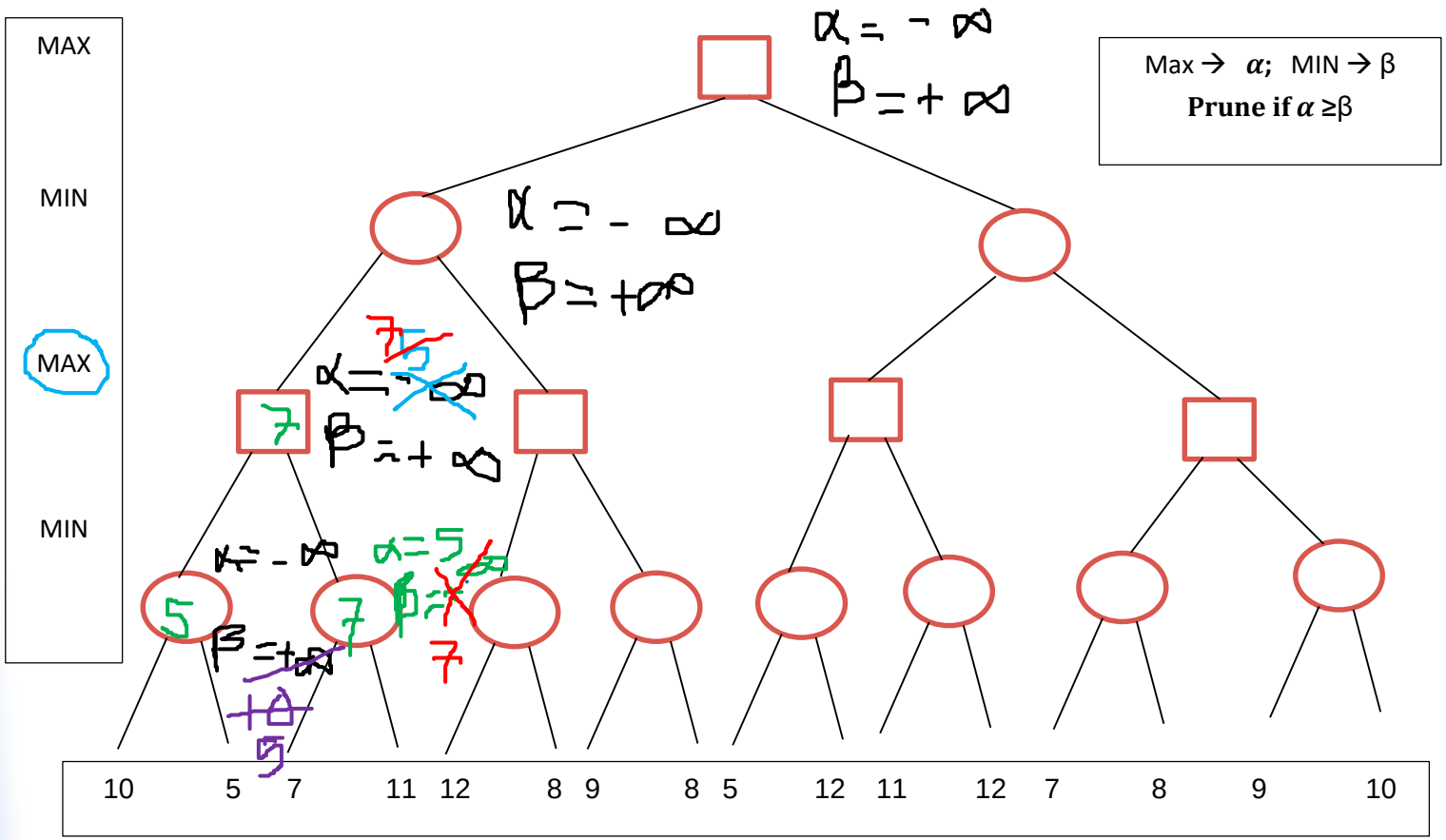

MAX
MIN
MAX
MIN
Max → α; MIN → β
Prune if α ≥β
10 5 7 11 12 8 9 8 5 12 11 12 7 8 9 10

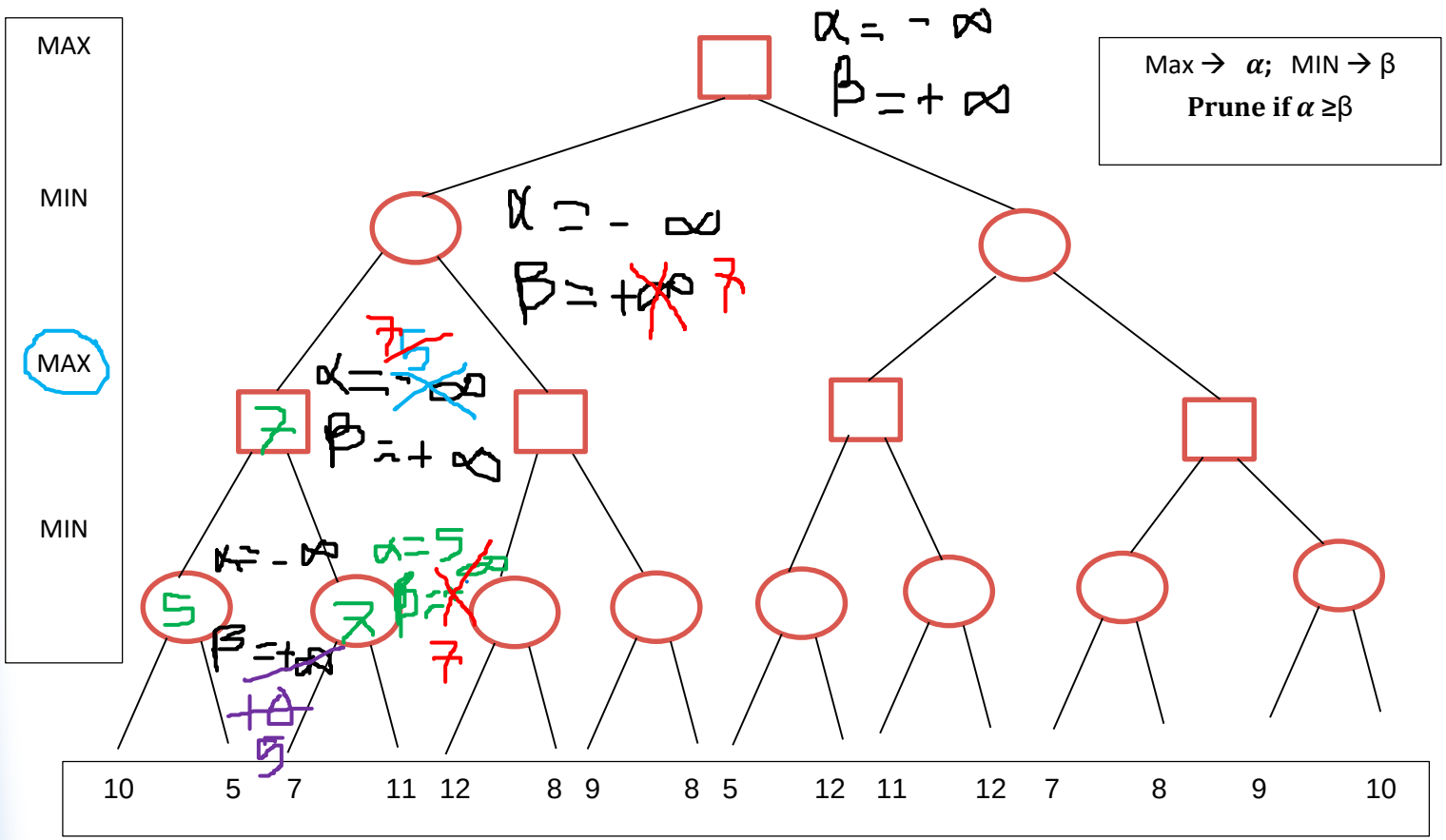

MAX
MIN
MAX
MIN
Max → α; MIN → β
Prune if α ≥β
10 5 7 11 12 8 9 8 5 12 11 12 7 8 9 10

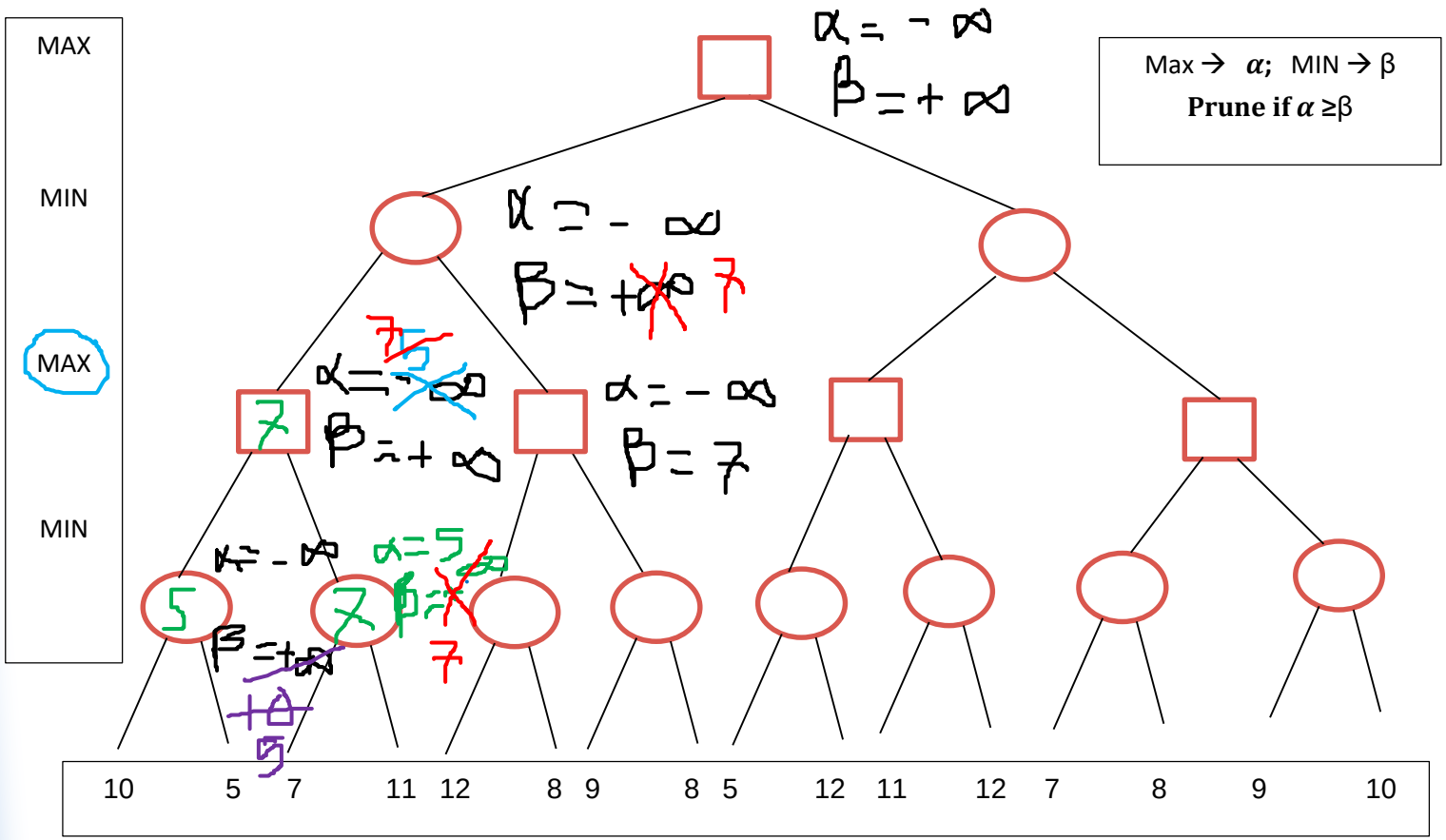

MAX
MIN
MAX
MIN
Max → α; MIN → β
Prune if α ≥β
10 5 7 11 12 8 9 8 5 12 11 12 7 8 9 10

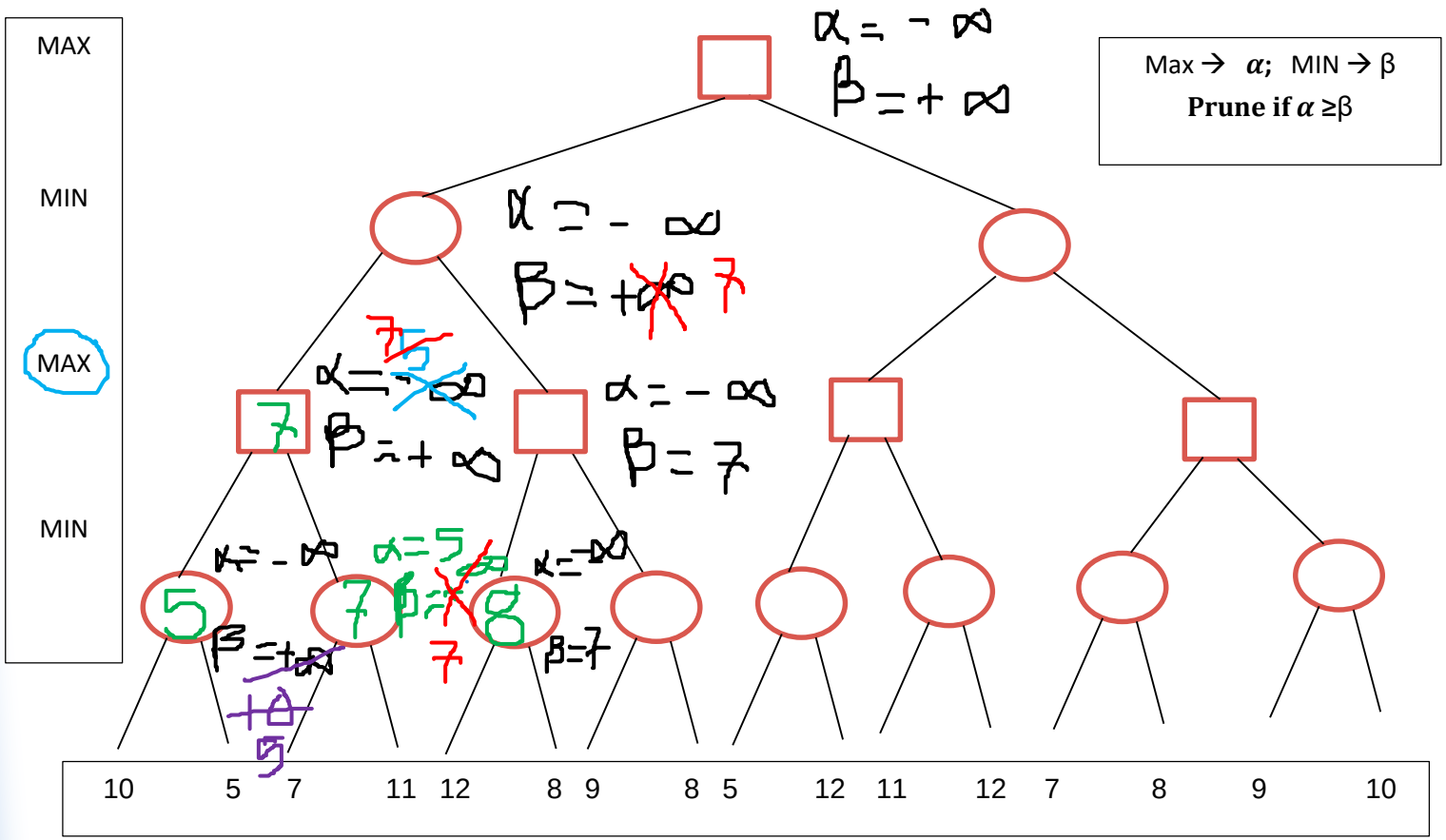

MAX
MIN
MAX
MIN
Max → α; MIN → β
Prune if α ≥β
10 5 7 11 12 8 9 8 5 12 11 12 7 8 9 10

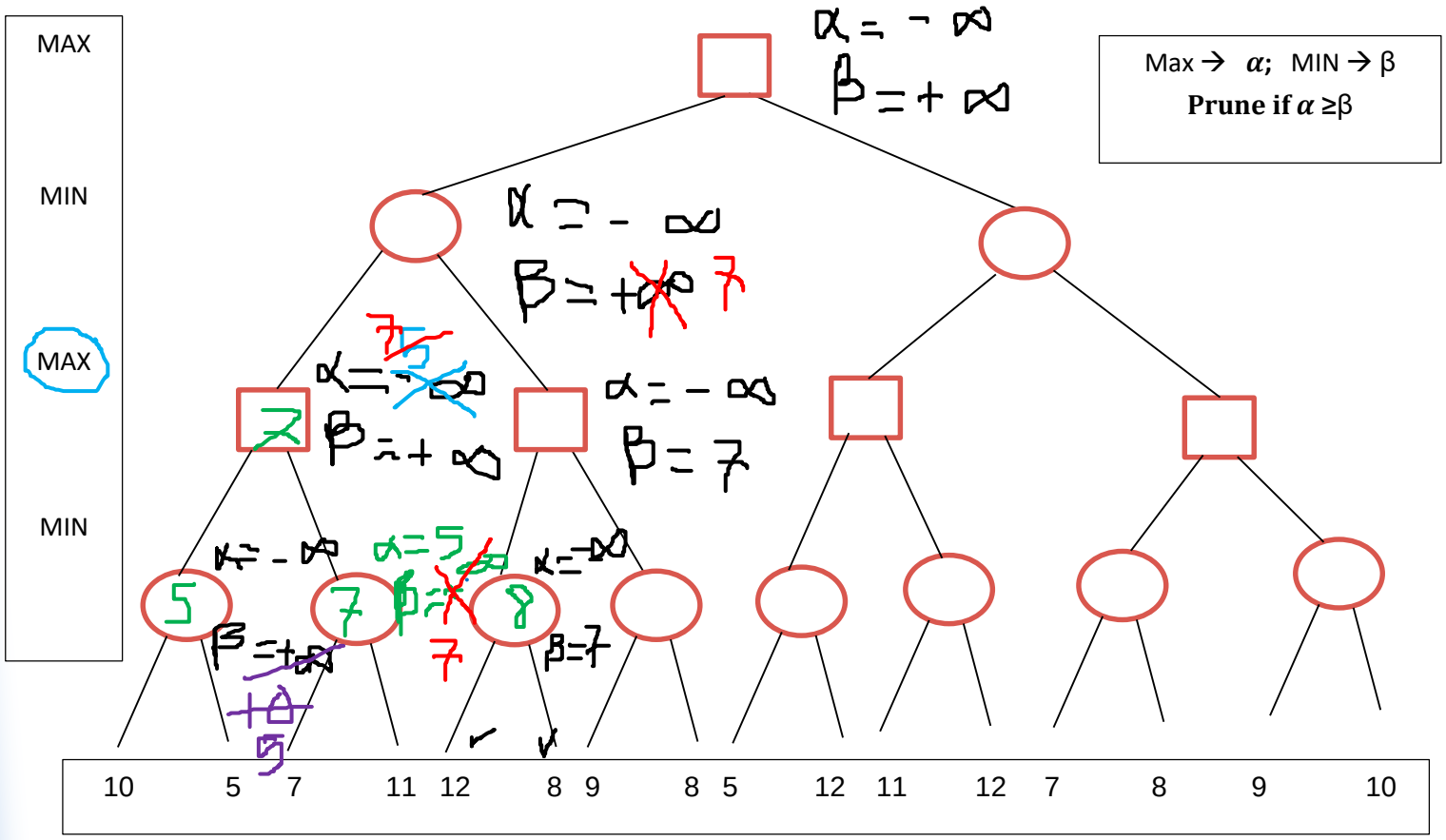

MAX
MIN
MAX
MIN
Max → α; MIN → β
Prune if α ≥β
10 5 7 11 12 8 9 8 5 12 11 12 7 8 9 10

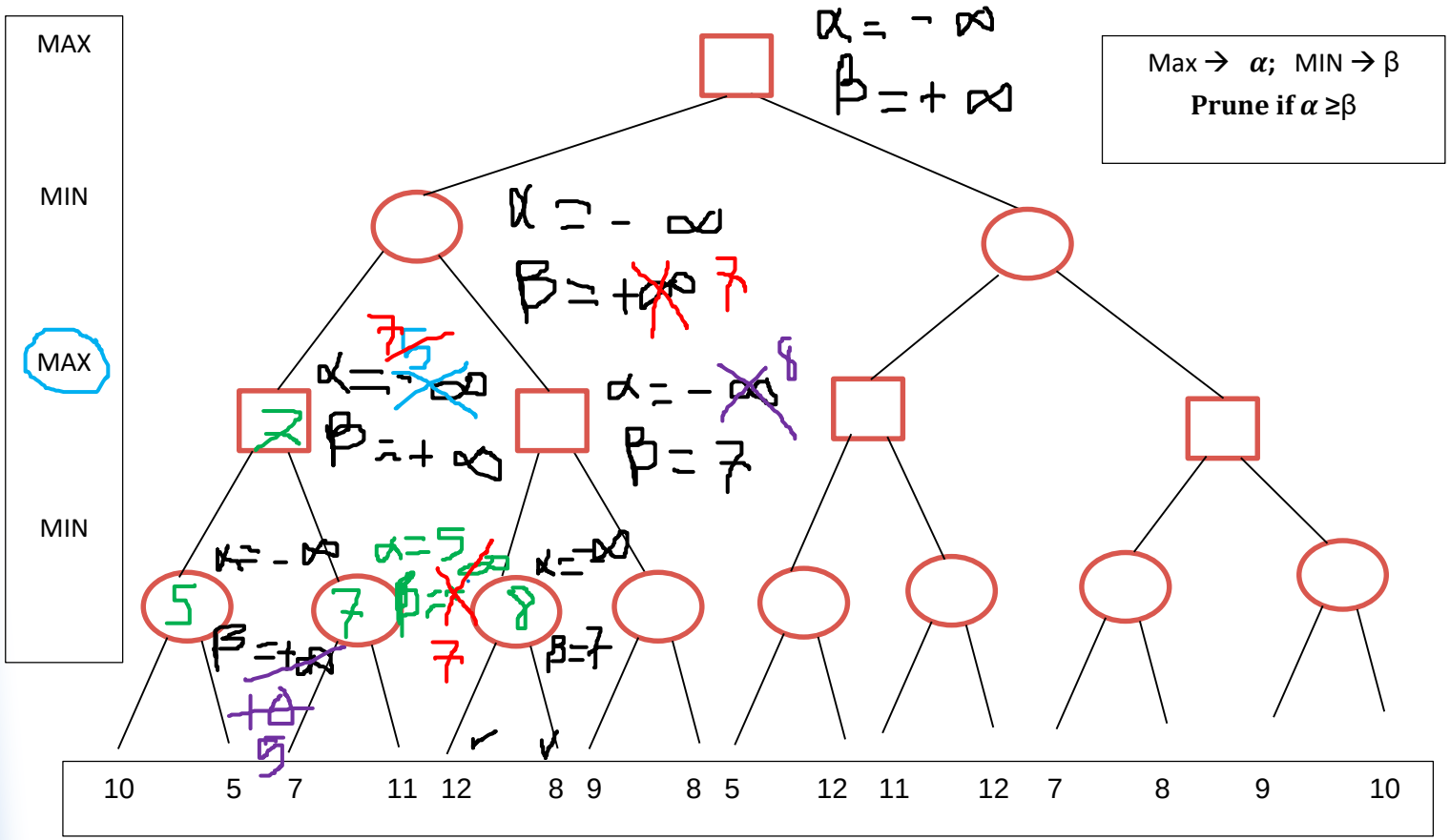

MAX
MIN
MAX
MIN
Max → α; MIN → β
Prune if α ≥β
10 5 7 11 12 8 9 8 5 12 11 12 7 8 9 10

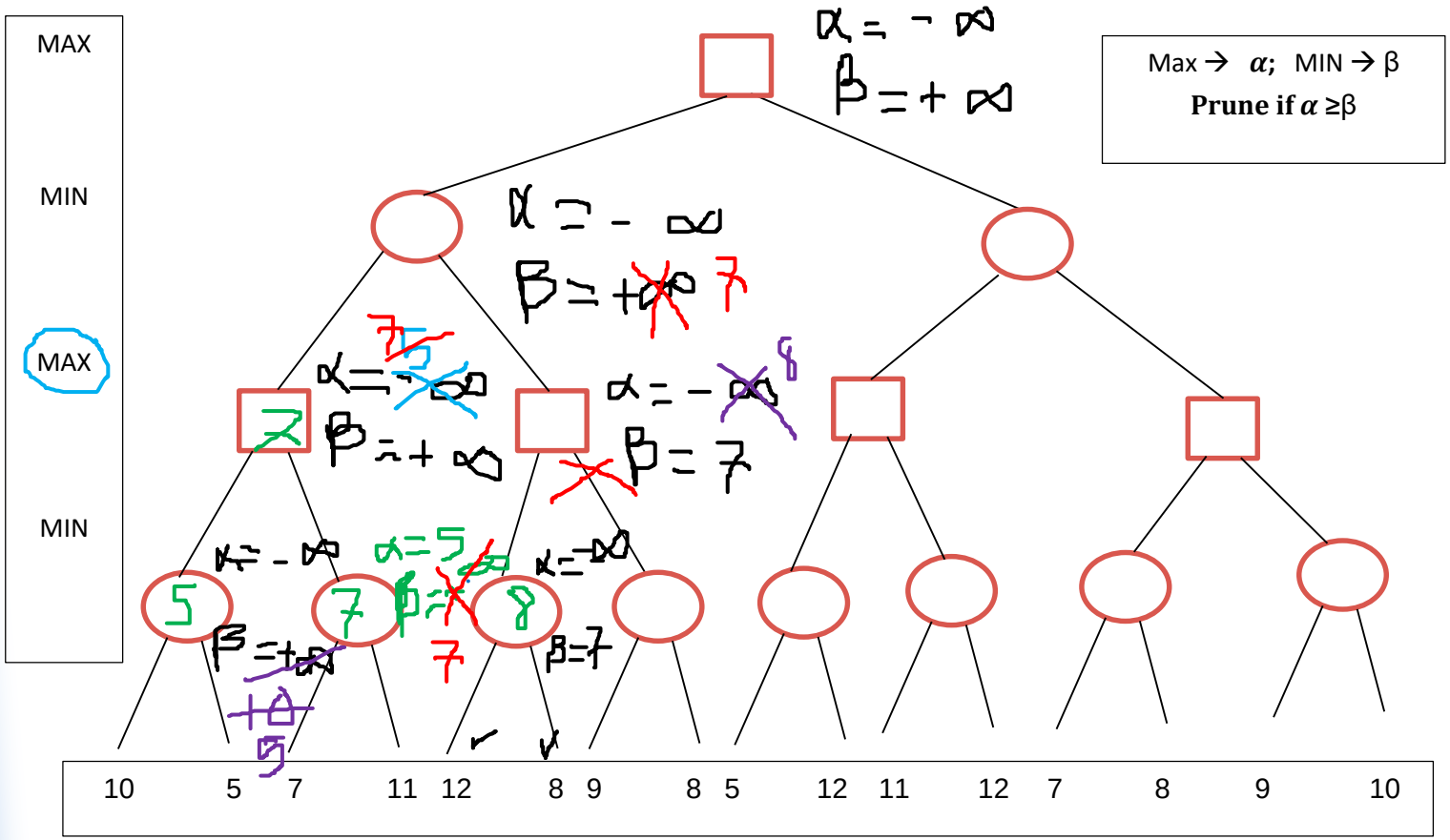

MAX
MIN
MAX
MIN
Max → α; MIN → β
Prune if α ≥β
10 5 7 11 12 8 9 8 5 12 11 12 7 8 9 10

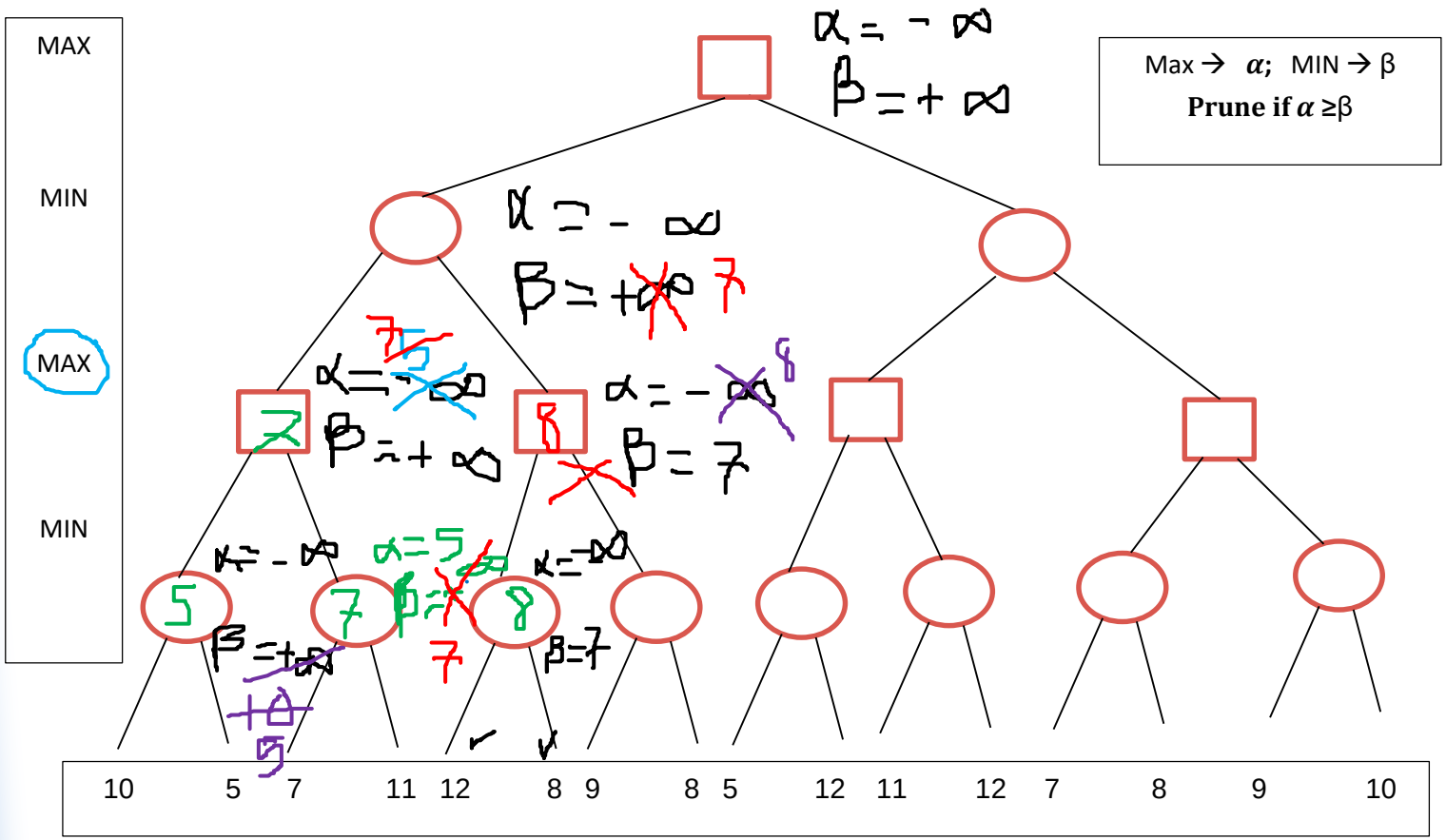

MAX
MIN
MAX
MIN
Max → α; MIN → β
Prune if α ≥β
10 5 7 11 12 8 9 8 5 12 11 12 7 8 9 10

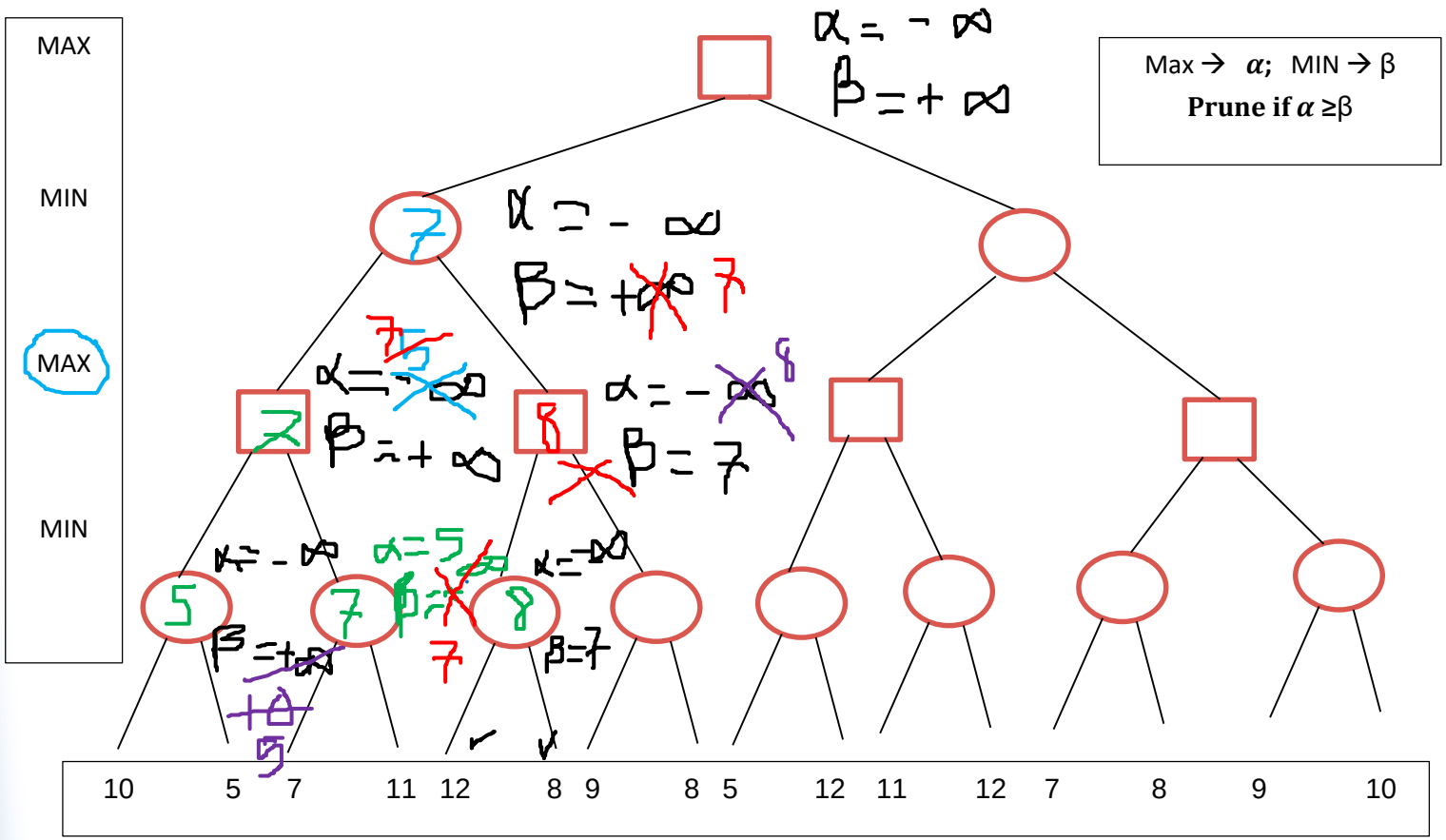

MAX
MIN
MAX
MIN
Max → α; MIN → β
Prune if α ≥β
10 5 7 11 12 8 9 8 5 12 11 12 7 8 9 10

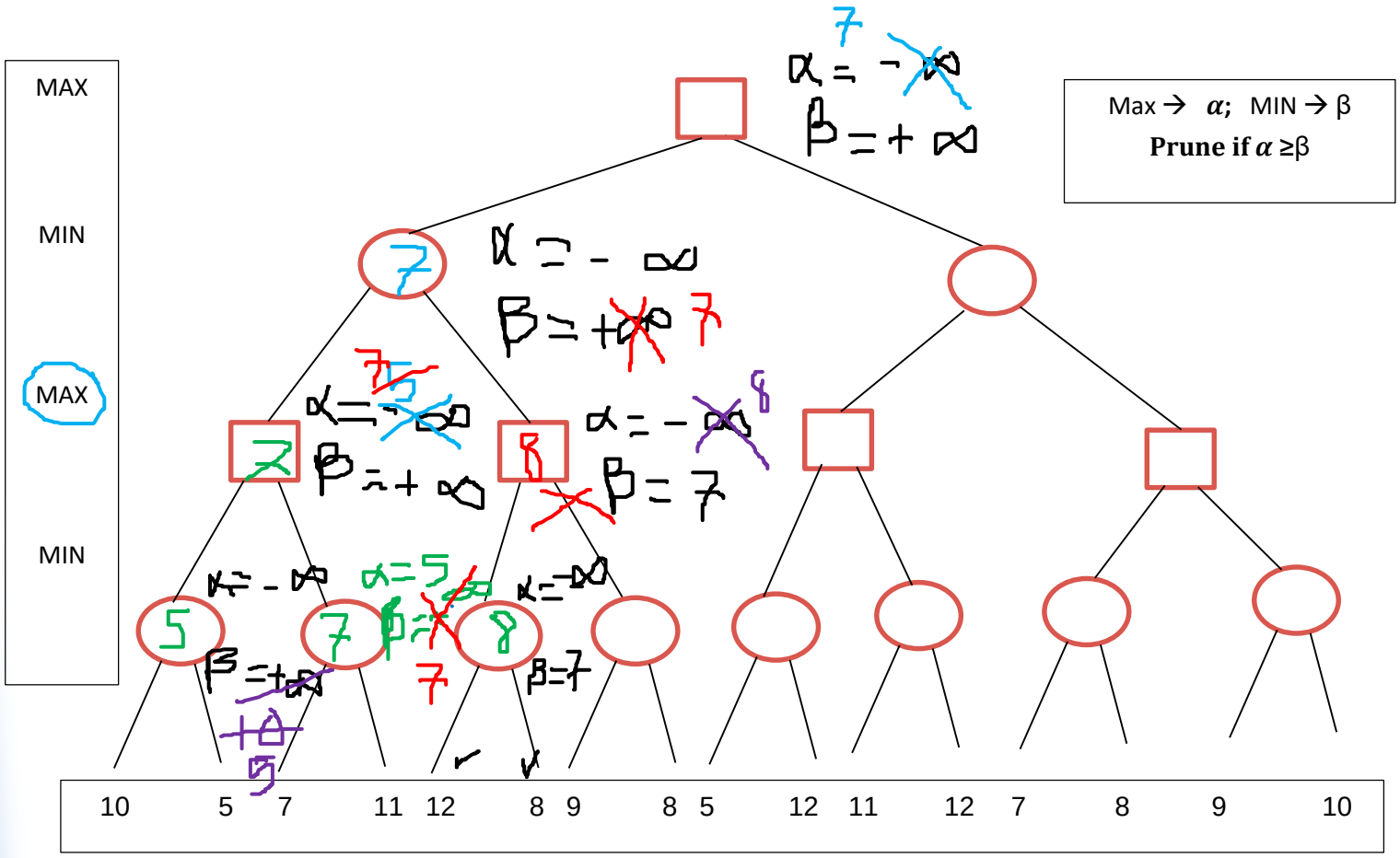

MAX
MIN
MAX
MIN
Max → α; MIN → β
Prune if α ≥β
10 5 7 11 12 8 9 8 5 12 11 12 7 8 9 10

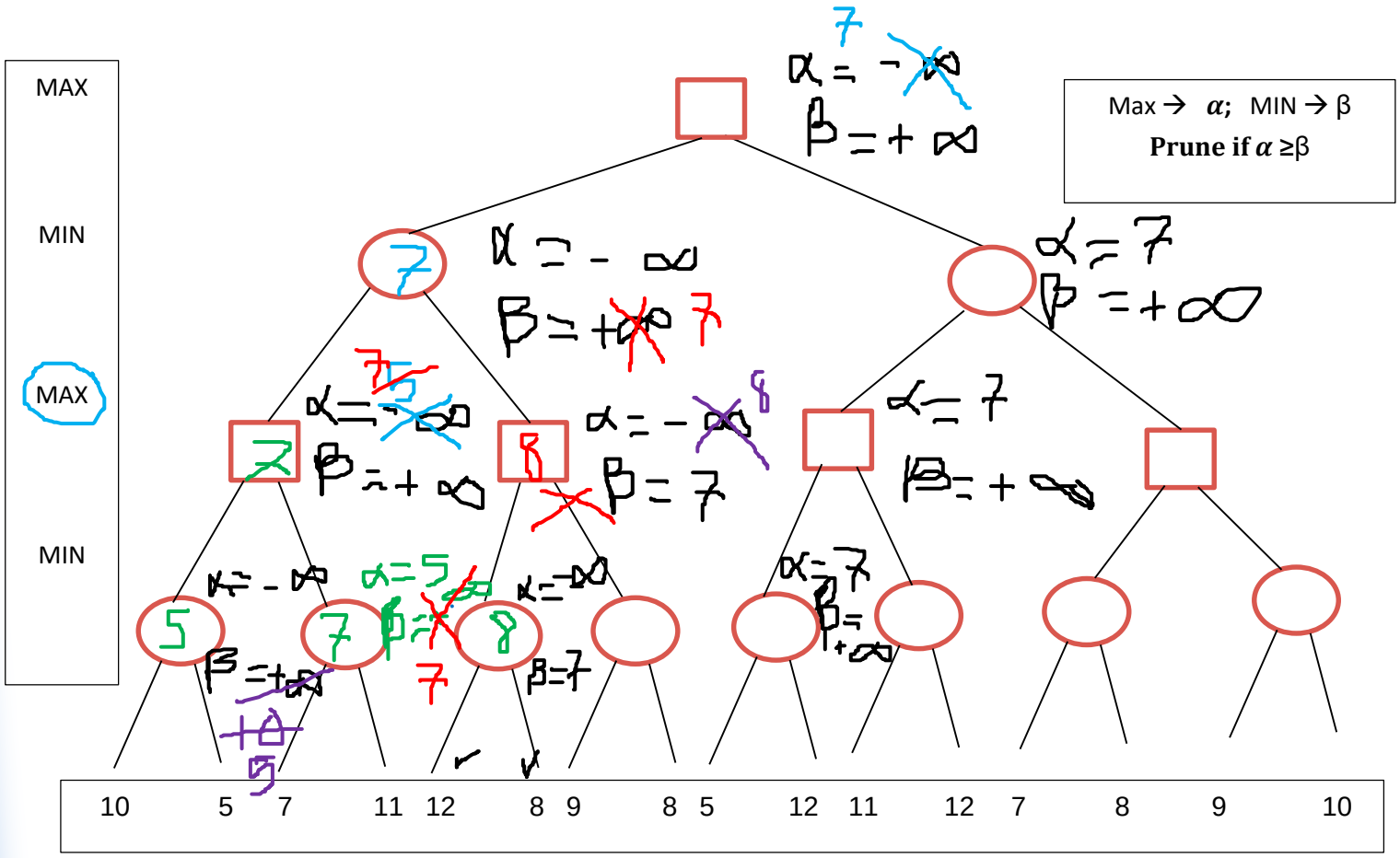

MAX
MIN
MAX
MIN
Max → α; MIN → β
Prune if α ≥β
10 5 7 11 12 8 9 8 5 12 11 12 7 8 9 10

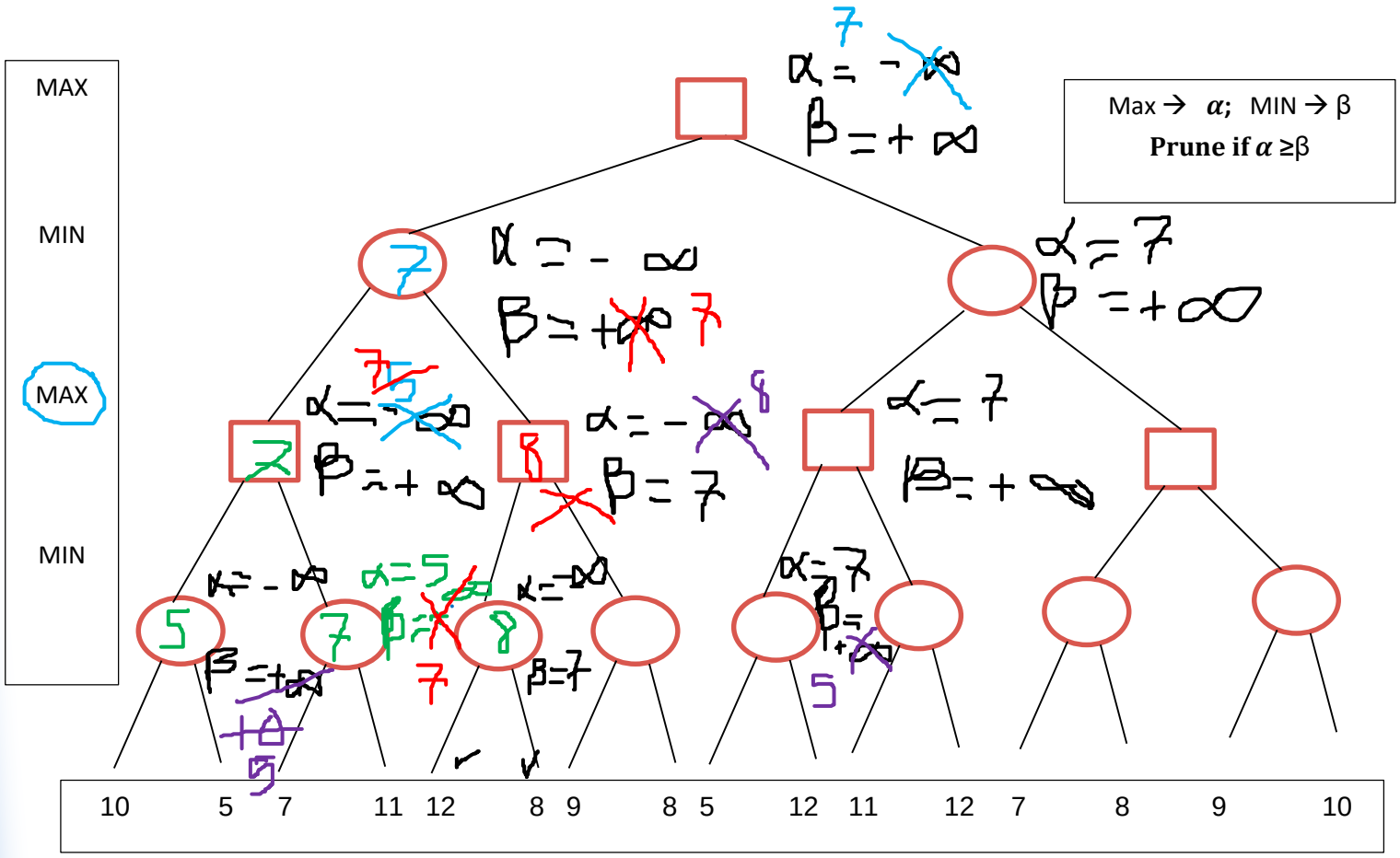

MAX
MIN
MAX
MIN
Max → α; MIN → β
Prune if α ≥β
10 5 7 11 12 8 9 8 5 12 11 12 7 8 9 10

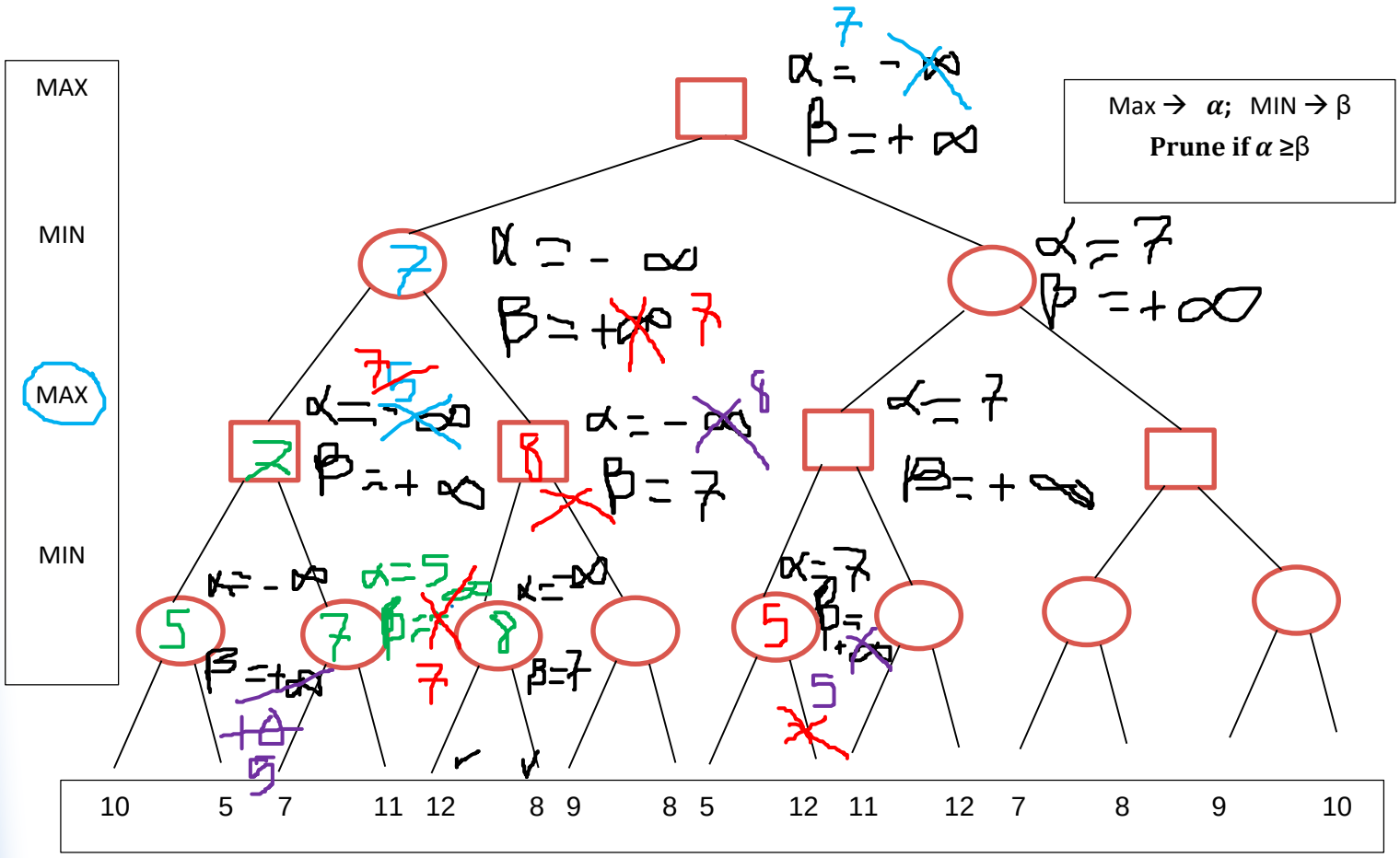

MAX
MIN
MAX
MIN
Max → α; MIN → β
Prune if α ≥β
10 5 7 11 12 8 9 8 5 12 11 12 7 8 9 10

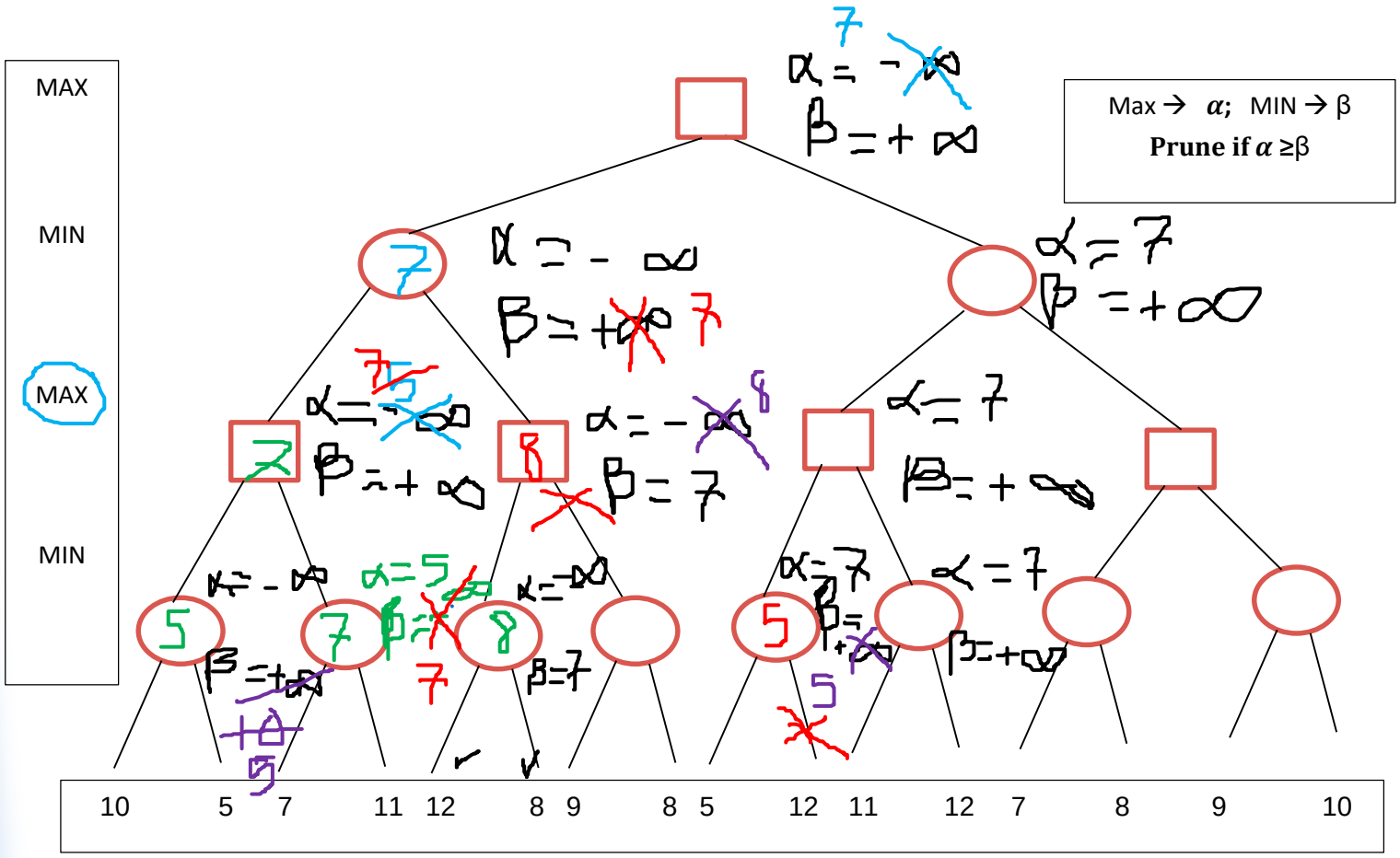

MAX
MIN
MAX
MIN
Max → α; MIN → β
Prune if α ≥β
10 5 7 11 12 8 9 8 5 12 11 12 7 8 9 10

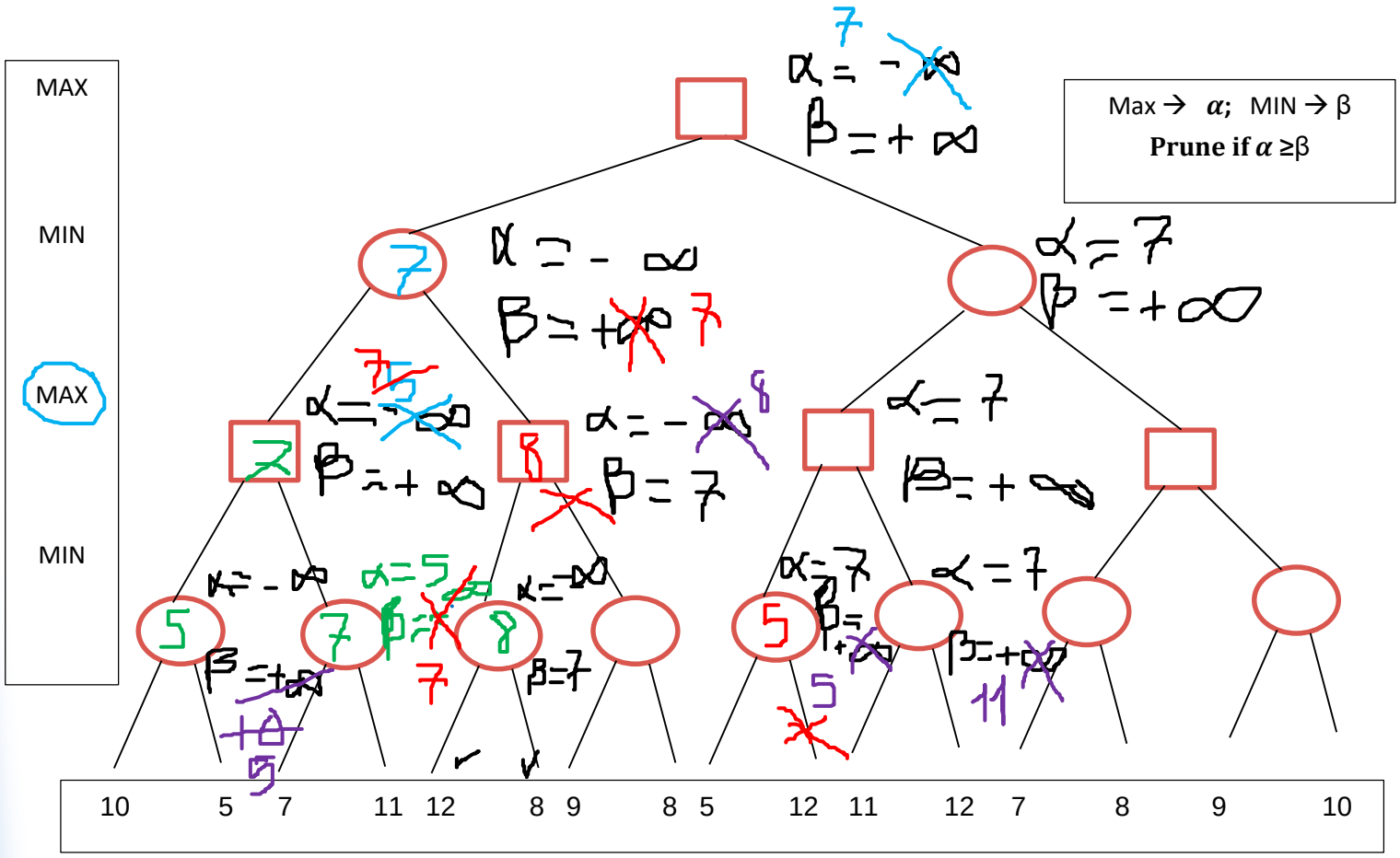

MAX
MIN
MAX
MIN
Max → α; MIN → β
Prune if α ≥β
10 5 7 11 12 8 9 8 5 12 11 12 7 8 9 10

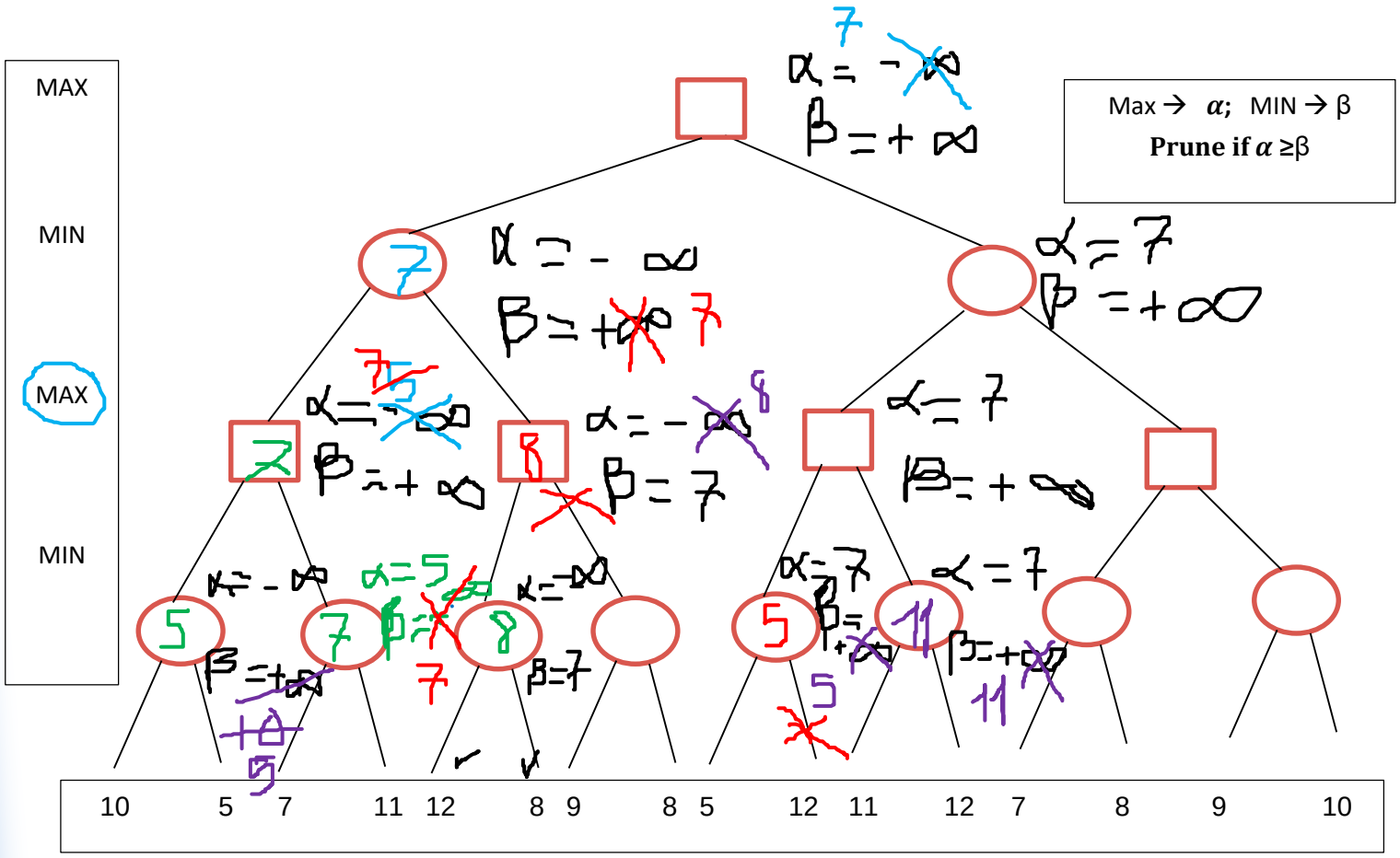

MAX
MIN
MAX
MIN
Max → α; MIN → β
Prune if α ≥β
10 5 7 11 12 8 9 8 5 12 11 12 7 8 9 10

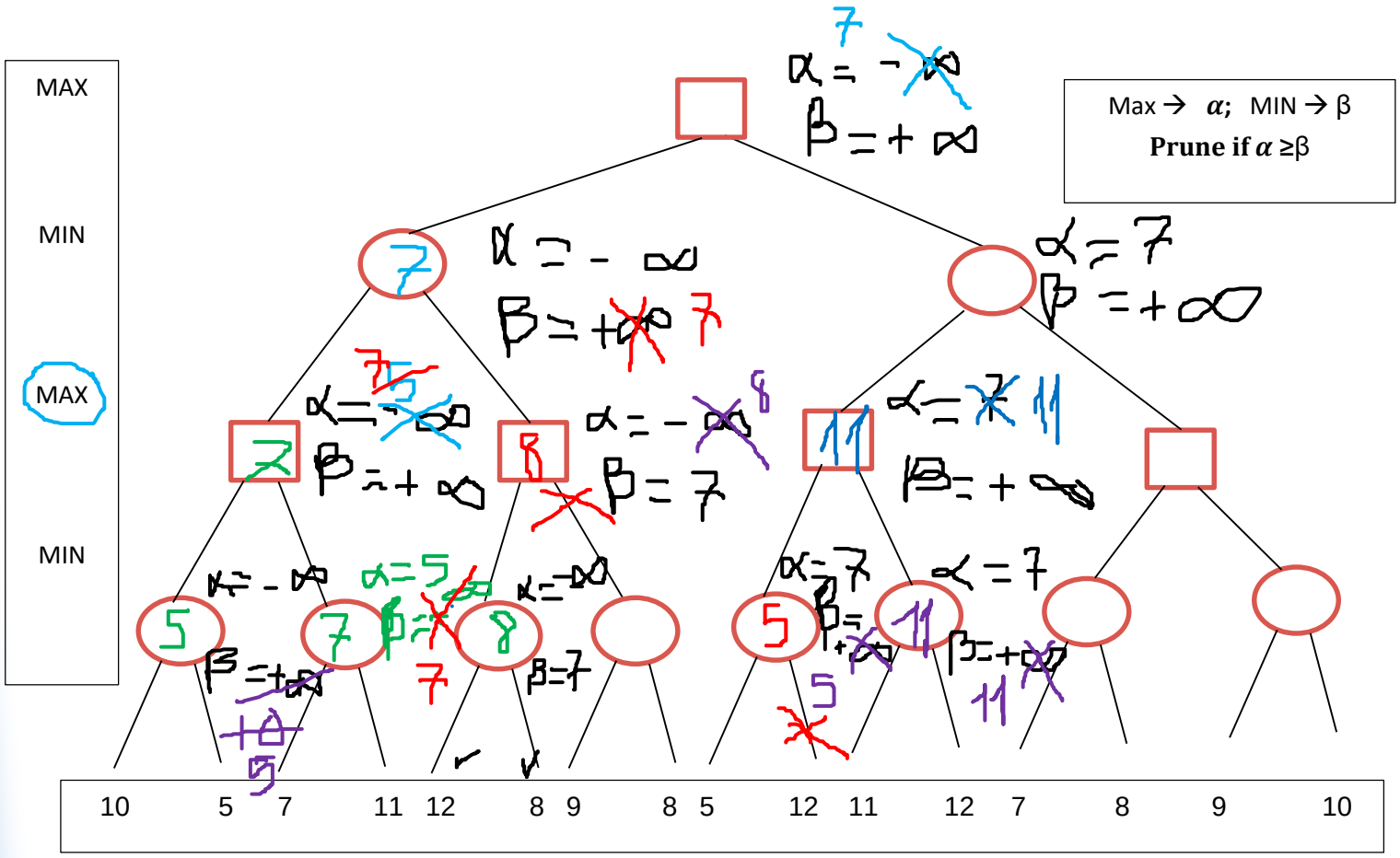

MAX
MIN
MAX
MIN
Max → α; MIN → β
Prune if α ≥β
10 5 7 11 12 8 9 8 5 12 11 12 7 8 9 10

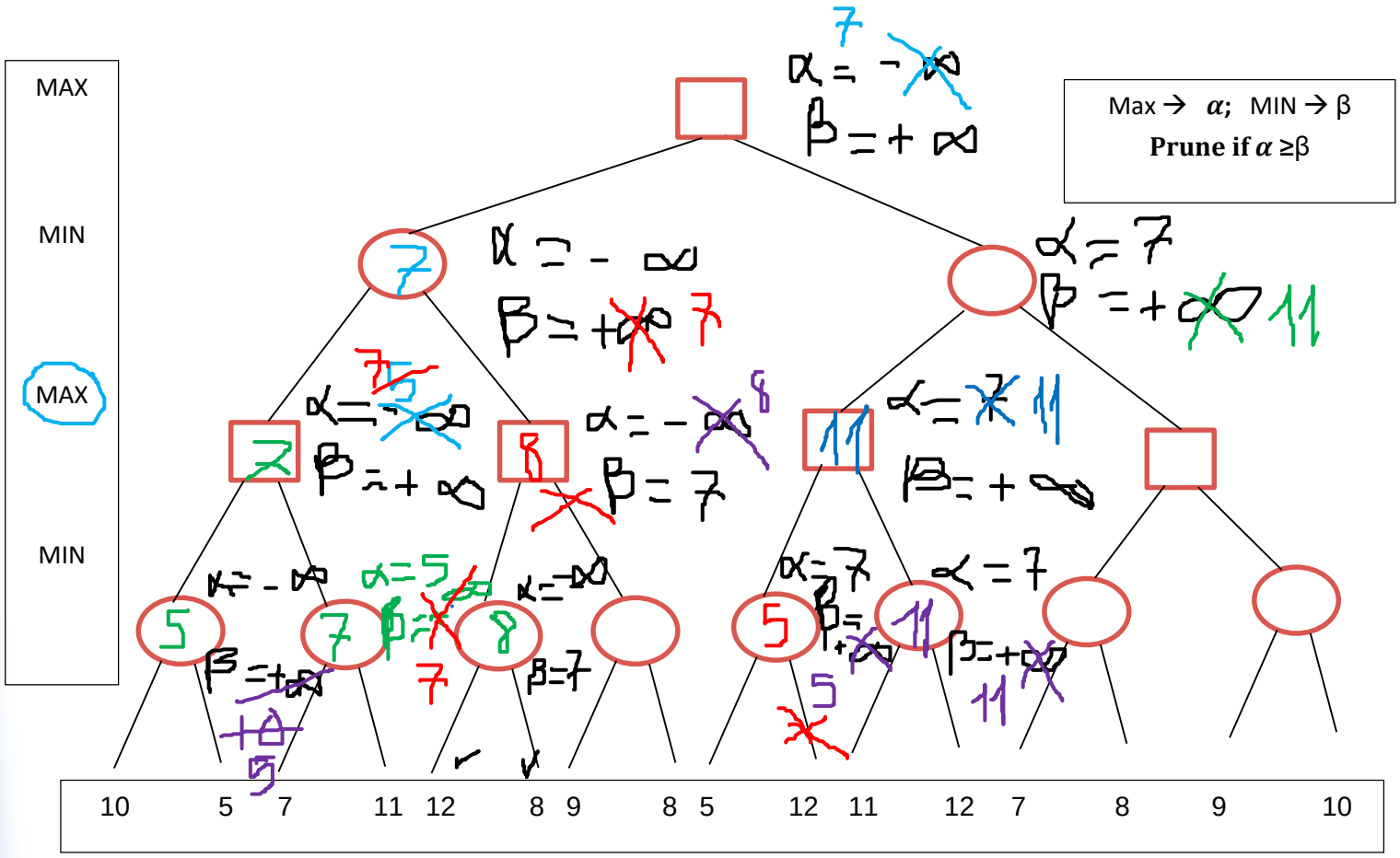

MAX
MIN
MAX
MIN
Max → α; MIN → β
Prune if α ≥β
10 5 7 11 12 8 9 8 5 12 11 12 7 8 9 10

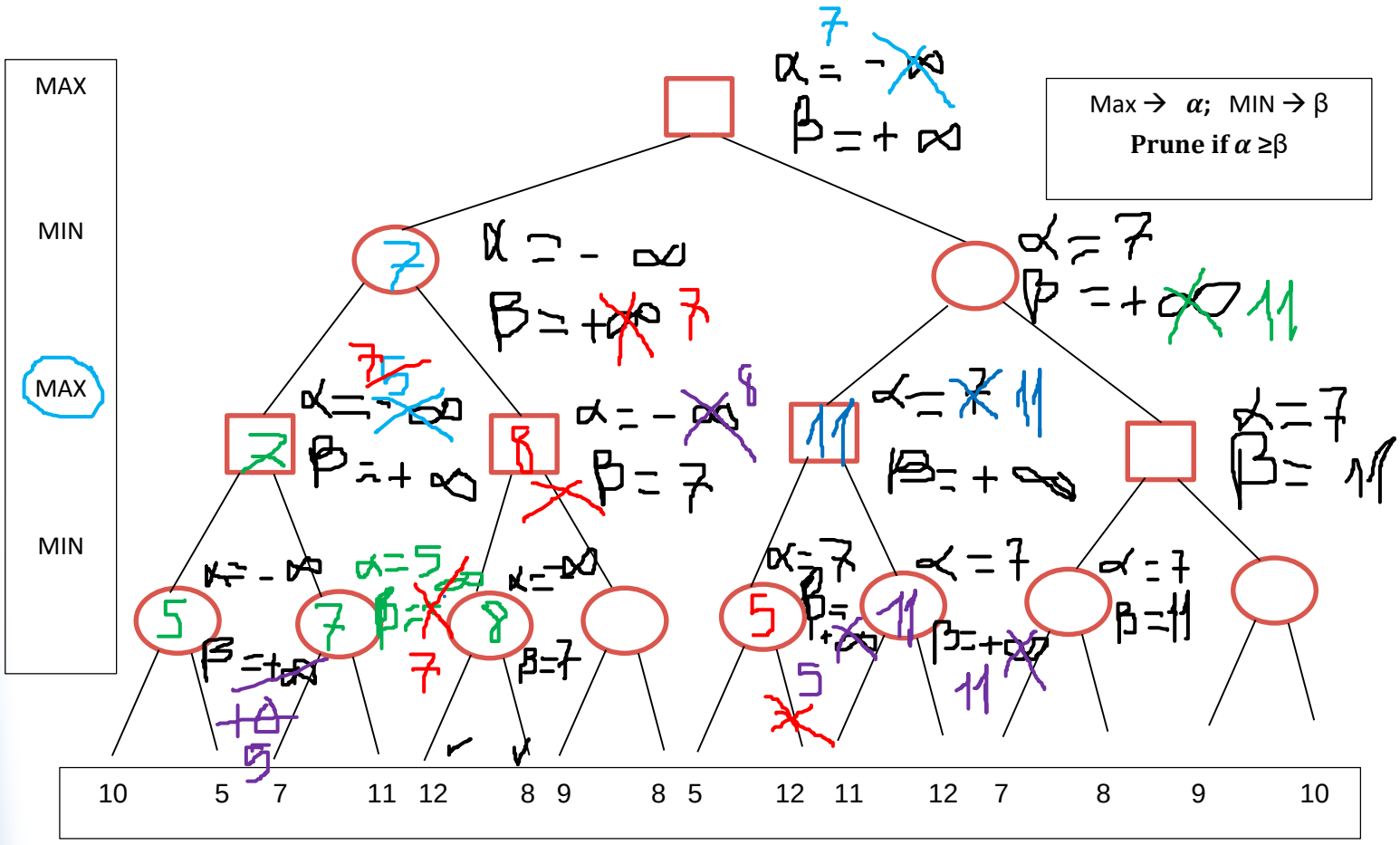

MAX
MIN
MAX
MIN
Max → α; MIN → β
Prune if α ≥β
10 5 7 11 12 8 9 8 5 12 11 12 7 8 9 10

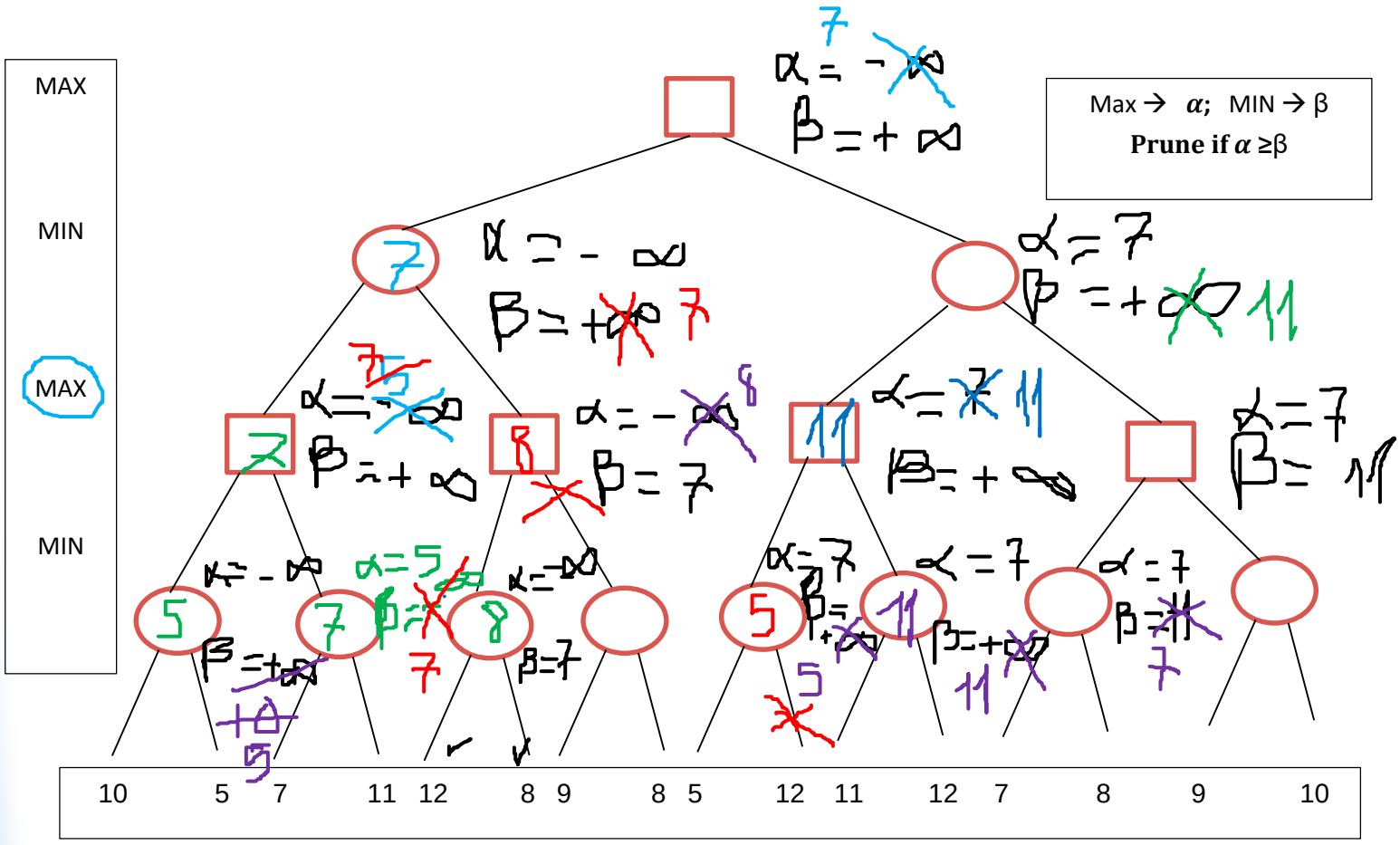

MAX
MIN
MAX
MIN
Max → α; MIN → β
Prune if α ≥β
10 5 7 11 12 8 9 8 5 12 11 12 7 8 9 10

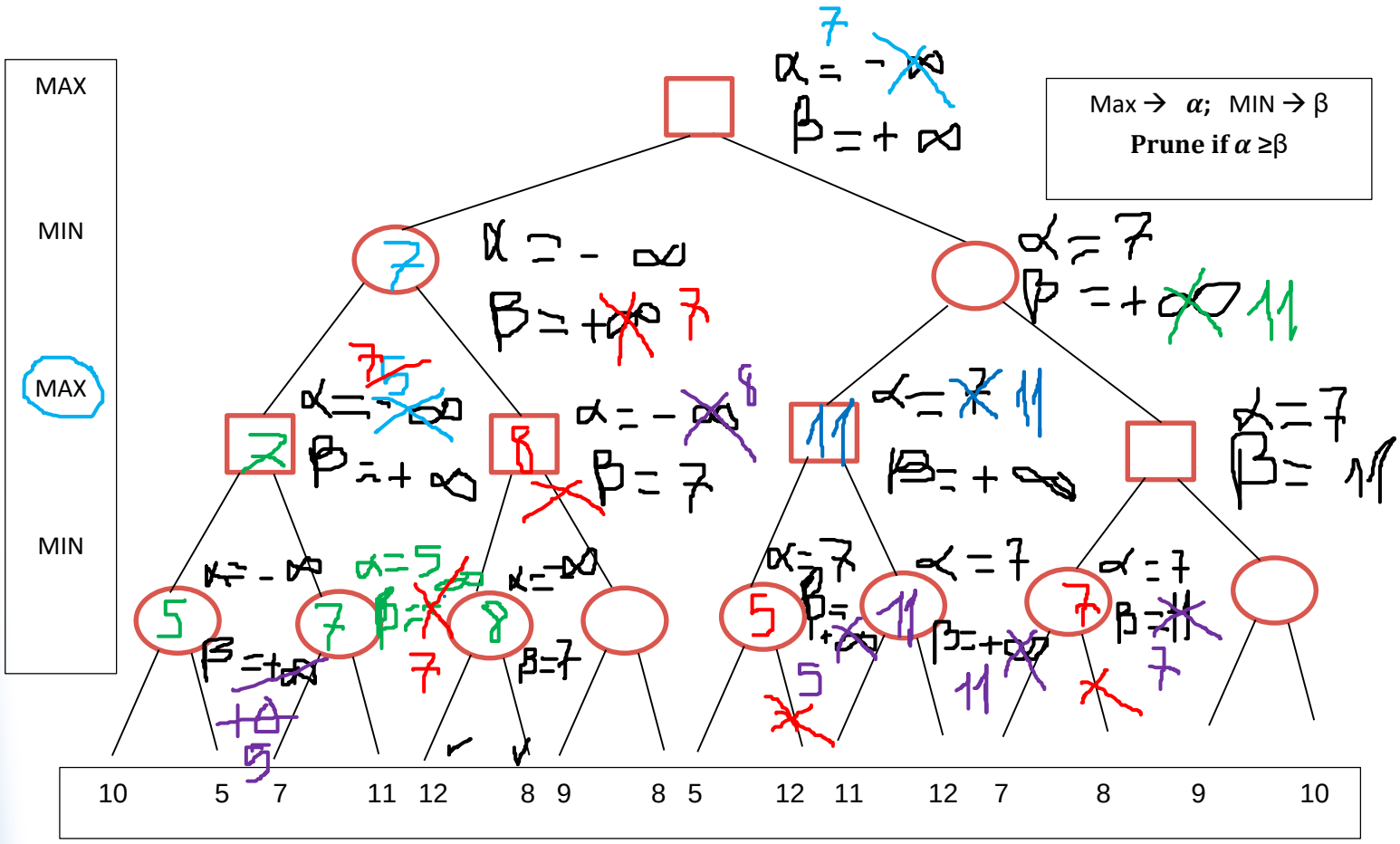

MAX
MIN
MAX
MIN
Max → α; MIN → β
Prune if α ≥β
10 5 7 11 12 8 9 8 5 12 11 12 7 8 9 10

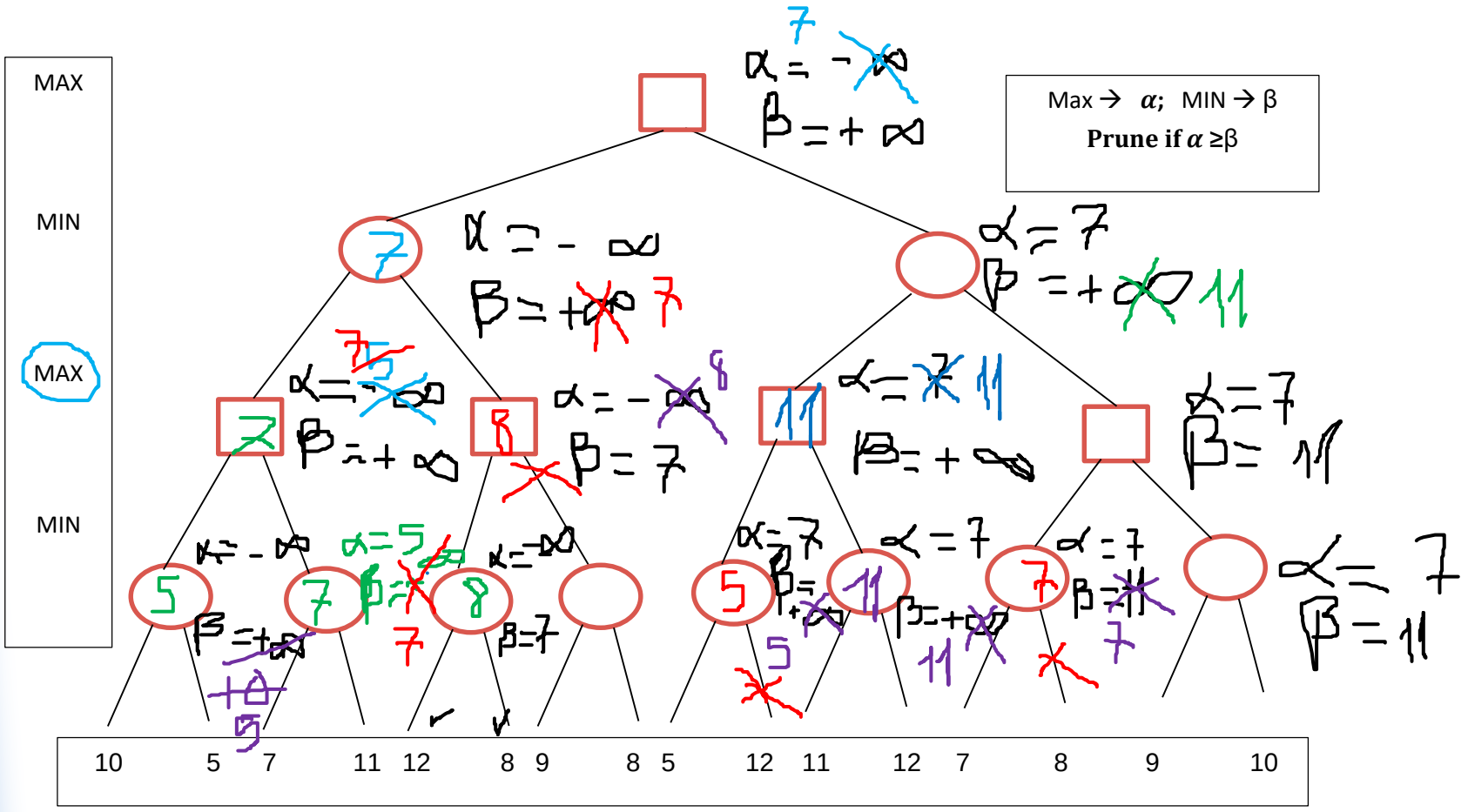

MAX
MIN
MAX
MIN
Max → α; MIN → β
Prune if α ≥β
10 5 7 11 12 8 9 8 5 12 11 12 7 8 9 10

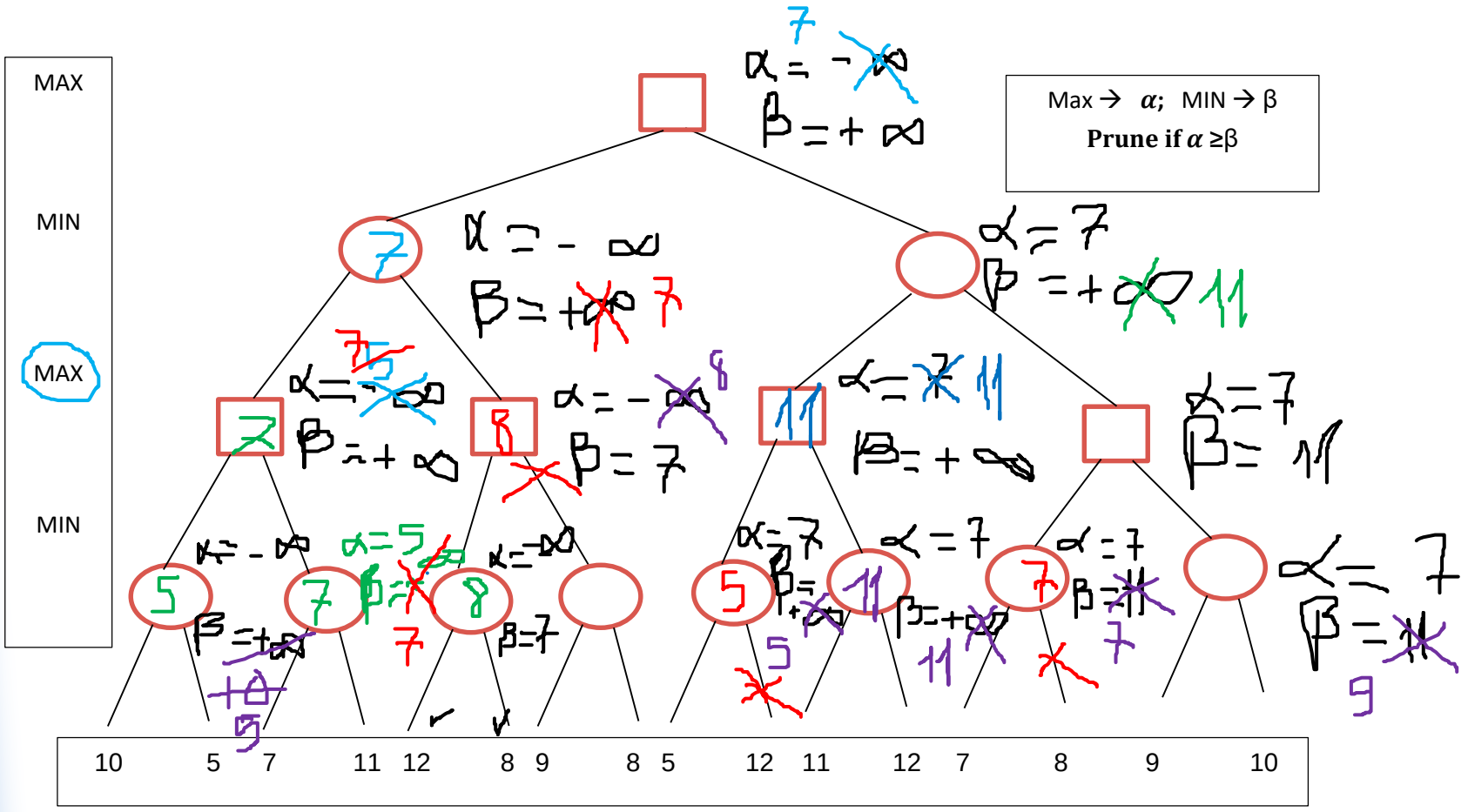

MAX
MIN
MAX
MIN
Max → α; MIN → β
Prune if α ≥β
10 5 7 11 12 8 9 8 5 12 11 12 7 8 9 10

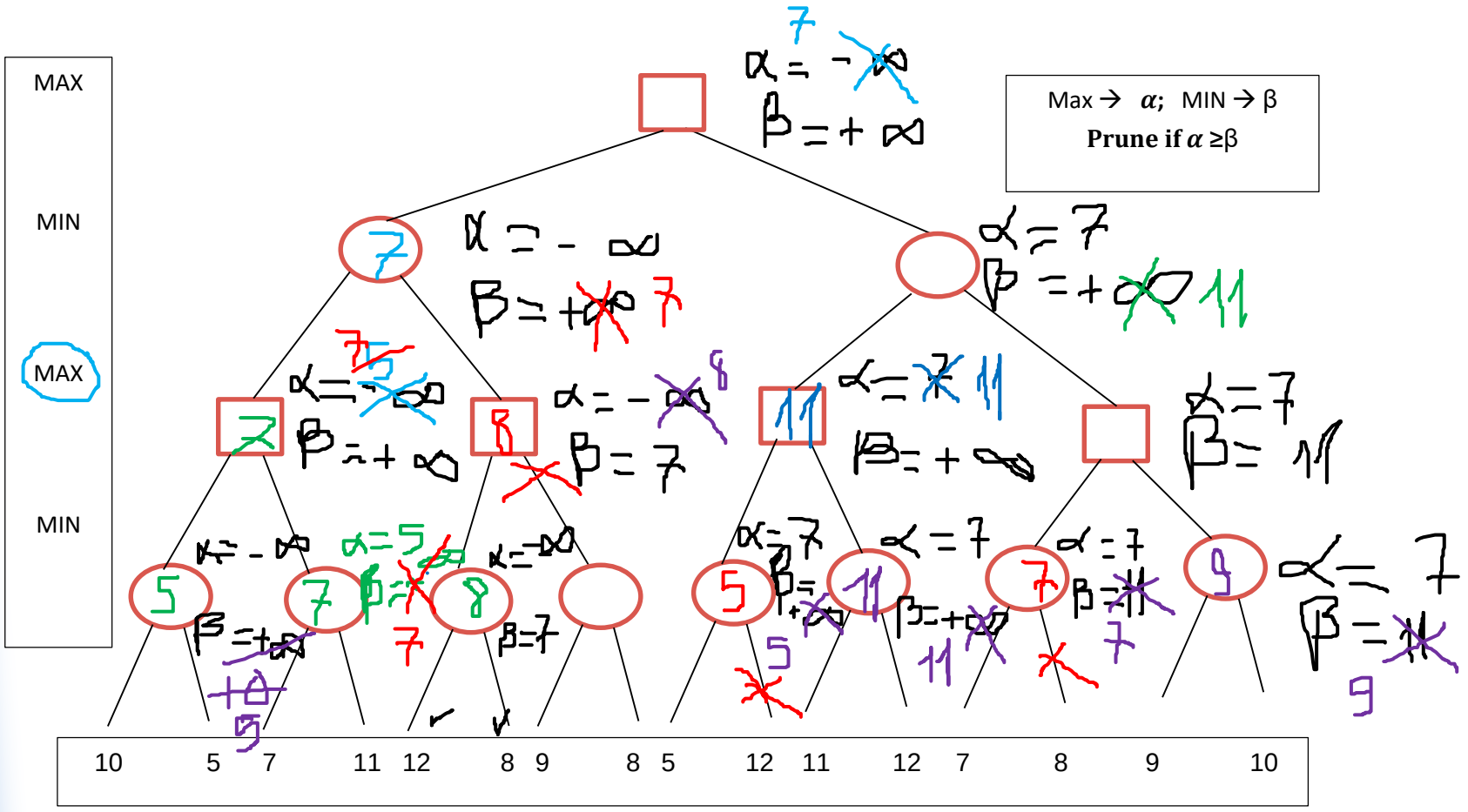

MAX
MIN
MAX
MIN
Max → α; MIN → β
Prune if α ≥β
10 5 7 11 12 8 9 8 5 12 11 12 7 8 9 10

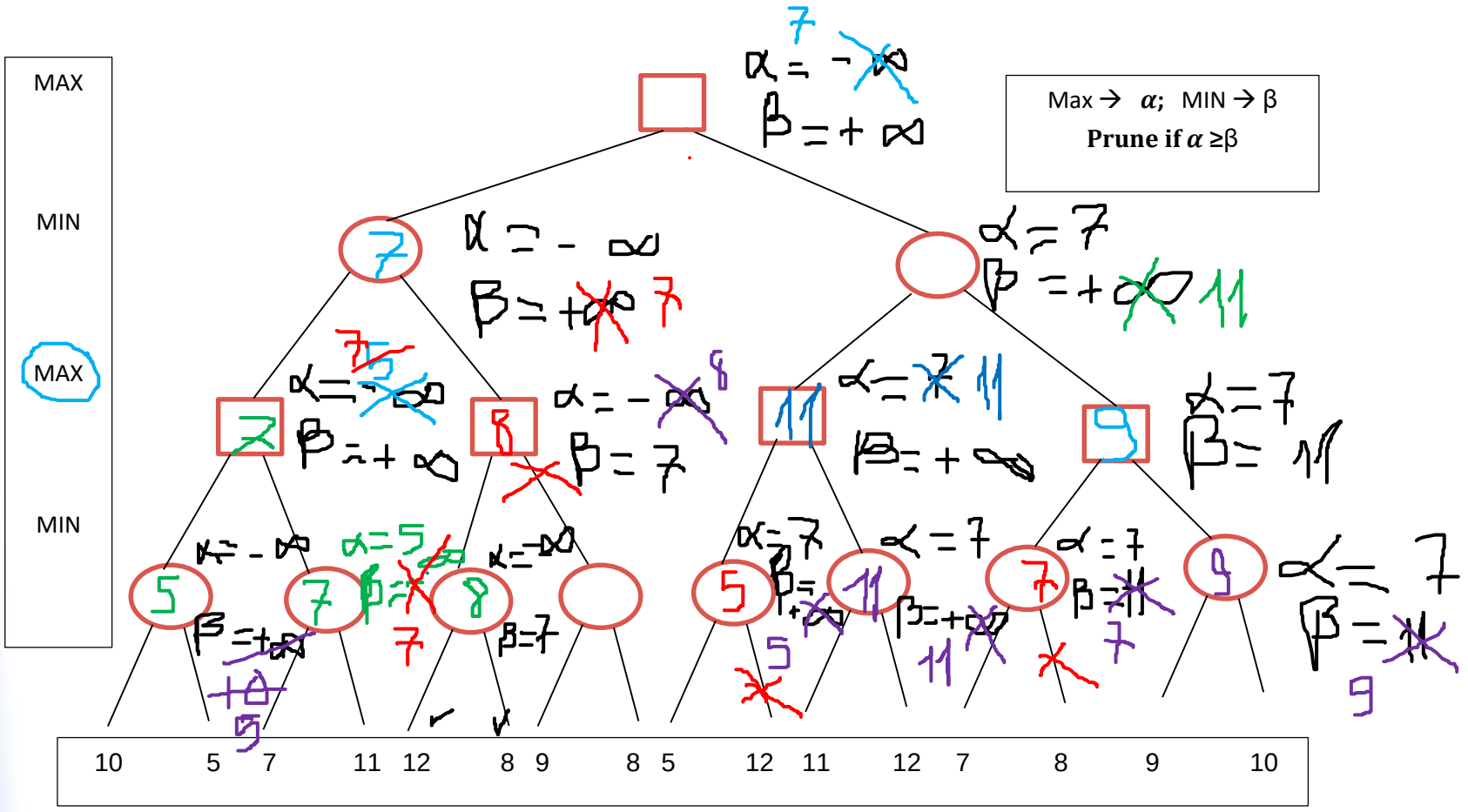

MAX
MIN
MAX
MIN
Max → α; MIN → β
Prune if α ≥β
10 5 7 11 12 8 9 8 5 12 11 12 7 8 9 10

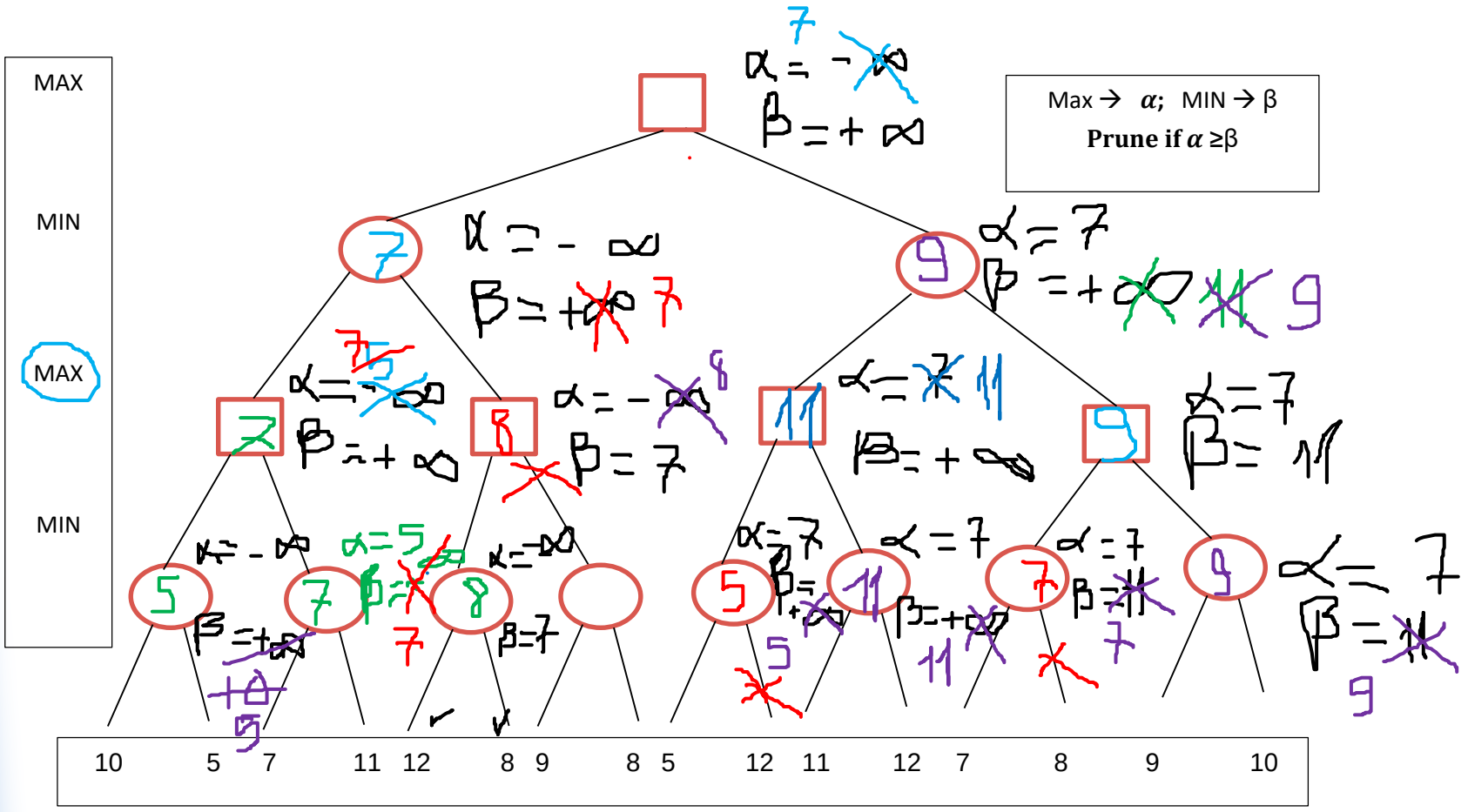

MAX
MIN
MAX
MIN
Max → α; MIN → β
Prune if α ≥β
10 5 7 11 12 8 9 8 5 12 11 12 7 8 9 10

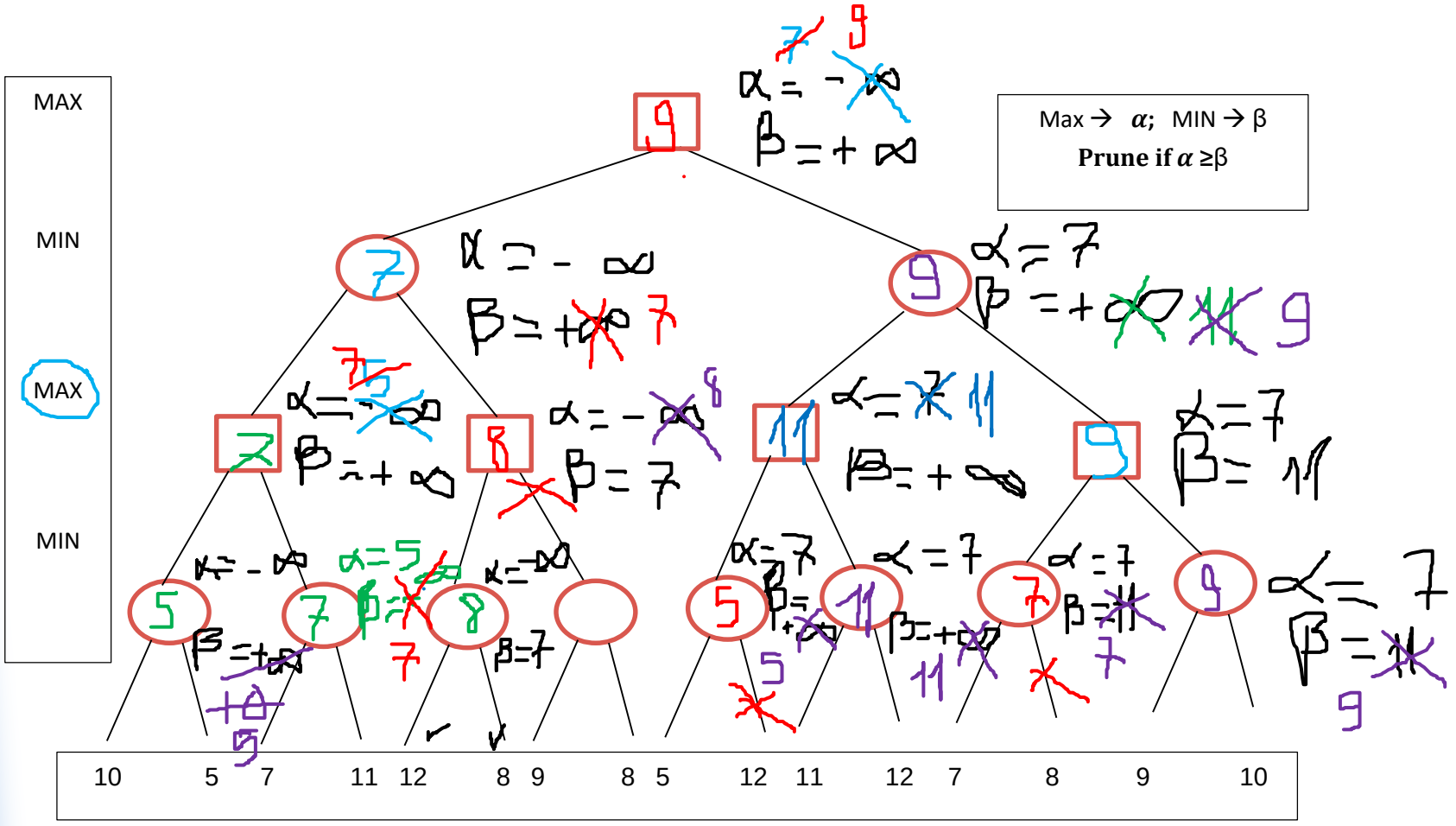

MAX
MIN
MAX
MIN
Max → α; MIN → β
Prune if α ≥β
10 5 7 11 12 8 9 8 5 12 11 12 7 8 9 10


MAX
MIN
MAX
MIN
Max → α; MIN → β
Prune if α ≥β
10 5 7 11 12 8 9 8 5 12 11 12 7 8 9 10


MAX
MIN
MAX
MIN
Max → α; MIN → β
Prune if α ≥β
10 5 7 11 12 8 9 8 5 12 11 12 7 8 9 10


MAX
MIN
MAX
MIN
Max → α; MIN → β
Prune if α ≥β
10 5 7 11 12 8 9 8 5 12 11 12 7 8 9 10


MAX
MIN
MAX
MIN
Max → α; MIN → β
Prune if α ≥β
10 5 7 11 12 8 9 8 5 12 11 12 7 8 9 10


MAX
MIN
MAX
MIN
Max → α; MIN → β
Prune if α ≥β
10 5 7 11 12 8 9 8 5 12 11 12 7 8 9 10

```
function ALPHA-BETA-SEARCH(state) returns an action
    v ← MAX-VALUE(state, −∞, +∞)
    return the action in ACTIONS(state) with value v

function MAX-VALUE(state, α, β) returns a utility value
    if TERMINAL-TEST(state) then return UTILITY(state)
    v ← −∞
    for each a in ACTIONS(state) do
        v ← MAX(v, MIN-VALUE(RESULT(s,a), α, β))
        if v ≥ β then return v
        α ← MAX(α, v)
    return v

function MIN-VALUE(state, α, β) returns a utility value
    if TERMINAL-TEST(state) then return UTILITY(state)
    v ← +∞
    for each a in ACTIONS(state) do
        v ← MIN(v, MAX-VALUE(RESULT(s,a) , α, β))
        if v ≤ α then return v
        β ← MIN(β, v)
    return v
```

Figure 5.7 The alpha–beta search algorithm. Notice that these routines are the same as the MINIMAX functions in Figure 5.3, except for the two lines in each of MIN-VALUE and MAX-VALUE that maintain α and β (and the bookkeeping to pass these parameters along).

Alpha–Beta Pruning: Move Ordering

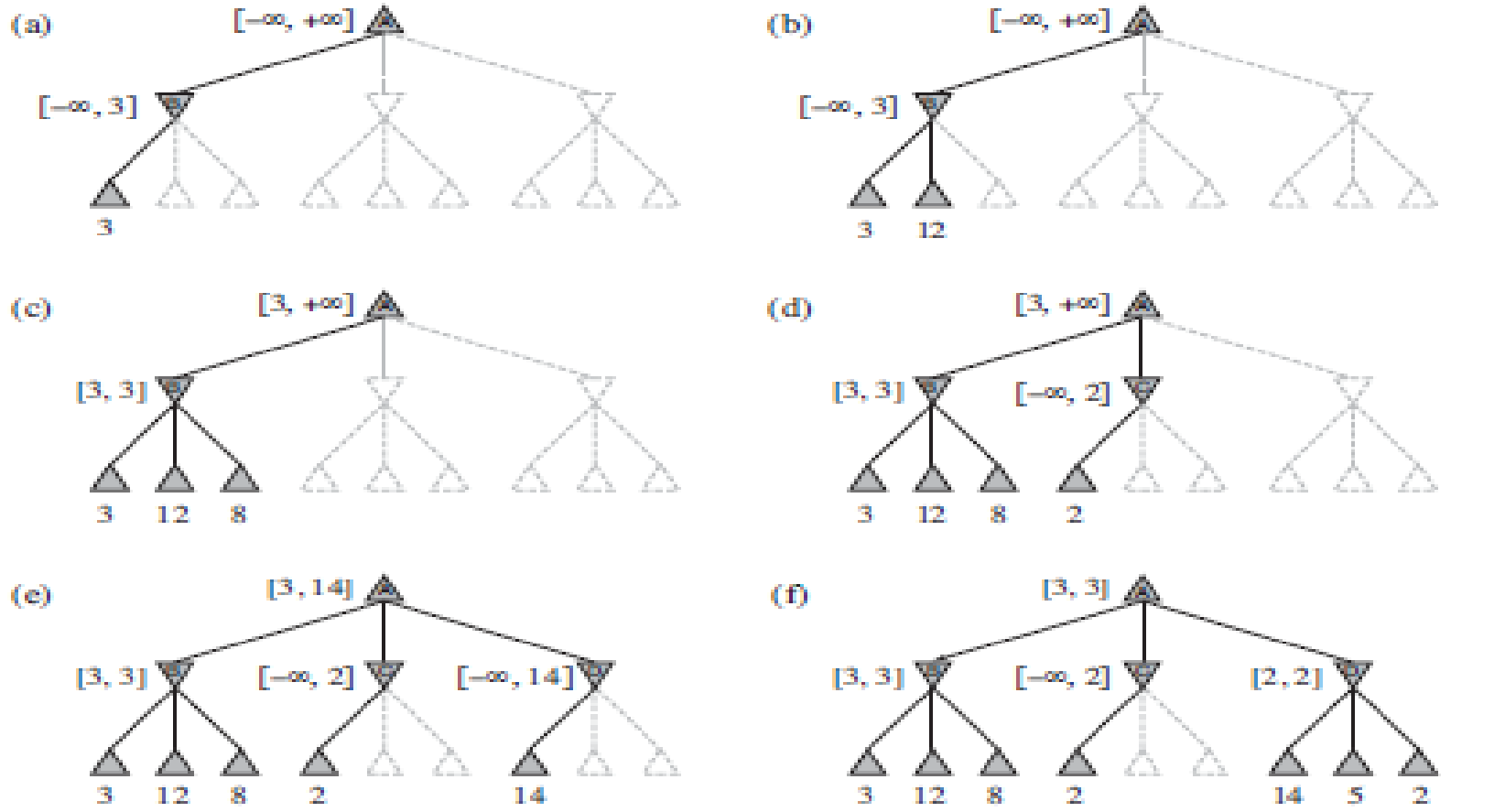


Figure 5.5 Stages in the calculation of the optimal decision for the game tree in Figure 5.2. At each point, we show the range of possible values for each node. (a) The first leaf below B has the value 3. Hence, B, which is a MIN node, has a value of *at most* 3. (b) The second leaf below B has a value of 12; MIN would avoid this move, so the value of B is still at most 3. (c) The third leaf below B has a value of 8; we have seen all B's successor states, so the

Move Ordering

- The effectiveness of alpha–beta pruning is highly dependent on the order in which the states are examined.
- Example,
  - In previous figure cases (e) and (f), we could not prune any successors of D because the worst successors (for MIN) were generated first.
  - If the third successor of D had been generated first, we would have been able to prune the other two.
- This suggests that it might be worth it to try to examine first the successors that are likely to be best.

Move Ordering

- If this can be done, then it turns out that alpha–beta needs to examine only $O(b^{m/2})$ nodes to pick the best move, instead of $O(b^m)$ for minimax.

  ➔ the effective branching factor becomes $\sqrt{b}$ instead of b.

  (For chess, this is ~6 instead of 35).
- I.e., alpha–beta can solve a tree roughly twice as deep as minimax in the same amount of time.
- If successors are examined in random order rather than best-first, the total number of nodes examined will be roughly $O(b^{3m/4})$.
- For chess, a fairly simple ordering function (such as trying captures first, then threats, then forward moves, and then backward moves) gets you to within about a factor of 2 of the best-case $O(b^{m/2})$ result.

Move Ordering

- Adding dynamic move-ordering schemes, such as trying first the moves that were found to be best in the past, brings us quite close to the theoretical limit.
- One way to gain information from the current move is with iterative deepening search.
  - First, search 1 ply deep and record the best path of moves.
  - Then search 1 ply deeper, but use the recorded path to inform move ordering.
- The best moves are often called **killer moves** and to try them first is called the killer move heuristic.

Move Ordering

- In many games, repeated states occur frequently because of **transpositions**— i.e. different permutations of the move sequence that end up in the same position.
- ➔ better to store the evaluation of the resulting position in a hash table the first time it is encountered so that we don't have to recompute it on subsequent occurrences.
- The hash table of previously seen positions is traditionally called a **transposition table**.
- Using a transposition table can have a dramatic effect, sometimes as much as doubling the reachable search depth in chess.
- Since it is not practical to keep *all* of the nodes in the transposition table, various strategies have been used to choose which nodes to keep and which to discard.

IMPERFECT REAL-TIME DECISIONS

- The *minimax* algorithm generates the entire game search space, whereas the alpha–beta algorithm allows us to prune large parts of it.
- Alpha–beta still has to search all the way to terminal states for at least a portion of the search space.
- Moves must be made in a reasonable amount of time—typically a few minutes at most.
- Claude Shannon: cut off the search earlier and apply a heuristic **evaluation function** to states in the search ➔ turns nonterminal nodes into terminal leaves.

IMPERFECT REAL-TIME DECISIONS

- So alter minimax or alpha–beta in two ways:
  - replace the utility function by a heuristic evaluation function EVAL, which estimates the position's utility, and
  - replace the terminal test by a **cutoff test** that decides when to apply EVAL.

$$\text{H\_MINIMAX}(s, d) = \begin{cases} EVAL(s) & if\,(CUTOFF\_TEST(s,d) \\ max_{a \in Actions(s)} \; H\_MINIMAX(RESULT(s,a), d+1) & if\ PLAYER(s) = MAX \\ min_{a \in Actions(s)} \; H\_MINIMAX(RESULT(s,a), d+1) & if\ PLAYER(s) = MIN \end{cases}$$

Evaluation functions

- An **evaluation function** returns an *estimate* of the expected utility of the game from a given position.
- How to design good evaluation functions?
  1. It should order the *terminal* states in the same way as the true *utility function*: states that are wins must evaluate better than draws, which in turn must be better than losses.
  2. The computation must not take too long!
  3. For nonterminal states, the evaluation function should be strongly correlated with the actual chances of winning.
- Cutting off search at nonterminal states ➔ the algorithm will necessarily be uncertain about the final outcomes of those states.

Evaluation functions

- Since it is a limited amount of computation that the evaluation function is allowed to do for a given state, it is best to make a guess about the final outcome.
- Most evaluation functions work by calculating various **features** of the state.
- E.g., in chess, we would have features for the numbers of white pawns, black pawns, white queens, black queens, …
- Practically, compute separate numerical contributions from each feature and then *combine* them to find the total value.

Evaluation functions

- In chess: common to give an approximate **material value** for each piece:
  - Each pawn is worth 1, a knight or bishop is worth 3, a rook 5, and the queen 9.
  - Other features such as "*good pawn structure*" and "*king safety*": ~ 1/2 pawn, for instance.
  - These feature values are then simply added up to obtain the evaluation of the position.
- Mathematically: we use a **weighted linear function** expressed as:

$$EVAL(s) = \mathrm{w1\ f1(s)} + \mathrm{w2\ f2(s)} + \cdots + \mathrm{wn\ fn(s)}$$

$$= \sum_{i=1}^{n} \mathrm{wi\ fi(s)}$$

Evaluation functions

- However, adding up the values of features involves a strong assumption: the contribution of each feature is *independent* of the values of the other features.
  - BUT, e.g. the value of a bishop is not fixed; bishops are more powerful in the endgame, when they have a lot of space to maneuver.

➔ current programs for chess and other games use *nonlinear* combinations of features.

  - E.g., a pair of bishops might be worth slightly more than twice the value of a single bishop, and
  - A bishop is worth more in the endgame.

- In games where this kind of human experience is not available, the weights of the evaluation function can be estimated by Machine Learning techniques.

Cutting off search

- So need to modify ALPHA-BETA-SEARCH so that it will call the heuristic EVAL function when it is appropriate to cut off the search.
- In the Alpha-Beta algorithm, just replace

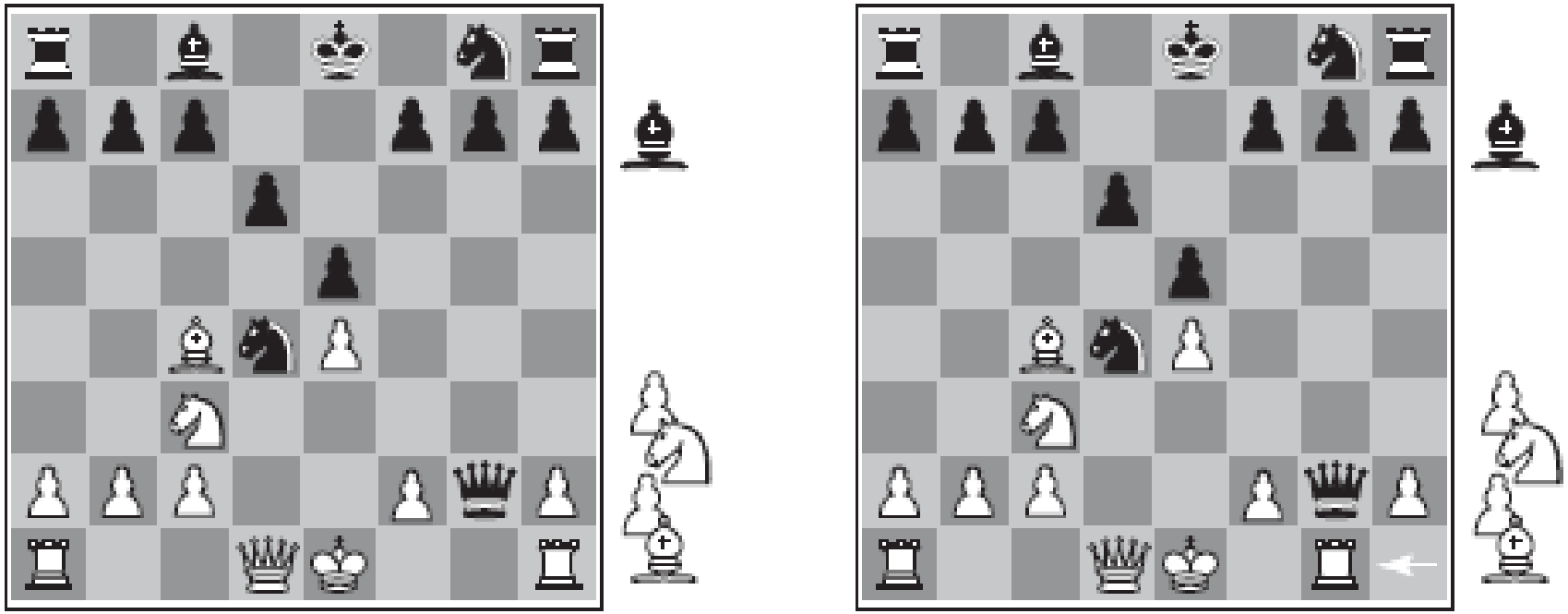
  if CUTOFF-TEST(state, depth) **then return** UTILITY(state)

  by

  if CUTOFF-TEST(state, depth) **then return** EVAL(state)
- We also need to keep track of the depth.
- To control the amount of search, set a fixed depth limit so that CUTOFF-TEST(state, depth) returns true for any depth greater than some fixed depth *d* (and for all terminal states).
- Depth *d* is chosen so a move is selected within allocated time.
- More robust approach: apply iterative deepening. When time runs out, the program returns the move selected by the deepest completed search.

Piece advantage and EVAL

Be careful: piece advantage can be misleading!



(a) White to move

(b) White to move

Figure 5.8 Two chess positions that differ only in the position of the rook at lower right. In (a), Black has an advantage of a knight and two pawns, which should be enough to win the game. In (b), White will capture the queen, giving it an advantage that should be strong enough to win.

Cutting off search

- Note that the depth limit can prevent to see interesting potential developments which can be seen only by looking ahead one more ply.
- The evaluation function should be applied only to positions that are **quiescent**— i.e., unlikely to exhibit wild swings in value in the near future.
- E.g., in chess, positions in which favourable captures can be made are not quiescent for an evaluation function that just counts material.
- Non-quiescent positions can be expanded further until quiescent positions are reached. (**quiescence search**)

Cutting off search

- The **horizon effect:** arises when the program is facing an opponent's move that causes serious damage and is ultimately unavoidable, but can be temporarily avoided by delaying tactics.
- The **horizon effect** is more difficult to eliminate
- One strategy to lighten the horizon effect is **singular extension**: a move "clearly better" than all other moves in a given position. (This move is memorized during the tree search until the depth limit.)
- Singular extension makes the tree deeper, but because there will be few singular extensions, it does not add many total nodes to the tree.

Horizon Effect

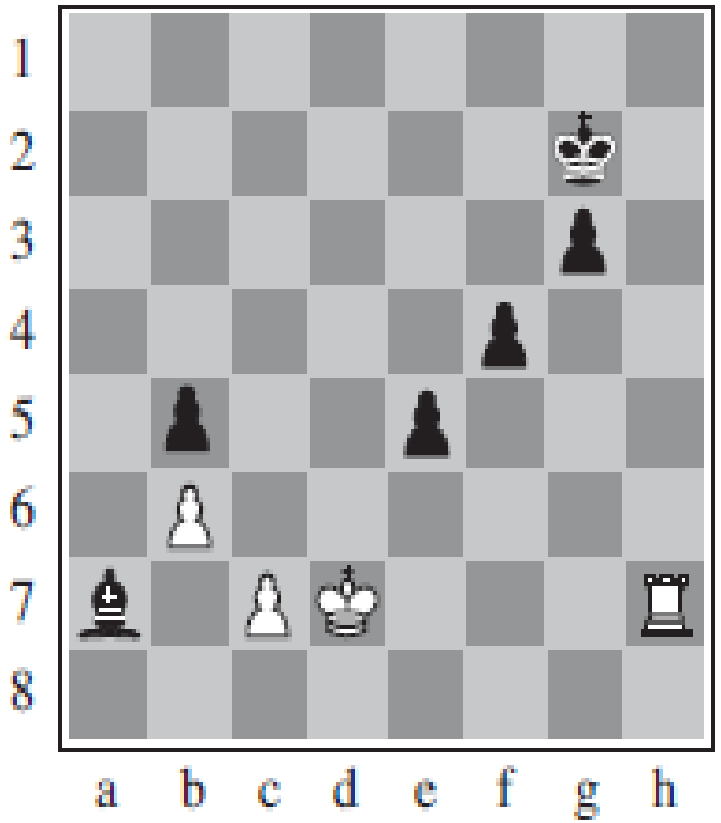


Figure 5.9 The horizon effect. With Black to move, the black bishop is surely doomed. But Black can forestall that event by checking the white king with its pawns, forcing the king to capture the pawns. This pushes the inevitable loss of the bishop over the horizon, and thus the pawn sacrifices are seen by the search algorithm as good moves rather than bad ones.

Forward pruning

- **Forward pruning**: some moves at a node are pruned immediately without further consideration.
  - Humans playing chess consider only a few moves from each position.
- One approach to forward pruning is **beam search**: on each ply, consider only a "beam" of the n best moves (according to the evaluation function) instead of all possible moves.
- Problem: risk of pruning a better/best move!

Playing games?

- To sum up: need to…
  - implement an evaluation function for chess,
  - a reasonable cutoff test with a quiescence search,
  - a large transposition table,
  - using minimax search, and
  - alpha-beta pruning.
- A lot of programming optimization to generate as many nodes per second as possible. And…
- Powerful computing facilities!

Search versus lookup

- Many game-playing programs use ***table lookup*** rather than search for the opening and ending of games which rely on the **expertise of humans**.
- Computers can also gather statistics from a database of previously played games to see which opening sequences most often lead to a win.
- Usually after ten moves we end up in a rarely seen position, and the program must switch from table lookup to search.
- Near the end of the game: fewer possible positions ➔ bigger chance to do lookup. But here it is the computer that has the expertise: **computer analysis of endgames** goes far beyond anything achieved by humans.
- A computer can completely *solve* the endgame by producing a **policy**, which is a mapping from every possible state to the best move in that state.

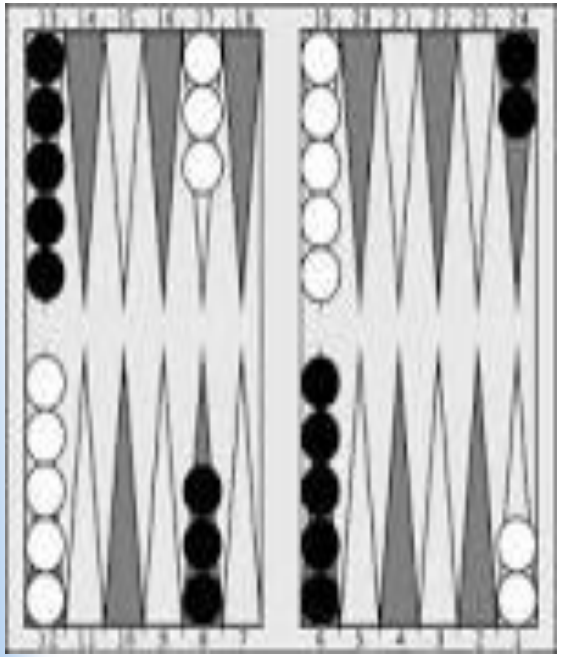
Outline

- Game Theory
- Optimal Decisions in Games
- Heuristic Alpha--Beta Tree Search
- Stochastic Games
- Partially Observable Games

Stochastic Games

- Many games mirror an unpredictability by including a random element, such as the throwing of dice.
- These are called **stochastic games**.
- Backgammon is a typical game that combines luck and skill.

Backgammon initial state



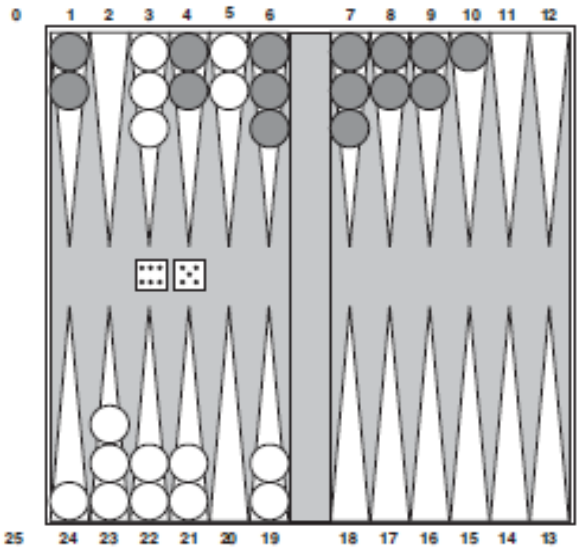


Figure 5.10 A typical backgammon position. The goal of the game is to move all one's pieces off the board. White moves clockwise toward 25, and Black moves counterclockwise toward 0. A piece can move to any position unless multiple opponent pieces are there; if there is one opponent, it is captured and must start over. In the position shown, White has rolled 6–5 and must choose among four legal moves: (5–10,5–11), (5–11,19–24), (5–10,10–16), and (5–11,11–16), where the notation (5–11,11–16) means move one piece from position 5 to 11, and then move a piece from 11 to 16.

Stochastic Games

- White cannot construct a standard game tree like that in chess and tic-tac-toe.
- A game tree in backgammon must include **chance nodes** in addition to MAX and MIN nodes.
- The branches leading from each chance node denote the possible dice rolls; each branch below a chance node is labeled with the roll and its proba.
- Backgammon: 36 ways to roll two dice: 6 doubles (proba. 1/36 each); 21 distinct ways of outcomes of throwing two dice (rolls). ➔
- p(double) = 1/36; p(any other two dice): 1/18

  e.g. p(1-1) = 1/36; p(5-2) = 1/18

Stochastic Games: Backgammon

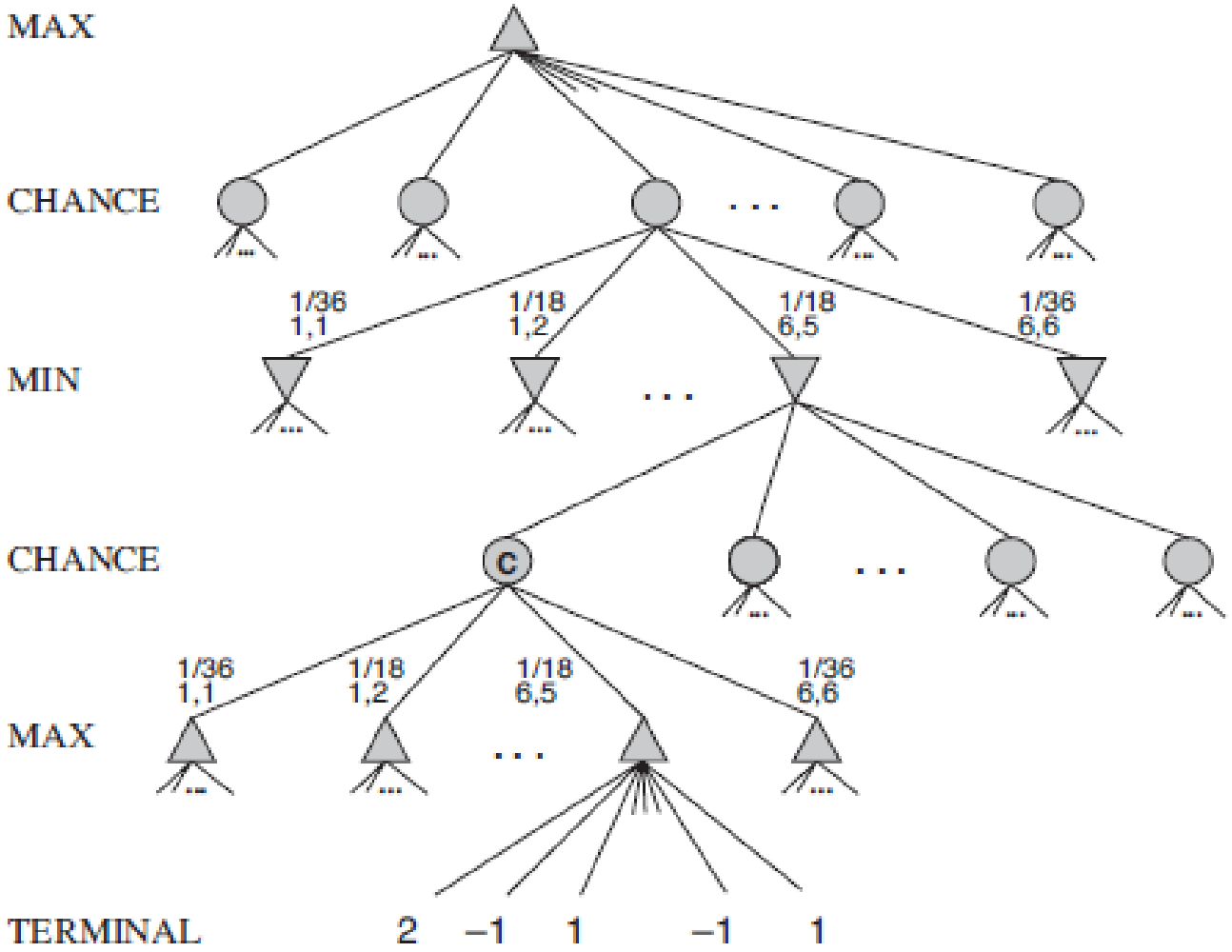


Figure 5.11 Schematic game tree for a backgammon position.

Stochastic Games

- Positions do not have definite minimax values.
- We can only calculate the **expected value** of a position: the average over all possible outcomes of the chance nodes.
- Need to generalize the **minimax value** for deterministic games to an **expectiminimax value** for games with chance nodes:

$$\text{EXPECTIMINIMAX}(s) = \begin{cases} UTILITY(s) & if\,(TERMINAL\_TEST(s) \\ max_a\,\text{EXPECTIMINIMAX}\big(RESULT(s,a)\big) & if\ PLAYER(s) = MAX \\ min_a\,\text{EXPECTIMINIMAX}\big(RESULT(s,a)\big) & if\ PLAYER(s) = MIN \\ \sum_r P(r)\ \text{EXPECTIMINIMAX}\big(RESULT(s,r)\big) & if\ PLAYER(s) = CHANCE \end{cases}$$

r represents a possible dice roll (or other chance event)

Evaluation functions for games of chance

- As with minimax, the approximation to make with *expectiminimax* is to cut the search off at some point and apply an evaluation function to each leaf.
- Evaluation functions for games such as backgammon are not exactly like evaluation functions for chess.
- The presence of chance makes things more sensitive. See next figure.

Evaluation functions for games of chance

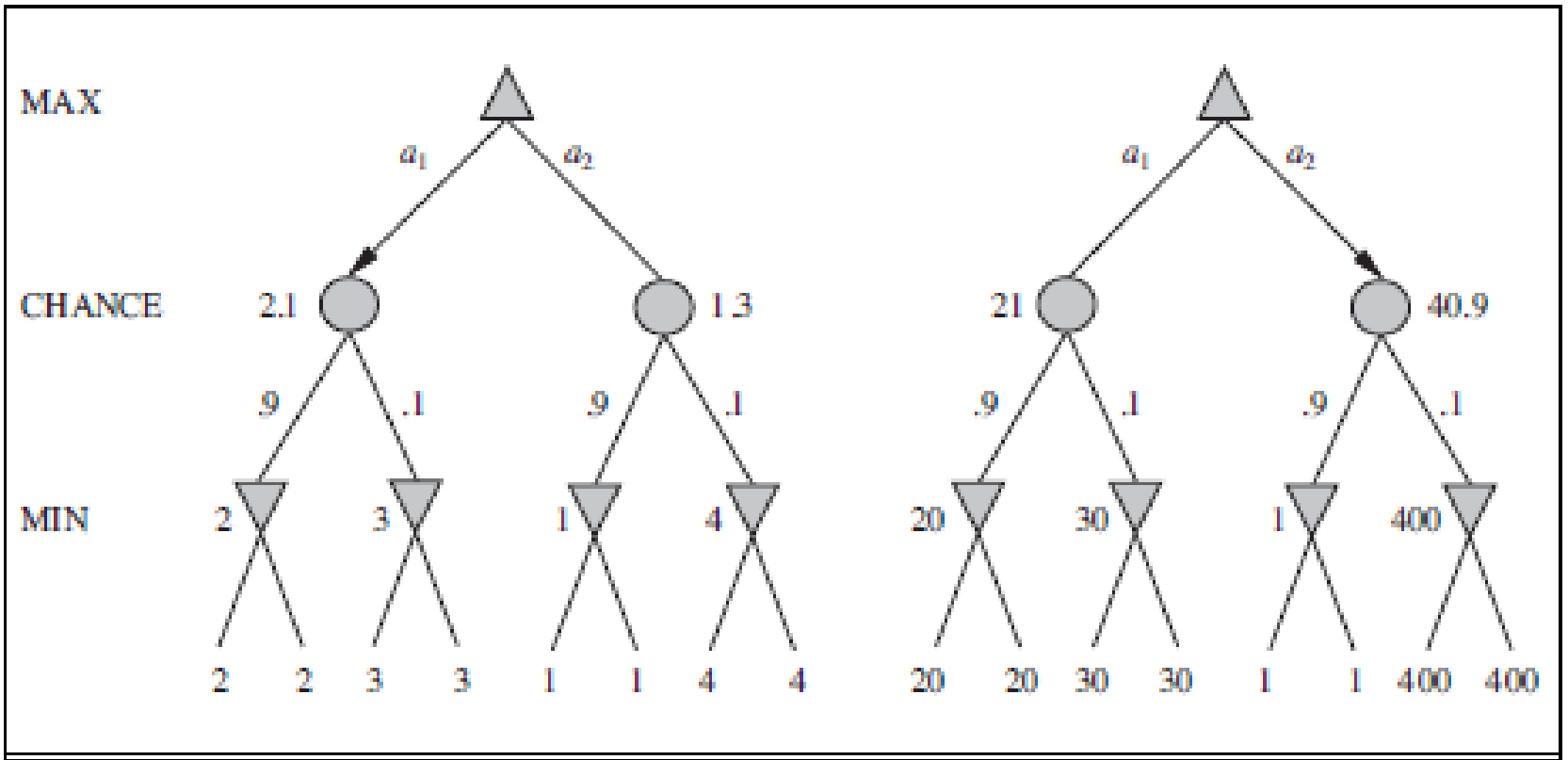


Figure 5.12 An order-preserving transformation on leaf values changes the best move.

The program behaves totally differently if we make a change in the scale of the evaluation

Evaluation functions for games of chance

- To avoid this sensitivity: the evaluation function must be a positive linear transformation of the probability of the expected utility of the position.
- This is an important and general property of situations in which uncertainty is involved.
- Because ***expectiminimax*** is also considering all the possible dice-roll sequences, it will take $\mathbf{O(b^m n^m)}$, where *b* is the branching factor, *m* the maximum depth of the game tree, and *n* is the number of distinct rolls.
- In backgammon: n is 21; b is usually ~20 (but in some situations can be as high as 4000 for dice rolls that are doubles). Usually 3 plies is probably all that would be possible.

Alpha–beta pruning?

- Alpha–beta pruning could be applied to game trees with chance nodes.
- The analysis for MIN and MAX nodes is unchanged, but we can also prune chance nodes, using a bit of ingenuity.
- It is possible to find an upper bound on the value of a chance node before we have looked at all its children.
- If we put bounds on the possible values of the utility function, then we can arrive at bounds for the average without looking at every number.
- E.g. suppose that all utility values are between −2 and +2.
- ➔ the value of leaf nodes is bounded ➔ we *can* place an upper bound on the value of a chance node without looking at all its children.
- **Exercise**: Think about it!

Outline

- Game Theory
- Optimal Decisions in Games
- Heuristic Alpha--Beta Tree Search
- Stochastic Games
- Partially Observable Games

Partially Observable Games: Card games

- Card games provide many examples of *stochastic* partial observability.
- Missing information is generated randomly.
- In many games, cards are dealt randomly at the beginning of the game. Each player receives a hand that is not visible to the other players.
- **Idea**:
  - consider all possible deals of the invisible cards;
  - solve each one as if it were a fully observable game; and then
  - choose the move that has the best outcome averaged over all the deals.

Partially Observable Games: Card games

- Suppose that each deal s occurs with probability P(s); then the move we want is:

$$\underset{a}{\operatorname{argmax}} \sum_s P(s)\, MINIMAX(RESULT(s,a))$$

- We run exact MINIMAX if computationally feasible; otherwise, run H-MINIMAX.
- **Problem**: in most card games, the number of possible deals is rather large.
- Instead of adding up *all* the deals, we take a *random sample* of N deals, where the probability of deal s appearing in the sample is proportional to P(s):

$$\underset{a}{\operatorname{argmax}} \frac{1}{N} \sum_{i=1}^{N} MINIMAX(RESULT(s,a))$$

Section 5.7: State-Of-The-Art Game Programs

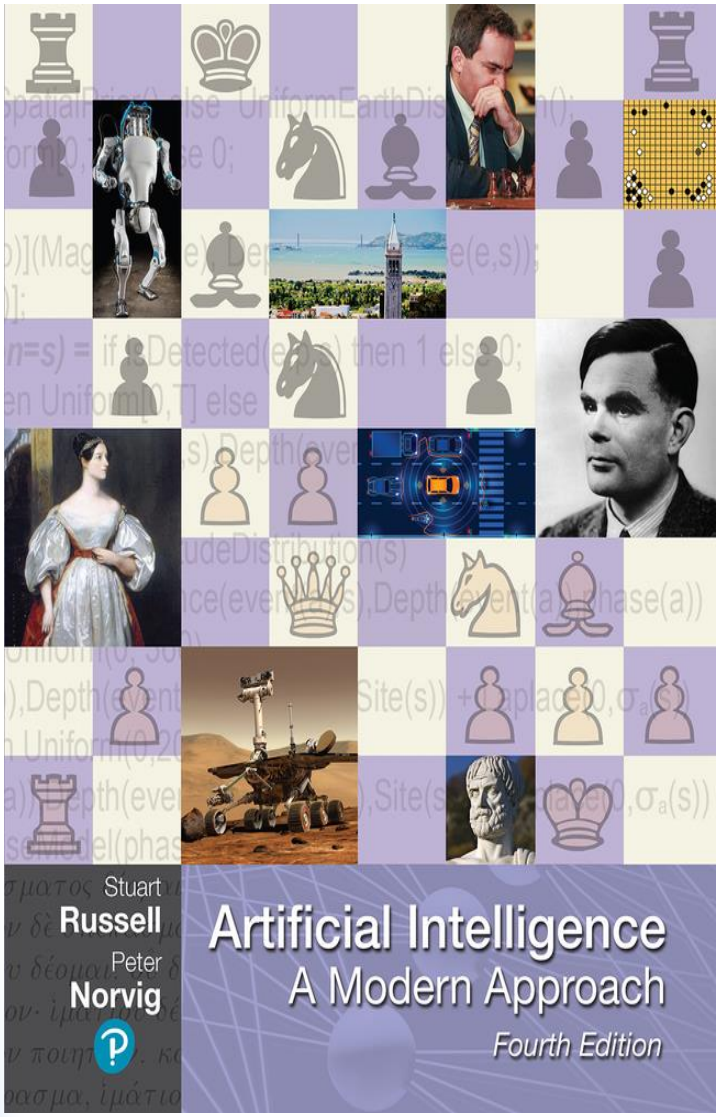
Section 5.8: Alternative Approaches

Section: Bibliographical and Historical Notes

Left as individual reading. ☹

;-)

Slides based on the textbook



- Russel, S. and Norvig, P. (2020) Artificial Intelligence, A Modern Approach (4th Edition), Pearson Education Limited.