Session 7

LOCAL SEARCH ALGORITHMS

Session outcomes

- **Students will learn about Local Search Algorithms**
- **Students will learn about state space search based problems**

- This systematicity is achieved by keeping one or more paths in memory and by recording which alternatives have been explored at each paint along the path.

- When a goal is found, the path to that goal also constitutes a solution to the problem.

- In many problems, how-ever, the path to the goal is irrelevant. For example, in the 8-queens problem . what matters is the final configuration of queens, not the order in which they are added.

- **The same general property holds for many important applications such as integrated-circuit de-sign, factory-flour layout, job-shop scheduling, automatic programming, telecommunications network optimization, vehicle routing. and portfolio management.**

LOCAL SEARCH ALGORITHMS

- Local search algorithms operate using a single current node (**rather than multiple paths**) and generally move only to neighbors of that node.
- Typically, the paths followed by the search are not retained.
- Although local search algorithms are not systematic, they have two key advantages:

(1) they use very little memory—usually a constant amount; and

(2) they can often find reasonable solutions in large or infinite(continuous) state spaces for which systematic algorithms are unsuitable.

- In addition to finding goals, local search algorithms are useful for solving pure optimization problems, in which the aim is to find the best state according to an objective function.
- A complete local search algorithm always finds a goal if one exists;
- An optimal algorithm always finds a global minimum/maximum.

HILL CLIMIBING

- Hill climbing is a simple local optimization method that “climbs” up the hill until a local optimum is found (assuming a maximization goal).

- The method works by iteratively searching for new solutions within the neighborhood of current solution, adopting new solutions if they are better.

- There are several hill climbing variants

- ✓ **Steepest Ascent Hill Climbing:** which searches for upto N solutions in the neighborhood of S and then adops the best one. It is time consuming but gives an optimum result.
- ✓ **Stochastic Hill Climbing:** which replaces the deterministic select function, selecting new solution with a probability of P.

Hill Climbing

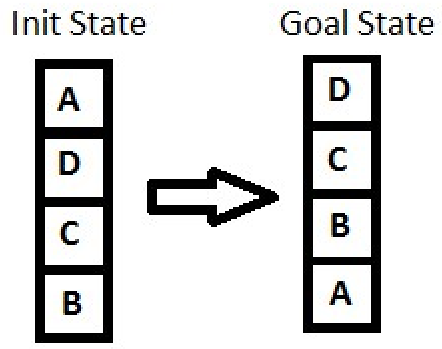
Algorithm:

1. Evaluate the initial state. If it is also goal state, then return it and quit. Otherwise continue with the initial state as the current state.
2. Loop until a solution is found or until there are no new operators left to be applied in the current state:
   a. Select an operator that has not yet been applied to the current state and apply it to produce a new state
   b. Evaluate the new state
      i. If it is the goal state, then return it and quit.
      ii. If it is not a goal state but it is better than the current state, then make it the current state.
      iii. If it is not better than the current state, then continue in the loop.

BLOCKS WORLD PROBLEM

Hill Climbing Algorithm can be categorized as an informed search. So we can implement any node-based search or problems like the n-queens problem using it. To understand the concept easily, we will take up a very simple example.

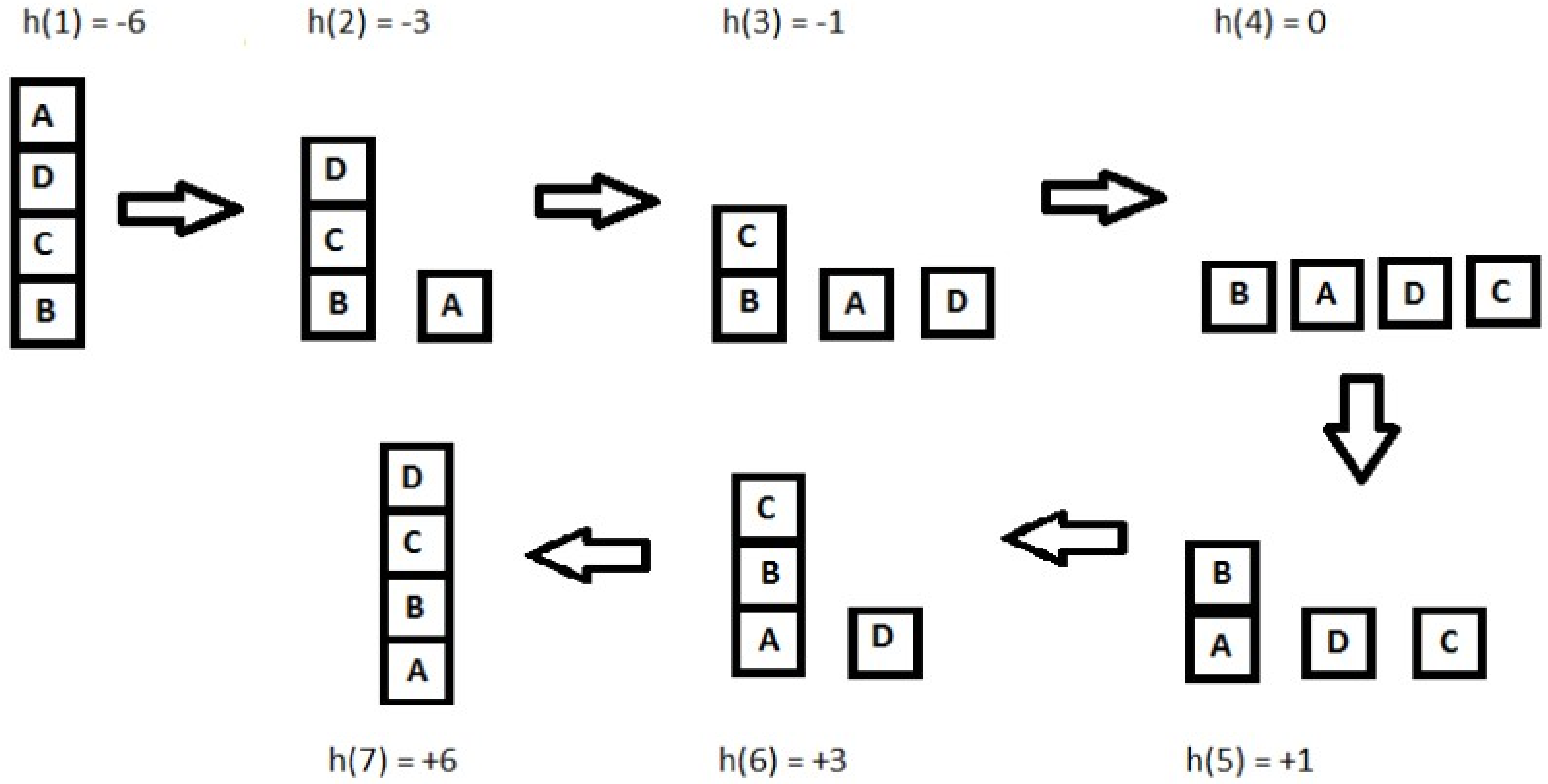
Let's look at the image below:



Key point while solving any hill-climbing problem is to choose an appropriate heuristic function.

Let's define such function *h:*

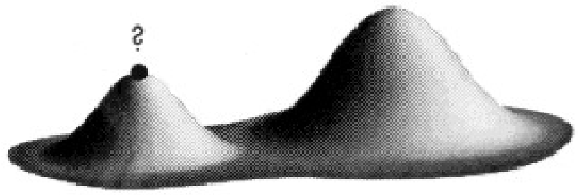
h(x) **= +1 for all the blocks in the support structure if the block is correctly positioned otherwise -1 for all the blocks in the support structure.**


h(1) = -6
A
D
C
B
h(2) = -3
D
C
B
A
h(3) = -1
C
B
A
D
h(4) = 0
B
A
D
C
B
A
D
C
h(5) = +1
C
B
A
D
h(6) = +3
D
C
B
A
h(7) = +6

Limitations of Hill-climbing

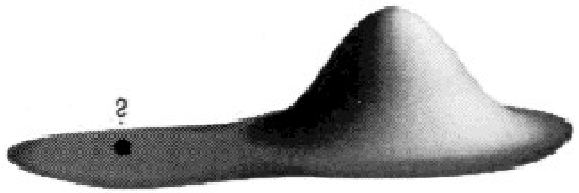
1. **Local Maxima**: a local maximum as opposed to global maximum.

   Way Out: Backtrack to some earlier node and try going in a different direction



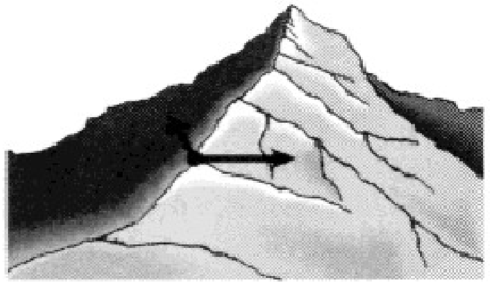
2. **Plateaus**: An area of the search space where evaluation function is flat, thus requiring random walk.

   Way out: Make a big jump to try to get in a new section



3. **Ridge**: Where there are steep slopes and the search direction is not towards the top but towards the side.

   Way out: Moving in several directions at once.



Steepest-Ascent Hill Climbing

- This is a variation of simple hill climbing which considers all the moves from the current state and selects the best one as the next state. Also known as <u>Gradient search</u>

Algorithm: Steepest-Ascent Hill Climbing

1.Evaluate the initial state. If it is also a goal state, then return it and quit. Otherwise, continue with the initial state as the current state.

2. Loop until a solution is found or until a complete iteration produces no change to current state:
   a. Let SUCC be a state such that any possible successor of the current state will be better than SUCC
   b. For each operator that applies to the current state do:
      i. Apply the operator and generate a new state
      ii. Evaluate the new state. If is is a goal state, then return it and quit. If not, compare it to SUCC. If it is better, then set SUCC to this state. If it is not better, leave SUCC alone.
   c. If the SUCC is better than the current state, then set current state to SUCC,

Simulated Annealing

- Simulated Annealing (SA) is an effective and general form of optimization.

  It is useful in finding global optima in the presence of large numbers of local optima. “Annealing” refers to an analogy with thermodynamics, specifically with the way that metals cool and anneal.

  Simulated annealing uses the objective function of an optimization problem instead of the energy of a material. Implementation of SA is surprisingly simple. The algorithm is basically hill-climbing except instead of picking the best move, it picks a random move. If the selected move improves the solution, then it is always accepted. Otherwise, the algorithm makes the move anyway *with some probability* less than 1. The probability decreases exponentially with the “badness” of the move, which is the amount deltaE by which the solution is worsened (i.e., energy is increased.)

Local Beam Search

In this algorithm, it holds k number of states at any given time. At the start, these states are generated randomly. The successors of these k states are computed with the help of objective function. If any of these successors is the maximum value of the objective function, then the algorithm stops.

Otherwise the (initial k states and k number of successors of the states = 2k) states are placed in a pool. The pool is then sorted numerically. The highest k states are selected as new initial states. This process continues until a maximum value is reached.

```
function BeamSearch( problem, k), returns a solution stat
start with k randomly generated states
loop
    generate all successors of all k states
    if any of the states = solution, then return the state
    else select the k best successors
end
```

Session 8

Adversarial Search

Session Outcomes

- Students will learn alpha beta pruning
- Students will learn the importance of pruning

Adversarial Search

- Examine the problems that arise when we try to plan ahead in a world where other agents are planning against us.
- A good example is in board games.
- Adversarial games, while much studied in AI, are a small part of game theory in economics.

Typical AI assumptions

- Two agents whose actions alternate
- Utility values for each agent are the opposite of the other
  - creates the adversarial situation
- Fully observable environments
- In game theory terms: Zero-sum games of perfect information.
- We'll relax these assumptions later.

Search versus Games

- Search – no adversary
  - Solution is (heuristic) method for finding goal
  - Heuristic techniques can find *optimal* solution
  - Evaluation function: estimate of cost from start to goal through given node
  - Examples: path planning, scheduling activities

- Games – adversary
  - Solution is **strategy** (strategy specifies move for every possible opponent reply).
  - **Optimality depends on opponent.** Why?
  - Time limits force an *approximate* solution
  - Evaluation function: evaluate “goodness” of game position
  - Examples: chess, checkers, Othello, backgammon

Types of Games

	deterministic	Chance moves
Perfect information	Chess, checkers, go, othello	Backgammon, monopoly
Imperfect information (Initial Chance Moves)	Bridge, Skat	Poker, scrabble, blackjack

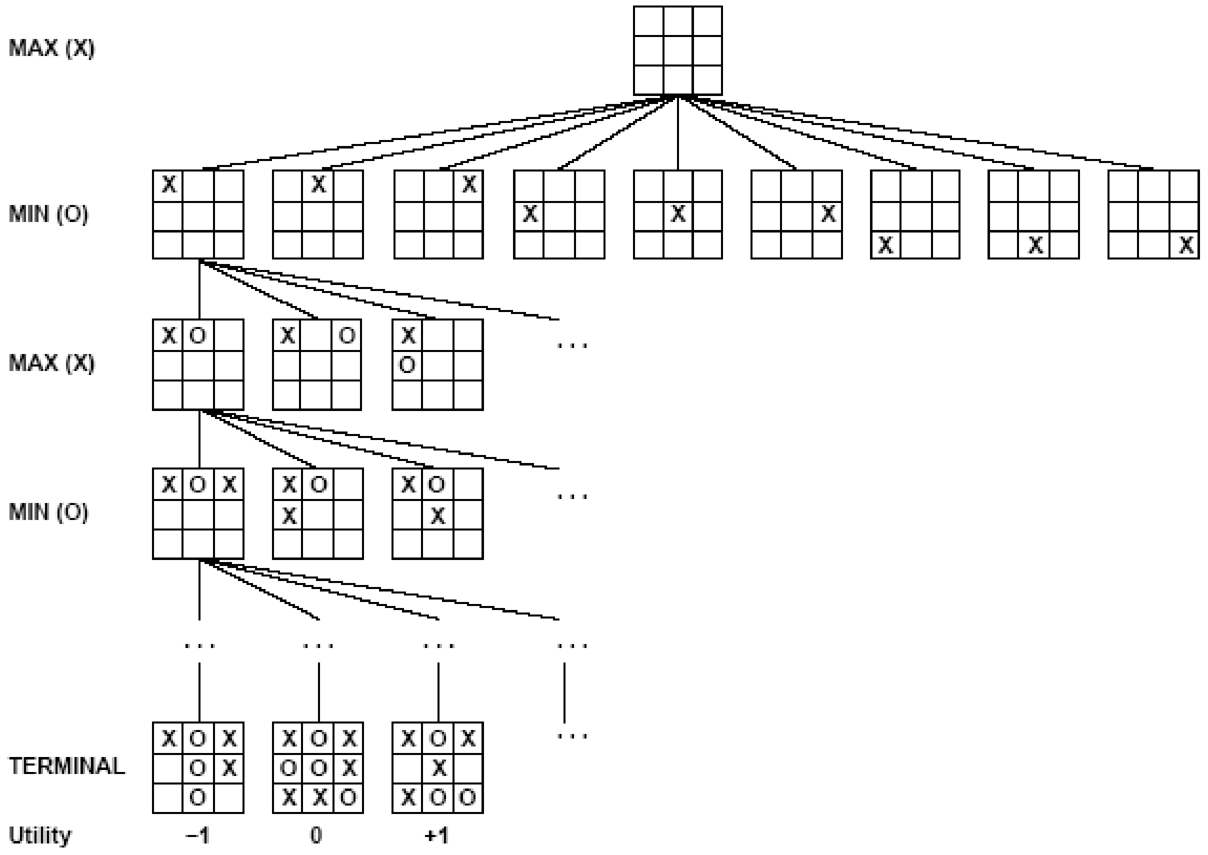
Game Setup

- Two players: MAX and MIN
- MAX moves first and they take turns until the game is over
  - Winner gets award, loser gets penalty.
- Games as search:
  - Initial state: e.g. board configuration of chess
  - Successor function: list of (move,state) pairs specifying legal moves.
  - Terminal test: Is the game finished?
  - Utility function: Gives numerical value of terminal states. E.g. win (+1), lose (-1) and draw (0) in tic-tac-toe or chess

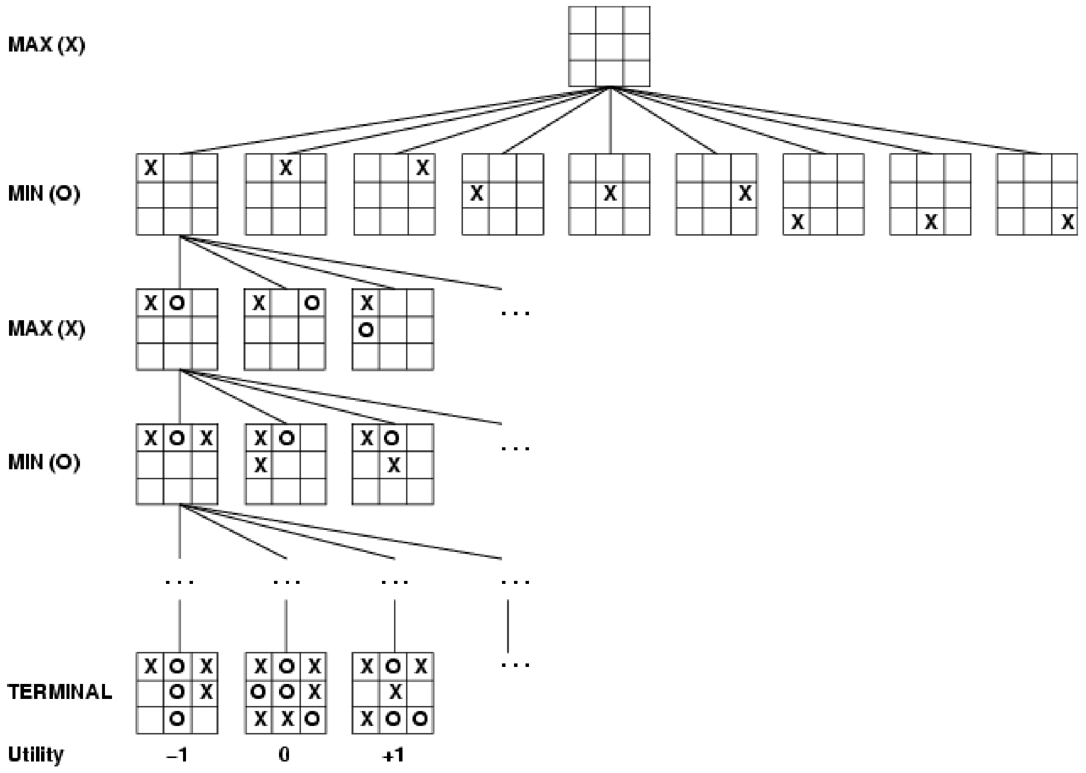
Size of search trees

- b = branching factor
- d = number of moves by both players
- Search tree is O(b^d)
- Chess
  - b ~ 35
  - D ~100
    - search tree is ~ 10^{154} (!!)
    - completely impractical to search this
- Game-playing emphasizes being able to make optimal decisions in a finite amount of time
  - Somewhat realistic as a model of a real-world agent
  - Even if games themselves are artificial

Partial Game Tree for Tic-Tac-Toe



Game tree (2-player, deterministic, turns)



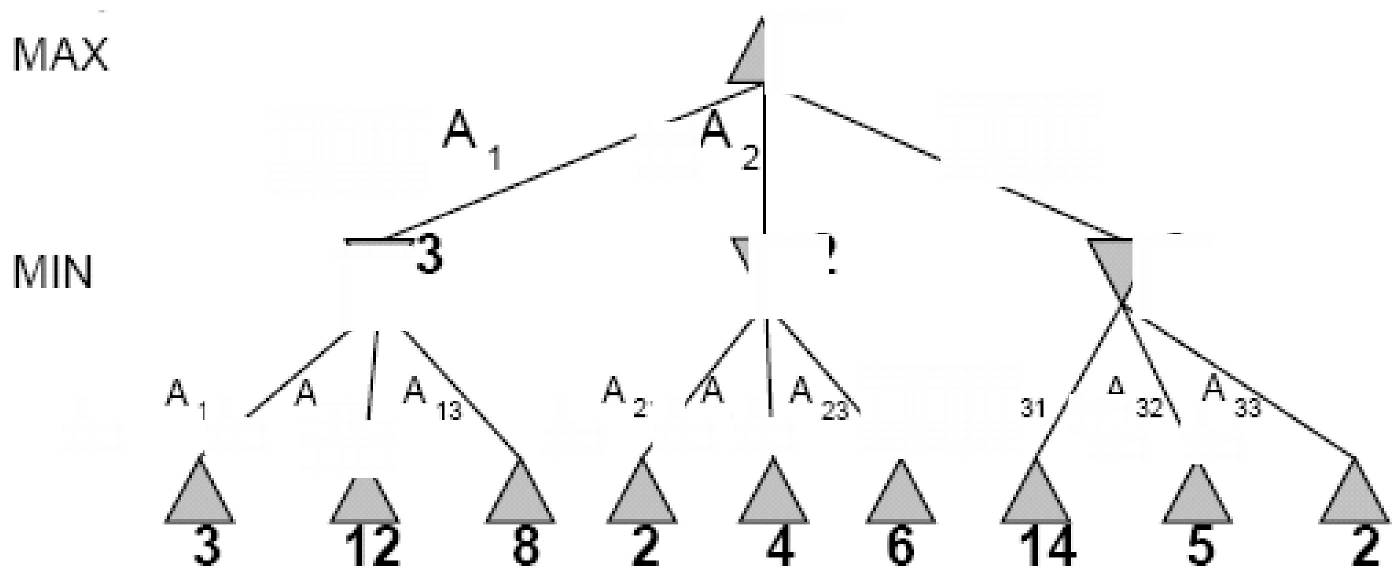
How do we search this tree to find the optimal move?

Minimax strategy:

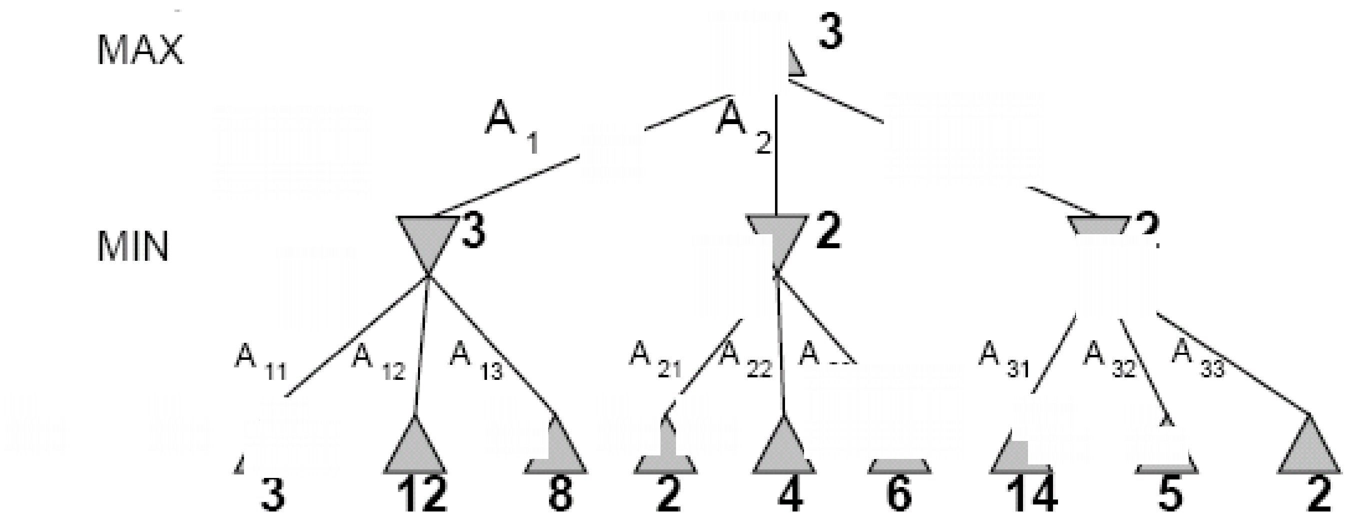
- Find the optimal *strategy* for MAX assuming an infallible MIN opponent
  - Need to compute this all the down the tree
  - Game Tree Search Demo

- Assumption: Both players play optimally!
- Given a game tree, the optimal strategy can be determined by using the **minimax value of each node.**
- Zermelo 1912.

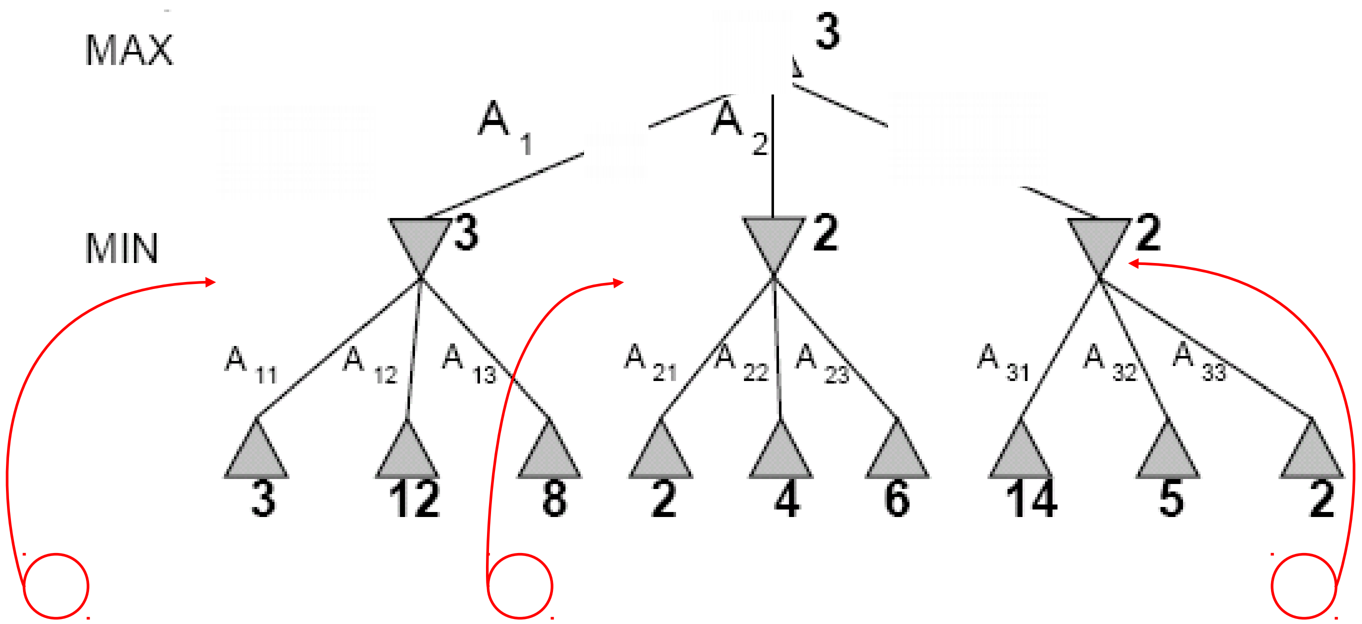
Two-Ply Game Tree



Two-Ply Game Tree

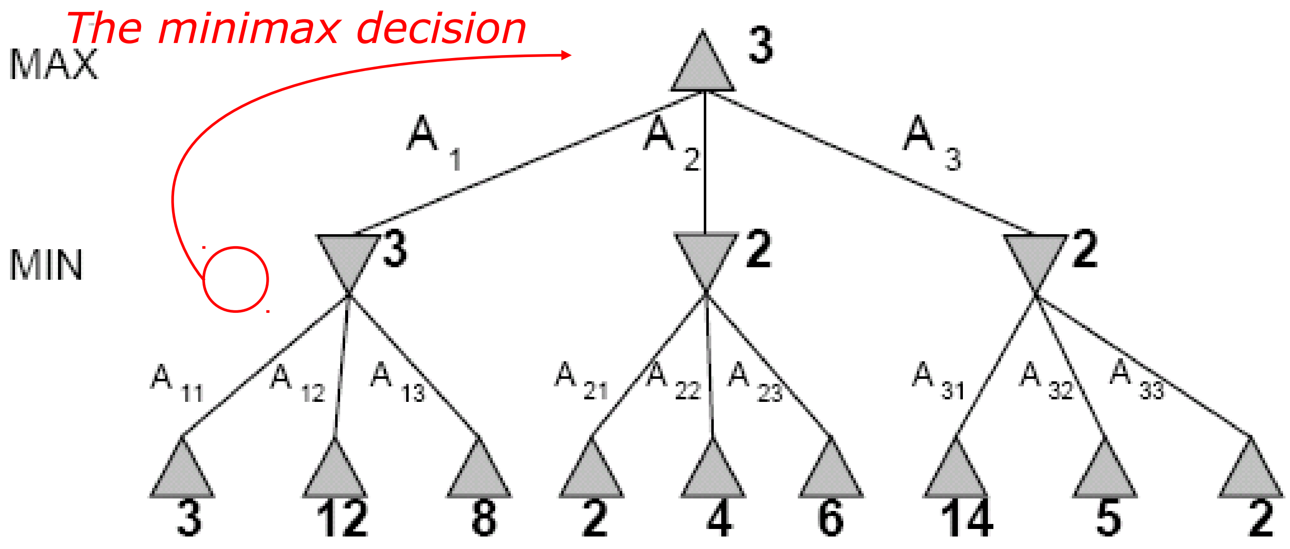


Two-Ply Game Tree



Two-Ply Game Tree

Minimax maximizes the utility for the worst-case outcome for max



Pseudocode for Minimax Algorithm

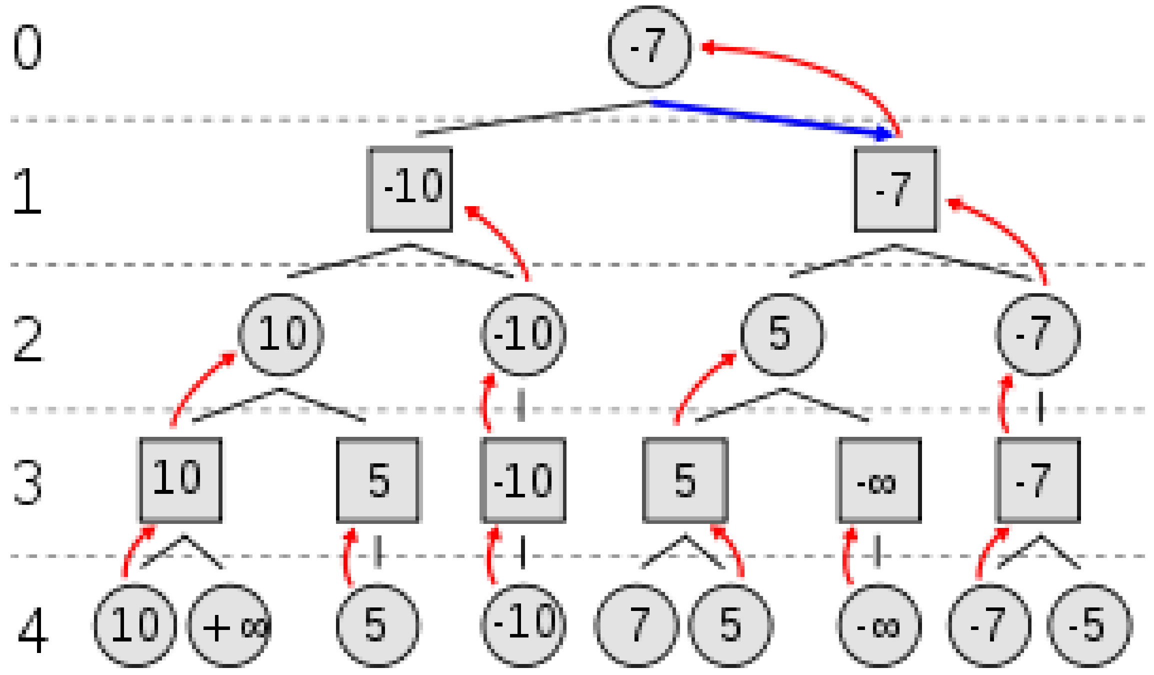
```
function MINIMAX-DECISION(state) returns an action
  inputs: state, current state in game
  v ← MAX-VALUE(state)
  return the action in SUCCESSORS(state) with value v
```

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

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

Example of Algorithm Execution

MAX to move



Minimax Algorithm

- Complete depth-first exploration of the game tree
- Assumptions:
  - Max depth = d, b legal moves at each point
  - E.g., Chess: d ~ 100, b ~35

Criterion	**Minimax**
Time	$O(b^d)$
Space	$O(bd)$

End of Session

Alpha Beta Pruning

Session Outcomes

- Students will learn applicability of AI in game playing
- Students will learn Optimal decisions in games

Introduction

- *Alpha-beta pruning* is a way of finding the optimal minimax solution while avoiding searching subtrees of moves which won't be selected. In the search tree for a two-player game, there are two kinds of nodes, nodes representing *your* moves and nodes representing *your opponent's* moves.
- Alpha-beta pruning gets its name from two parameters.
  - They describe bounds on the values that appear anywhere along the path under consideration:
    - α = the value of the best (i.e., highest value) choice found so far along the path for MAX
    - β = the value of the best (i.e., lowest value) choice found so far along the path for MIN

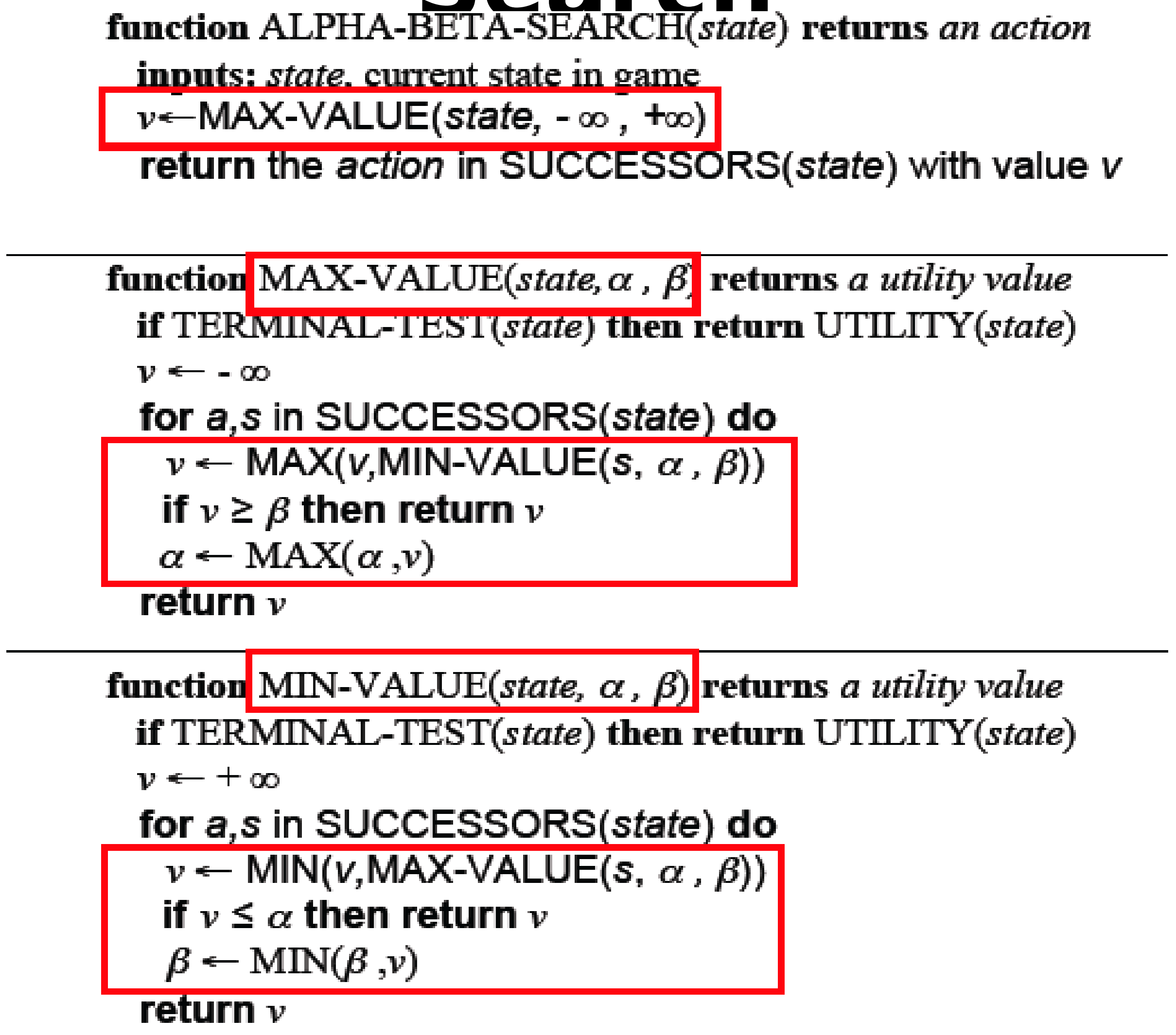
Alpha Beta Pruning

- Alpha-beta pruning gets its name from two bounds that are passed along during the calculation, which restrict the set of possible solutions based on the portion of the search tree that has already been seen. Specifically,
- Beta is the *minimum upper bound* of possible solutions
- Alpha is the *maximum lower bound* of possible solutions
- Thus, when any new node is being considered as a possible path to the solution, it can only work if:

$$\boxed{\alpha \leq N \leq \beta}$$

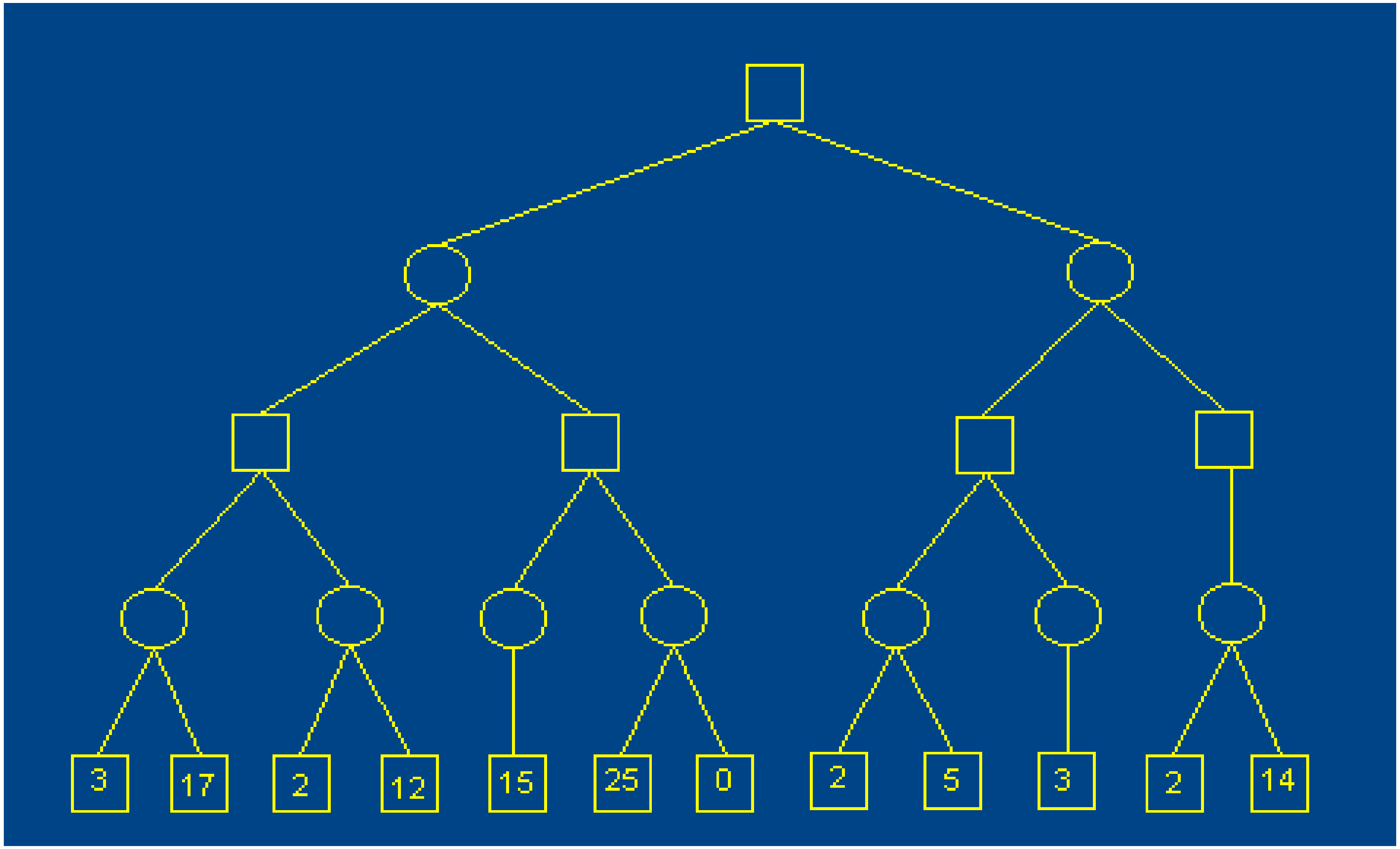
where N is the current estimate of the value of the node

Algorithm: Alpha Beta Search



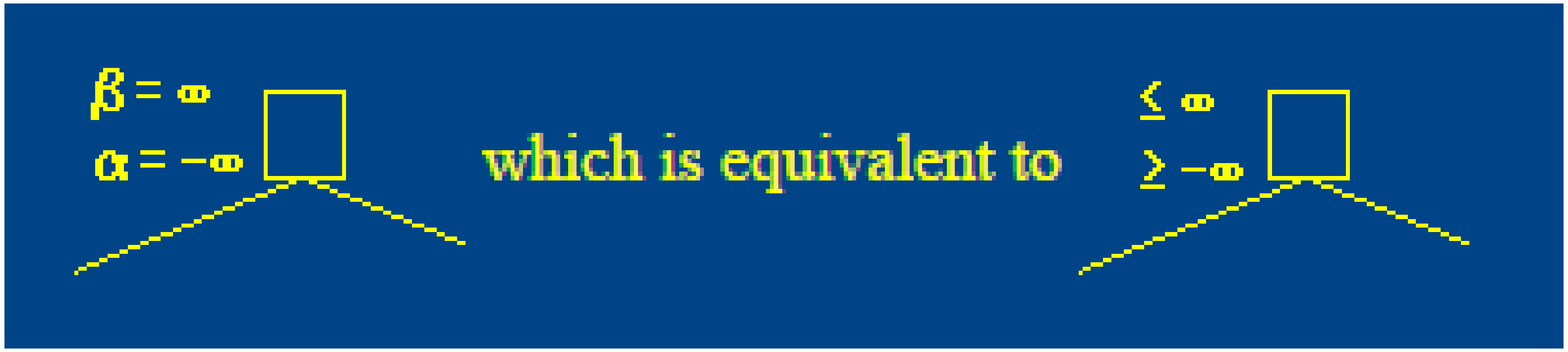
```
function ALPHA-BETA-SEARCH(state) returns an action
  inputs: state, current state in game
  $v \leftarrow$ MAX-VALUE(state, $-\infty$, $+\infty$)
  return the action in SUCCESSORS(state) with value $v$

function MAX-VALUE(state, $\alpha$, $\beta$) returns a utility value
  if TERMINAL-TEST(state) then return UTILITY(state)
  $v \leftarrow -\infty$
  for a,s in SUCCESSORS(state) do
    $v \leftarrow$ MAX($v$, MIN-VALUE(s, $\alpha$, $\beta$))
    if $v \geq \beta$ then return $v$
    $\alpha \leftarrow$ MAX($\alpha$, $v$)
  return $v$

function MIN-VALUE(state, $\alpha$, $\beta$) returns a utility value
  if TERMINAL-TEST(state) then return UTILITY(state)
  $v \leftarrow +\infty$
  for a,s in SUCCESSORS(state) do
    $v \leftarrow$ MIN($v$, MAX-VALUE(s, $\alpha$, $\beta$))
    if $v \leq \alpha$ then return $v$
    $\beta \leftarrow$ MIN($\beta$, $v$)
  return $v$
```

Example



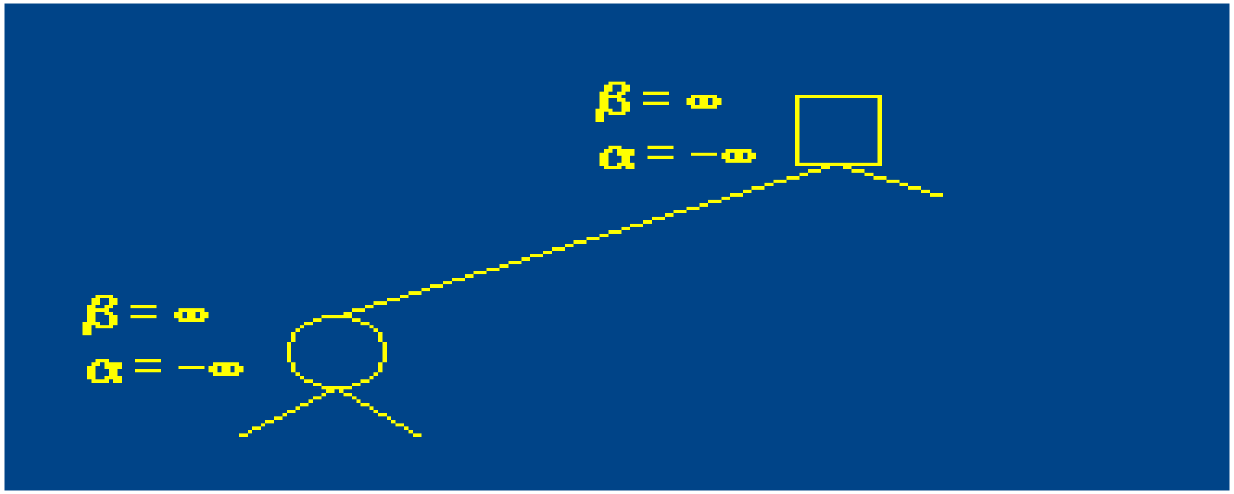
Initial Assumption for Alpha and Beta

- At the start of the problem, you see only the current state (i.e. the current position of pieces on the game board). As for upper and lower bounds, all you know is that it's a number less than infinity and greater than negative infinity. Thus, here's what the initial situation looks like:



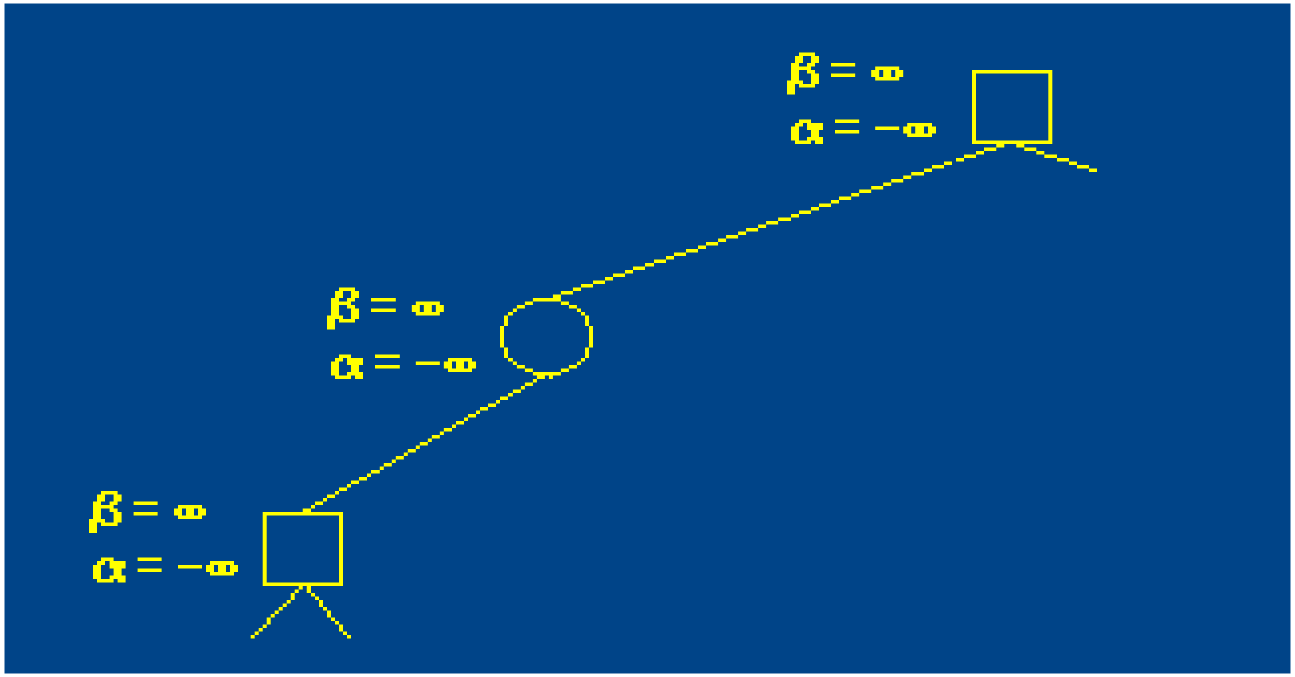
Example

- Since the bounds still contain a valid range, we start the problem by generating the first child state, and passing along the current set of bounds. At this point our search looks like this:



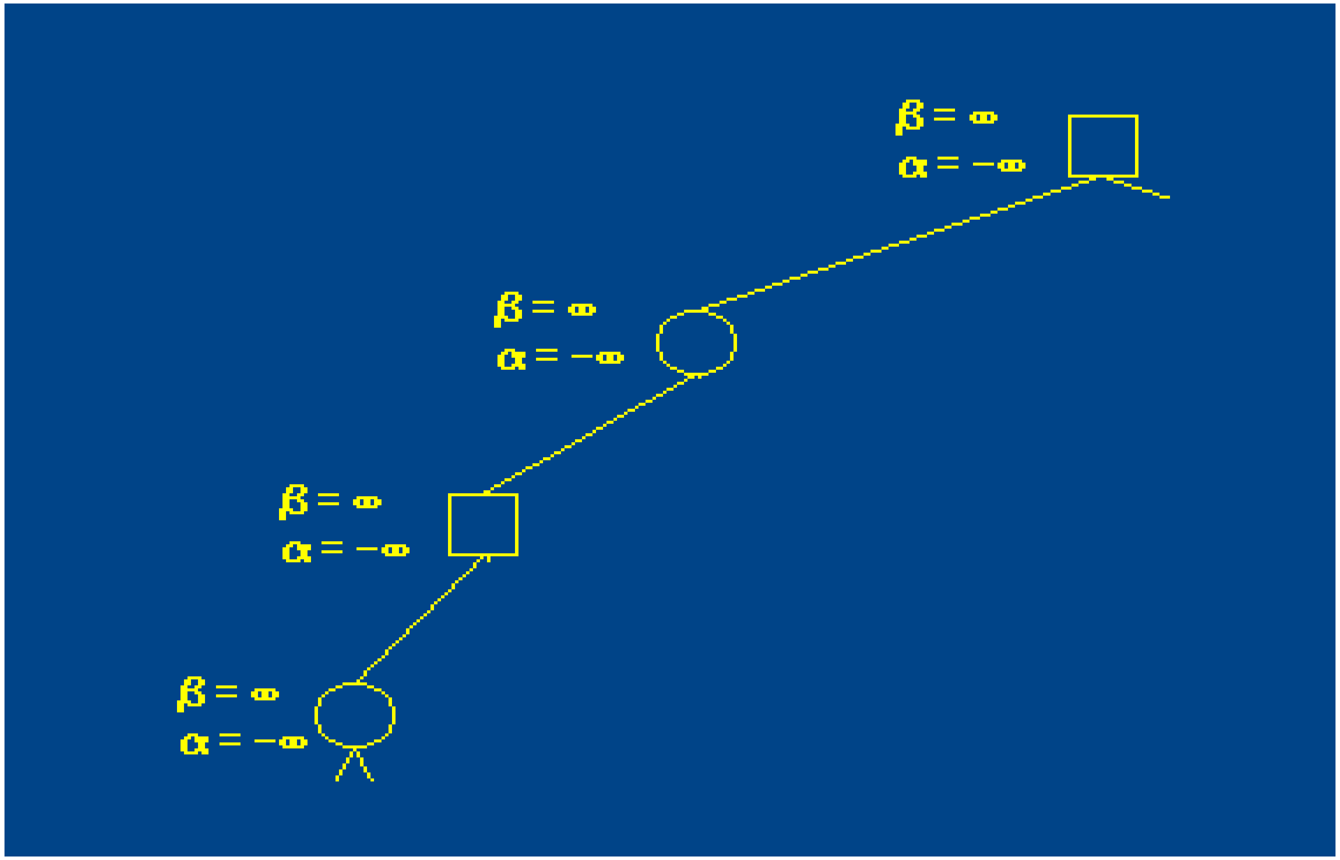
Example

- We're still not down to depth 4, so once again we generate the first child node and pass along our current alpha and beta values:



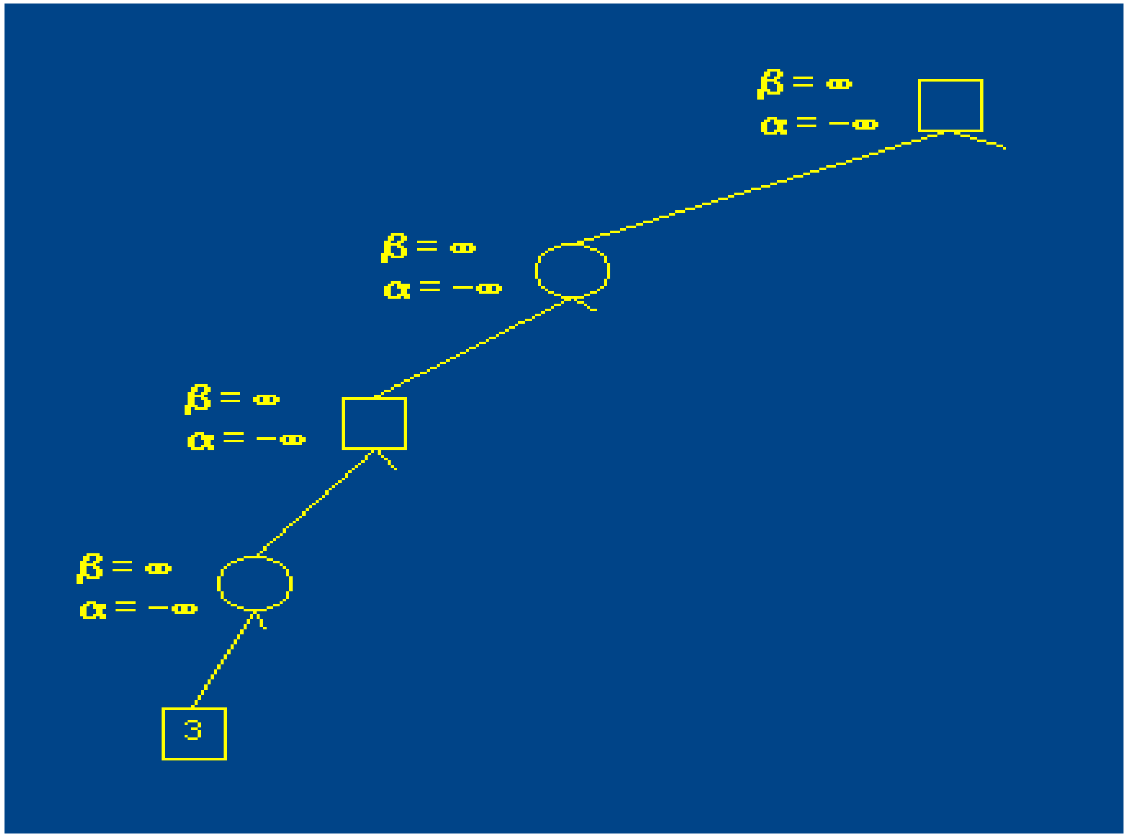
Example

- And one more time



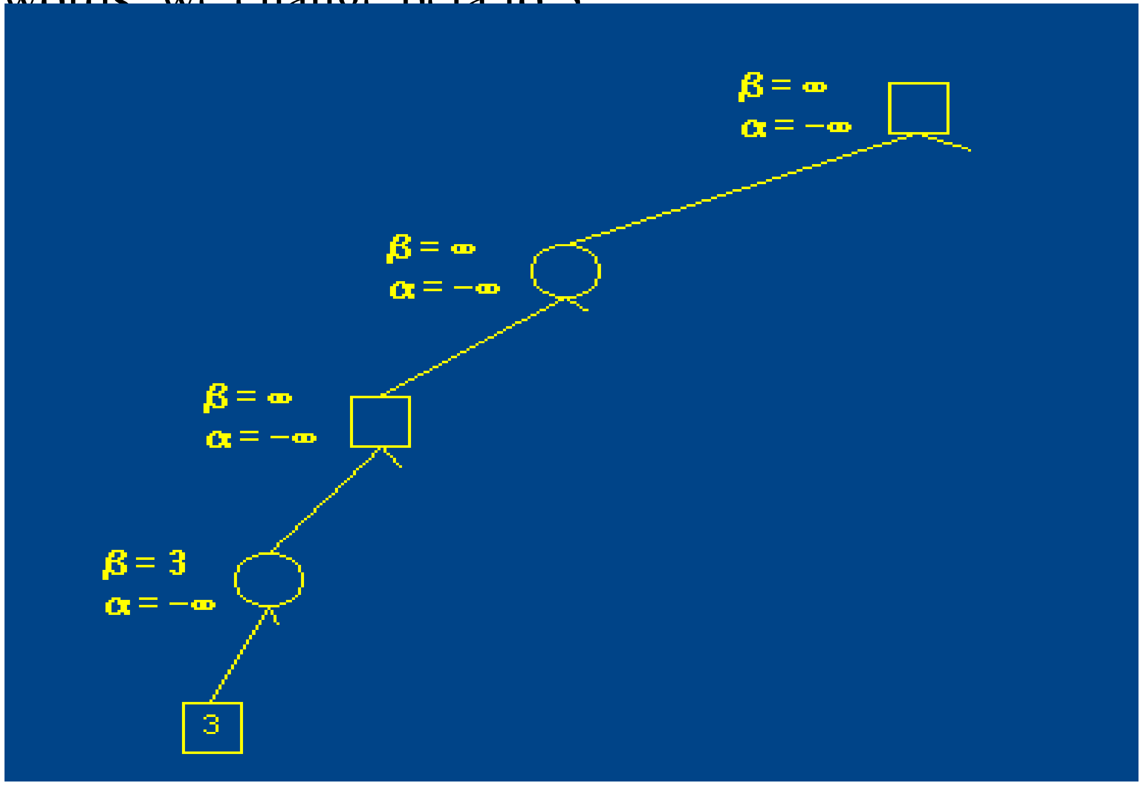
Example

- When we get to the first node at depth 4, we run our evaluation function on the state, and get the value 3. Thus we have this:



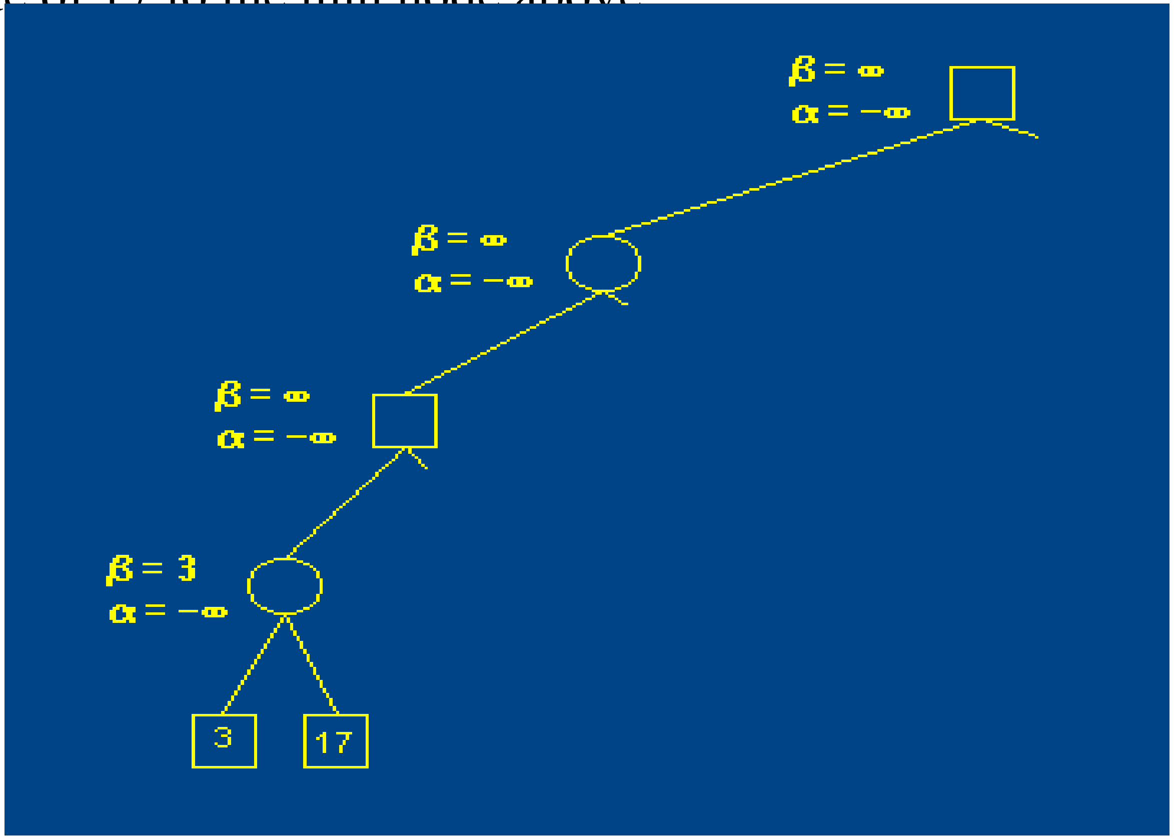
Example

- We pass this node back to the min node above. Since this is a min node, we now know that the minimax value of this node must be less than or equal to
  3. In other words, we change beta to 3.



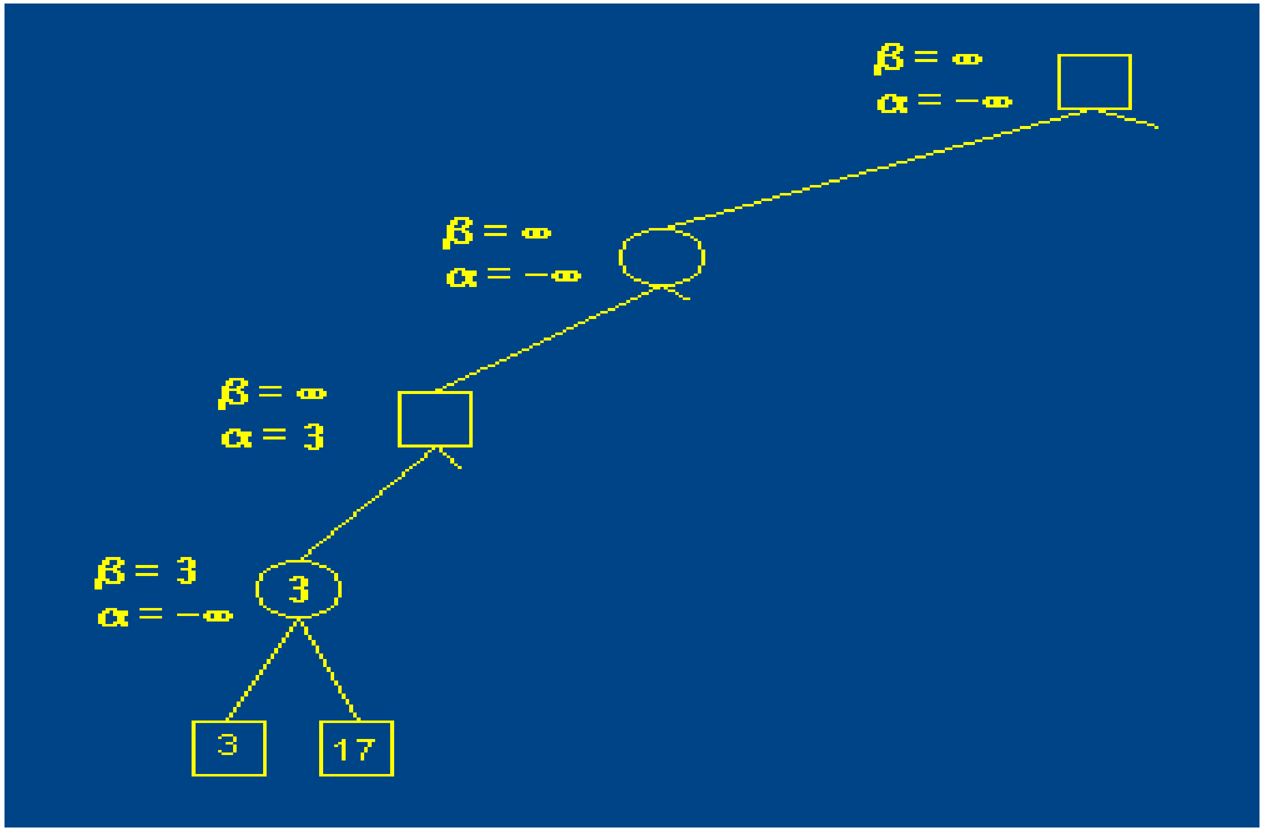
Example

- Next we generate the next child at depth 4, run our evaluation function, and
  return a value of 17 to the min node above:



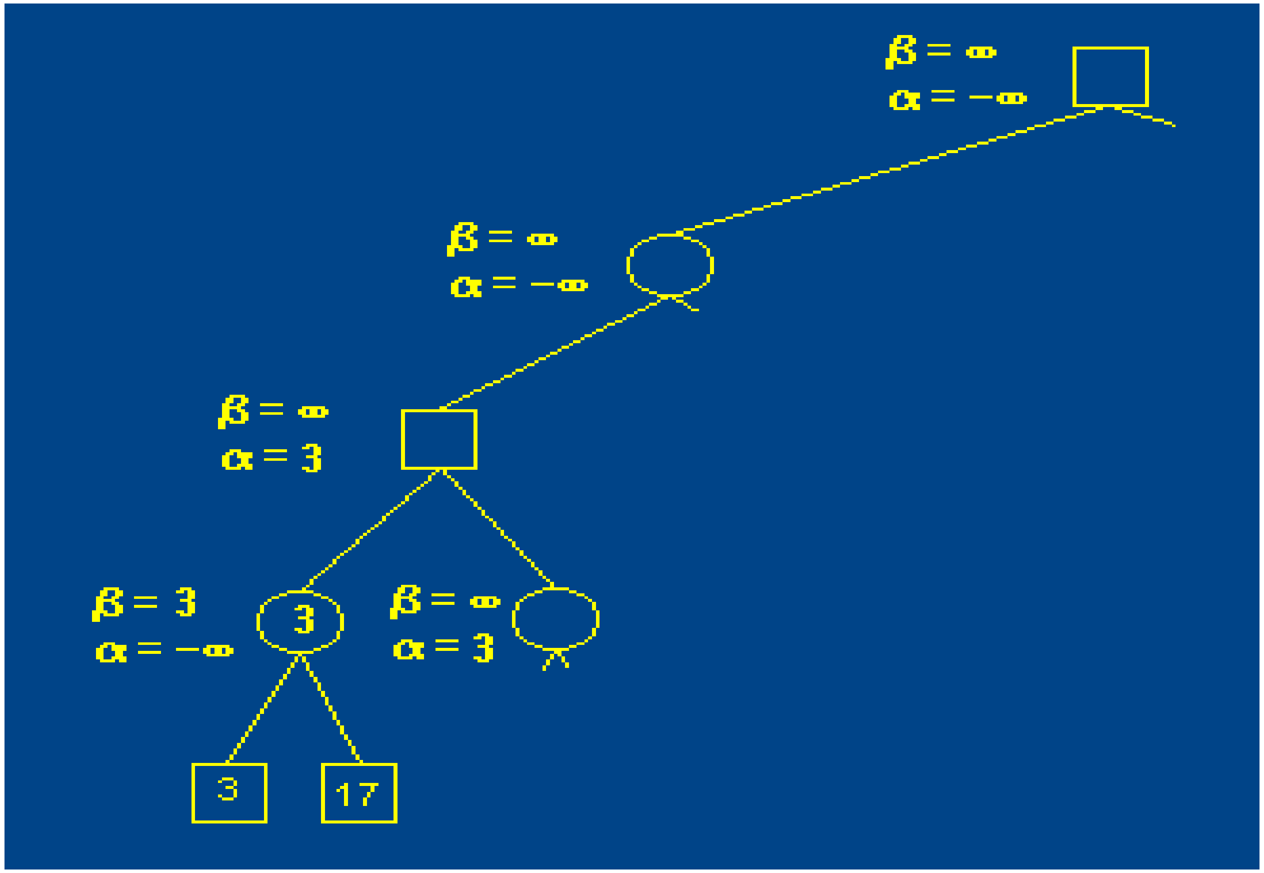
Example

- Since this is a min node and 17 is greater than 3, this child is ignored. Now we've seen all of the children of this min node, so we return the beta value to the max node above. Since it is a max node, we now know that it's value will be greater than or equal to 3, so we change alpha to 3:



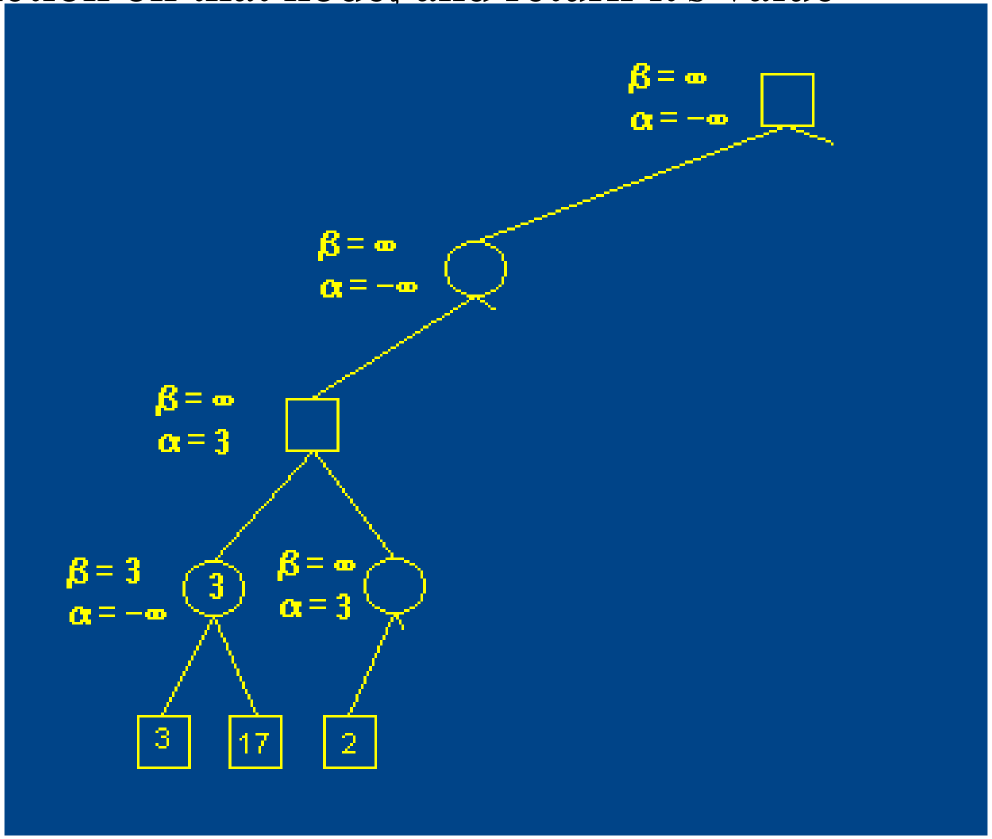
Example

- We generate the next child and pass the bounds along



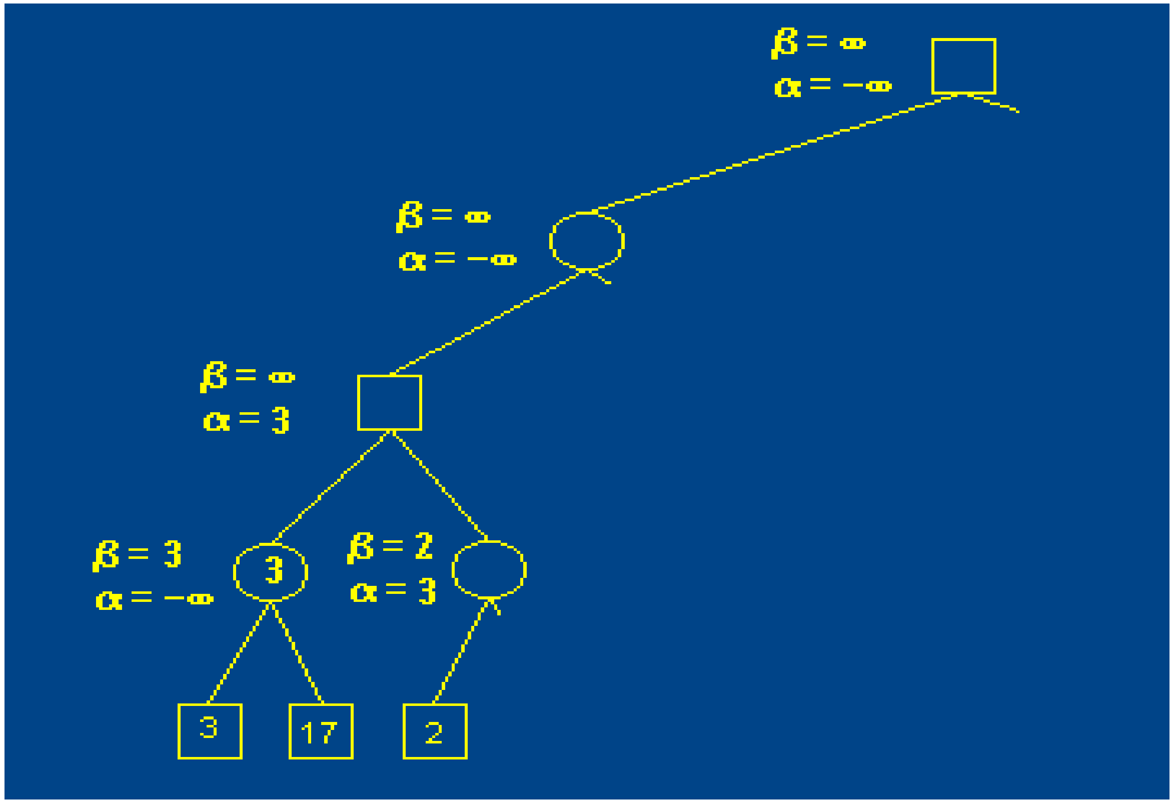
Exampl e

- Since this node is not at the target depth, we generate its first child, run the
evaluation function on that node, and return it's value



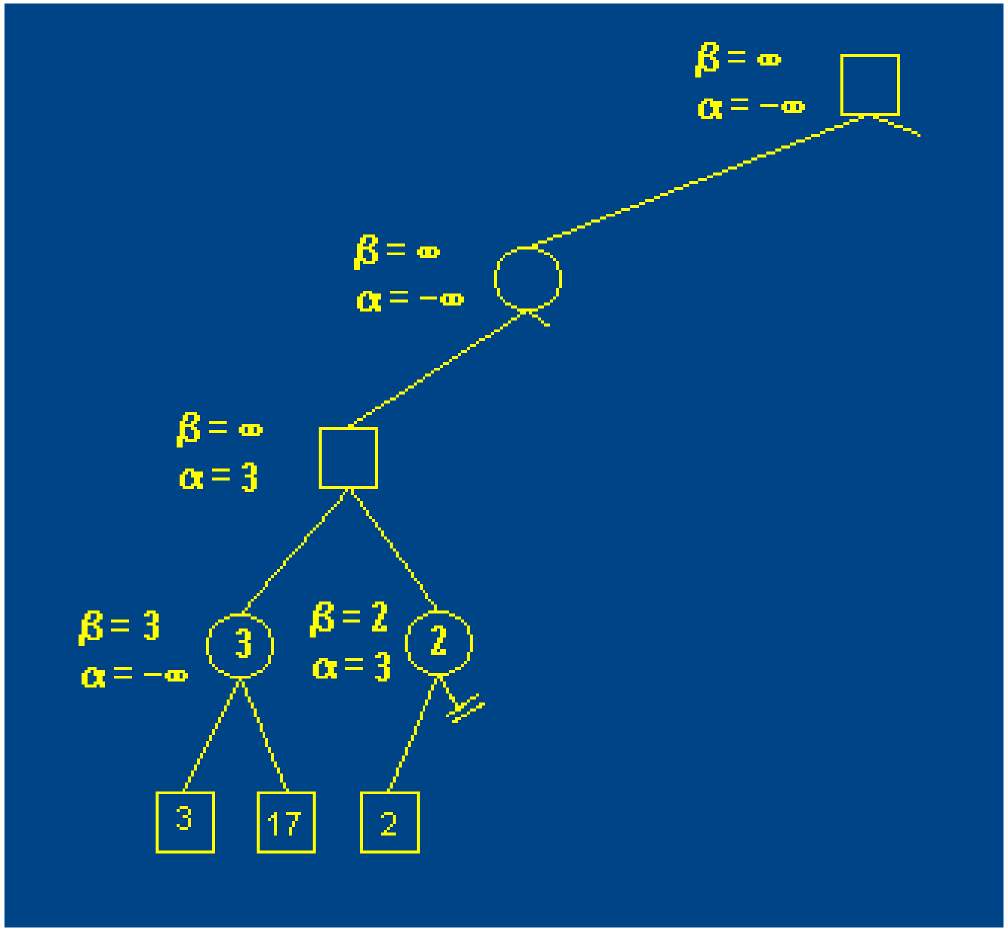
Exampl
e

- Since this is a min node, we now know that the value of this node will be
  less than or equal to 2, so we change beta to 2:



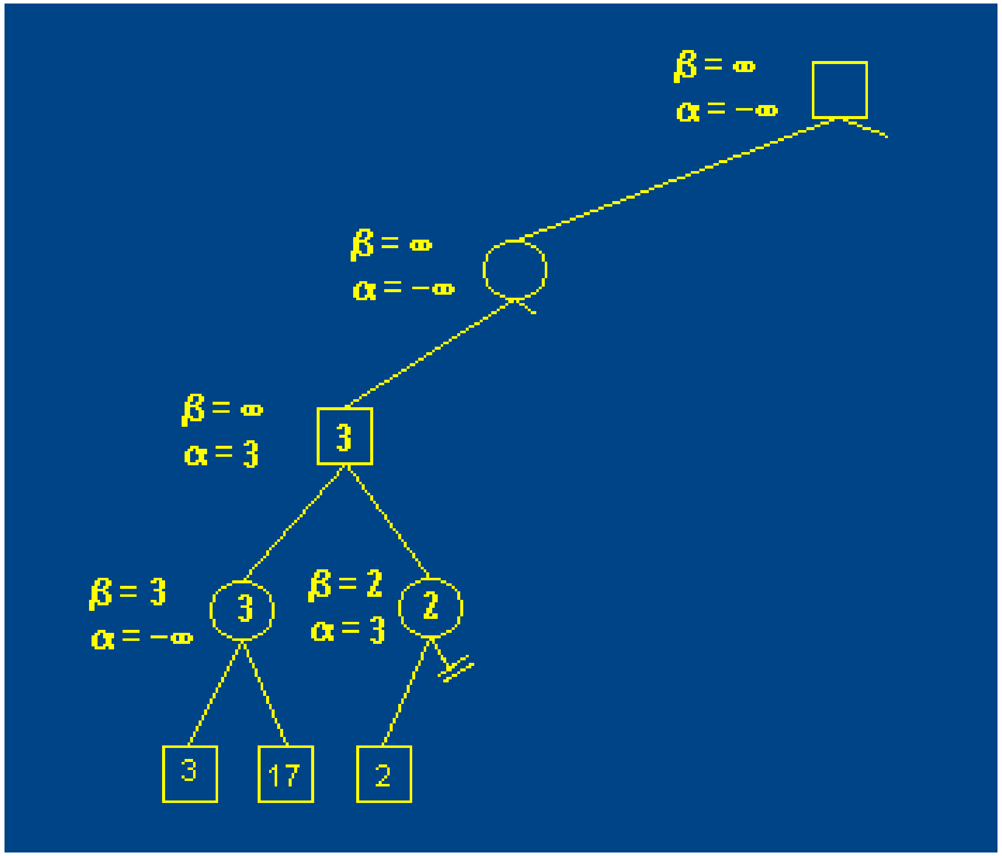
Example

- Admittedly, we don't know the actual value of the node. There could be a 1 or 0 or -100 somewhere in the other children of this node. But even if there was such a value, searching for it won't help us find the optimal solution in the search tree. The 2 alone is enough to make this subtree fruitless, so we can prune any other children and return it.
- **That's all there is to beta pruning!**



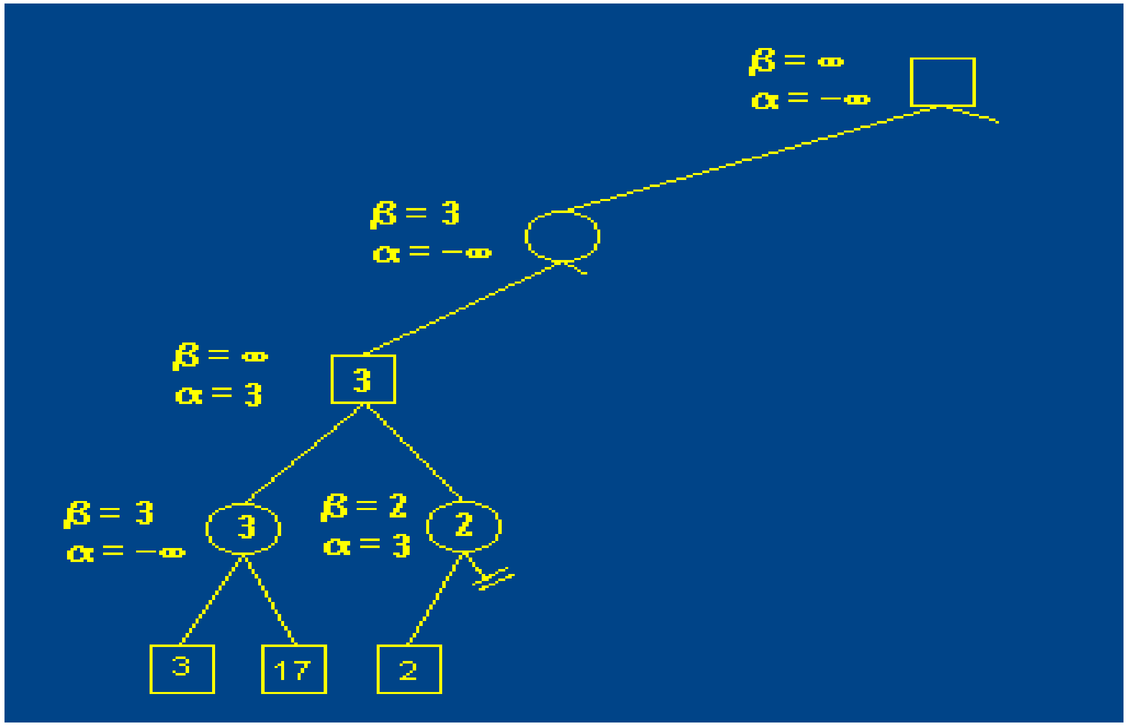
Example

- Back at the parent max node, our alpha value is already 3, which is more restrictive than 2, so we don't change it. At this point we've seen all the children of this max node, so we can set its value to the final alpha value:



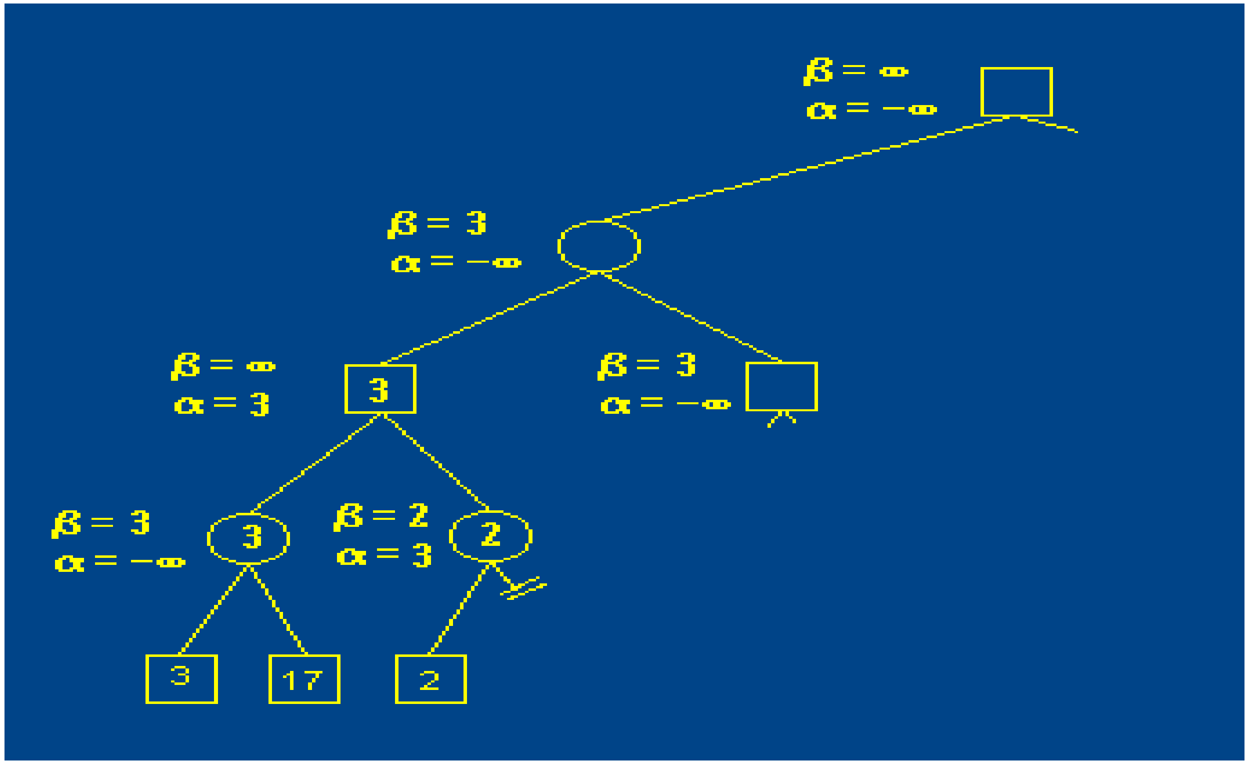
Example

- Now we move on to the parent min node. With the 3 for the first child value, we know that the value of the min node must be less than or equal to 3, thus we set beta to 3:



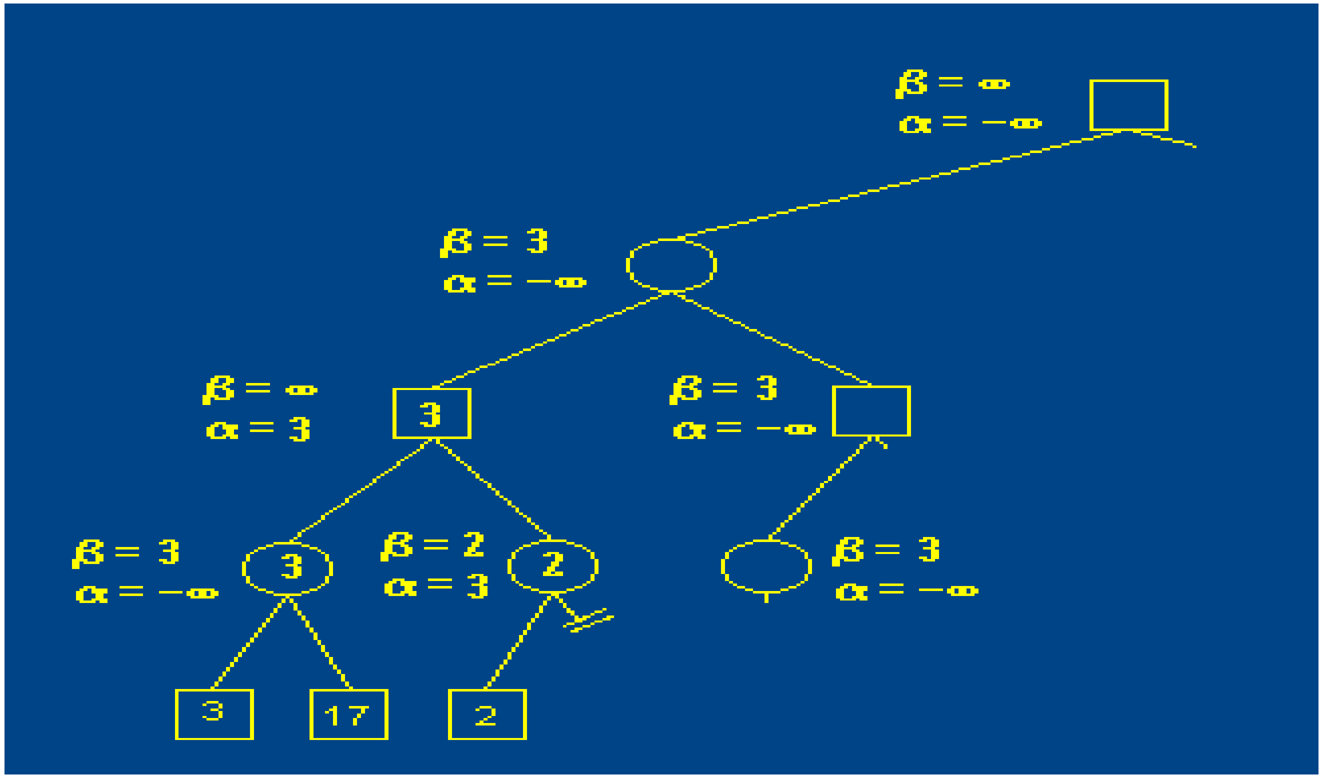
Example

- Since we still have a valid range, we go on to explore the next child. We generate the max node...



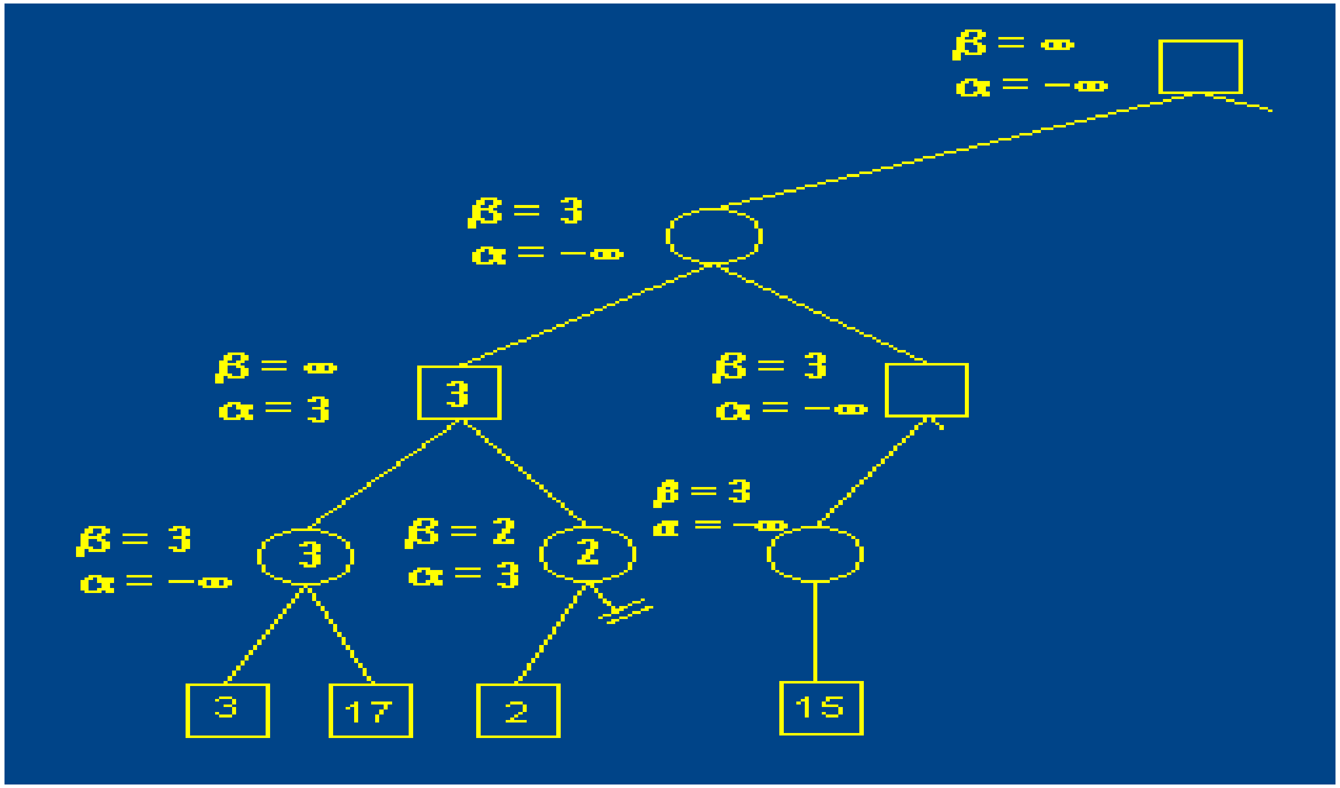
Example

- ... it's first child min node ...



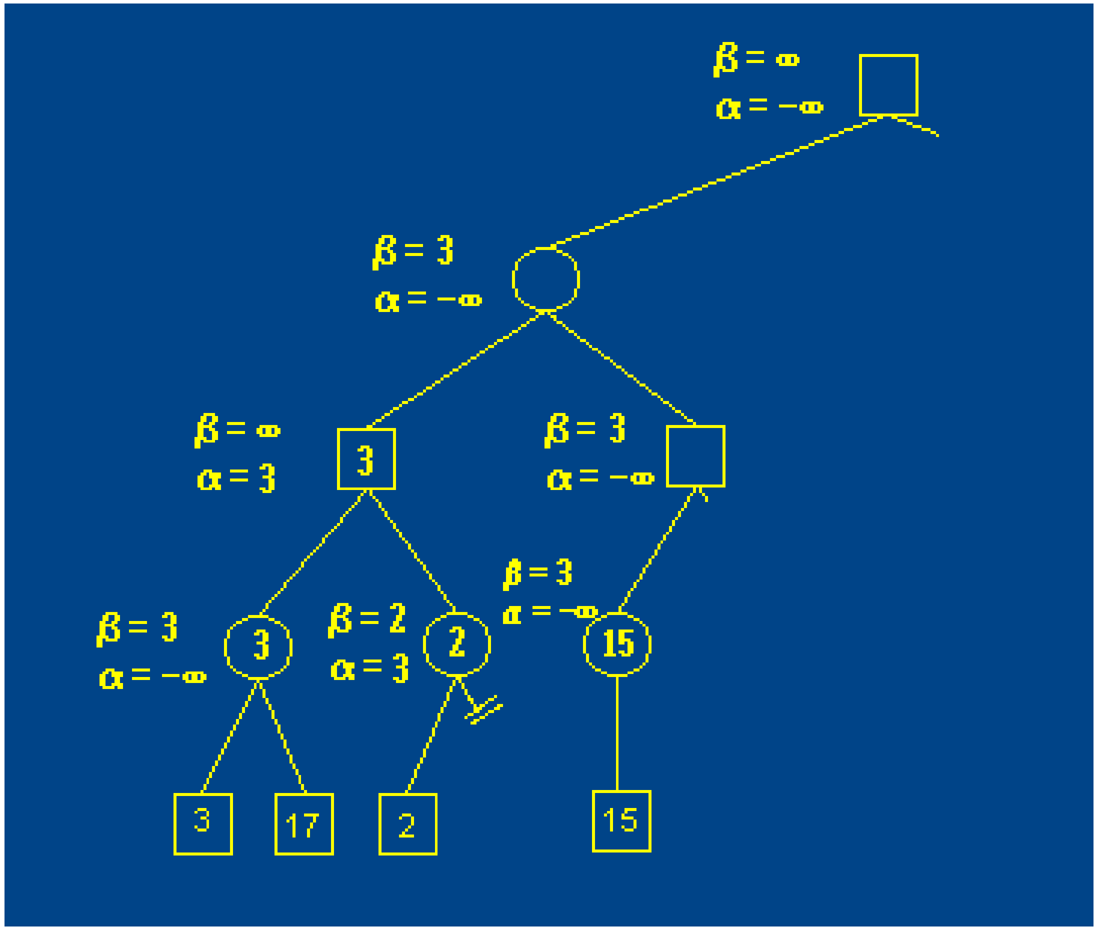
Example

- ... and finally the max node at the target depth. All along this path, we merely pass the alpha and beta bounds along.



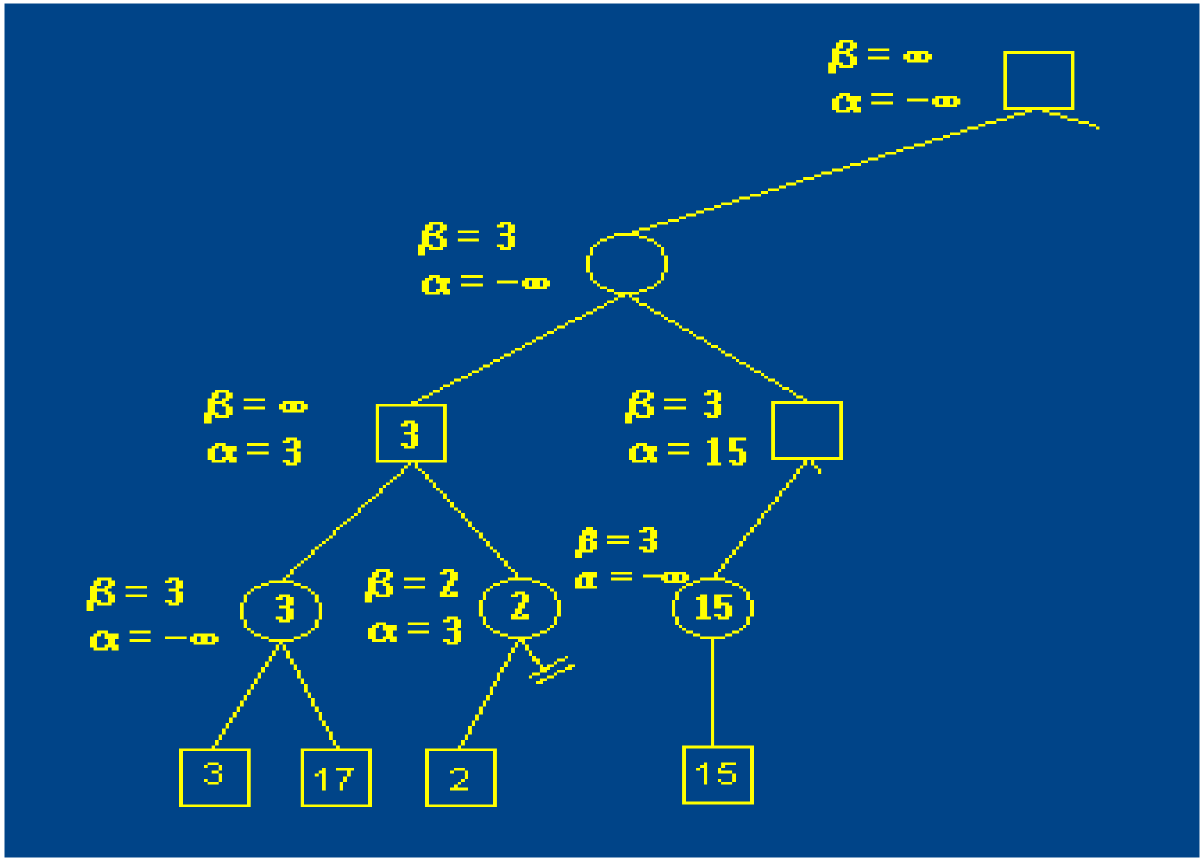
Example

- At this point, we've seen all of the children of the min node, and we haven't changed the beta bound. Since we haven't exceeded the bound, we should return the actual min value for the node. Notice that this is different than the case where we pruned, in which case you returned the beta value. The reason for this will become apparent shortly.



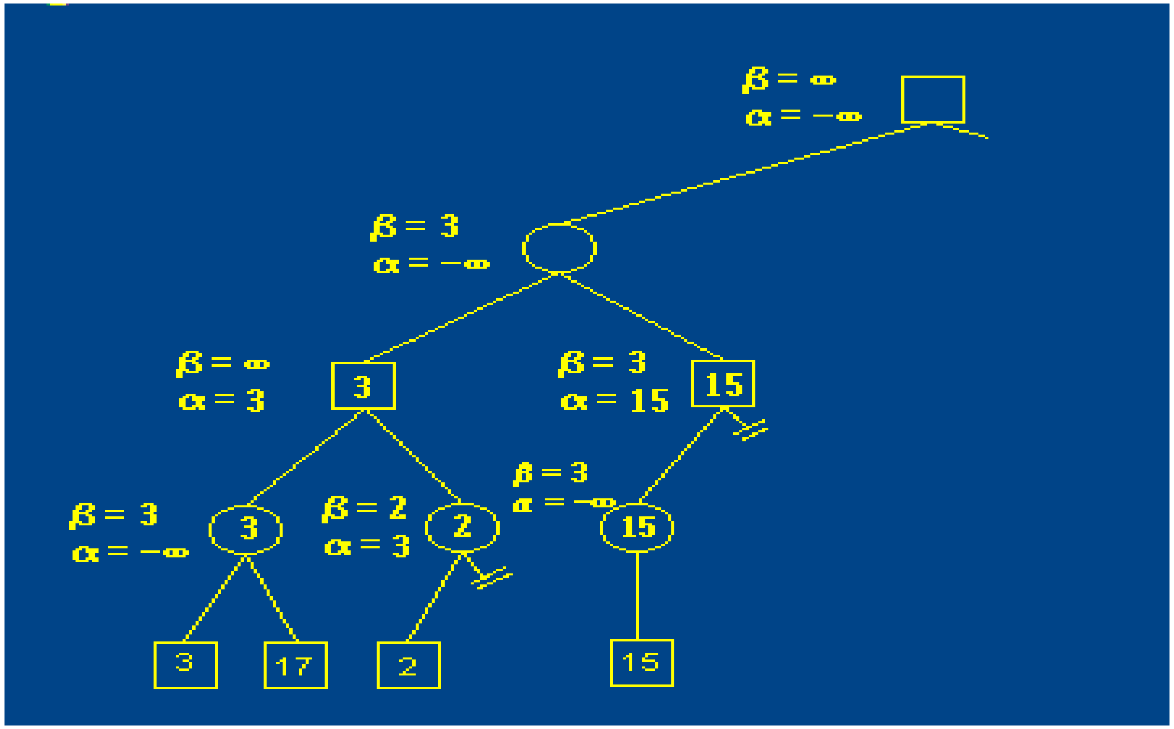
Example

- Now we return the value to the parent max node. Based on this value, we know that this max node will have a value of 15 or greater, so we set alpha to 15:



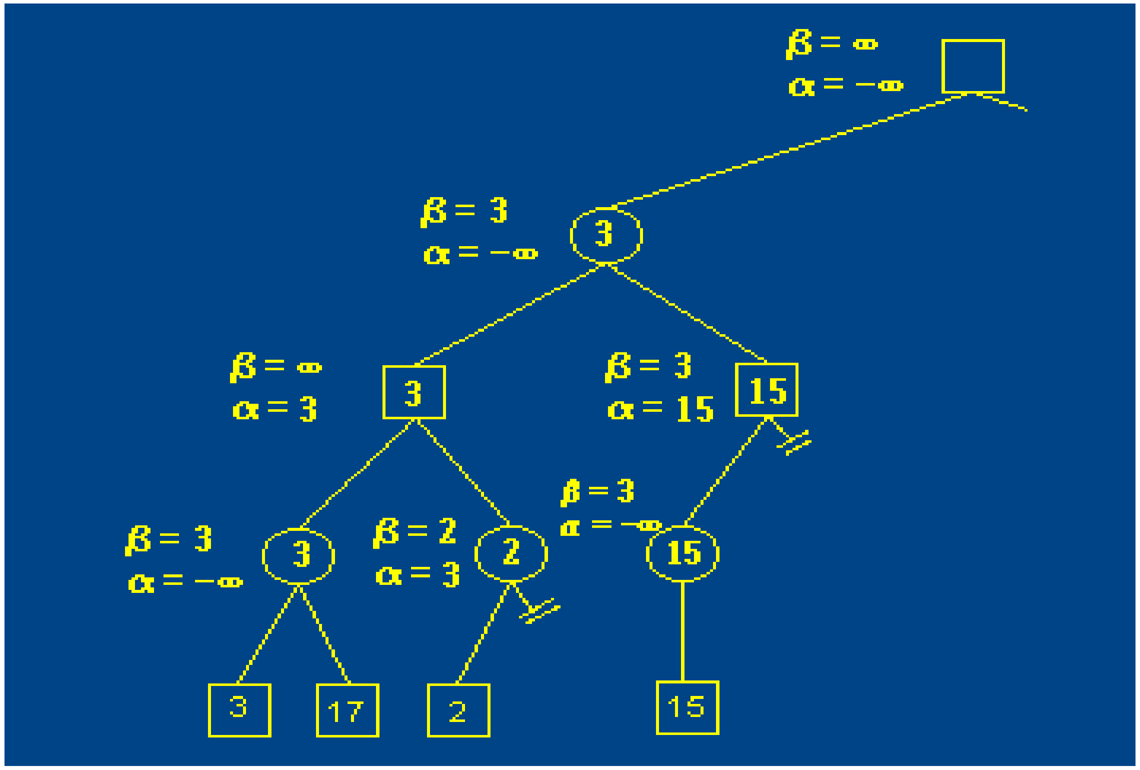
Exampl e

- Once again the alpha and beta bounds have crossed, so we can prune the rest of this node's children and return the value that exceeded the bound (i.e. 15). Notice that if we had returned the beta value of the child min node
(3) instead of the actual value (15), we wouldn't have been able to prune
here.



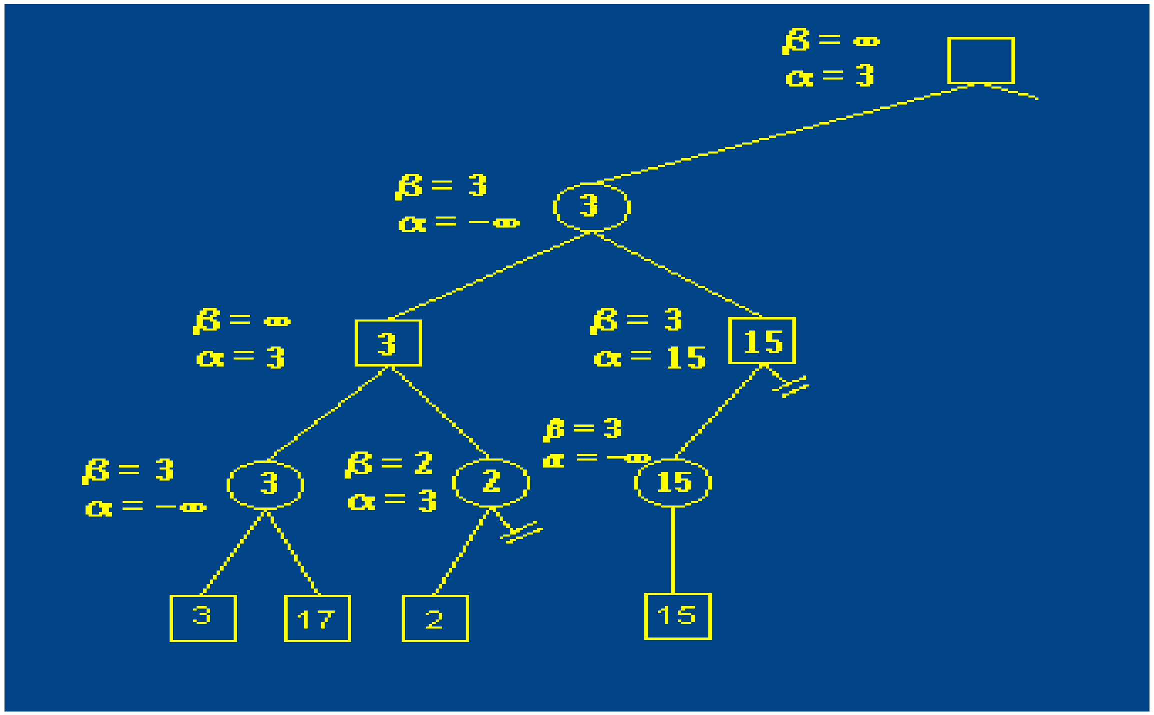
Example

- Now the parent min node has seen all of it's children, so it can select the minimum value of it's children (3) and return.



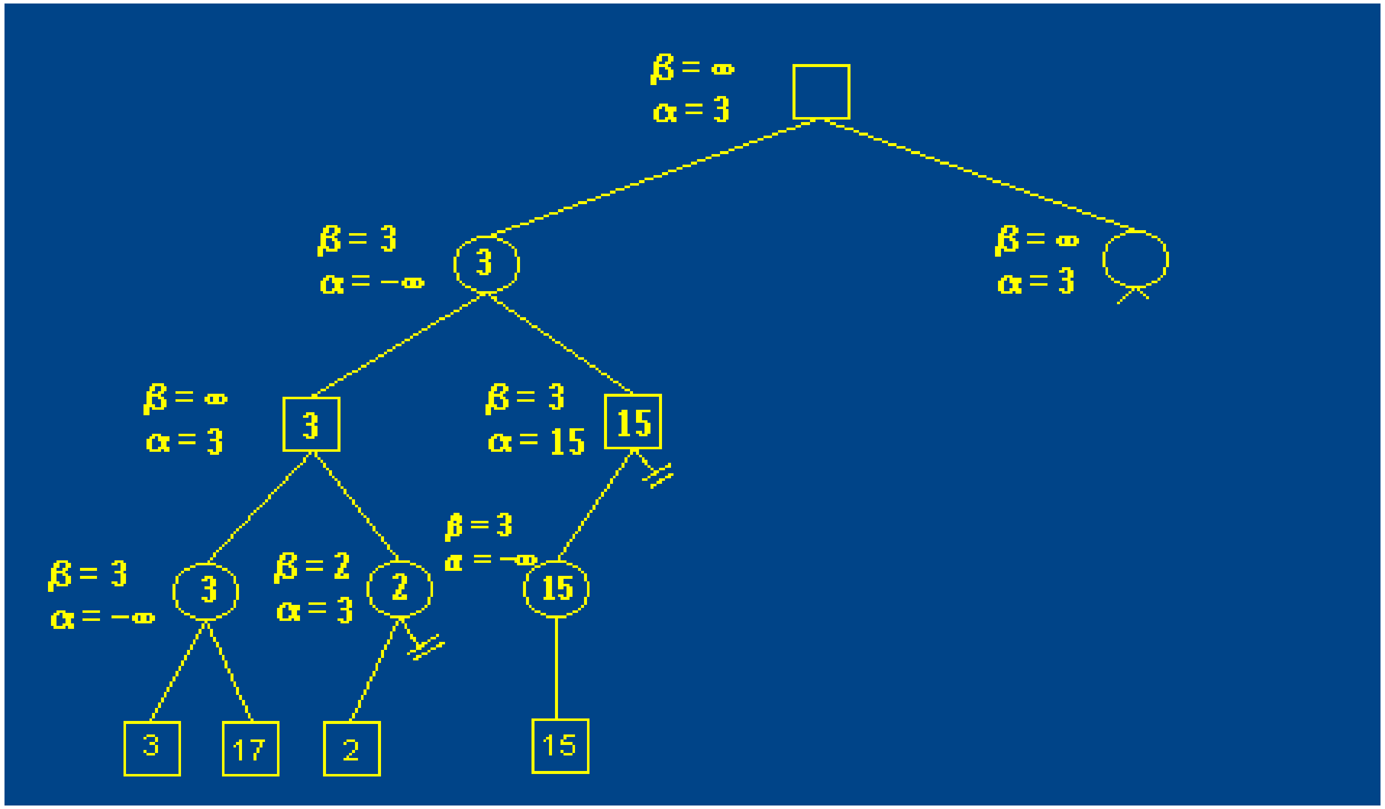
Exampl e

- Finally we've finished with the first child of the root max node. We now know our solution will be at least 3, so we set the alpha value to 3 and go on to the second child.



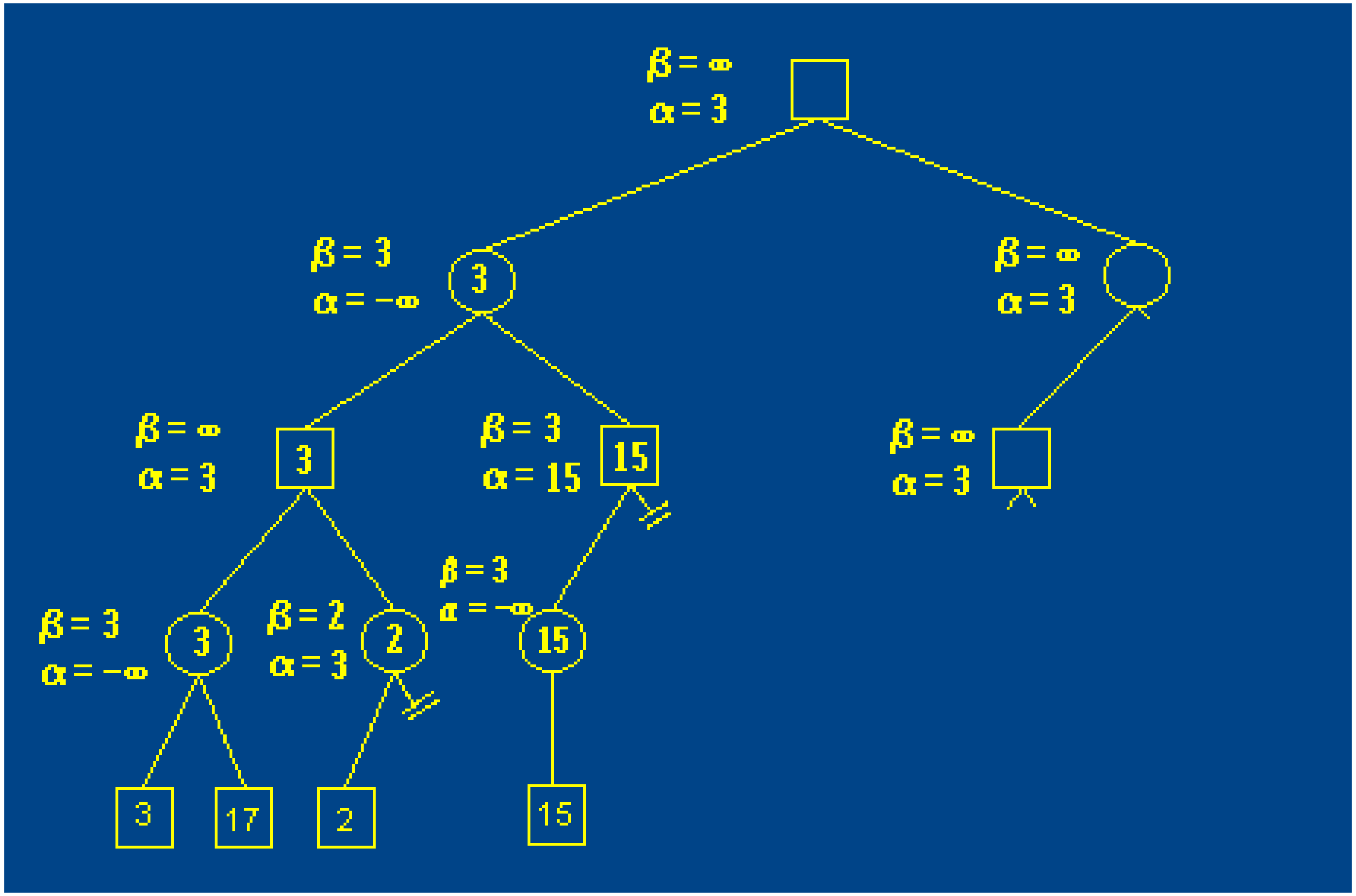
Exampl e

- Passing the alpha and beta values along as we go, we generate the second
  child of the root node...

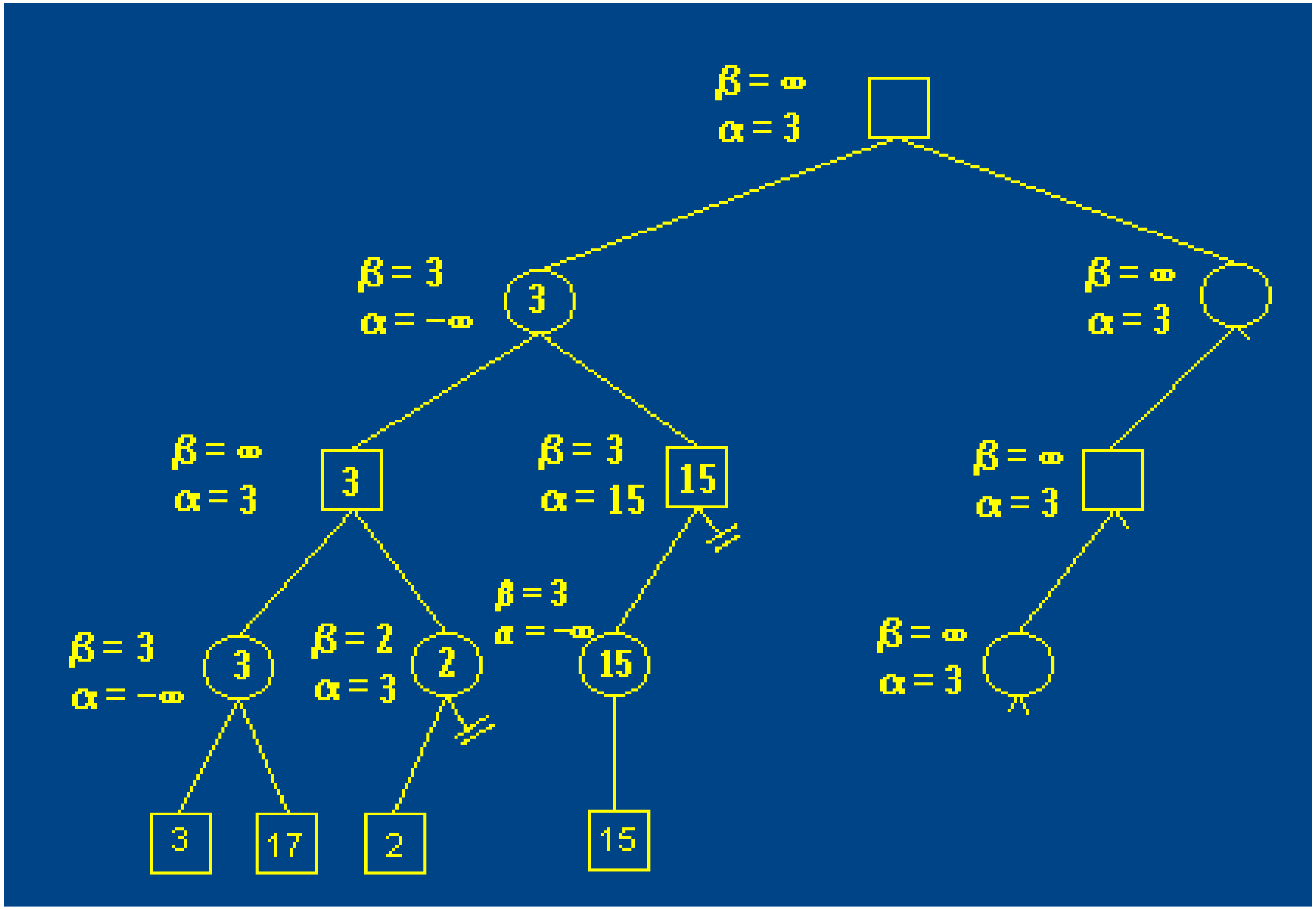


Example

- ... and its first child ...

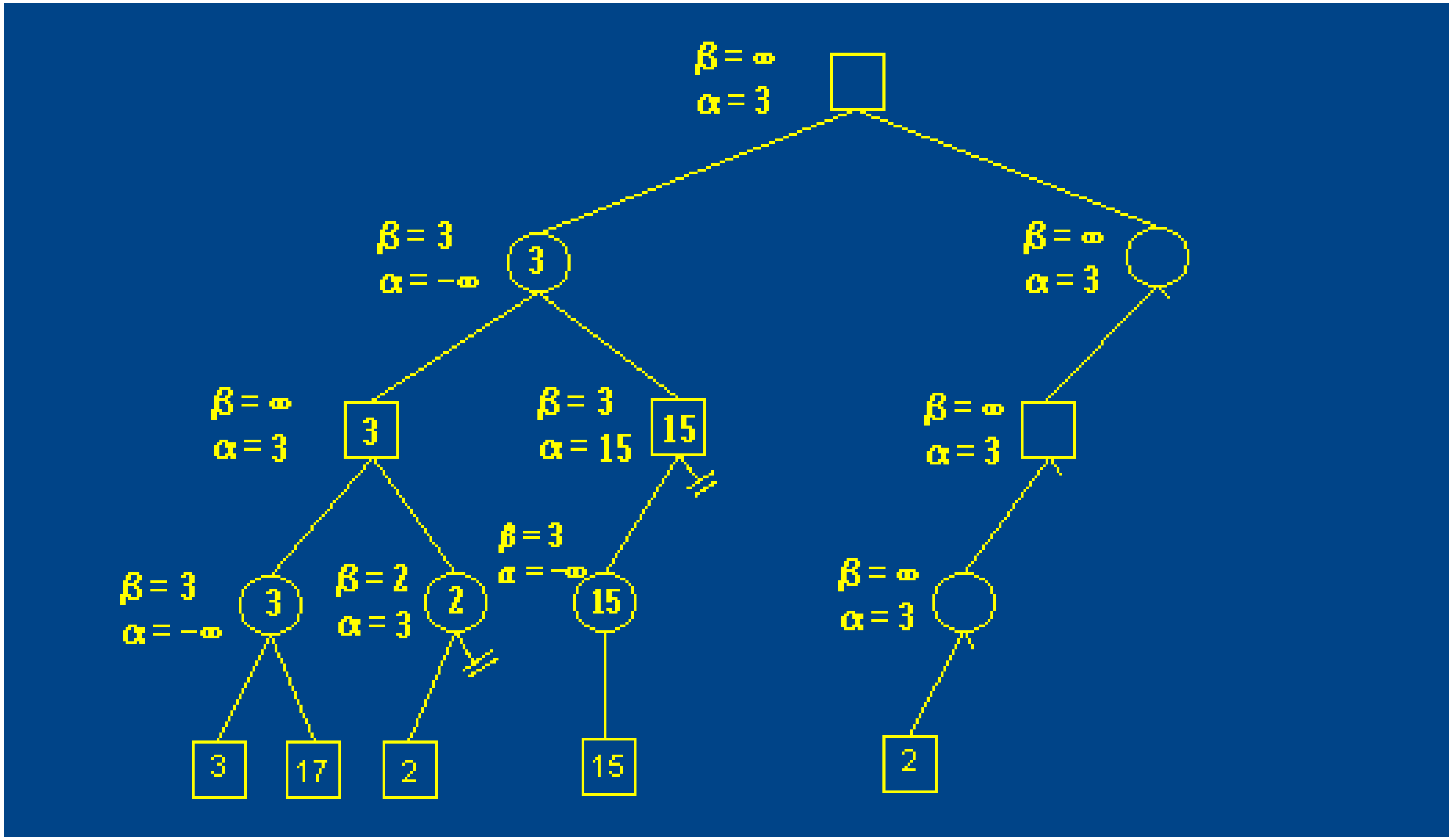


Exampl



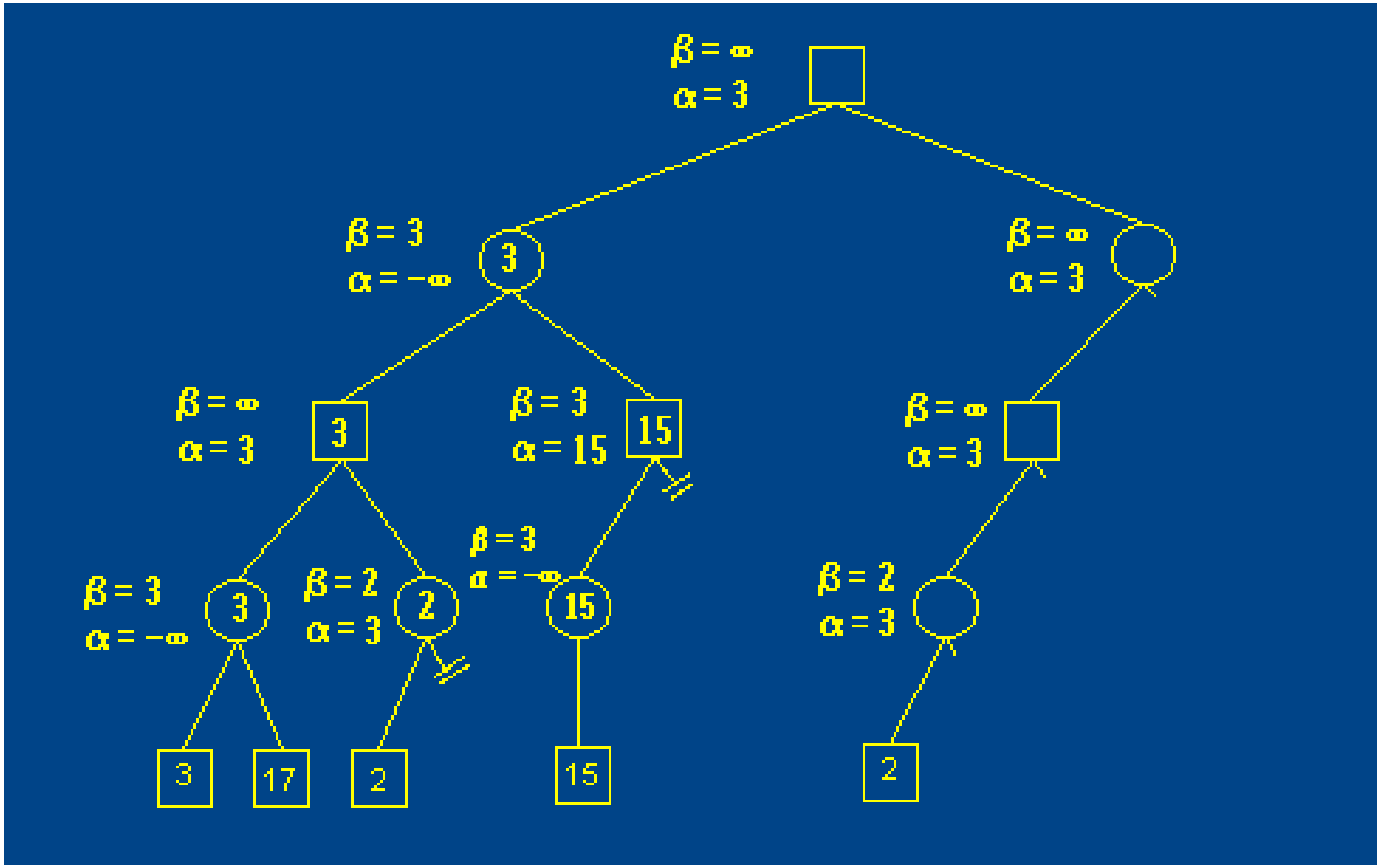
$\beta = \infty$
$\alpha = 3$
$\beta = 3$
$\alpha = -\infty$
3
$\beta = \infty$
$\alpha = 3$
$\beta = \infty$
$\alpha = 3$
3
$\beta = 3$
$\alpha = 15$
15
$\beta = \infty$
$\alpha = 3$
$\beta = 3$
$\alpha = -\infty$
$\beta = 3$
$\alpha = -\infty$
3
$\beta = 2$
$\alpha = 3$
2
15
$\beta = \infty$
$\alpha = 3$
3
17
2
15

Exampl



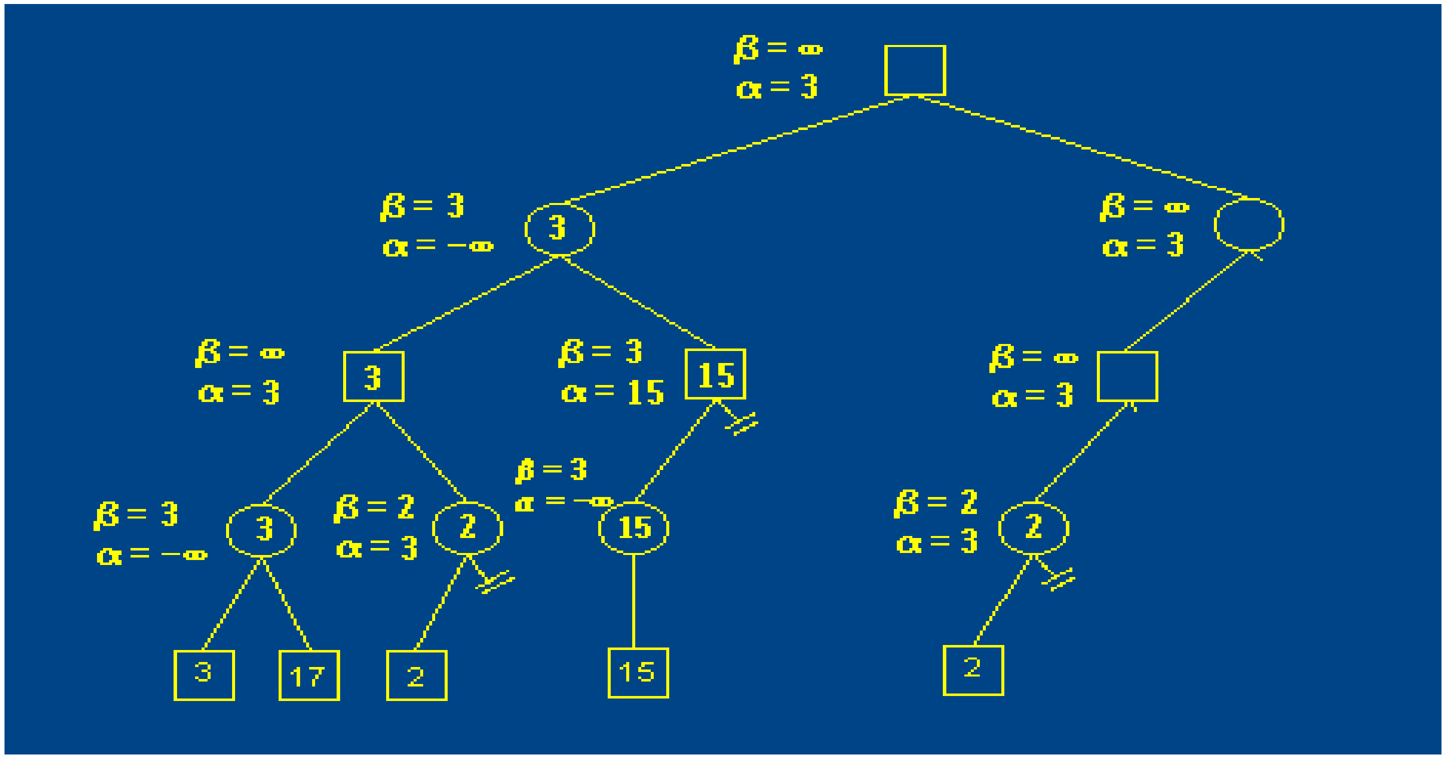
Exampl e

- The min node parent uses this value to set it's beta value to 2:



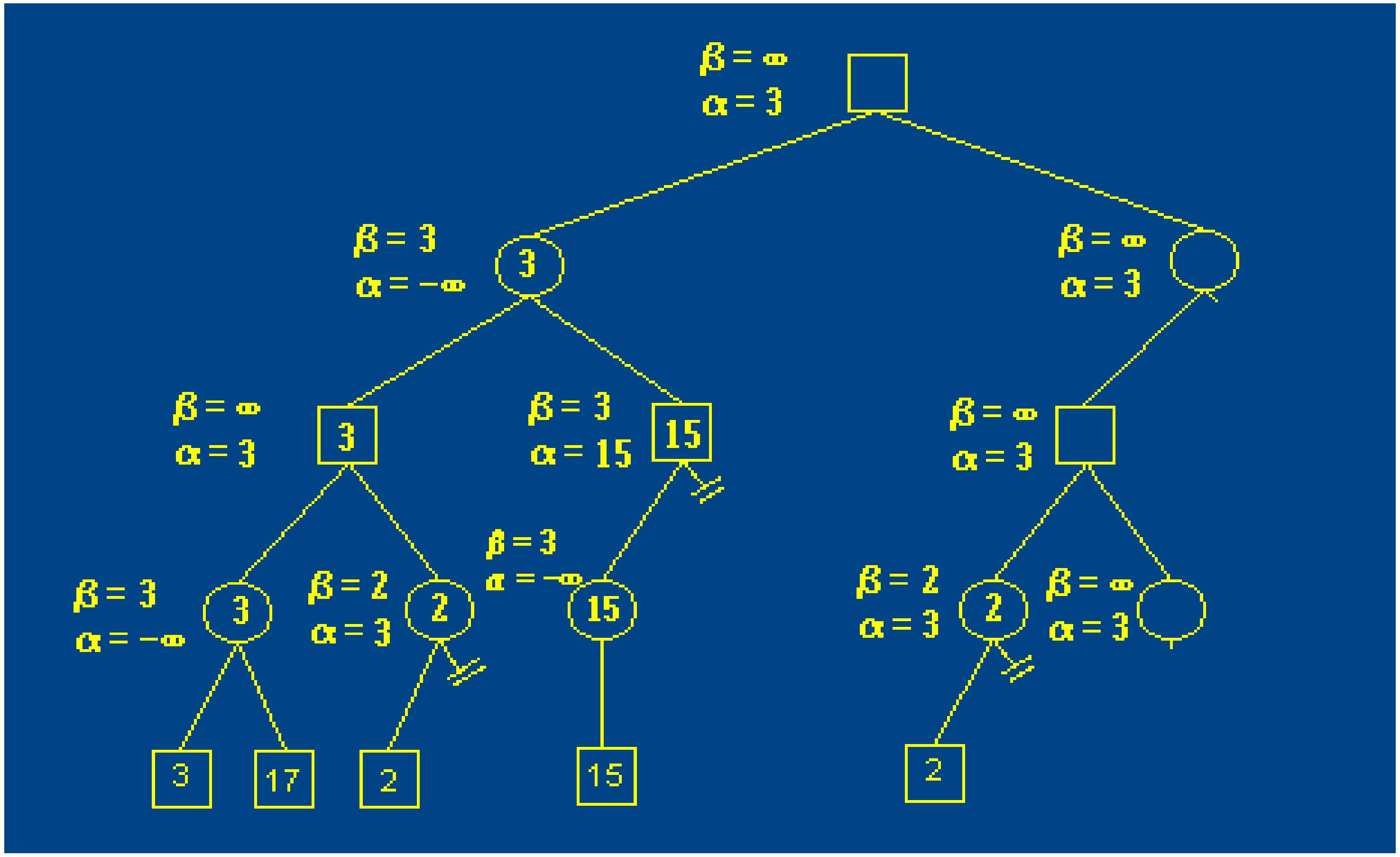
Exampl e

- Once again we are able to prune the other children of this node and return the value that exceeded the bound. Since this value isn't greater than the alpha bound of the parent max node, we don't change the bounds.



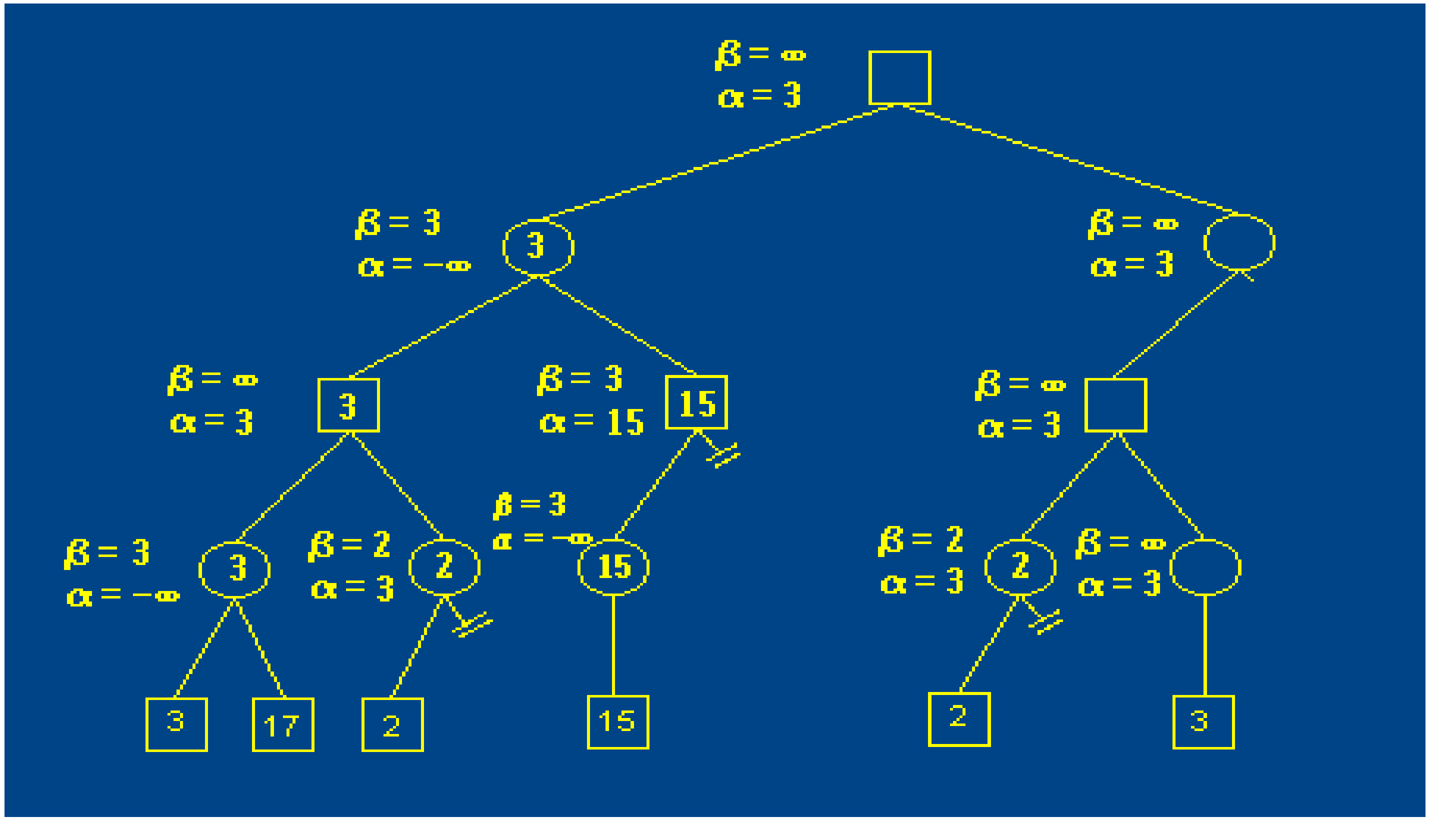
Example

- From here, we generate the next child of the max node:



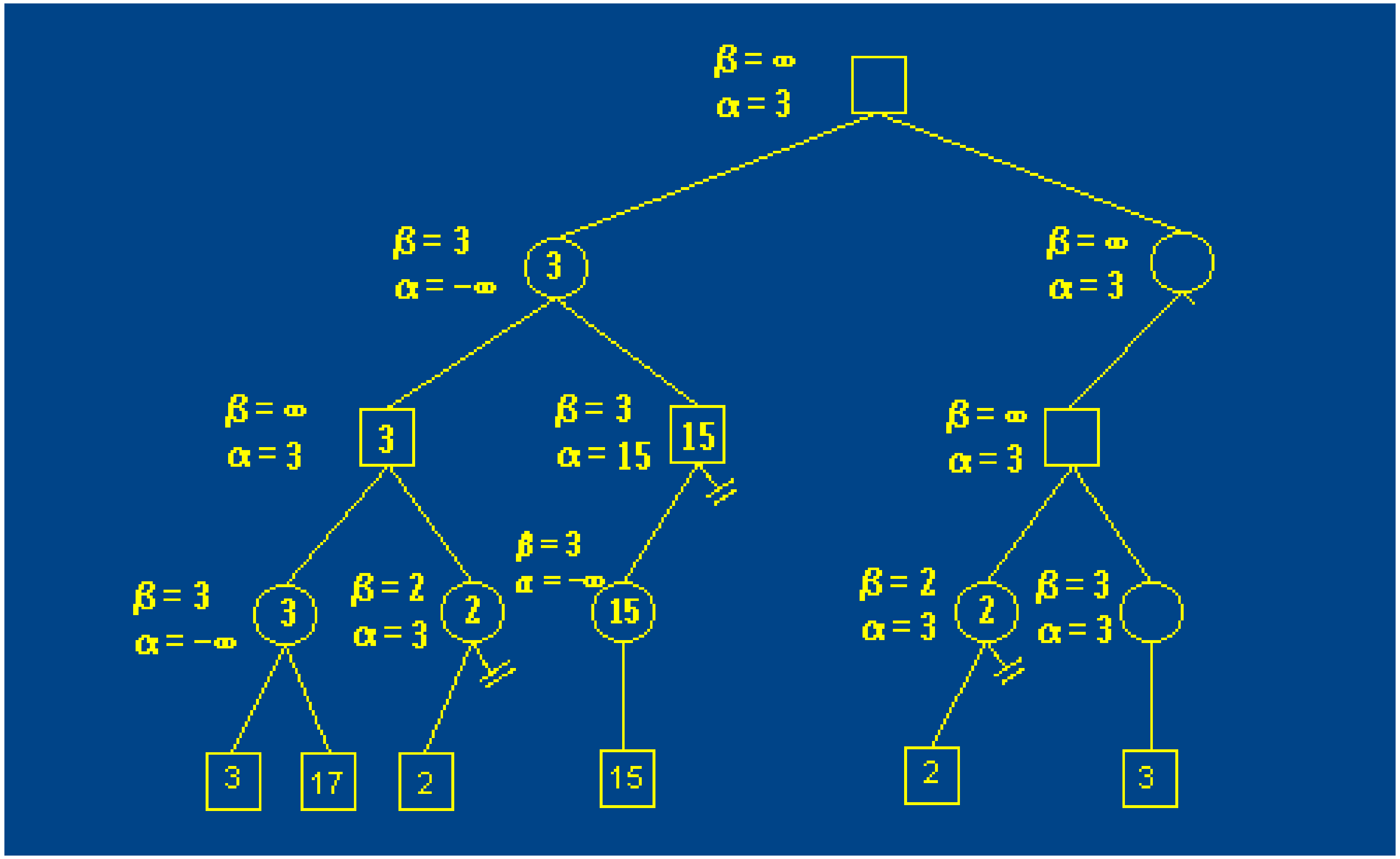
Example

- Then we generate its child, which is at the target depth. We call the evaluation function and get its value of 3.



Example

- The parent min node uses this value to set its upper bound (beta) to 3:

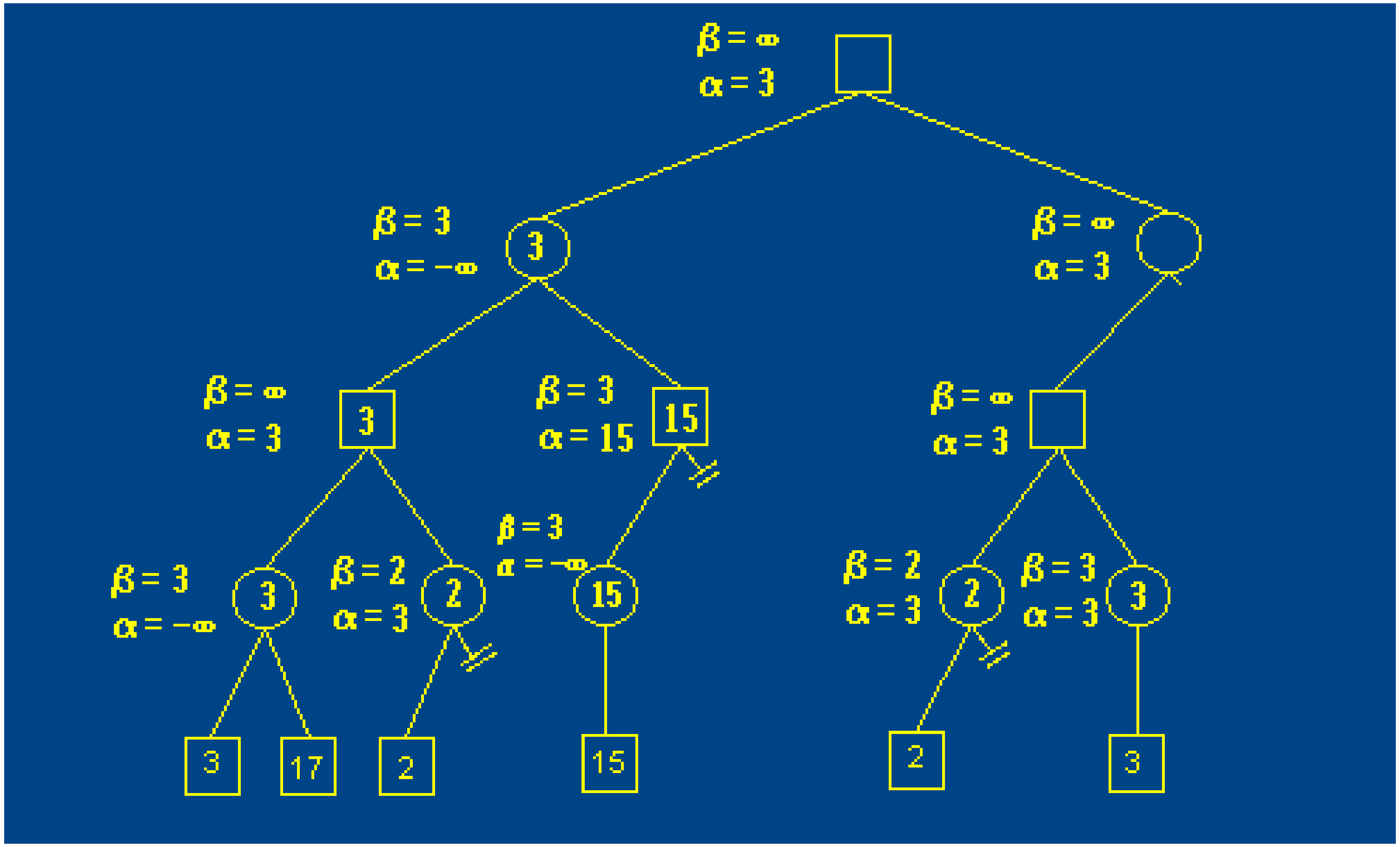


Example

- In other words, at this point alpha = beta. Should we prune here? We haven't actually*exceeded* the bounds, but since alpha and beta are equal, we know we can't really do *better* in this subtree.

- The answer is yes, we should prune. The reason is that even though we can't do better, we might be able to do worse. Remember, the task of minimax is to find the best move to make at the state represented by the top level max node. As it happens we've finished with this node's children anyway, so we return the min value 3.

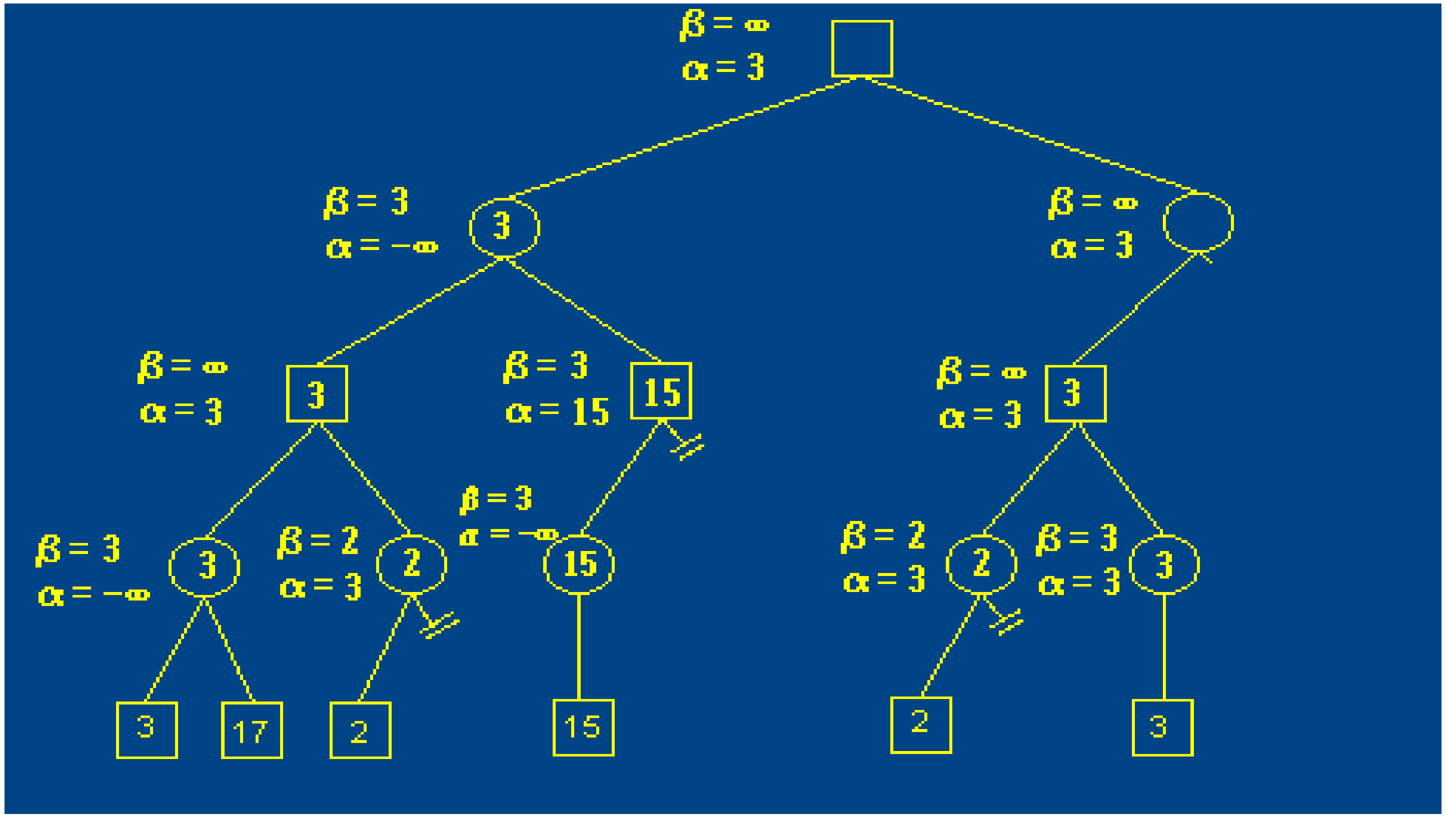
Example



β = ∞
α = 3
β = 3
α = −∞
3
β = ∞
α = 3
β = ∞
α = 3
3
β = 3
α = 15
15
β = ∞
α = 3
β = 3
α = −∞
3
β = 2
α = 3
2
β = 3
α = −∞
15
β = 2
α = 3
2
β = 3
α = 3
3
3
17
2
15
2
3

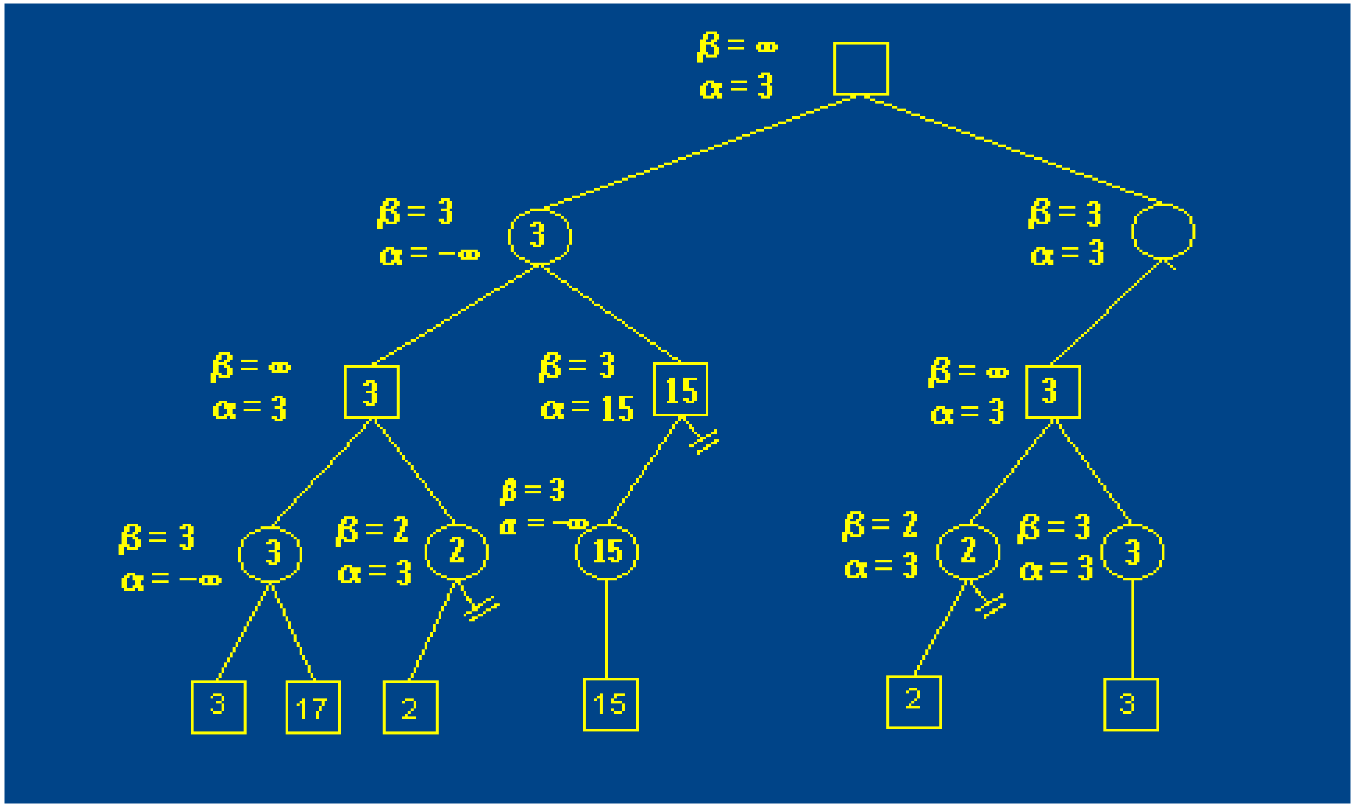
Example

- The max node above has now seen all of its children, so it returns the maximum value of those it has seen, which is 3.



Example

- This value is returned to its parent min node, which then has a new upper bound of 3, so it sets beta to 3:



- Once again, we're at a point where alpha and beta are tied, so we prune. Note that a real solution doesn't just indicate a number, but what move led to that number.
- If you were to run minimax on the list version presented at the start of the example, your minimax would return a value of 3 and 6 terminal nodes would have been examined

Conclusion

- Pruning does not affect final results.
- Entire subtrees can be pruned, not just leaves.
- Good move *ordering* improves effectiveness of pruning.
- With *perfect ordering*, time complexity is $O(b^{m/2})$.
  - Effective branching factor of sqrt(b)
  - Consequence: alpha-beta pruning can look twice as deep as minimax in the same amount of time.

Forward Pruning

Forward Pruning

- Forward pruning- some moves at a given node are pruned immediately without further consideration.
- Humans playing chess consider only a few moves from each position.
- **FORWARD PRUNING** – “Beam Search” Beam of n best moves rather than the all possible moves.

(Not Best Approach).

- PROBCUT /Probabilistic cut algorithm version of Alpha beta search-gain previous experience (less chance to pruned)

Forward Pruning

Alpha–beta search prunes any node that is *provably outside* current (α, β) window. PROBCUT also prunes nodes that are *probably outside the* window.
past experience to estimate how likely it is that a score of v at depth d in the tree would be outside (α, β).
Let us assume we have implemented an evaluation function for chess, a reasonable cutoff test with a quiescence search, and a large transposition table.

Forward Pruning

Generate and evaluate around a million nodes/s on the latest PC, allowing us to search roughly 200 million nodes per move under standard time controls.
The branching factor for chess is about 35, on average, and 355 is about 50 million, so if we used minimax search, we could look ahead only about five plies.
With alpha–beta search we get to about 10 plies, which results in an expert level of play.
To reach grandmaster status we would need an extensively tuned evaluation function and a large database of optimal opening and endgame moves.

Cutting off Search

Cutting of Search

modify **ALPHA-BETA-SEARCH** so that it will call the heuristic EVAL function when it is appropriate to cut off the search.

if CUTOFF-TEST(state, depth) then return EVAL(state)

We also must arrange for some bookkeeping so that the current depth is incremented on each recursive call.

The most straightforward approach to controlling the amount of search is to set a fixed depth limit so that CUTOFF-TEST(state, depth) returns true for all depth greater than some fixed depth d.

Cutting of Search

The depth d is chosen so that a move is selected within the allocated time. A more robust approach is to apply iterative deepening.
A more robust approach is to apply iterative deepening.
Consider again the simple evaluation function for chess based on material advantage

QUIESCENT

More Sophisticated Cutoff test is needed

QUIESCENT: to exhibit wild swings in value in near future.

EX: *Chess-*

(a) Favorable captures are NOT Quiescent for evaluation function.

(b) Nonquiescent positions can be expanded further until quiescent positions are reached.

This extra search called Quiescence Search- certain types of moves resolve uncertainty in position.

Horizon Effect

The Horizon Effect is difficult to Eliminate.
When Program facing opponent move cause series damage and avoidable but temporarily avoid delay tactics.
Ex: Chess game- No way for black bishop to
escape. (Over the Horizon)

Singular Extension

To mitigate the Horizon Effect is Singular Extension.
A move is “clearly better” than all moves in given position.
Course of search this single move is remembered.
Search reaches normal depth limit algorithm checks if singular extension is legal move and algorithm allows to consideration.
The tree deeper, but because there will be few singular extensions, it does not add many total nodes to the tree.

Genetic Algorithm

Session Outcomes

- **Students will learn about Genetic Algorithms**
- **Students will learn about Online search agents**

Genetic Algorithm

- Nature has always been a great source of inspiration to all mankind. Genetic Algorithms (GAs) are search based algorithms based on the concepts of natural selection and genetics. GAs are a subset of a much larger branch of computation known as **Evolutionary Computation**.

Basic Terminology

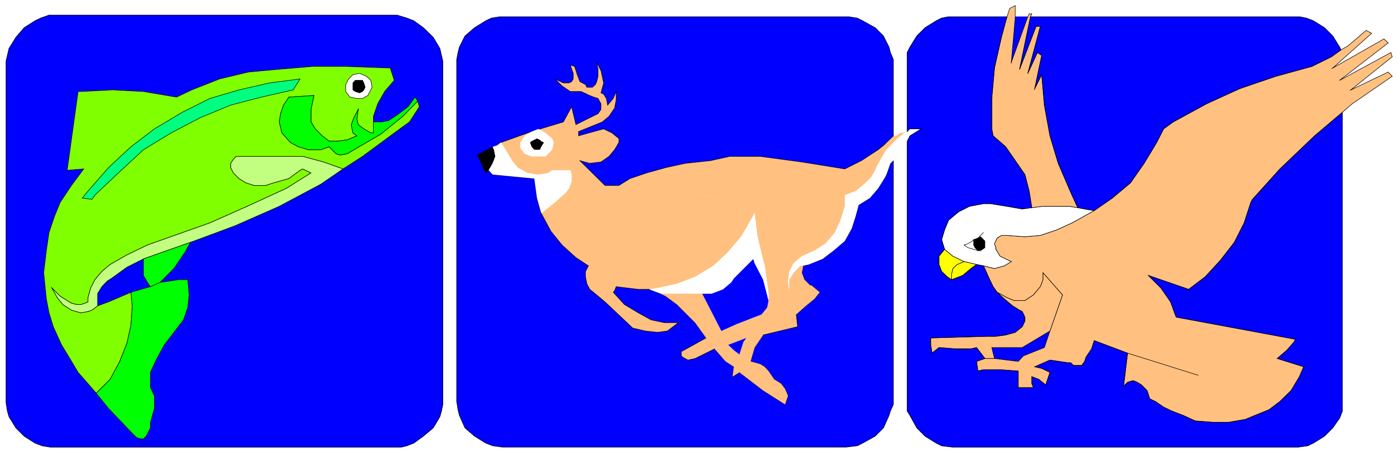
- **Population** − It is a subset of all the possible (encoded) solutions to the given problem. The population for a GA is analogous to the population for human beings except that instead of human beings, we have Candidate Solutions representing human beings.
- **Chromosomes** − A chromosome is one such solution to the given problem.
- **Gene** − A gene is one element position of a chromosome.
- **Allele** − It is the value a gene takes for a particular chromosome.

- **Genotype** − Genotype is the population in the computation space. In the computation space, the solutions are represented in a way which can be easily understood and manipulated using a computing system.
- **Phenotype** − Phenotype is the population in the actual real world solution space in which solutions are represented in a way they are represented in real world situations.
- **Decoding and Encoding** − For simple problems, the **phenotype and genotype** spaces are the same. However, in most of the cases, the phenotype and genotype spaces are different. Decoding is a process of transforming a solution from the genotype to the phenotype space, while encoding is a process of transforming from the phenotype to genotype space. Decoding should be fast as it is carried out repeatedly in a GA during the fitness value calculation.

Simple Genetic Algorithm

```
{
  initialize population;
  evaluate population;
  while TerminationCriteriaNotSatisfied
  {
    select parents for reproduction;
    perform recombination and mutation;
    evaluate population;
  }
}
```

A Simple Example



"The Gene is by far the most sophisticated program around."

- Bill Gates, *Business Week*, June 27, 1994

A Simple Example

The Traveling Salesman Problem:

Find a tour of a given set of cities so that

- each city is visited only once
- the total distance traveled is minimized

Representation

Representation is an ordered list of city numbers known as an *order-based* GA.

1) London	3) Dunedin	5) Beijing	7) Tokyo
2) Venice	4) Singapore	6) Phoenix	8) Victoria

```
CityList1    (3  5  7  2  1  6  4  8)
CityList2    (2  5  7  6  8  1  3  4)
```

Crossover

Crossover combines inversion and recombination:

			*			*		
Parent1	(3	5	7	2	1	6	4	8)
Parent2	(2	5	7	6	8	1	3	4)
Child	(5	8	7	2	1	6	3	4)

This operator is called the *Order1* crossover.

Mutation

Mutation involves reordering of the list:

Before: $(5\quad 8\quad \overset{*}{\boxed{7}}\quad 2\quad 1\quad \overset{*}{\boxed{6}}\quad 3\quad 4)$

After: $(5\quad 8\quad \boxed{6}\quad 2 \xrightarrow{\quad} 1\quad \boxed{7}\quad 3\quad 4)$

Issues for GA Practitioners

- Choosing basic implementation issues:
  - representation
  - population size, mutation rate, ...
  - selection, deletion policies
  - crossover, mutation operators
- Termination Criteria
- Performance, scalability
- Solution is only as good as the evaluation function (often hardest part)

Online Search Agents and Unknown Environment

- Offline search algorithms compute a complete solution before setting foot in the real world and then execute the solution. In ONLINE SEARCH contrast, an **online search agent interleaves computation and action: first it takes an action, then it observes the environment and computes the next action.**

- Online search is a good idea in dynamic or semi-dynamic domains—domains where there is a penalty for sitting around and computing too long.

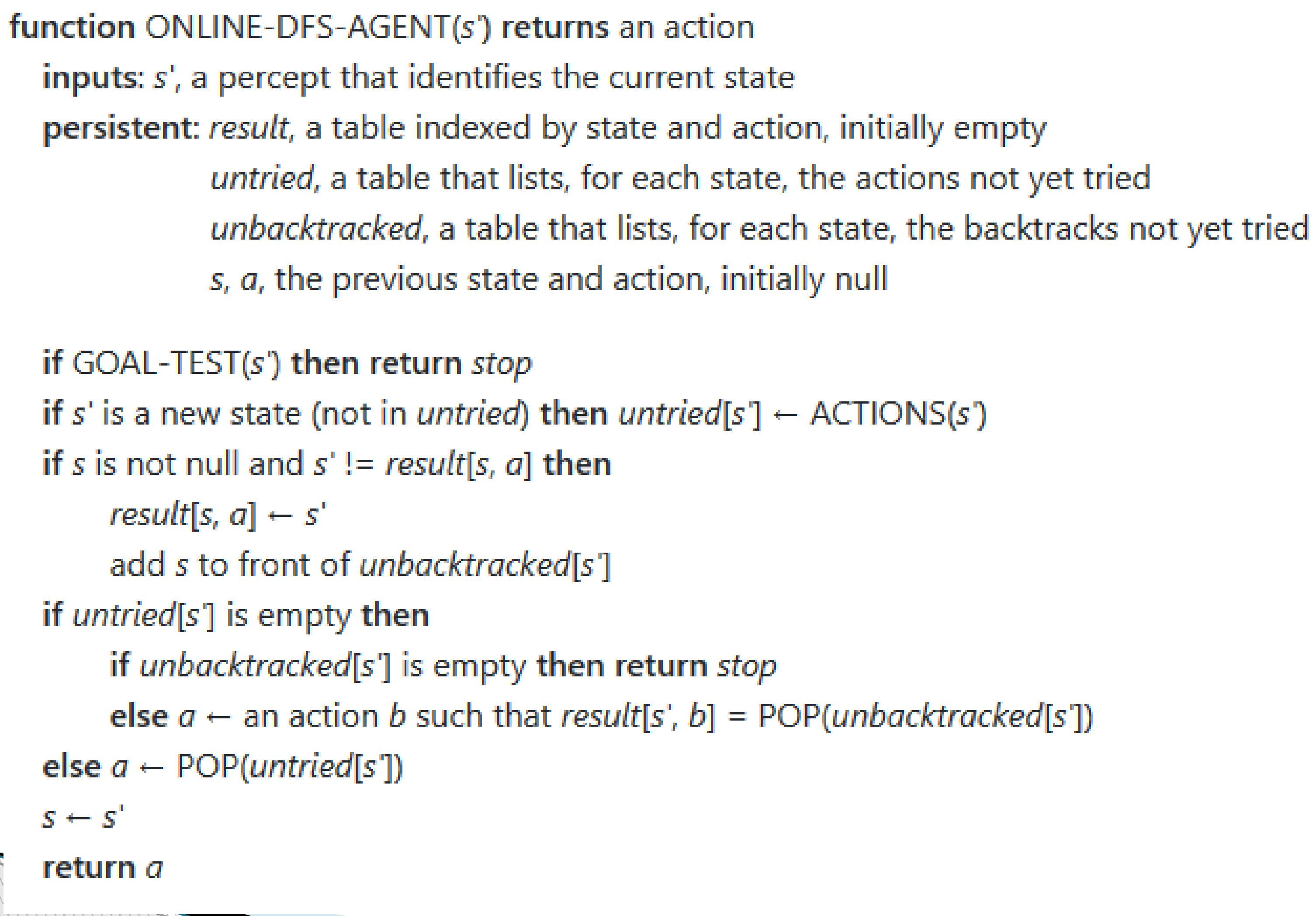
- Online search is also helpful in nondeterministic domains because it allows the agent to focus its computational efforts on the contingencies that actually arise rather than those that might happen but probably won't.
- Online search is a necessary idea for unknown environments, where the agent does not know what states exist or what its actions do. In this state of ignorance, the agent faces an **exploration problem** and must use its actions as experiments in order to learn enough to make deliberation worthwhile.

An agent knows only the following in online search:

- **ACTIONS(s)**, which returns a list of actions allowed in state s;
- **The step-cost function c(s, a, s')—**note that this cannot be used until the agent knows that s' is the outcome; and
- **GOAL-TEST(s**).

1. An online search problem must be solved by an agent executing actions, rather than by pure computation.
2. An agent cannot determine RESULT(s, a) except by actually being in s and doing a.
3. Competitive Ratio should be small and not moves to infinite.

Online DFS Agent



```
function ONLINE-DFS-AGENT(s') returns an action
  inputs: s', a percept that identifies the current state
  persistent: result, a table indexed by state and action, initially empty
              untried, a table that lists, for each state, the actions not yet tried
              unbacktracked, a table that lists, for each state, the backtracks not yet tried
              s, a, the previous state and action, initially null

  if GOAL-TEST(s') then return stop
  if s' is a new state (not in untried) then untried[s'] $\leftarrow$ ACTIONS(s')
  if s is not null and s' != result[s, a] then
      result[s, a] $\leftarrow$ s'
      add s to front of unbacktracked[s']
  if untried[s'] is empty then
      if unbacktracked[s'] is empty then return stop
      else a $\leftarrow$ an action b such that result[s', b] = POP(unbacktracked[s'])
  else a $\leftarrow$ POP(untried[s'])
  s $\leftarrow$ s'
  return a
```

End of Session

Constraint satisfaction problem

Session Outcomes

Students will learn about Constraint Satisfaction Problems.

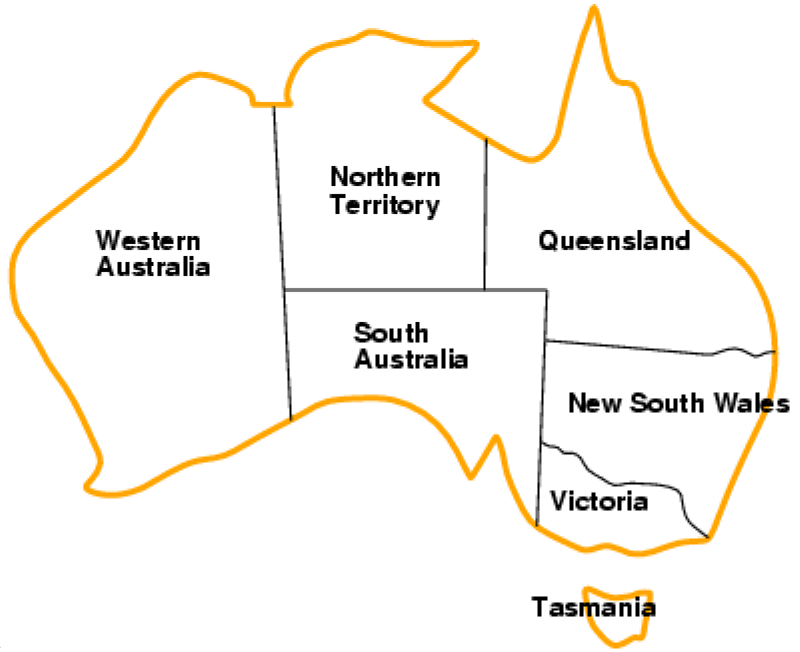
Constraint satisfaction problems (CSPs)

Standard search problem: state is a "black box“ – any data structure that supports successor function and goal test

- CSP:
  - state is defined by variables X_i with values from domain D_i
  - goal test is a set of constraints specifying allowable combinations of values for subsets of variables

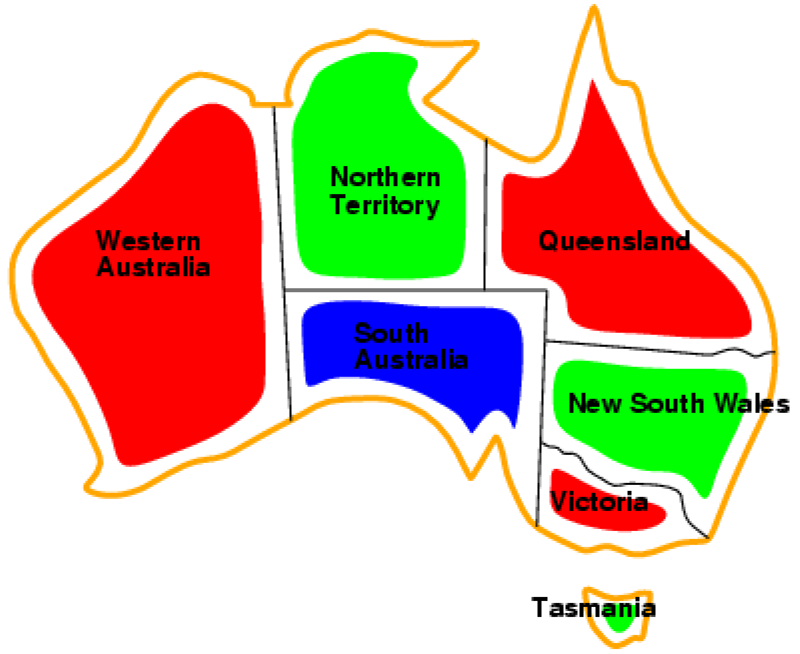
- Simple example of a formal representation language
- Allows useful general-purpose algorithms with more power than standard search algorithms

Example: Map-Coloring



- Variables *WA, NT, Q, NSW, V, SA, T*
- Domains D_i = {red,green,blue}
- Constraints: adjacent regions must have different colors
- e.g., WA ≠ NT, or (WA,NT) in {(red,green),(red,blue),(green,red), (green,blue),(blue,red),(blue,green)}

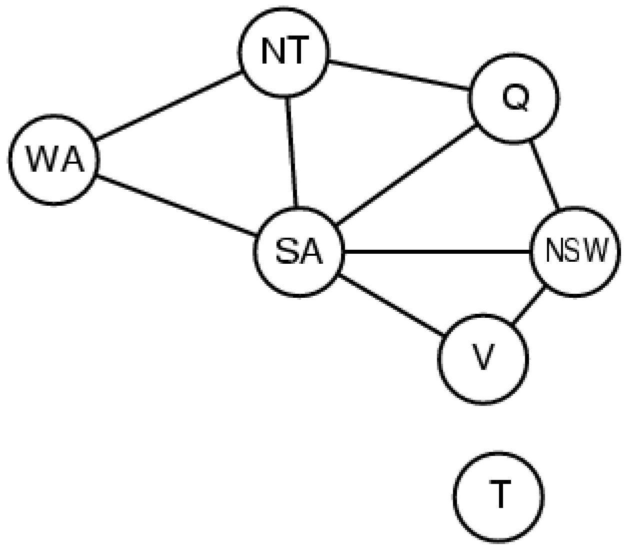
Example: Map-Coloring



- Solutions are complete and consistent assignments
- e.g., WA = red, NT = green, Q = red, NSW = green,V = red,SA = blue,T = green
-

Constraint graph

- Binary CSP: each constraint relates two variables
- Constraint graph: nodes are variables, arcs are constraints



Varieties of CSPs

- Discrete variables
  - finite domains:
    - *n* variables, domain size *d* ☾ $O(d^n)$ complete assignments
    - e.g., Boolean CSPs, incl. Boolean satisfiability (NP-complete)
  - infinite domains:
    - integers, strings, etc.
    - e.g., job scheduling, variables are start/end days for each job
    - need a constraint language, e.g., $StartJob_1 + 5 \leq StartJob_3$

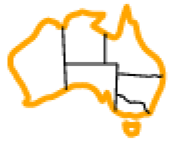
- Continuous variables
  - e.g., start/end times for Hubble Space Telescope observations
  - linear constraints solvable in polynomial time by LP

Varieties of constraints

- Unary constraints involve a single variable,
  - e.g., SA ≠ green
- Binary constraints involve pairs of variables,
  - e.g., SA ≠ WA
- Higher-order constraints involve 3 or more variables,
  - e.g., cryptarithmetic column constraints

Forward checking

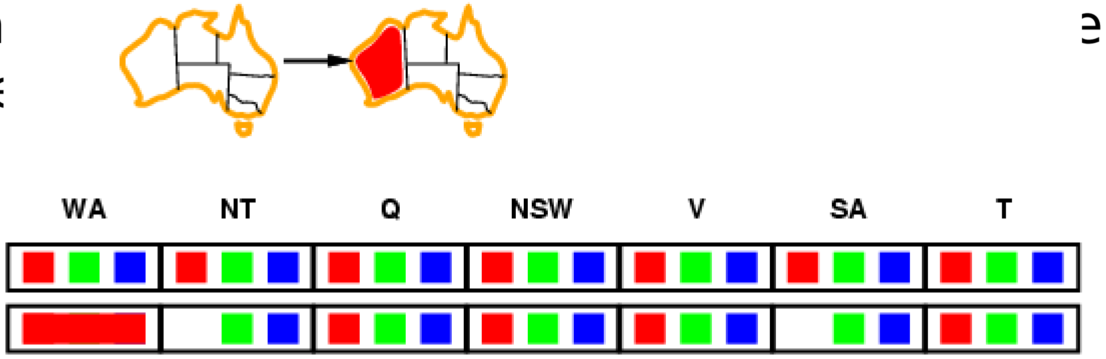
- Idea:
  - Keep track of remaining legal values for unassigned variables
  - Term egal valu
  -



WA	NT	Q	NSW	V	SA	T
🟥🟩🟦	🟥🟩🟦	🟥🟩🟦	🟥🟩🟦	🟥🟩🟦	🟥🟩🟦	🟥🟩🟦

Forward checking

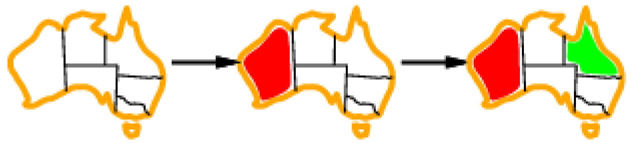
- Idea:
  - Keep track of remaining legal values for unassigned variables
  - Term egal valu
  -

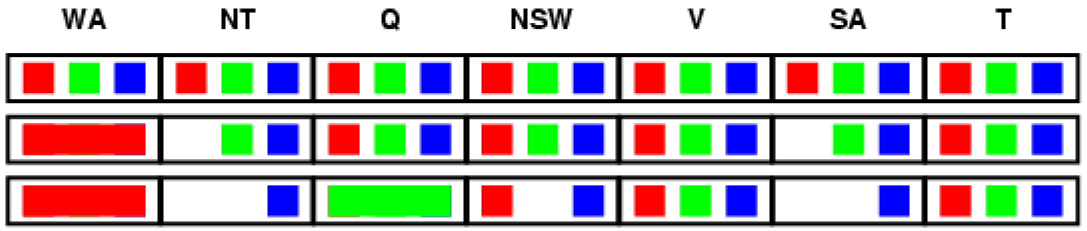


WA	NT	Q	NSW	V	SA	T
🟥🟩🟦	🟥🟩🟦	🟥🟩🟦	🟥🟩🟦	🟥🟩🟦	🟥🟩🟦	🟥🟩🟦
🟥	🟩🟦	🟥🟩🟦	🟥🟩🟦	🟥🟩🟦	🟩🟦	🟥🟩🟦

Forward checking

- Idea:
  - Keep track of remaining legal values for unassigned variables
  - Term egal value

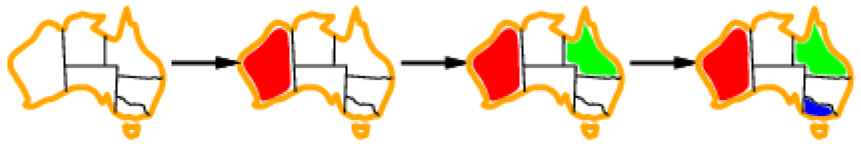


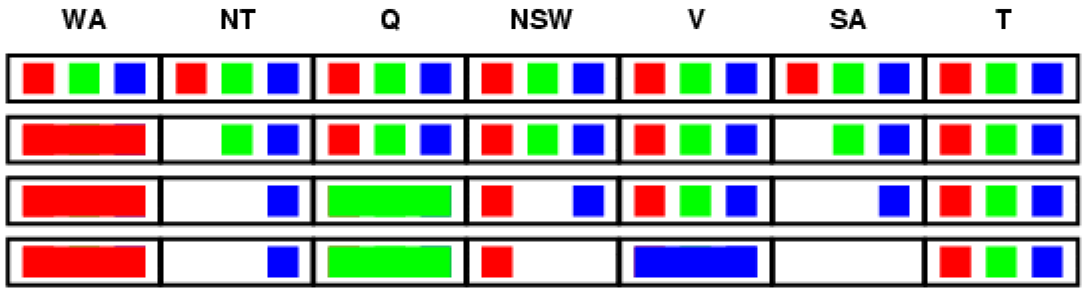


WA	NT	Q	NSW	V	SA	T
🟥🟩🟦	🟥🟩🟦	🟥🟩🟦	🟥🟩🟦	🟥🟩🟦	🟥🟩🟦	🟥🟩🟦
🟥	🟩🟦	🟥🟩🟦	🟥🟩🟦	🟥🟩🟦	🟩🟦	🟥🟩🟦
🟥	🟦	🟩	🟥🟦	🟥🟩🟦	🟦	🟥🟩🟦

Forward checking

- Idea:
  - Keep track of remaining legal values for unassigned variables
  - Term egal value
  -

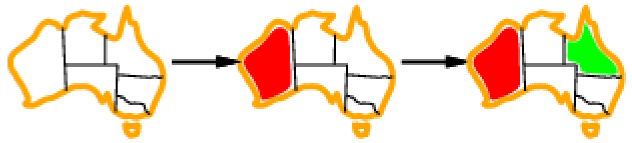


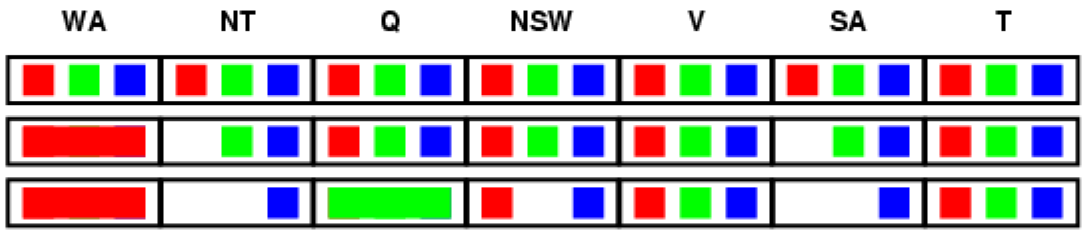


WA	NT	Q	NSW	V	SA	T
🟥🟩🟦	🟥🟩🟦	🟥🟩🟦	🟥🟩🟦	🟥🟩🟦	🟥🟩🟦	🟥🟩🟦
🟥	🟩🟦	🟥🟩🟦	🟥🟩🟦	🟥🟩🟦	🟩🟦	🟥🟩🟦
🟥	🟦	🟩	🟥🟦	🟥🟩🟦	🟦	🟥🟩🟦
🟥	🟦	🟩	🟥	🟦		🟥🟩🟦

Constraint propagation: Inferences in CSPs

Forward checking propagates information from assigned to unassigned variables, but doesn't provide early detection for all failures:

- 

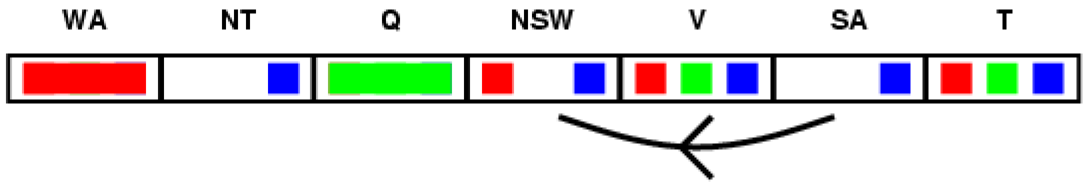


WA	NT	Q	NSW	V	SA	T
🟥🟩🟦	🟥🟩🟦	🟥🟩🟦	🟥🟩🟦	🟥🟩🟦	🟥🟩🟦	🟥🟩🟦
🟥	🟩🟦	🟥🟩🟦	🟥🟩🟦	🟥🟩🟦	🟩🟦	🟥🟩🟦
🟥	🟦	🟩	🟥🟦	🟥🟩🟦	🟦	🟥🟩🟦

- NT and SA cannot both be blue!
- Constraint propagation algorithms repeatedly enforce constraints locally…

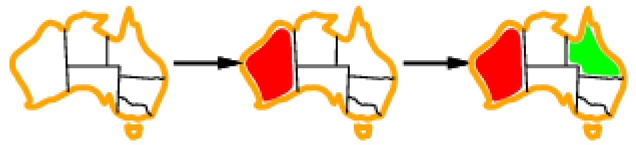
Arc consistency

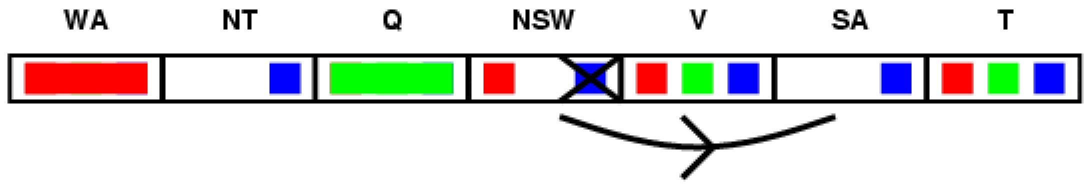
- Simplest form of propagation makes each arc consistent
- *X* ☾ *Y* is consistent iff
- for every value *x* of *X* there is some allowed *y*



Arc consistency

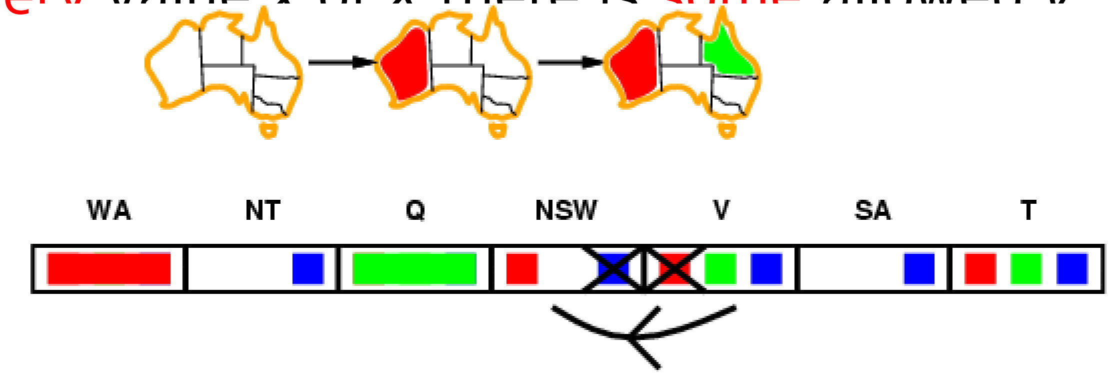
- Simplest form of propagation makes each arc consistent
- *X* ☾ *Y* is consistent iff
-
- for every value *x* of *X* there is some allowed *y*





Arc consistency

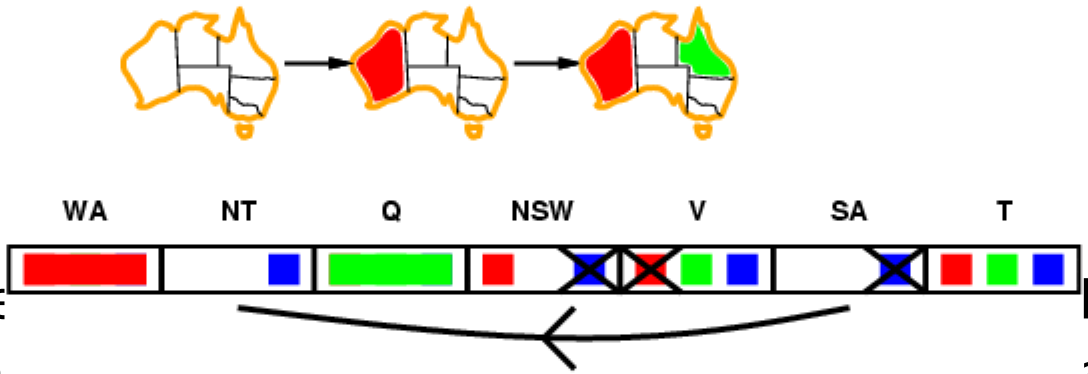
- Simplest form of propagation makes each arc consistent
- *X* ☾ *Y* is consistent iff
-
- for every value *x* of *X* there is some allowed *y*



- If *X* loses a value, neighbors of *X* need to be rechecked

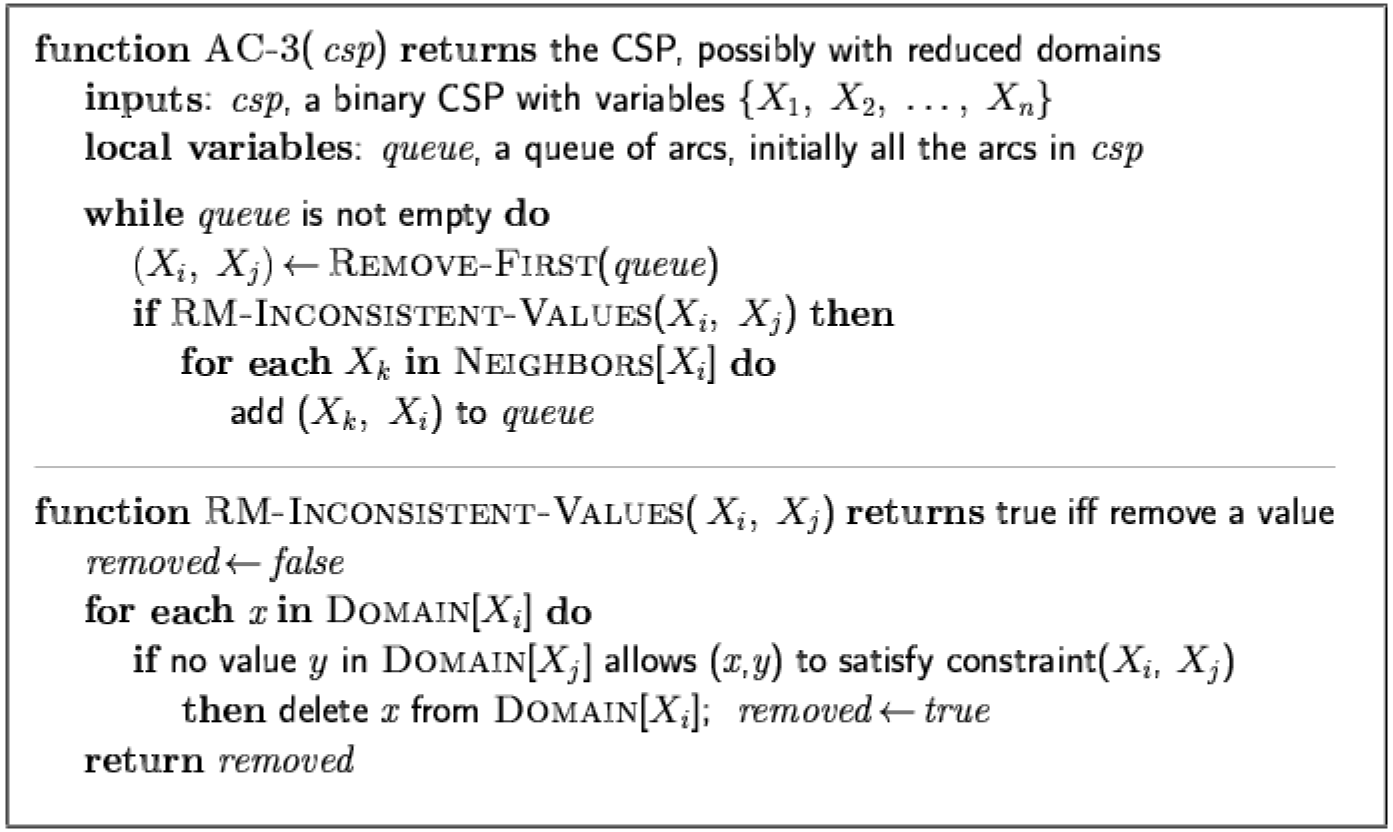
Arc consistency

- Simplest form of propagation makes each arc consistent
- *X* ☾ *Y* is consistent iff
-
- for every value *x* of *X* there is some allowed *y*



- If *X* lose hecked
- Arc consistency detects failure earlier than forward checking
- Can be run as a preprocessor or after each assignment
-

Arc consistency algorithm AC-3



```
function AC-3( $csp$) returns the CSP, possibly with reduced domains
    inputs: $csp$, a binary CSP with variables $\{X_1, X_2, \ldots, X_n\}$
    local variables: $queue$, a queue of arcs, initially all the arcs in $csp$

    while $queue$ is not empty do
        $(X_i, X_j) \leftarrow$ REMOVE-FIRST($queue$)
        if RM-INCONSISTENT-VALUES($X_i, X_j$) then
            for each $X_k$ in NEIGHBORS[$X_i$] do
                add $(X_k, X_i)$ to $queue$

function RM-INCONSISTENT-VALUES( $X_i, X_j$) returns true iff remove a value
    $removed \leftarrow false$
    for each $x$ in DOMAIN[$X_i$] do
        if no value $y$ in DOMAIN[$X_j$] allows $(x,y)$ to satisfy constraint($X_i, X_j$)
            then delete $x$ from DOMAIN[$X_i$];  $removed \leftarrow true$
    return $removed$
```

- Time complexity: O(#constraints |domain|^3)

Checking consistency of an arc is O(|domain|^2)

k-consistency

- A CSP is *k-consistent* if, for any set of k-1 variables, and for any consistent assignment to those variables, a consistent value can always be assigned to any kth variable
- 1-consistency is node consistency
- 2-consistency is arc consistency
- For binary constraint networks, 3-consistency is the same as *path consistency*
- Getting k-consistency requires time and space exponential in k
- *Strong k-consistency* means k'-consistency for all k' from 1 to k
    - Once strong k-consistency for k=#variables has been obtained, solution can be constructed trivially
- Tradeoff between propagation and branching
- Practitioners usually use 2-consistency and less commonly 3-consistency

End of Session

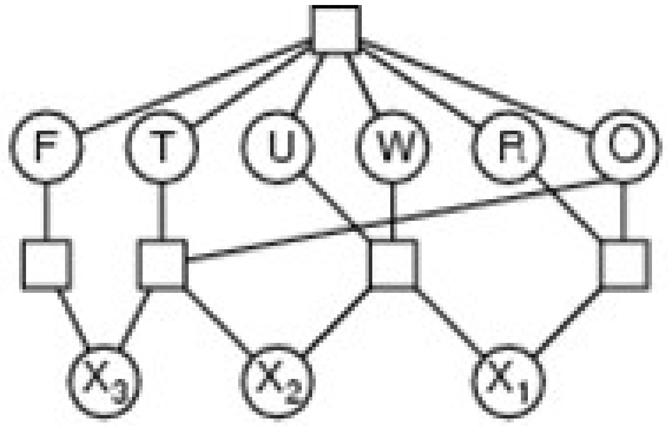
Back tracking for CSPs & Crypt arithmetic

Session Outcome

- **Students will learn backtracking and local search in Map Coloring.**
- **Students will learn to solve the CSP related problems.**

Example: Cryptarithmetic

$$\begin{array}{r} T\ W\ O \\ +\ T\ W\ O \\ \hline F\ O\ U\ R \end{array}$$



- Variables: *F T U W R O X_1 X_2 X_3*
- Domains: *{0,1,2,3,4,5,6,7,8,9}*
- Constraints: *Alldiff (F,T,U,W,R,O)*
-
  - $O + O = R + 10 \cdot X_1$
  -
  - $X_1 + W + W = U + 10 \cdot X_2$
  -
  - $X_2 + T + T = O + 10 \cdot X_3$
  - $X_3 = F$, $T \neq 0$, $F \neq 0$

Backtracking search

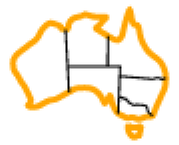
- Variable assignments are commutative, i.e.,
- [WA = red then NT = green] same as [NT = green then WA = red]

- => Only need to consider assignments to a single variable at each node

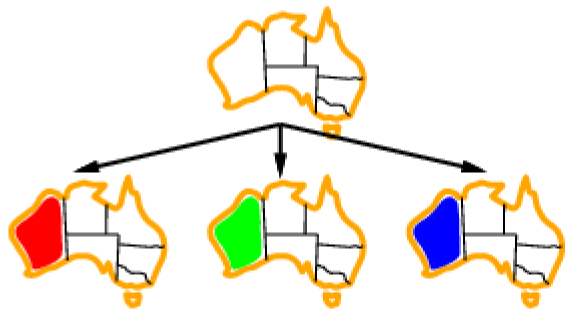
- Depth-first search for CSPs with single-variable assignments is called backtracking search

- Can solve *n*-queens for $n \approx 25$

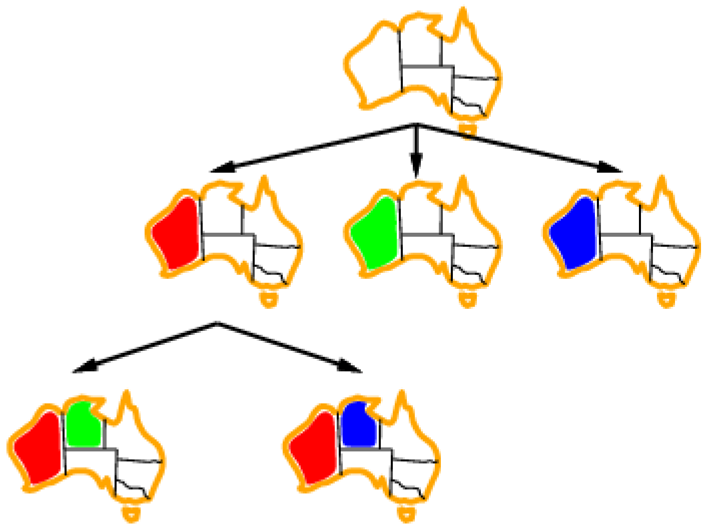
Backtracking example



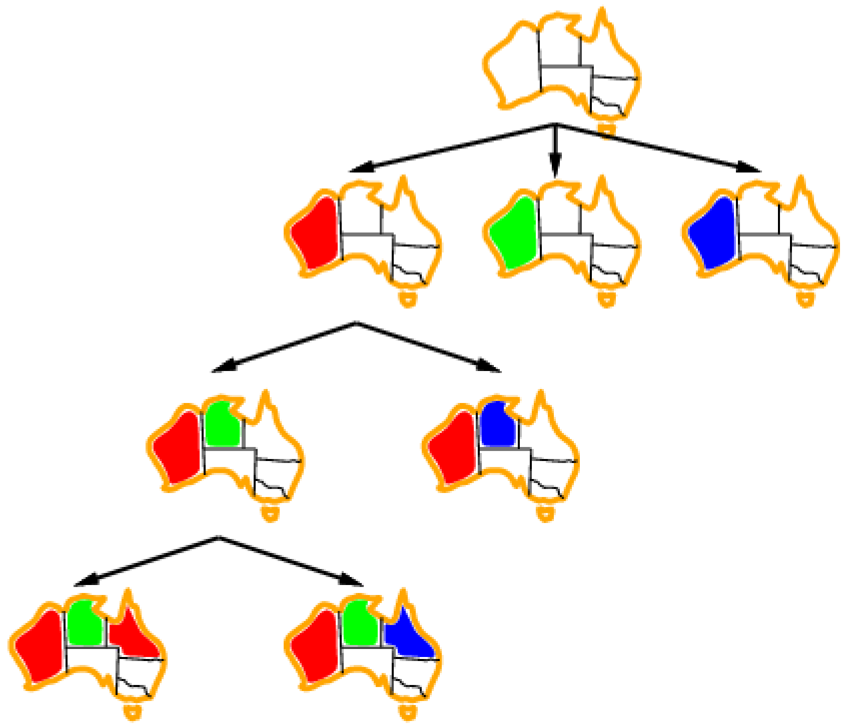
Backtracking example



Backtracking example



Backtracking example

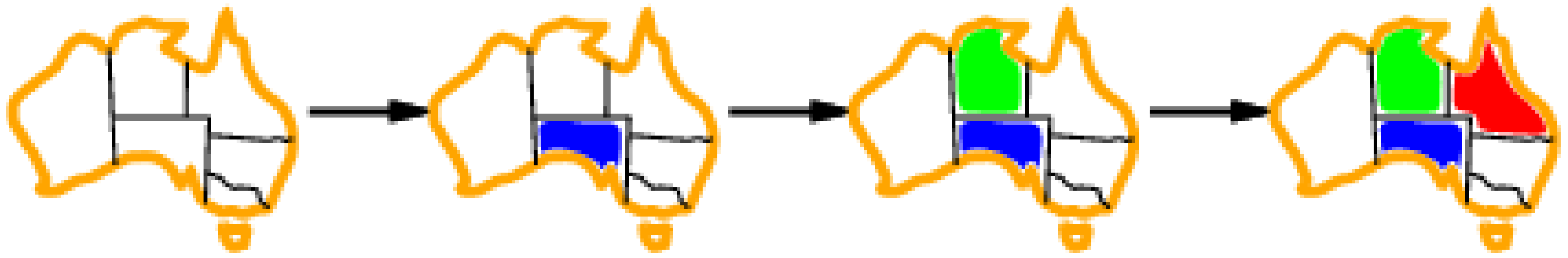


Improving backtracking efficiency

- General-purpose methods can give huge gains in speed:
  - Which variable should be assigned next?
  - In what order should its values be tried?
  - Can we detect inevitable failure early?

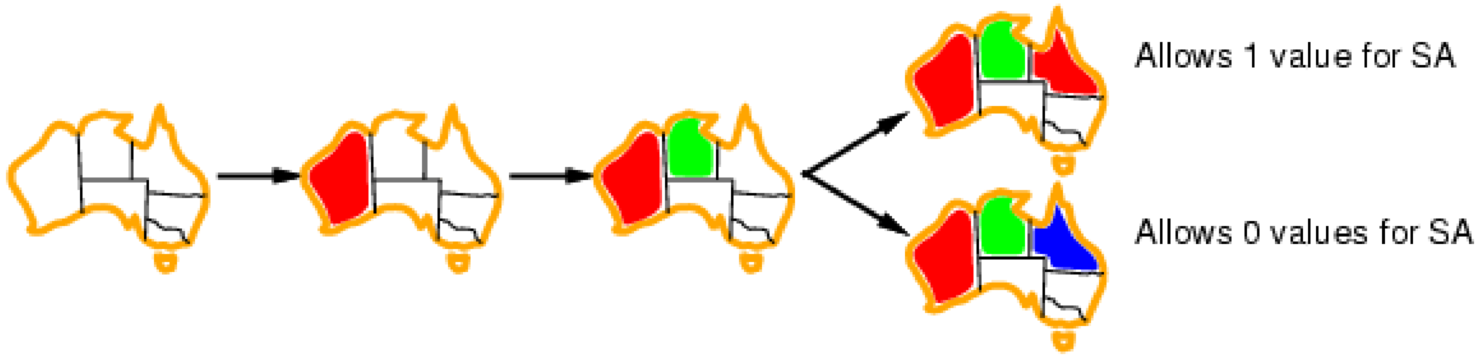
Most constraining variable

- A good idea is to use it as a tie-breaker among most constrained variables
- Most constraining variable:
- choose the variable with the most constraints on remaining variables
    -



Least constraining value

- Given a variable to assign, choose the least constraining value:
- the one that rules out the fewest values in the remaining variables
  -



- Combining these heuristics makes 1000 queens feasible