Chapter 3

Search techniques

Basanta Joshi, PhD

basanta@ioe.edu.np

Lecture notes can be downloaded from

www.basantajoshi.com.np

Informed Search

- uses domain specific information to improve the search pattern
  - Define a heuristic function, h(n), that estimates the "goodness" of a node n.
  - Specifically, h(n) = estimated cost (or distance) of minimal cost path from n to a goal state.
  - The heuristic function is an estimate, based on domain-specific information that is computable from the current state description, of how close we are to a goal

Informed search:

Use heuristics to guide the search

- Best first
- A*
- Heuristics
- Hill-climbing
- Simulated annealing

```
function TREE-SEARCH(problem, fringe) returns a solution, or failure
    fringe ← INSERT(MAKE-NODE(INITIAL-STATE[problem]), fringe)
    loop do
        if fringe is empty then return failure
        node ← REMOVE-FRONT(fringe)
        if GOAL-TEST[problem] applied to STATE(node) succeeds return node
        fringe ← INSERTALL(EXPAND(node, problem), fringe)
```

A strategy is defined by picking the order of node expansion

Best-first search

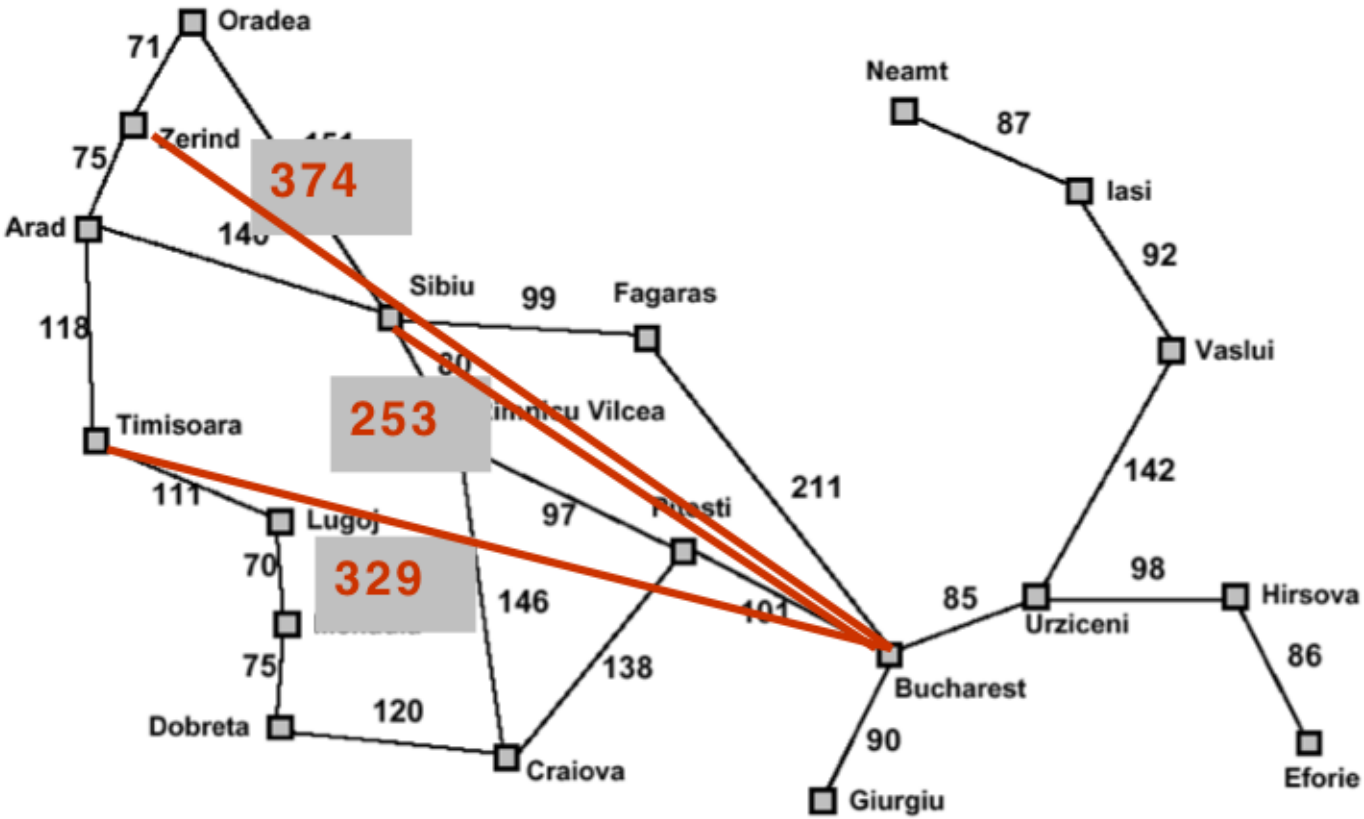
- Idea: use an evaluation function for each node
  - estimate of ***"desirability"***

Expand most desirable unexpanded node.

- Implementation:
  fringe is a queue sorted in decreasing order of desirability

- Special cases:
  greedy search
  A* search

Romania with step costs in km



Straight–line distance to Bucharest	
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	178
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	100
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

Greedy search

- Evaluation function **h**(**n**) (**h**euristic)

  = estimate of cost from **n** to closest goal

- For example:

  $h_{SLD}(n)$ = straight-line distance from **n** to Bucharest

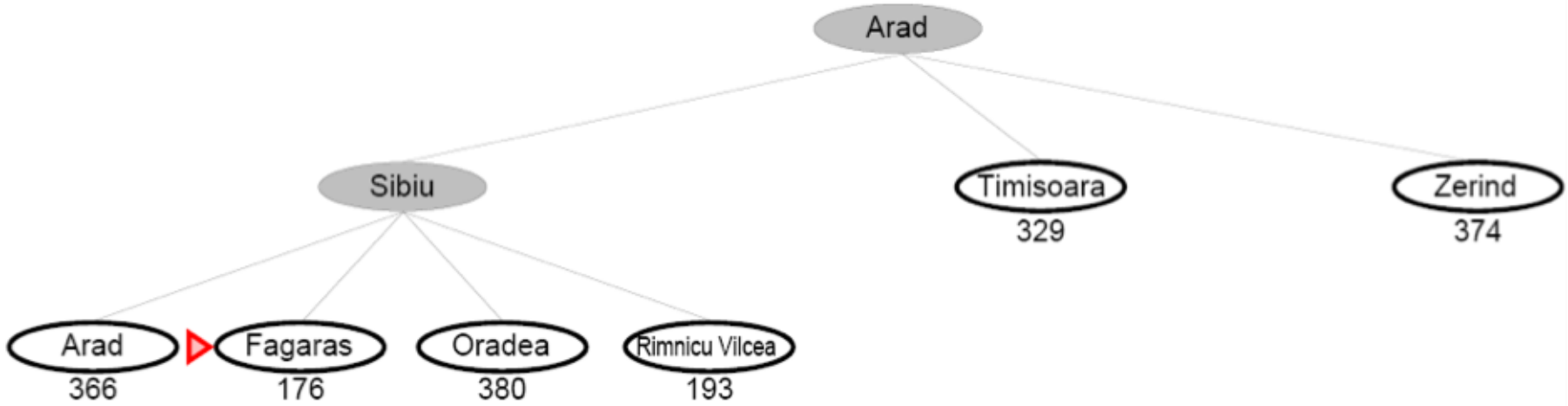
- Greedy search expands first the node that appears to be closest to the goal, according to **h**(**n**).

Greedy search example

Straight–line distance to Bucharest

Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	178
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	100
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

Greedy search example

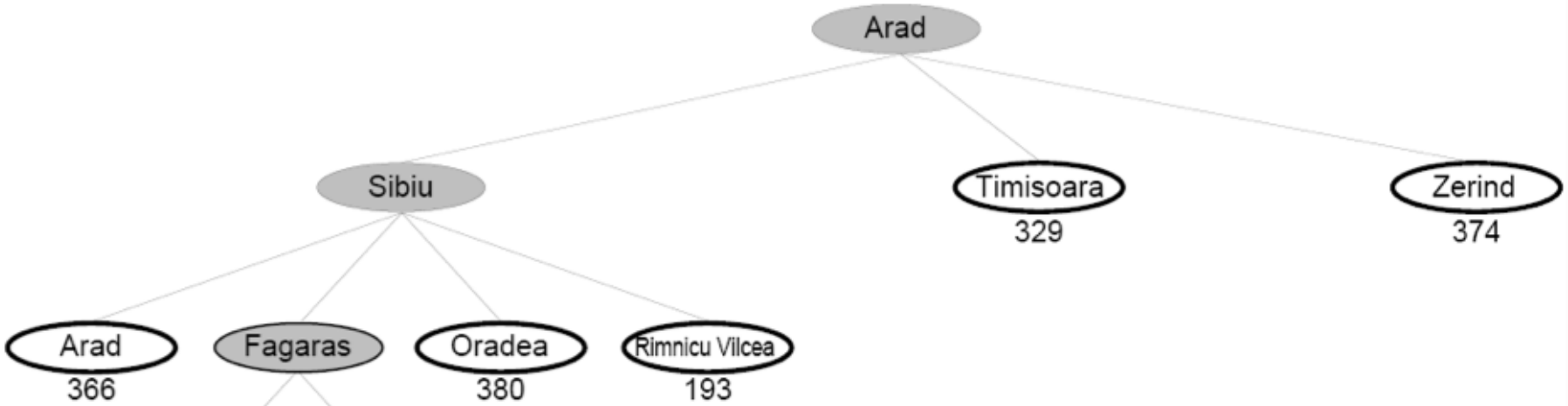


Straight–line distance to Bucharest	
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	178
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	100
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

Greedy search example

Straight–line distance to Bucharest	
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	178
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	100
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

Greedy search example



Straight–line distance to Bucharest

Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	178
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	100
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

Properties of Greedy Search

- Complete?

  No – can get stuck in loops
  e.g., Iasi > Neamt > Iasi > Neamt > .
  Complete in finite space with repeated-state checking.

- Time?

  $O(b^m)$ but a good heuristic can give dramatic improvement

- Space?

  $O(b^m)$ – keeps all nodes in memory

- Optimal?

  No.

A* search

- Idea: avoid expanding paths that are already expensive

  evaluation function: $\mathbf{f}(\mathbf{n}) = \mathbf{g}(\mathbf{n}) + \mathbf{h}(\mathbf{n})$

  where:

  $\mathbf{g}(\mathbf{n})$ – cost so far to reach n

  $\mathbf{h}(\mathbf{n})$ – estimated cost to goal from n

  $\mathbf{f}(\mathbf{n})$ – estimated total cost of path through n to goal

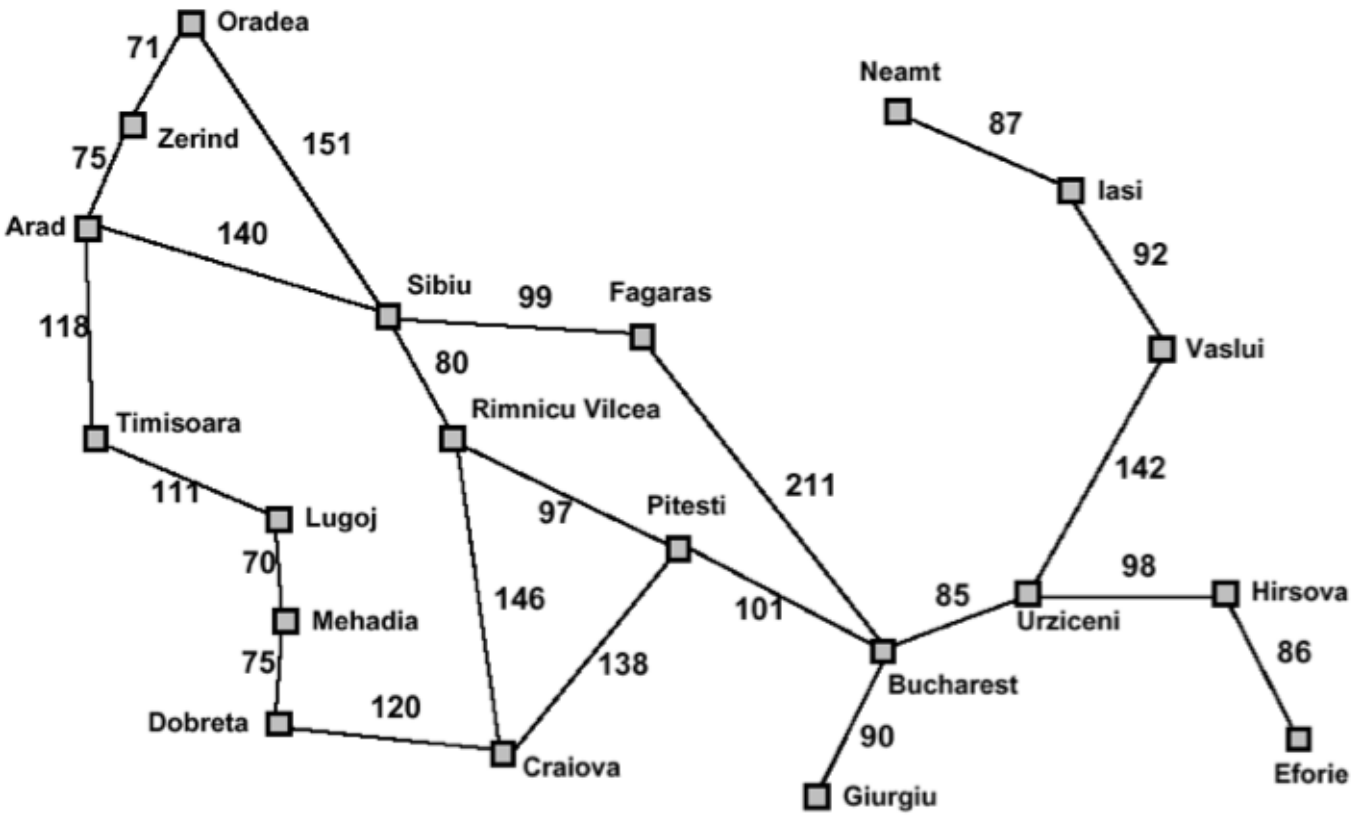
- A* search uses an *admissible* heuristic, that is,

  $h(n) \leq h^*(n)$ where $\mathbf{h}^*(\mathbf{n})$ is the *true* cost from n.

  For example: $\mathbf{h}_{SLD}(\mathbf{n})$ never overestimates actual road distance.

- Theorem: A* search is optimal

Romania with step costs in km



Straight–line distance to Bucharest

Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	178
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	98
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

A* Search Example

Arad

366=0+366

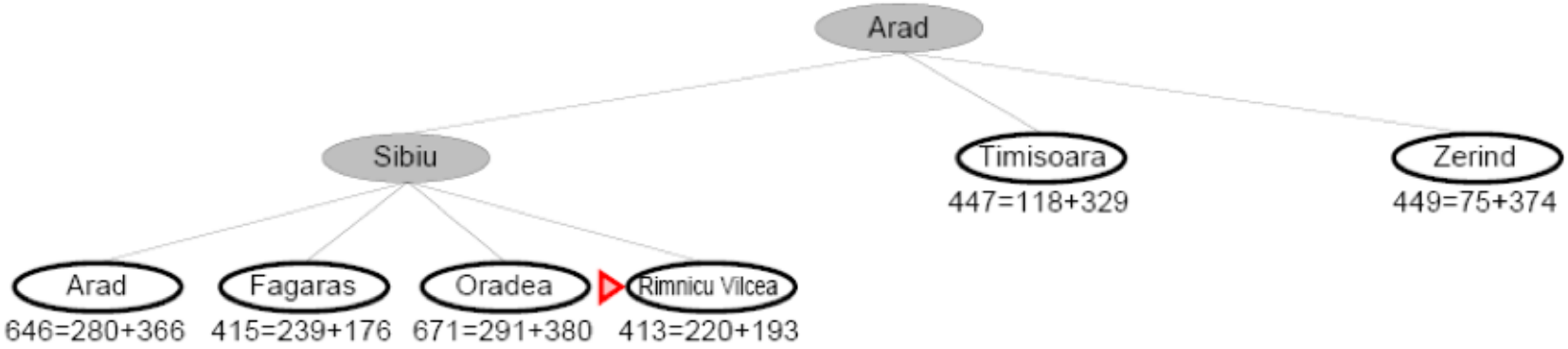
Straight-line distance to Bucharest

Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	178
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	100
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

A* Search Example

Straight-line distance to Bucharest	
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	178
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	100
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

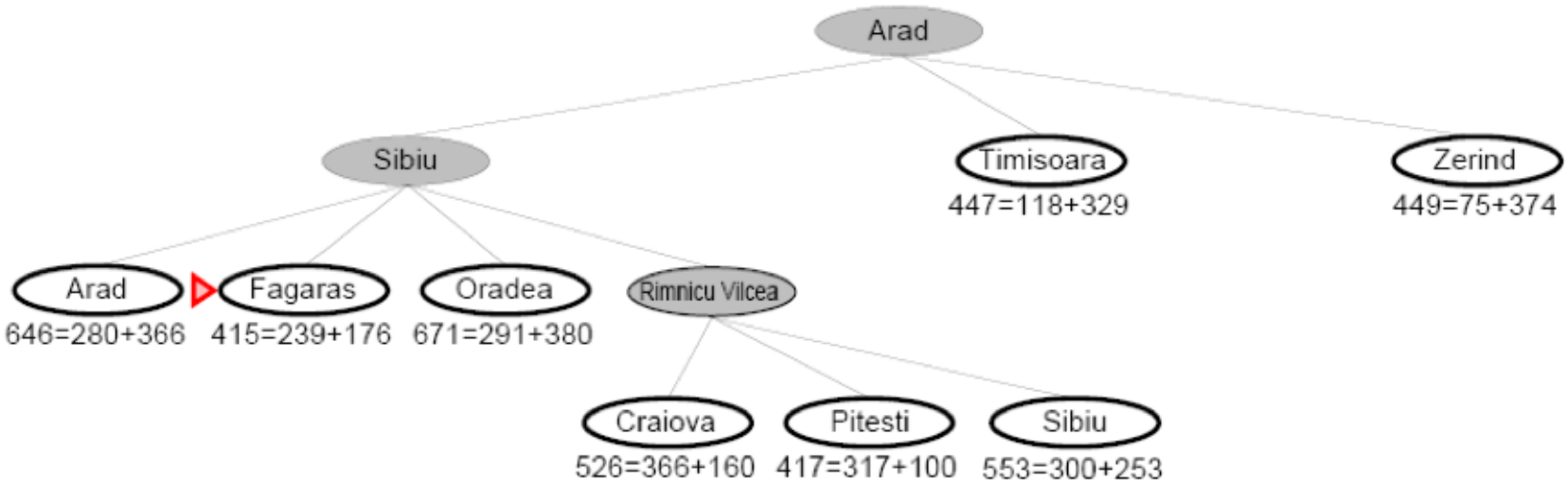
A* Search Example

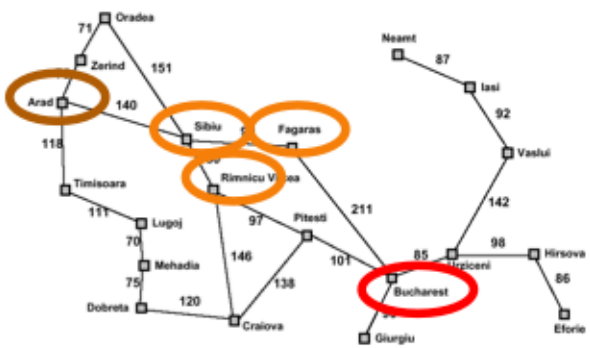


Straight-line distance to Bucharest

Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	178
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	100
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

A* Search Example

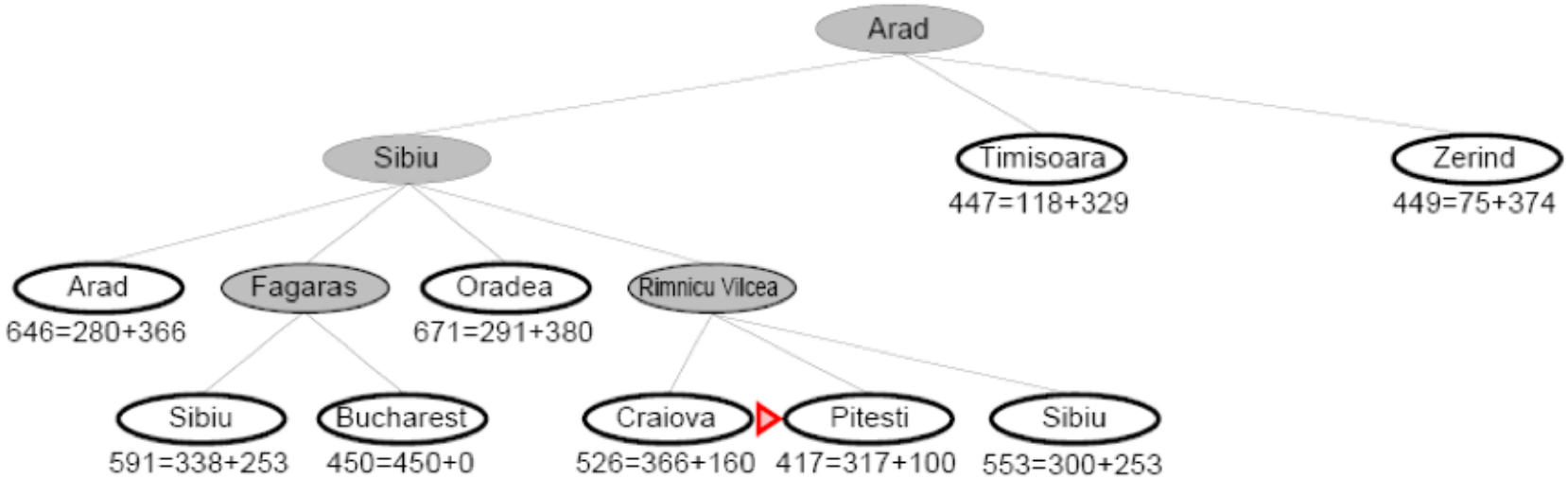




Straight-line distance to Bucharest

Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	178
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	100
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

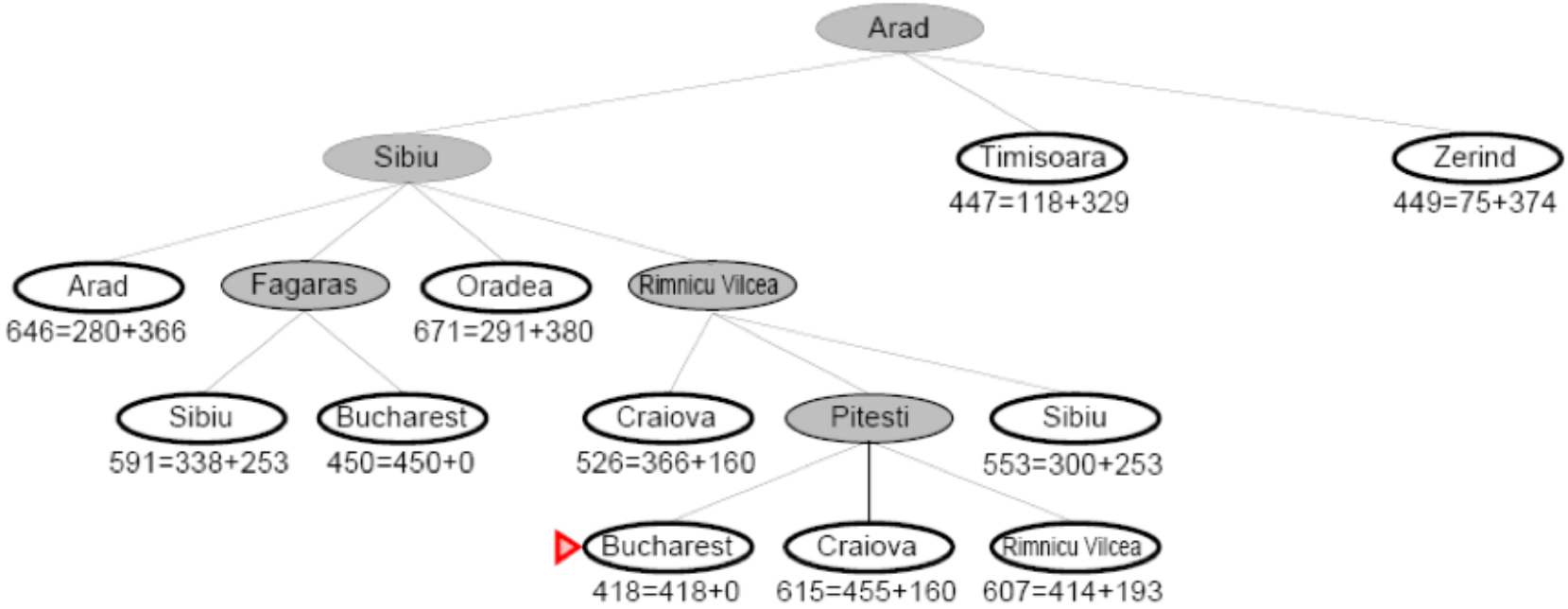
A* Search Example



Straight-line distance to Bucharest

Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	178
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	100
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

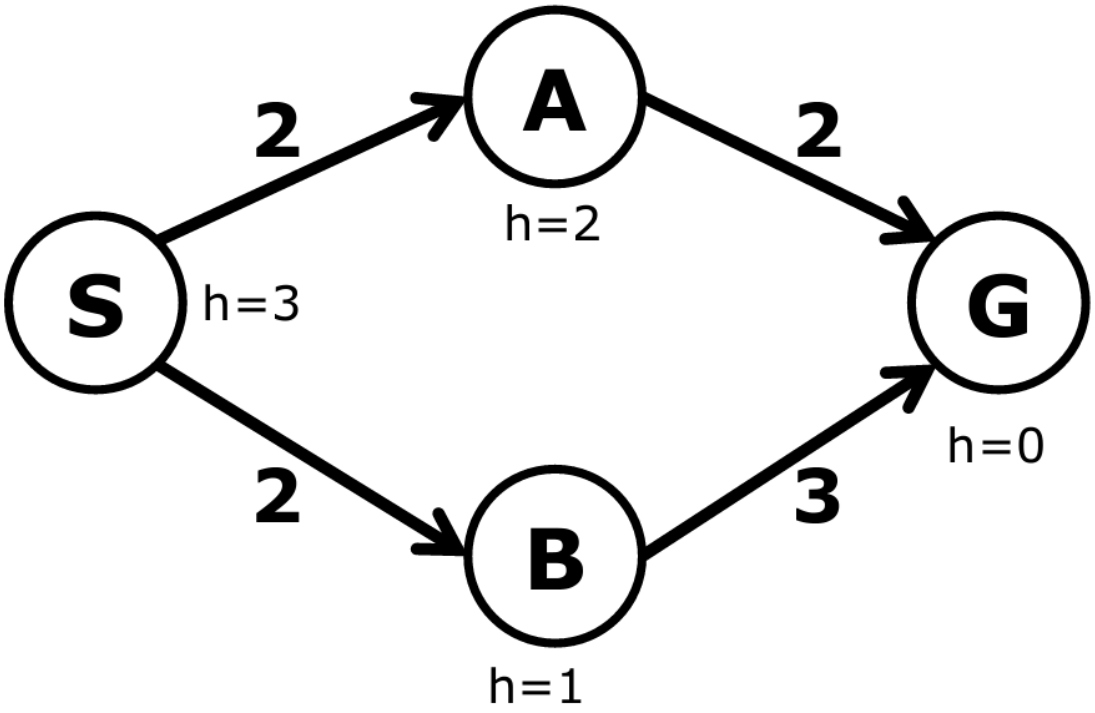
A* Search Example



Straight-line distance to Bucharest	
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	178
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	100
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

When should A* terminate?

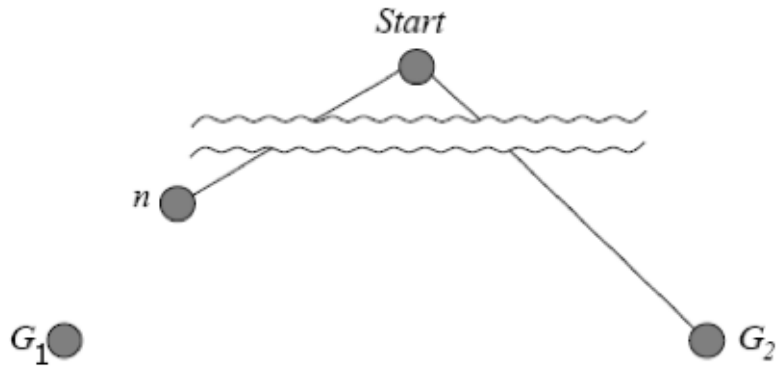
- Should we stop when we enqueue a goal?



- No: only stop when we dequeue a goal

Optimality of A* (standard proof)

Suppose some suboptimal goal G_2 has been generated and is in the queue.
Let **n** be an unexpanded node on a shortest path to an optimal goal G_1.



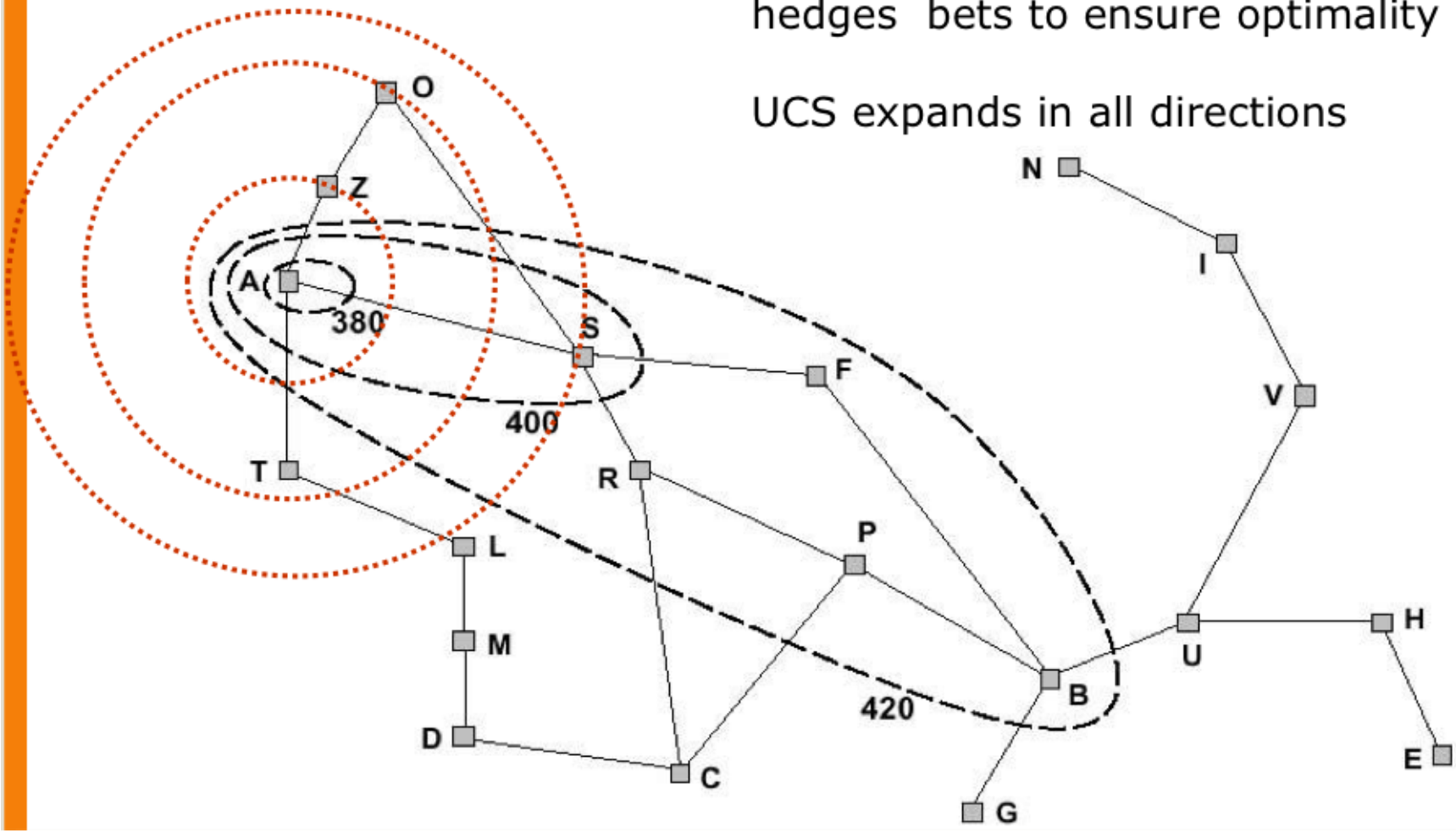
$$
\begin{aligned}
f(G_2) &= g(G_2) && \text{since } h(G_2) = 0 \\
&> g(G_1) && \text{since } G_2 \text{ is suboptimal} \\
&\geq f(n) && \text{since } h \text{ is admissible}
\end{aligned}
$$

Since $f(G_2) > f(n)$, A* will never select G_2 for expansion

UCS vs A* Contours

A* expands mainly towards goal, but hedges bets to ensure optimality

UCS expands in all directions



Properties of A*

- Complete?

  Yes, unless there are infinitely many nodes with $f \leq f(G)$
- Time?

  Exponential in [relative error in h x length of soln.]
- Space?

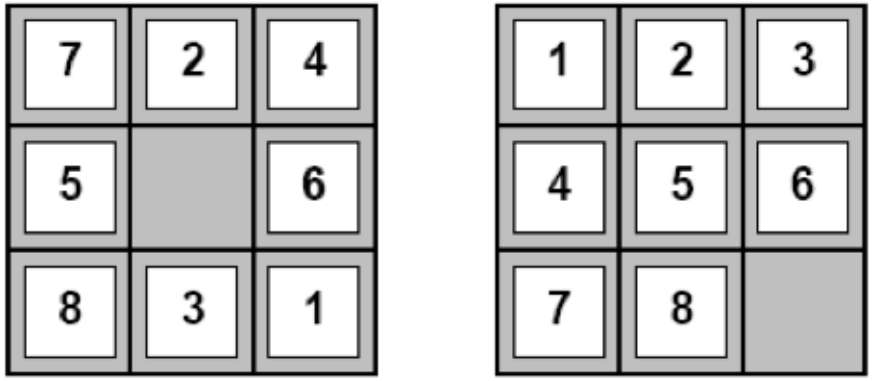
  Keeps all nodes in memory.
- Optimal?

  Yes – cannot expand f_{i+1} until f_i is finished.

Admissible heuristics

E.g., for the 8-puzzle:

$h_1(n)$ = number of misplaced tiles
$h_2(n)$ = total Manhattan distance
(i.e., no. of squares from desired location of each tile)



Start State Goal State

$h_1(S) = ??$ 6
$h_2(S) = ??$ $4+0+3+3+1+0+2+1 = 14$

Dominance

If $h_2(n) \geq h_1(n)$ for all n (both admissible)
then h_2 dominates h_1 and is better for search

Typical search costs:

$d = 14$ IDS $=$ 3,473,941 nodes
A$^*(h_1) =$ 539 nodes
A$^*(h_2) =$ 113 nodes
$d = 24$ IDS $\approx$ 54,000,000,000 nodes
A$^*(h_1) =$ 39,135 nodes
A$^*(h_2) =$ 1,641 nodes

Given any admissible heuristics h_a, h_b,

$$h(n) = \max(h_a(n), h_b(n))$$

is also admissible and dominates h_a, h_b

A* Applications

- Pathing / routing problems
- Resource planning problems
- Robot motion planning
- Language analysis
- Machine translation
- Speech recognition
- …

Graph Search

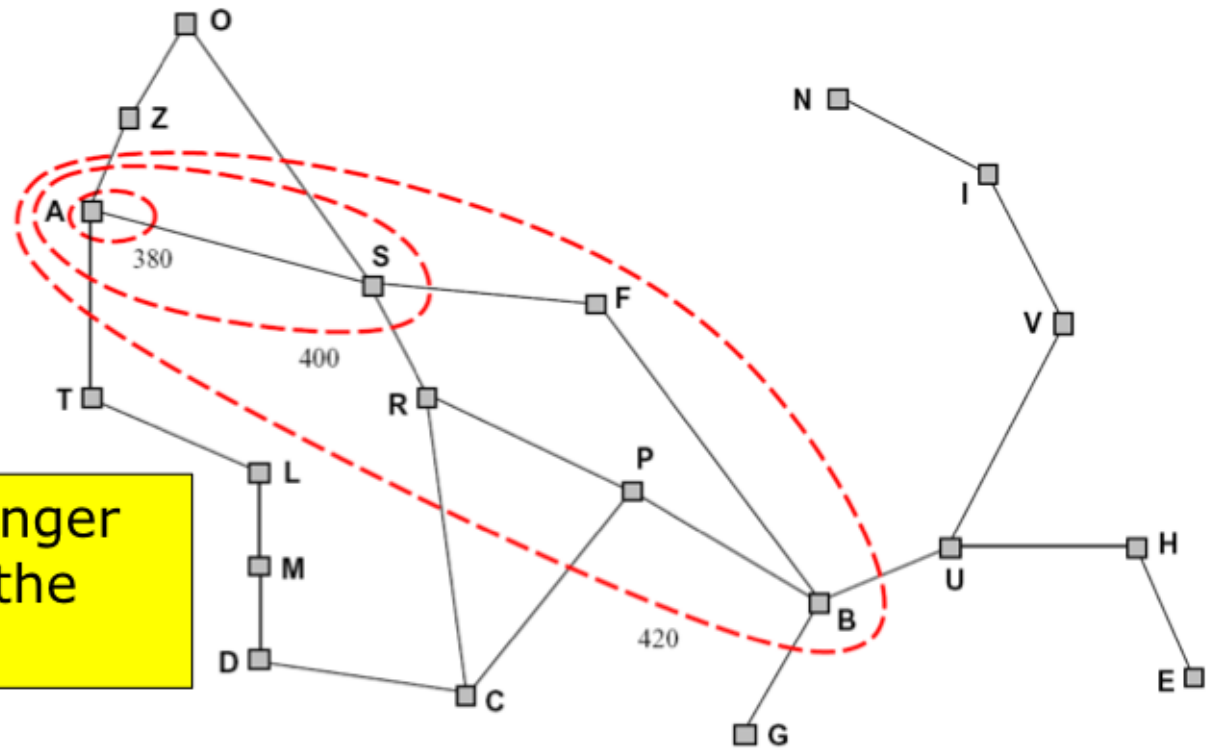
- Simple fix: never expand a state twice

```
function GRAPH-SEARCH( problem, fringe) returns a solution, or failure
    closed ← an empty set
    fringe ← INSERT(MAKE-NODE(INITIAL-STATE[problem]), fringe)
    loop do
        if fringe is empty then return failure
        node ← REMOVE-FRONT(fringe)
        if GOAL-TEST(problem, STATE[node]) then return node
        if STATE[node] is not in closed then
            add STATE[node] to closed
            fringe ← INSERTALL(EXPAND(node, problem), fringe)
    end
```

- Any effect on completeness and optimality?

Optimality of A* Graph Search

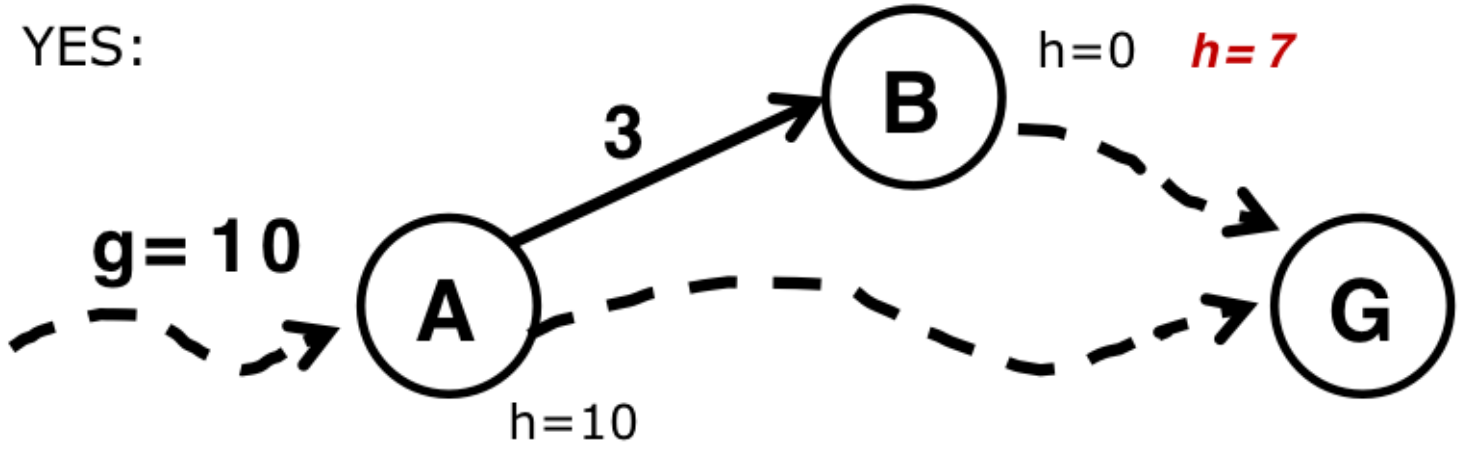
- Consider what A* does:
  - Expands nodes in increasing total f value (f-contours)
  - Proof idea: optimal goals have lower f value, so get expanded first



We're making a stronger assumption than in the last proof… what?

Consistency

- Wait, how do we know we expand in increasing f value?
- Couldn't we pop some node *n*, and find its child *n′ to* have lower *f* value?
- YES:



- What can we require to prevent these inversions?

$$c(n,a,n') \geq h(n) - h(n')$$

- Consistency:
  Real cost must always exceed reduction in heuristic

Optimality

- Tree search:
  - A* optimal if heuristic is admissible (and nonnegative)
  - UCS is a special case (h = 0)
- Graph search:
  - A* optimal if heuristic is consistent
  - UCS optimal (h = 0 is consistent)
- In general, natural admissible heuristics tend to be consistent

Summary: A*

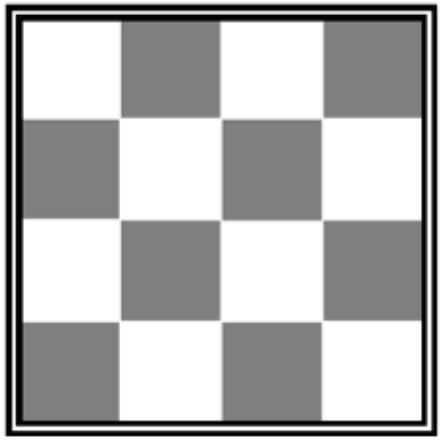
- Heuristic functions estimate costs of shortest paths
- Good heuristics can dramatically reduce search cost
- Greedy best-first search expands lowest **h**
  - incomplete and not always optimal
- A* search expands lowest **g + h**
  - complete and optimal
  - also optimally efficient (up to tie-breaks, for forward search)

Limited Memory Options

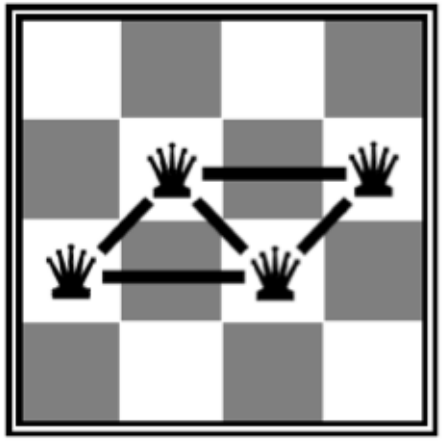
- Bottleneck: not enough memory to store entire fringe
- Hill-Climbing Search:
  - Only "best" node kept around, no fringe!
  - Usually prioritize successor choice by *h* (greedy hill climbing)
  - Compare to greedy backtracking, which still has fringe

Types of Problems

- Planning problems:
  - We want a path to a solution
  - Usually want an optimal path
  - *Incremental formulations*



- Identification problems:
  - We actually just want to know what the goal is
  - Usually want an optimal goal
  - *Complete-state formulations*
  - Iterative improvement algorithms



Iterative improvement

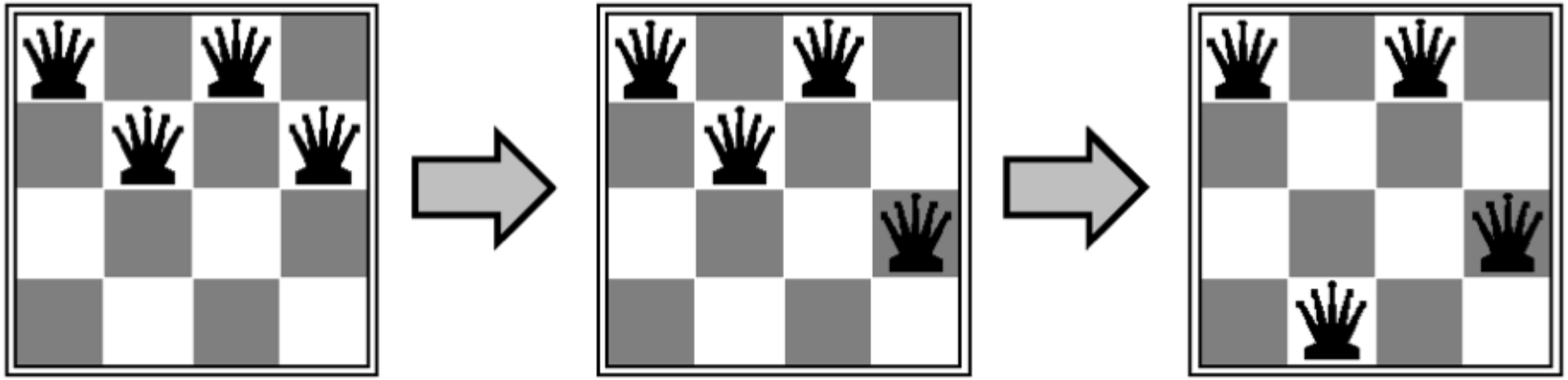
- In many optimization problems, path is irrelevant; the goal state itself is the solution.

- Then, state space = space of "complete" configurations.
  Algorithm goal:
  - find optimal configuration (e.g., TSP), or,
  - find configuration satisfying constraints (e.g., n-queens)

- In such cases, can use iterative improvement algorithms: keep a single "current" state, and try to improve it.

Iterative improvement example: n-queens

- Goal: Put **n** chess-game queens on an **n** x **n** board, with no two queens on the same row, column, or diagonal.
  → Here, goal state is initially unknown but is specified by constraints that it must satisfy.
- Move a queen to reduce number of conflicts



- Almost always solves **n**-queens problems almost instantaneously for very large **n**, e.g., **n** =1 million

Hill Climbing

- Simple, general idea:
  - Start wherever
  - Always choose the best neighbor
  - If no neighbors have better scores than current, quit
- Why can this be a terrible idea?
  - Complete?
  - Optimal?
- What's good about it?

Hill climbing (or gradient ascent/ descent)

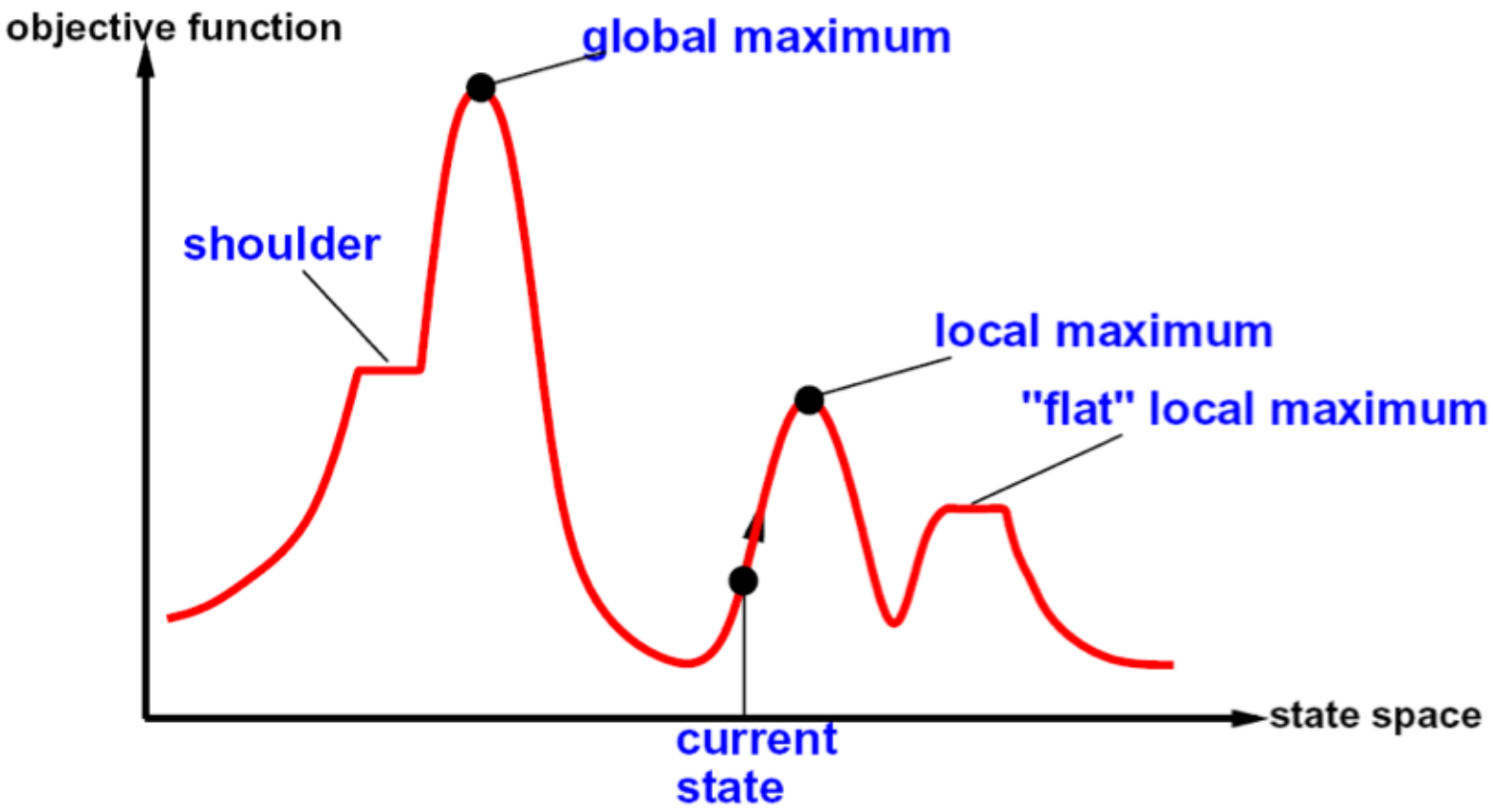
- Iteratively maximize "value" of current state, by replacing it by successor state that has highest value, as long as possible.

"Like climbing Everest in thick fog with amnesia"

```
function HILL-CLIMBING(problem) returns a state that is a local maximum
   inputs: problem, a problem
   local variables: current, a node
                    neighbor, a node

   current ← MAKE-NODE(INITIAL-STATE[problem])
   loop do
      neighbor ← a highest-valued successor of current
      if VALUE[neighbor] ≤ VALUE[current] then return STATE[current]
      current ← neighbor
   end
```

Hill Climbing diagram



Random restarts – overcomes local maxima – trivially complete
Random sideways moves – escape shoulders, loop when flat

Summary

- Best-first search = general search, where the minimum-cost nodes (according to some measure) are expanded first.

- Greedy search = best-first with the estimated cost to reach the goal as a heuristic measure.
  - Generally faster than uninformed search
  - not optimal
  - not complete.

- A* search = best-first with measure = path cost so far + estimated path cost to goal.
  - combines advantages of uniform-cost and greedy searches
  - complete, optimal and optimally efficient
  - space complexity still exponential

Summary

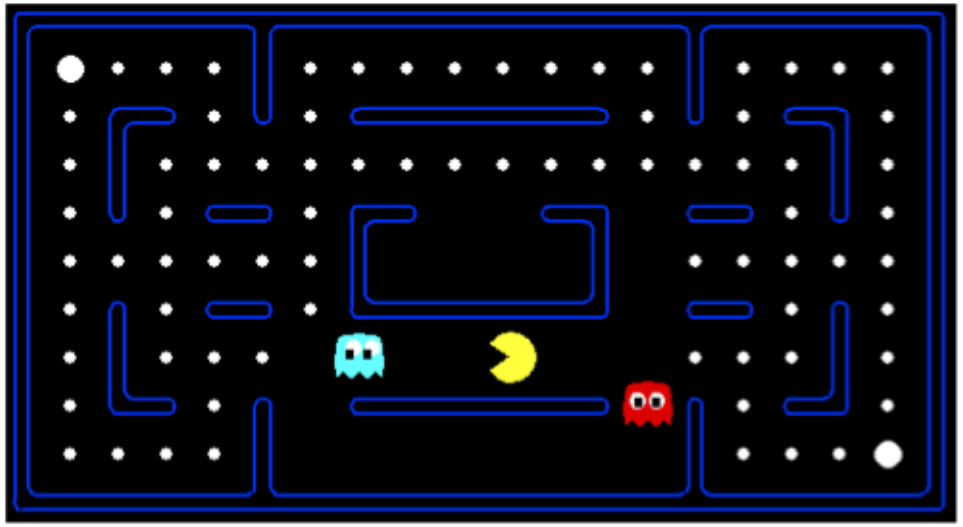
- Time complexity of heuristic algorithms depend on quality of heuristic function. Good heuristics can sometimes be constructed by examining the problem definition or by generalizing from experience with the problem class.

- Iterative improvement algorithms keep only a single state in memory.

Adversarial Search

Game playing

- Perfect play
  - The minimax algorithm
  - alpha-beta pruning
- Resource limitations
- Elements of chance
- Imperfect information



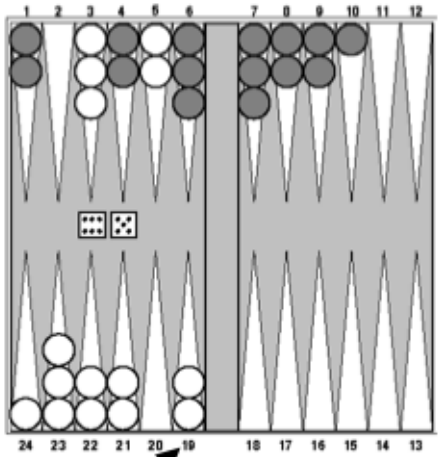
Game Playing State-of-the-Art

- **Checkers:** Chinook ended 40-year-reign of human world champion Marion Tinsley in 1994. Used an endgame database defining perfect play for all positions involving 8 or fewer pieces on the board, a total of 443,748,401,247 positions. Checkers is now solved!
- **Chess:** Deep Blue defeated human world champion Gary Kasparov in a six-game match in 1997. Deep Blue examined 200 million positions per second, used very sophisticated evaluation and undisclosed methods for extending some lines of search up to 40 ply. Current programs are even better, if less historic.
- **Othello:** Human champions refuse to compete against computers, which are too good.
- **Go:** Human champions are beginning to be challenged by machines, though the best humans still beat the best machines. In go, $b > 300$, so most programs use pattern knowledge bases to suggest plausible moves, along with aggressive pruning.
- **Pacman:** unknown

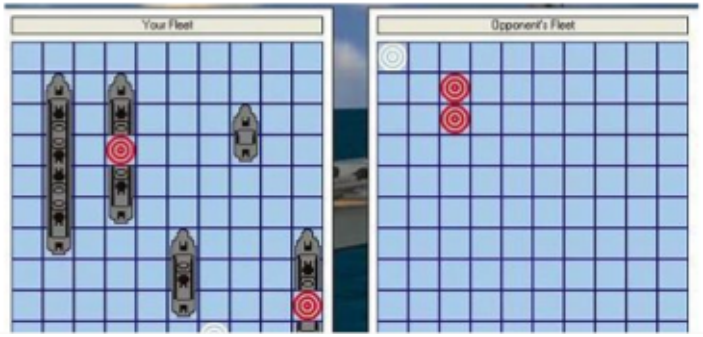
What kind of games?

- **Abstraction**: To describe a game we must capture every relevant aspect of the game. Such as:
  - Chess
  - Tic-tac-toe
  - …
- **Accessible environments:** Such games are characterized by perfect information
- **Search:** game-playing then consists of a search through possible game positions
- **Unpredictable opponent:** introduces **uncertainty** thus game-playing must deal with **contingency problems**

Type of games



	deterministic	chance
perfect information	chess, checkers, go, othello	backgammon monopoly
imperfect information	battleships, blind tictactoe	bridge, poker, scrabble nuclear war



Game Playing

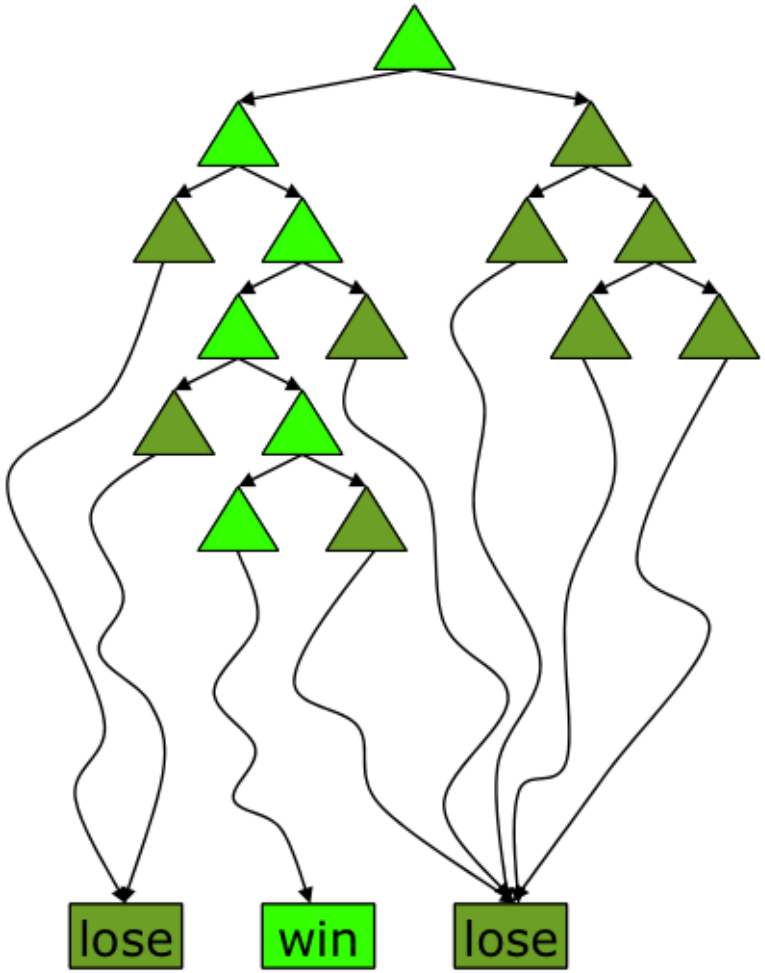
- Many different kinds of games!
- Axes:
  - Deterministic or stochastic?
  - One, two, or more players?
  - Perfect information (can you see the state)?
  - Turn taking or simultaneous action?
- Want algorithms for calculating a strategy (policy) which recommends a move in each state

Deterministic Games

- Many possible formalizations, one is:
  - States: S (start at s_0)
  - Players: P={1...N} (usually take turns)
  - Actions: A (may depend on player / state)
  - Transition Function: SxA $\rightarrow$ S
  - Terminal Test: S $\rightarrow$ {t,f}
  - Terminal Utilities: SxP $\rightarrow$ R
- Solution for a player is a policy: S $\rightarrow$ A

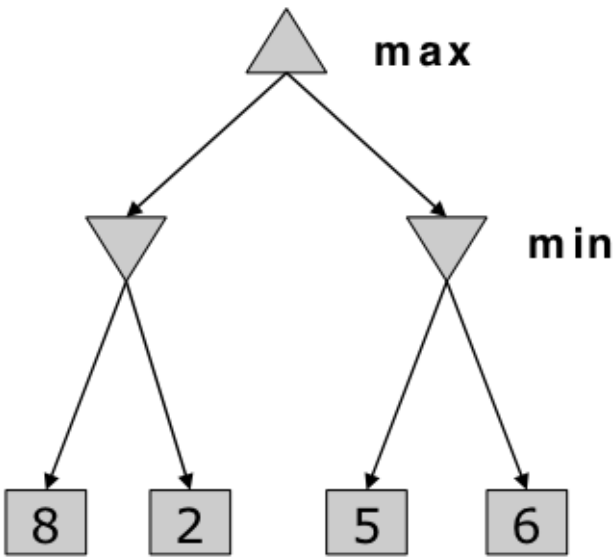
Deterministic Single-Player?

- Deterministic, single player, perfect information:
  - Know the rules
  - Know what actions do
  - Know when you win
  - E.g. Freecell, 8-Puzzle, Rubik's cube
- … it's just search!
- Slight reinterpretation:
  - Each node stores a value: the best outcome it can reach
  - This is the maximal outcome of its children (the max value)
  - Note that we don't have path sums as before (utilities at end)
- After search, can pick move that leads to best node



Deterministic Two-Player

- E.g. tic-tac-toe, chess, checkers
- Zero-sum games
  - One player maximizes result
  - The other minimizes result
- Minimax search
  - A state-space search tree
  - Players alternate
  - Each layer, or ply, consists of a round of moves*
  - Choose move to position with highest minimax value = best achievable utility against best play



** Slightly different from the book definition*

Games vs. search problems

- "Unpredictable" opponent ➔ solution is a strategy specifying a move for every possible opponent reply
- Time limits ➔ unlikely to find goal, must approximate
- Plan of attack:
  - Computer considers possible lines of play (Babbage, 1846)
  - Algorithm for perfect play (Zermelo, 1912; Von Neumann, 1944)
  - Finite horizon, approximate evaluation (Zuse, 1945; Wiener, 1948; Shannon, 1950)
  - First chess program (Turing, 1951)
  - Machine learning to improve evaluation accuracy (Samuel, 1952-57)
  - Pruning to allow deeper search (McCarthy, 1956)

Searching for the next move

- **Complexity:** many games have a huge search space
  - **Chess:** $b = 35, m=100 \Rightarrow nodes = 35^{100}$
    if each node takes about 1 ns to explore
    then each move will take about ***10^{50}* millennia**
    to calculate.

- **Resource (e.g., time, memory) limit:** optimal solution not feasible/possible, thus must approximate

1. **Pruning:** makes the search more efficient by discarding portions of the search tree that cannot improve quality result.

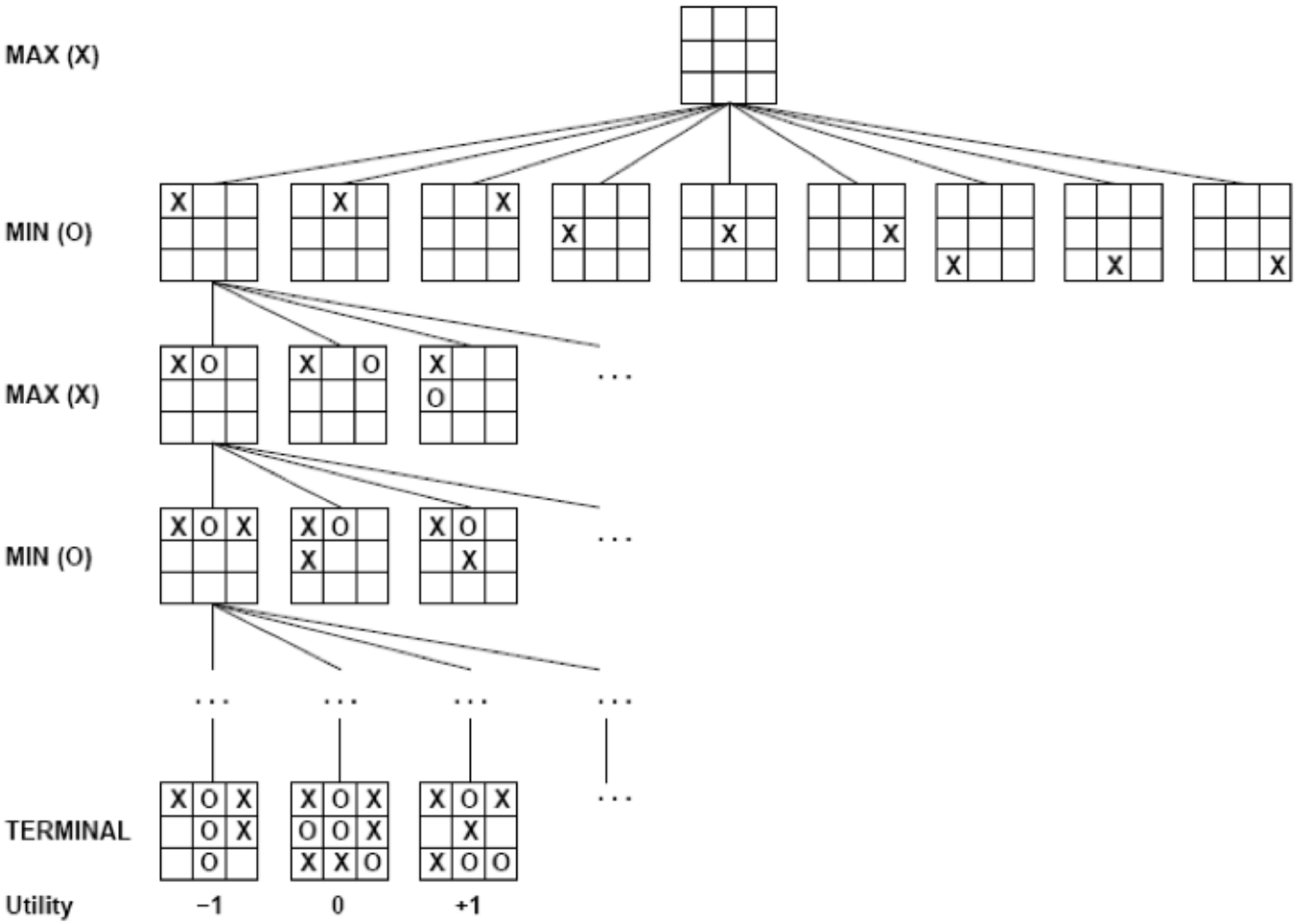
2. **Evaluation functions:** heuristics to evaluate utility of a state without exhaustive search.

Two-player games

- A game formulated as a search problem:

◦ Initial state:	board position and turn
◦ Operators:	definition of legal moves
◦ Terminal state:	conditions for when game is over
◦ Utility function:	a numeric value that describes the outcome of the game. E.g., -1, 0, 1 for loss, draw, win. (AKA **payoff function**)

Example: Tic-Tac-Toe

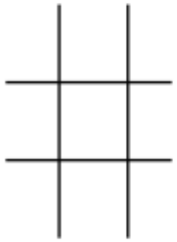


The minimax algorithm

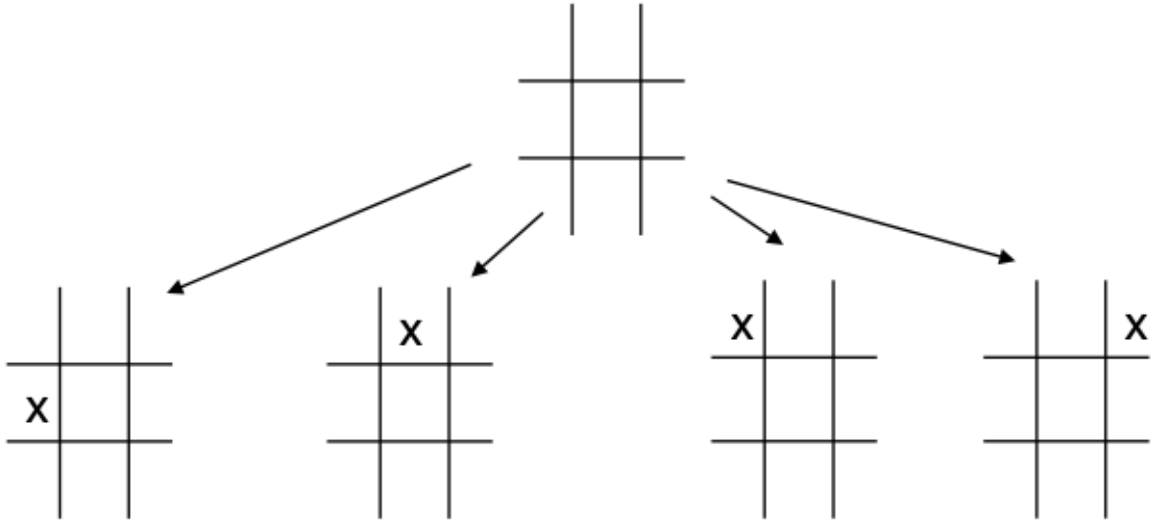
- Perfect play for deterministic environments with perfect information
- **Basic idea:** choose move with highest minimax value
  = best achievable payoff against best play
- **Algorithm:**
  1. Generate game tree completely
  2. Determine utility of each terminal state
  3. Propagate the utility values upward in the three by applying MIN and MAX operators on the nodes in the current level
  4. At the root node use <u>minimax decision</u> to select the move with the max (of the min) utility value

- Steps 2 and 3 in the algorithm assume that the opponent will play perfectly.

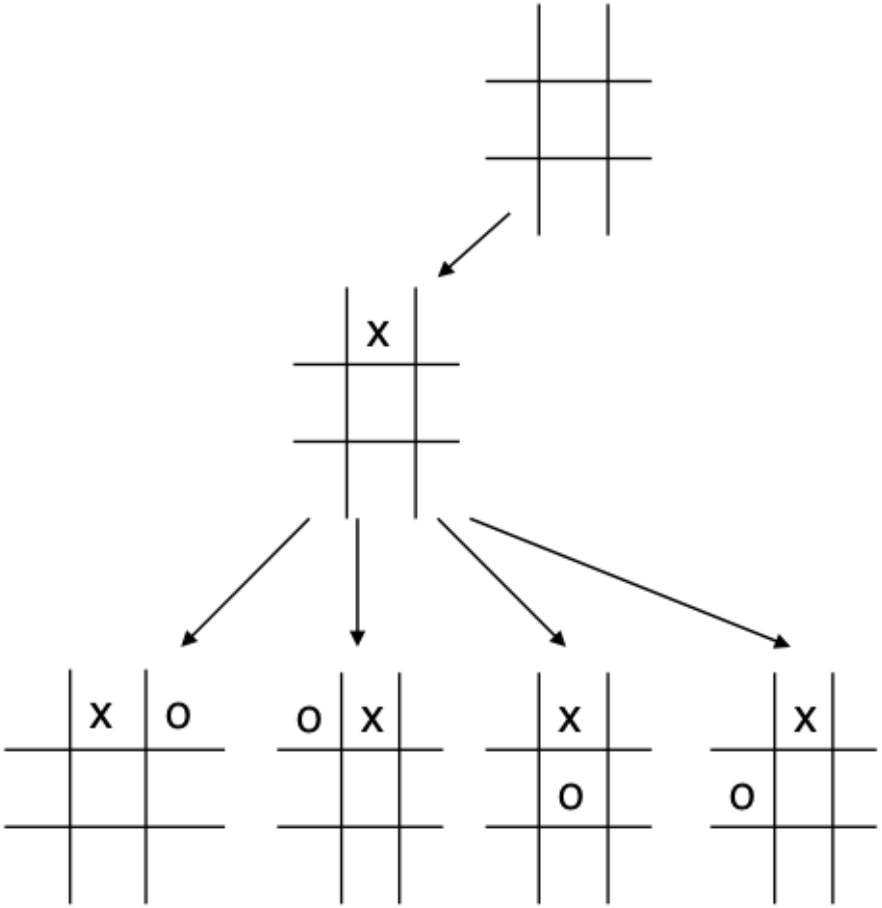
Generate Game Tree



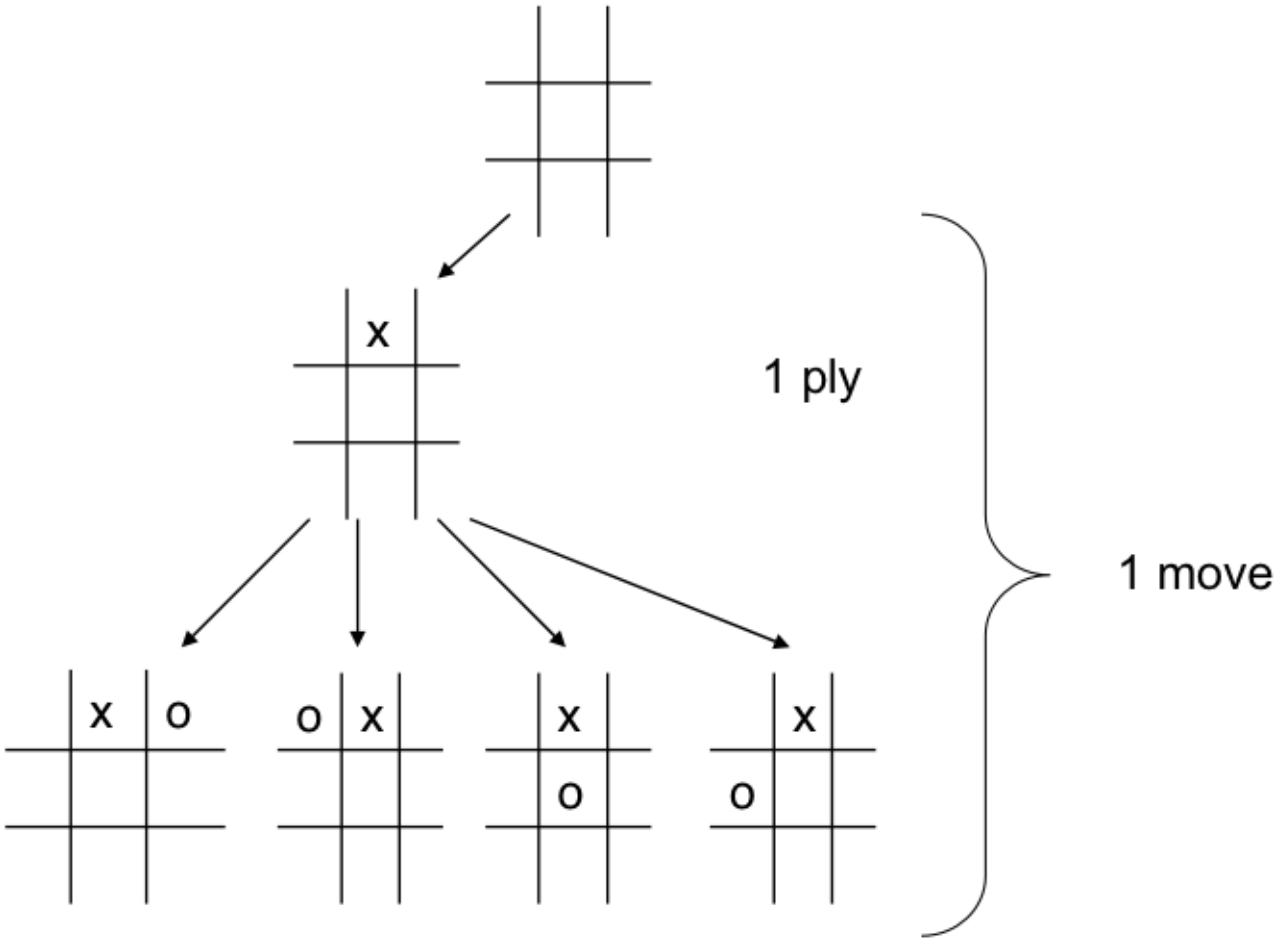
Generate Game Tree



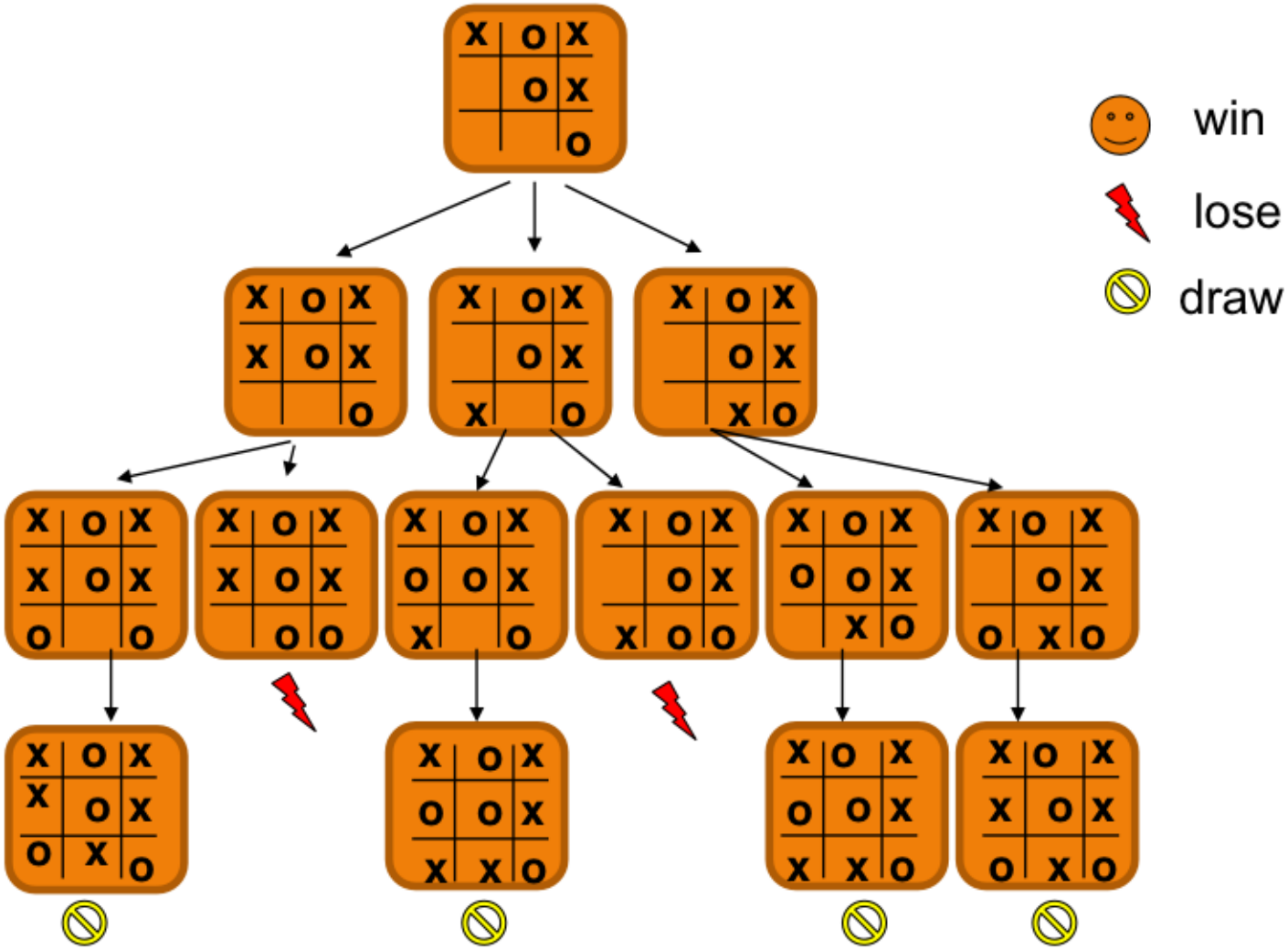
Generate Game Tree



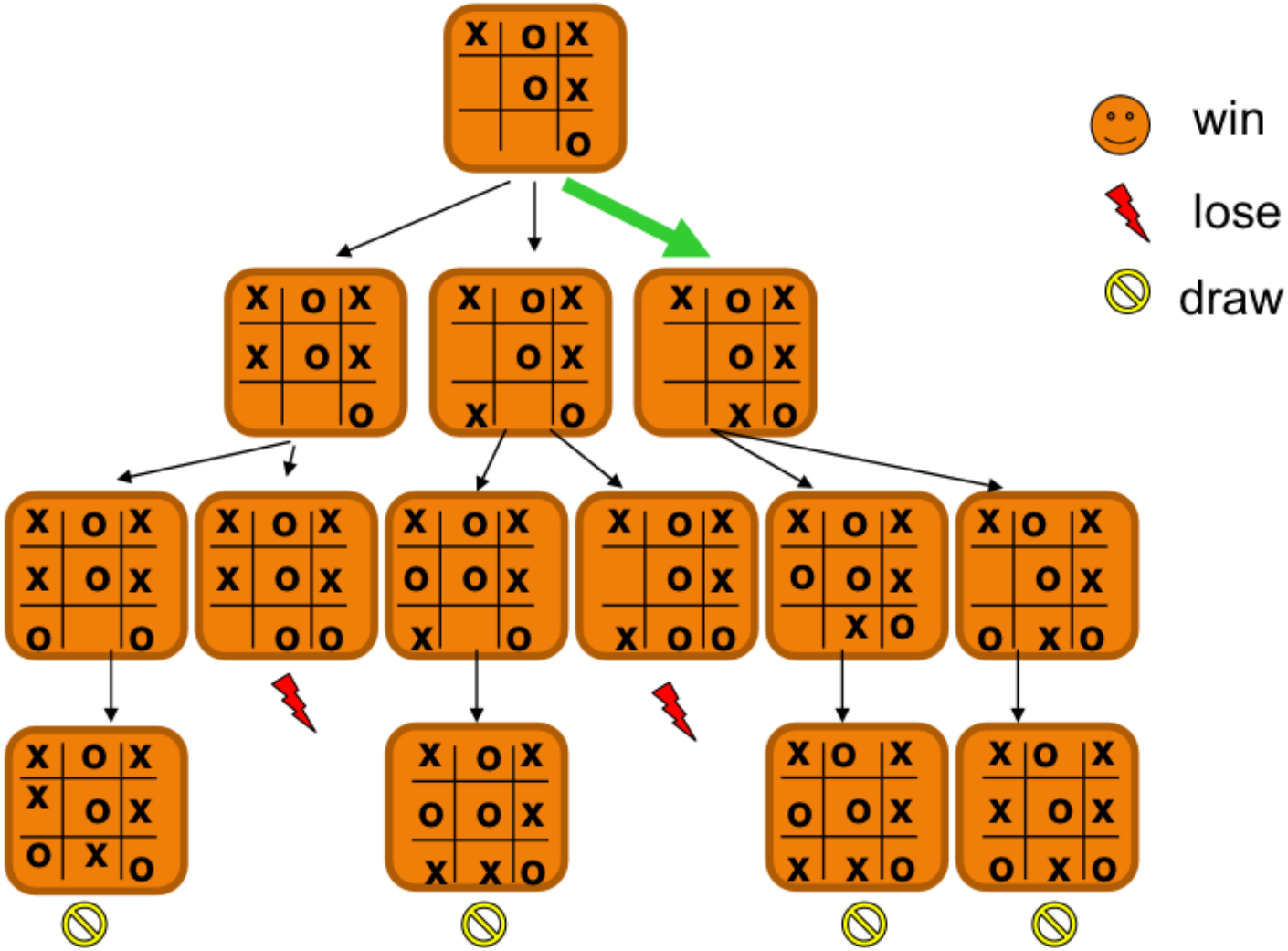
Generate Game Tree



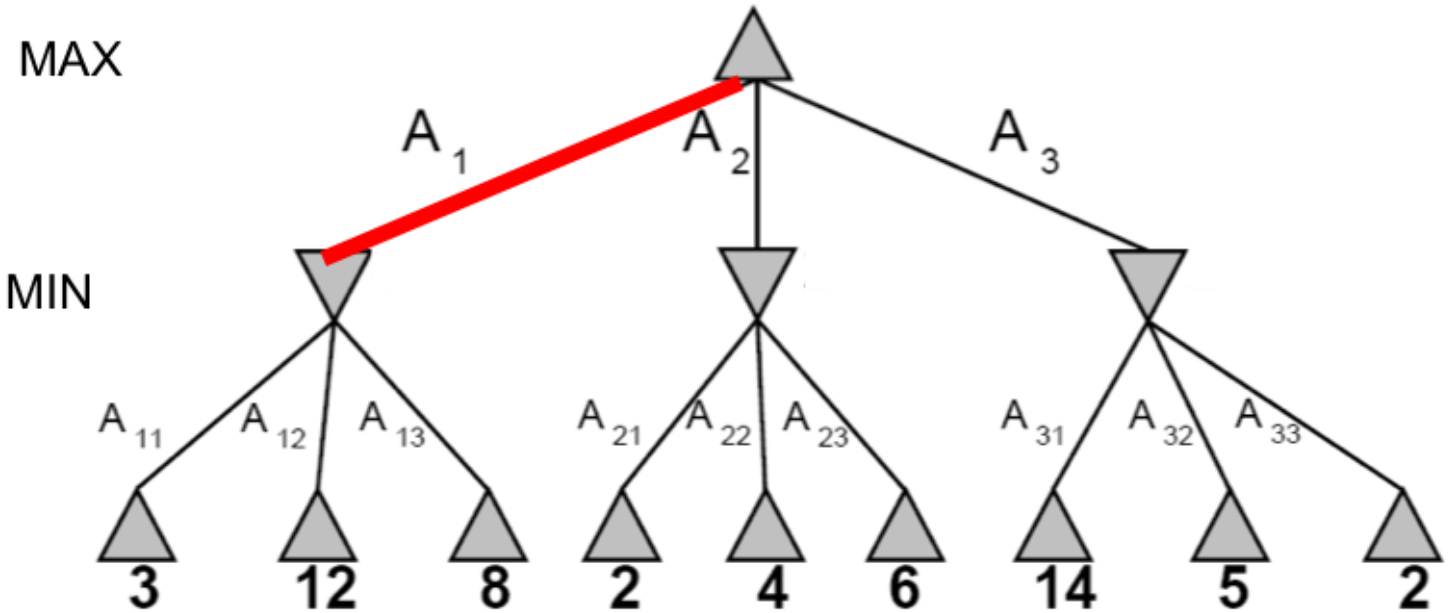
A subtree



What is a good move?



Minimax Example



Minimax: Recursive implementation

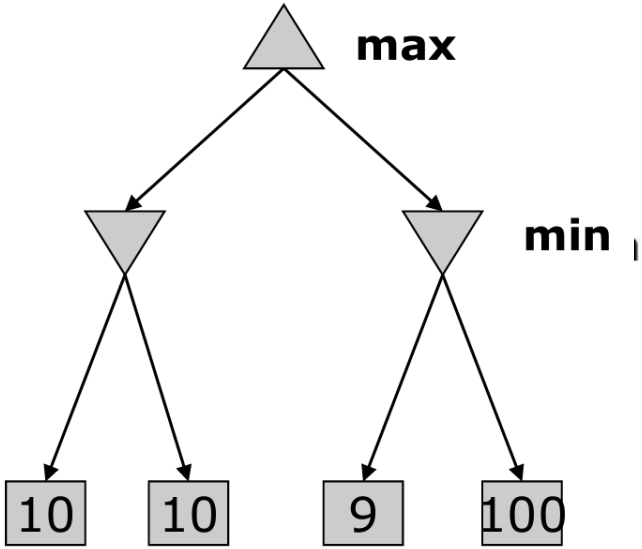
```
function MINIMAX-DECISION(state) returns an action
   inputs: state, current state in game
   return the a in ACTIONS(state) maximizing MIN-VALUE(RESULT(a, state))
```

```
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
```

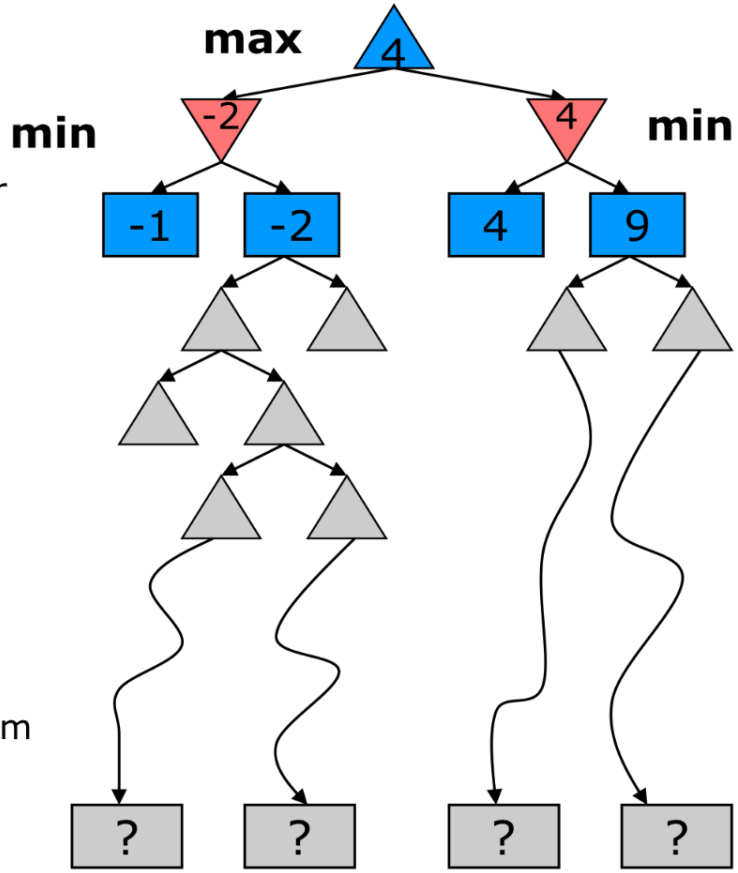
Minimax Properties

- Optimal against a perfect player. Otherwise?
- Time complexity?
  - $O(b^m)$
- Space complexity?
  - $O(bm)$
- For chess, $b \approx 35$, $m \approx 100$
  - Exact solution is completely infeasible
  - But, do we need to explore the whole tree?



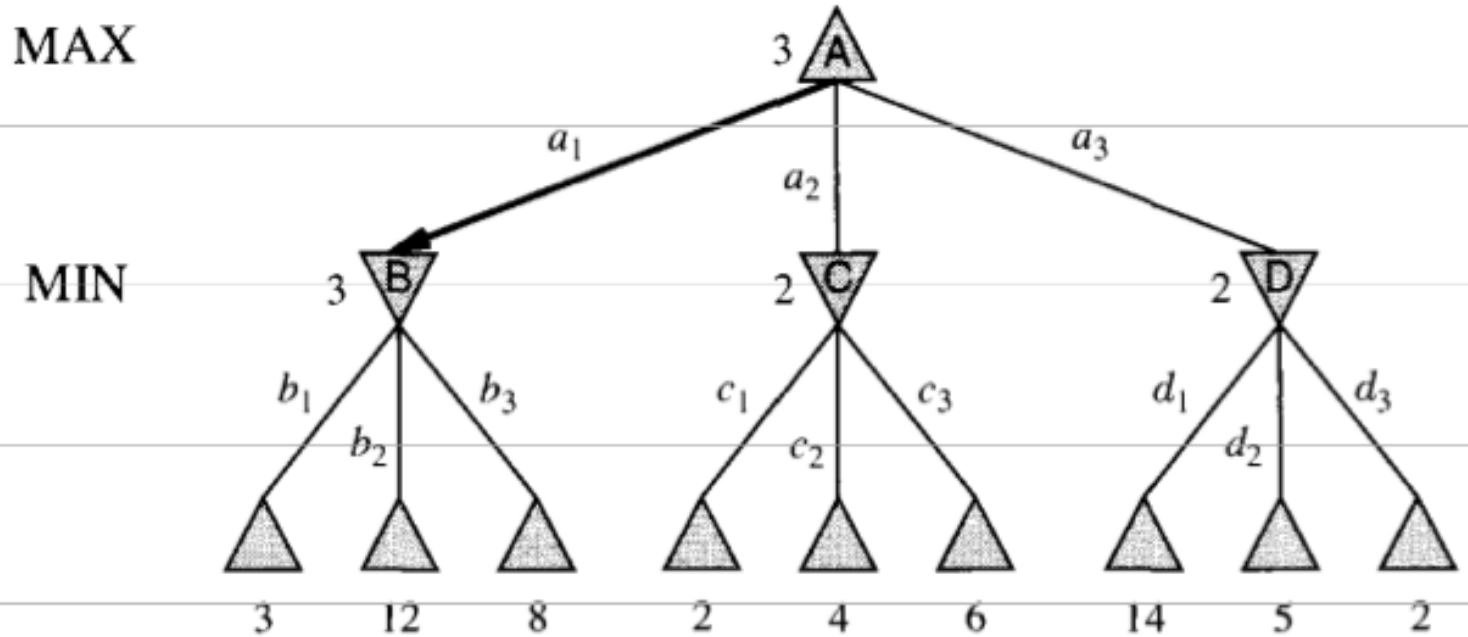
Resource Limits

- Cannot search to leaves
- Depth-limited search
  - Instead, search a limited depth of tree
  - Replace terminal utilities with an eval function for non-terminal positions
- Guarantee of optimal play is gone
- More plies makes a BIG difference
- Example:
  - Suppose we have 100 seconds, can explore 10K nodes / sec
  - So can check 1M nodes per move
  - α-β reaches about depth 8 – decent chess program



Minimax Example

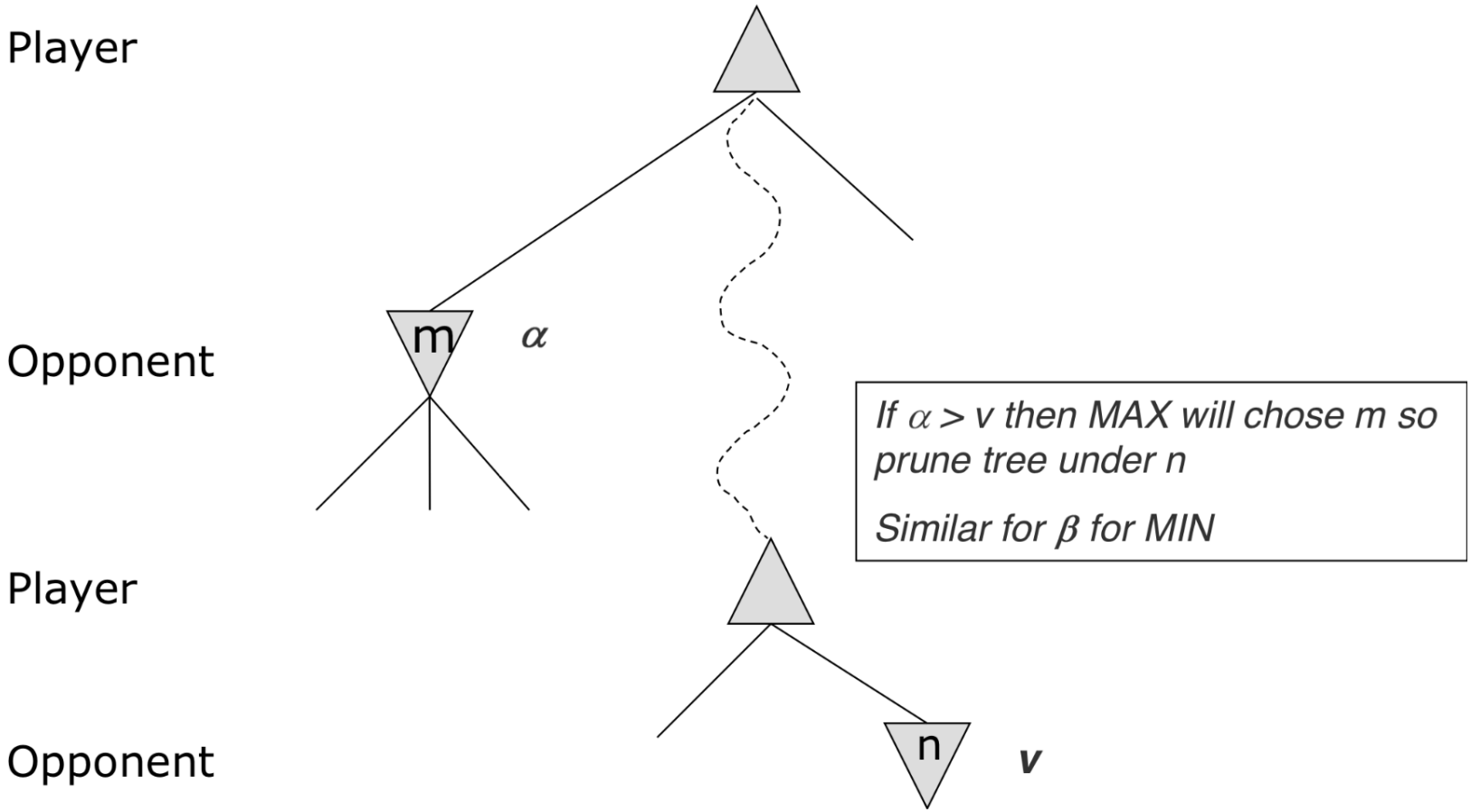
$$\text{MINIMAX-VALUE}(n) = \begin{cases} \text{UTILITY}(n) & \text{if } n \text{ is a terminal state} \\ \max_{s \in Successors(n)} \text{MINIMAX-VALUE}(s) & \text{if } n \text{ is a MAX node} \\ \min_{s \in Successors(n)} \text{MINIMAX-VALUE}(s) & \text{if } n \text{ is a MIN node.} \end{cases}$$



α-β pruning: general principle

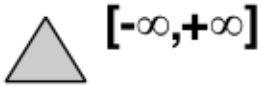
The problem with minimax search is that the number of game states it has to examine is exponential in the number of moves. Unfortunately we can't eliminate the exponent, but we can effectively cut it in half. The trick is that it is possible to compute the correct minimax decision without looking at every node in the game tree. That is, we can borrow the idea of **pruning** from Chapter 4 in order to eliminate large parts of the tree from consideration. The particular technique we will examine is called **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.

α-β pruning: general principle



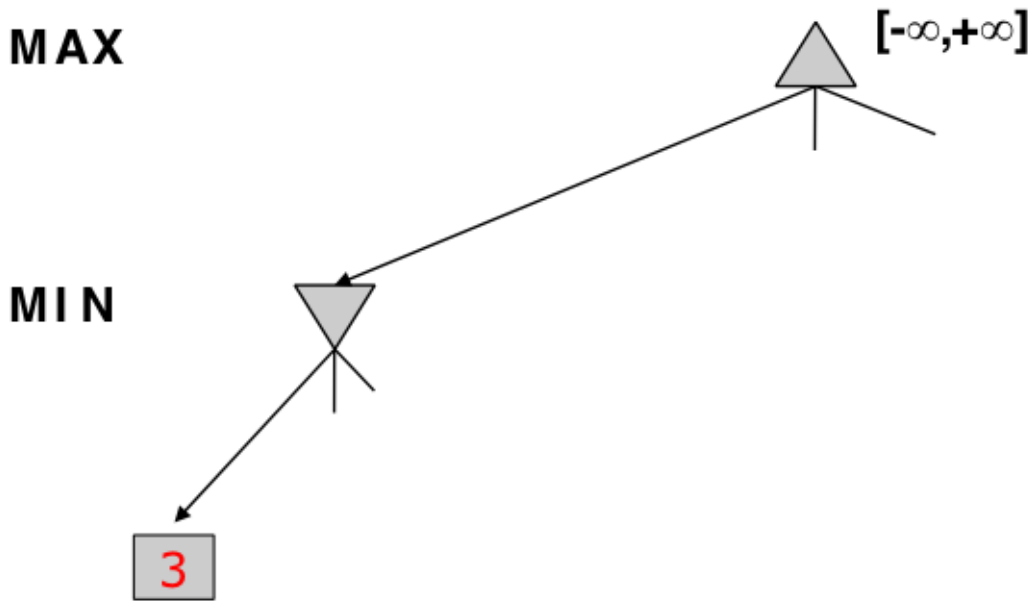
α-β pruning: example 1

MAX

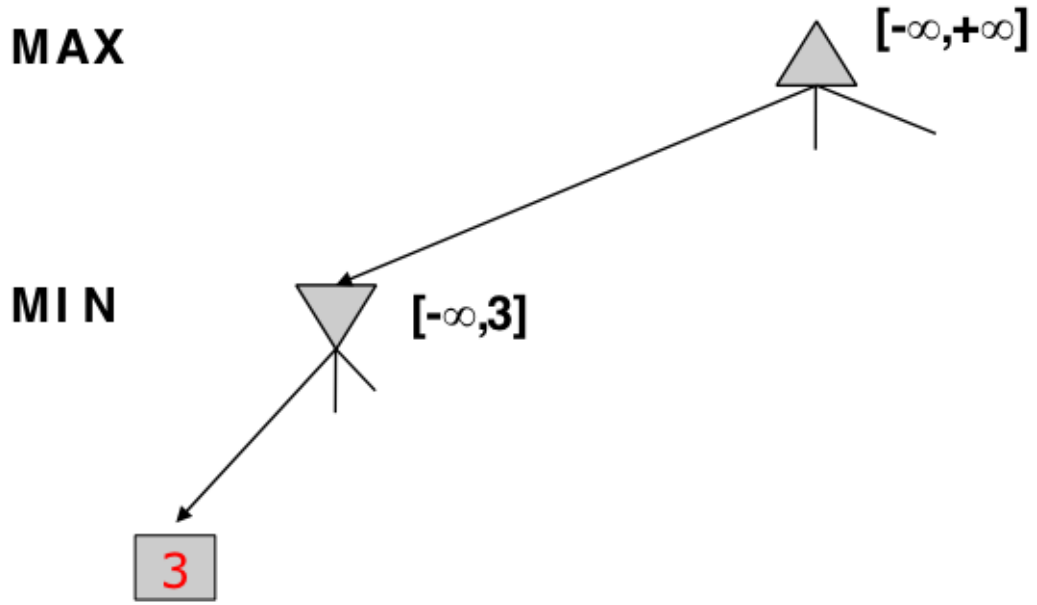


MIN

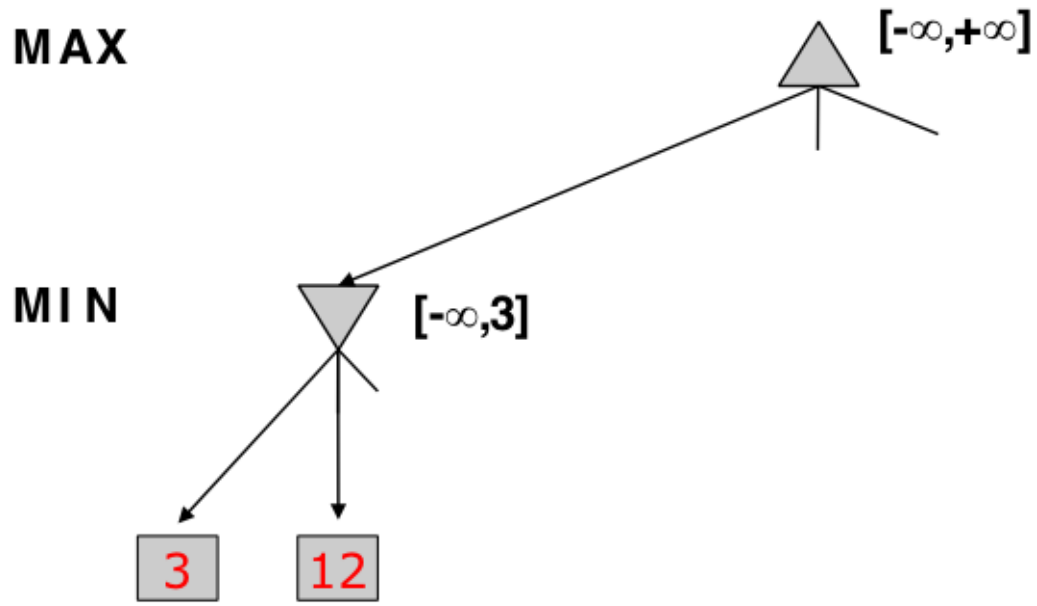
α-β pruning: example 1



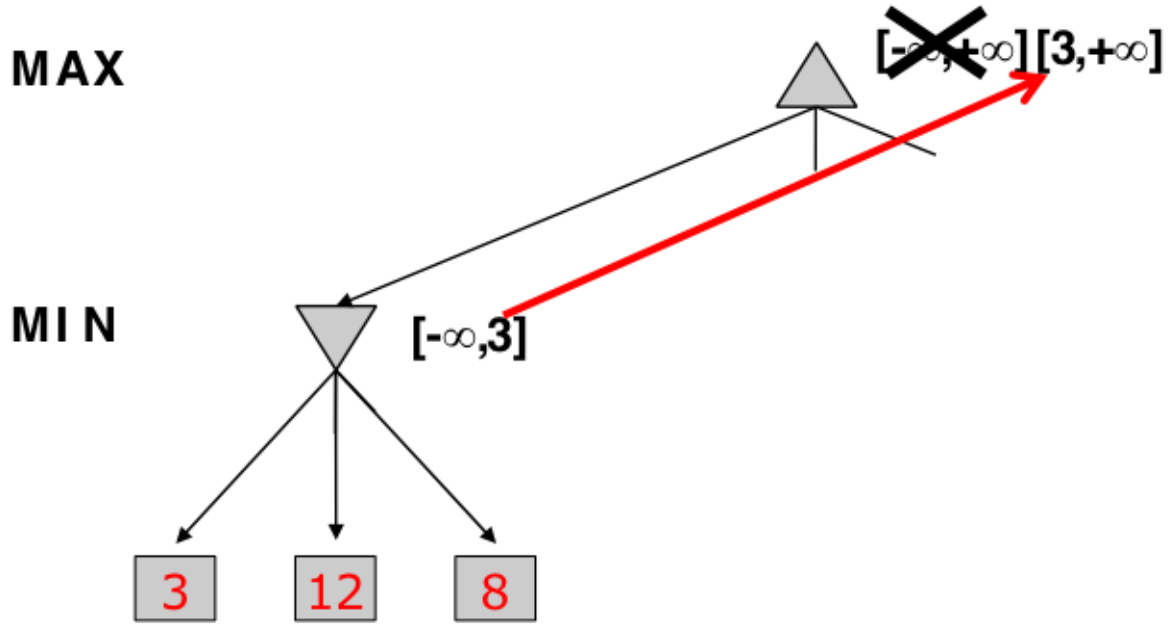
α-β pruning: example 1



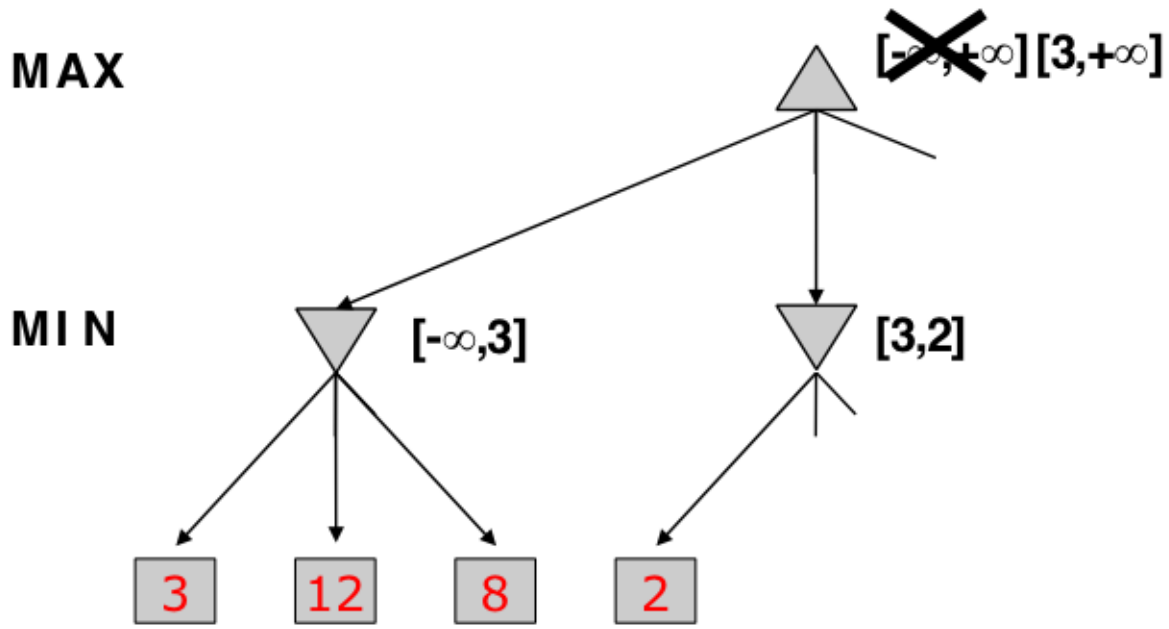
α-β pruning: example 1



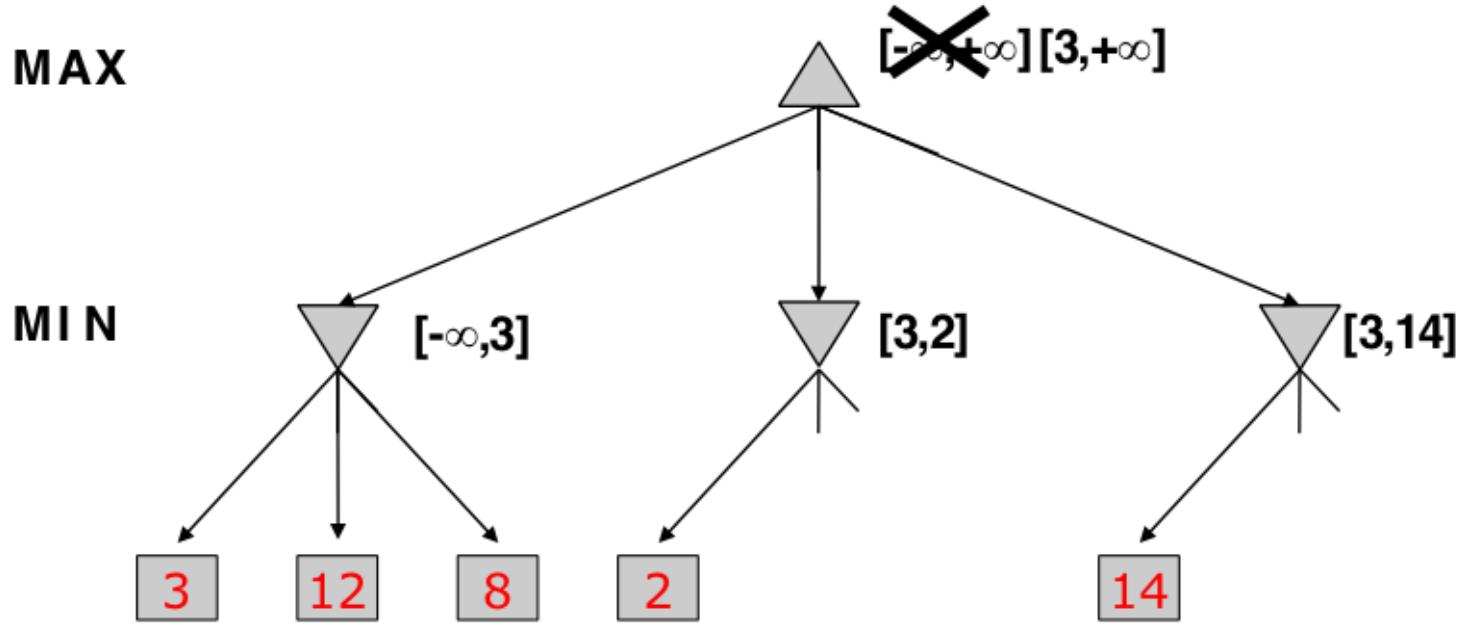
α-β pruning: example 1



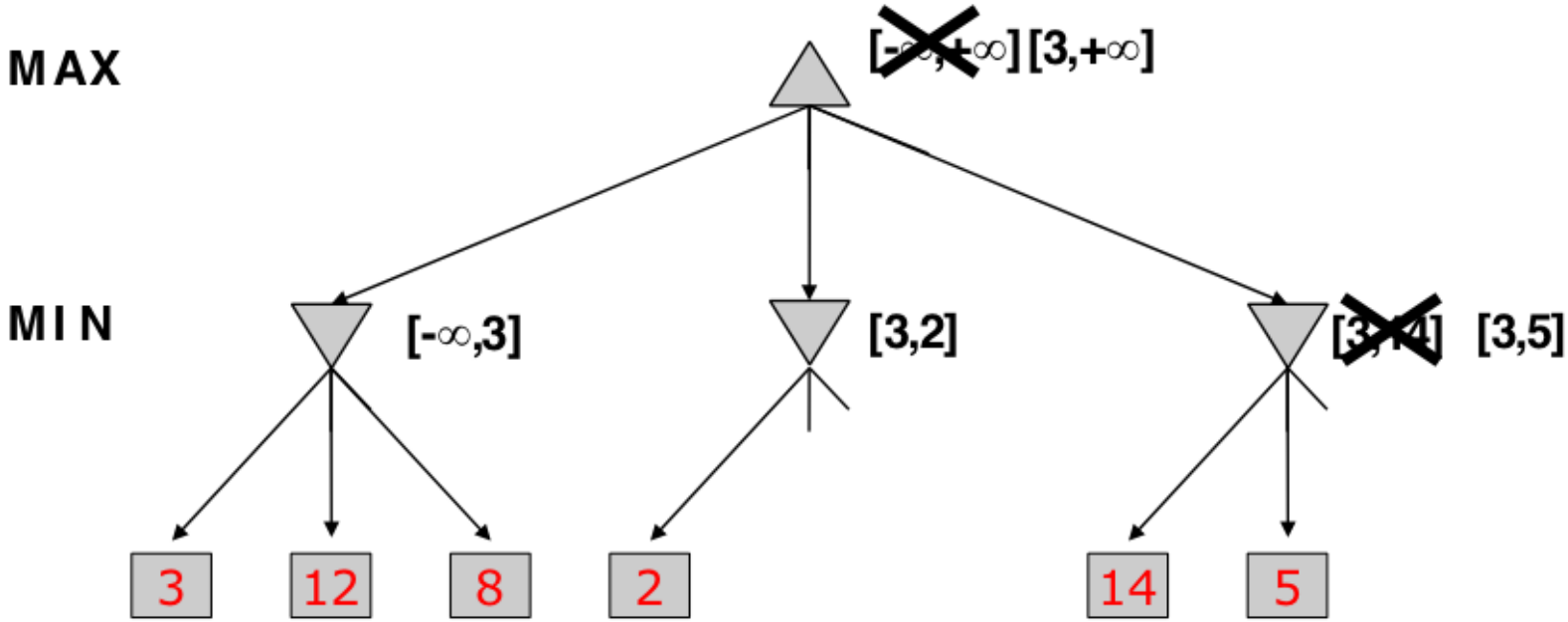
α-β pruning: example 1



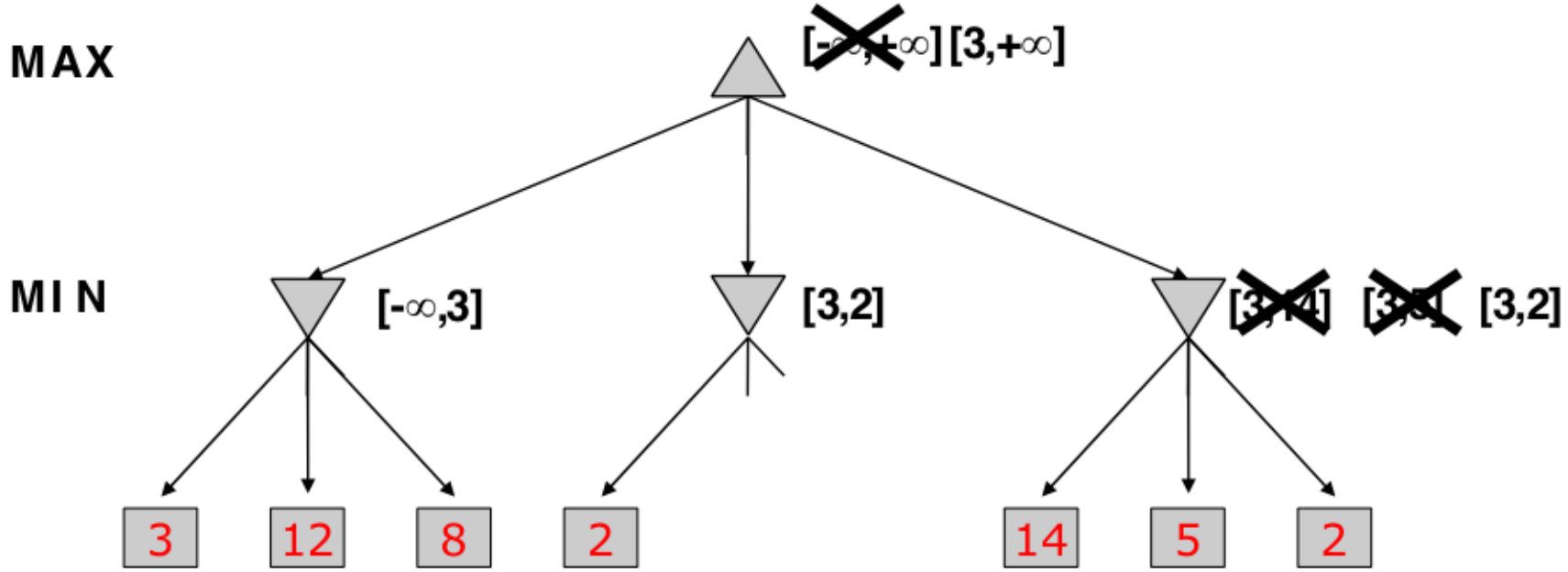
α-β pruning: example 1



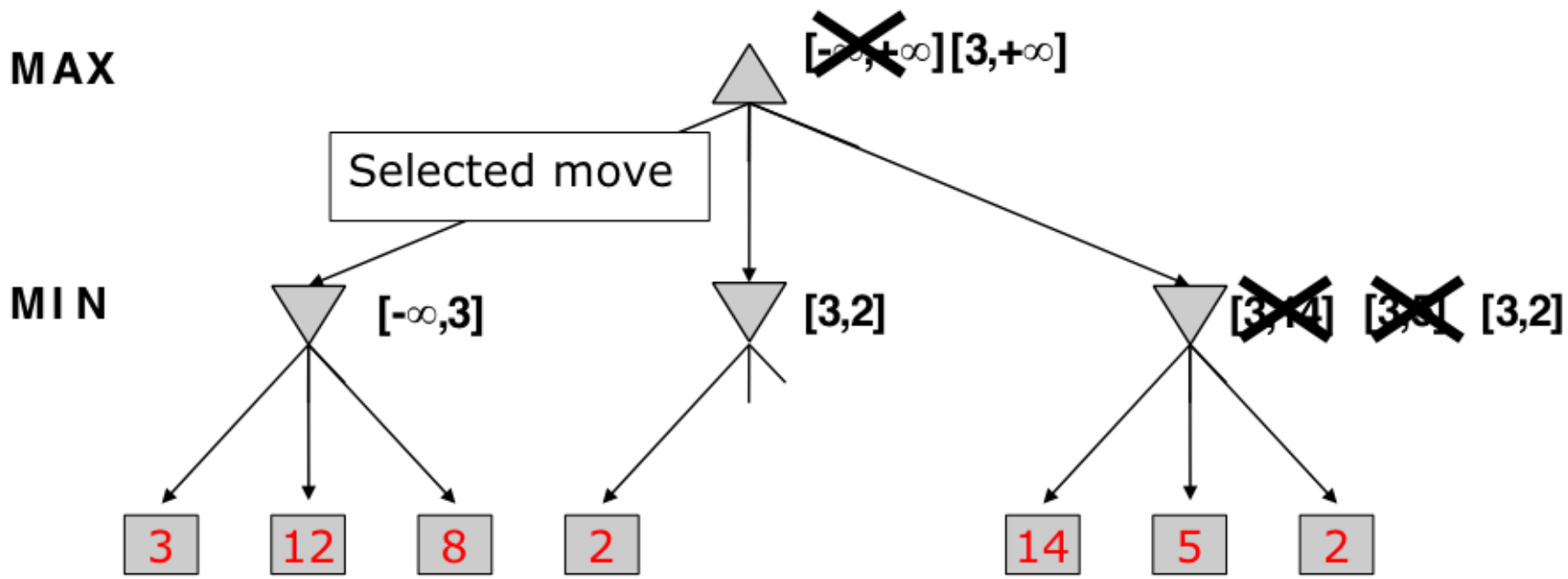
α-β pruning: example 1



α-β pruning: example 1



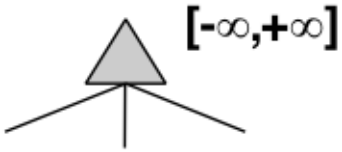
α-β pruning: example 1



$$
\begin{aligned}
\text{MINIMAX-VALUE}(root) &= \max(\min(3,12,8), \min(2,x,y), \min(14,5,2)) \\
&= \max(3, \min(2,x,y), 2) \\
&= \max(3, z, 2) \qquad \text{where } z \leq 2 \\
&= 3.
\end{aligned}
$$

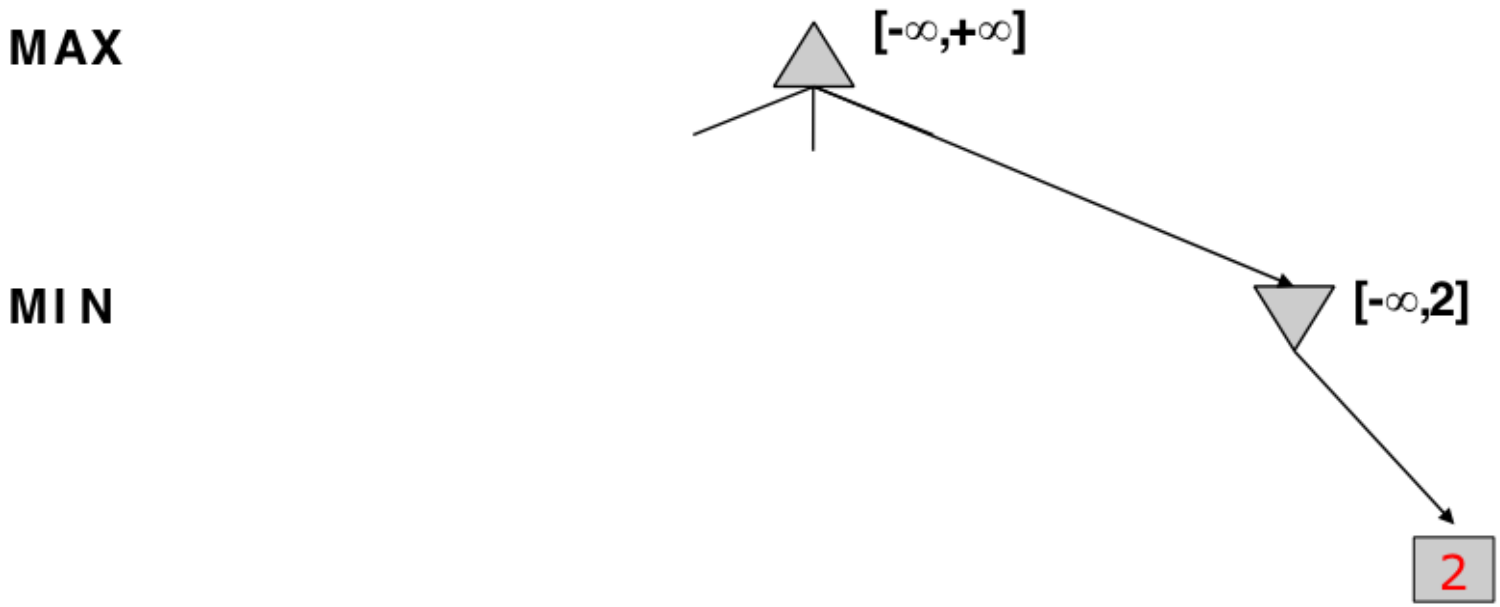
α-β pruning: example 2

MAX

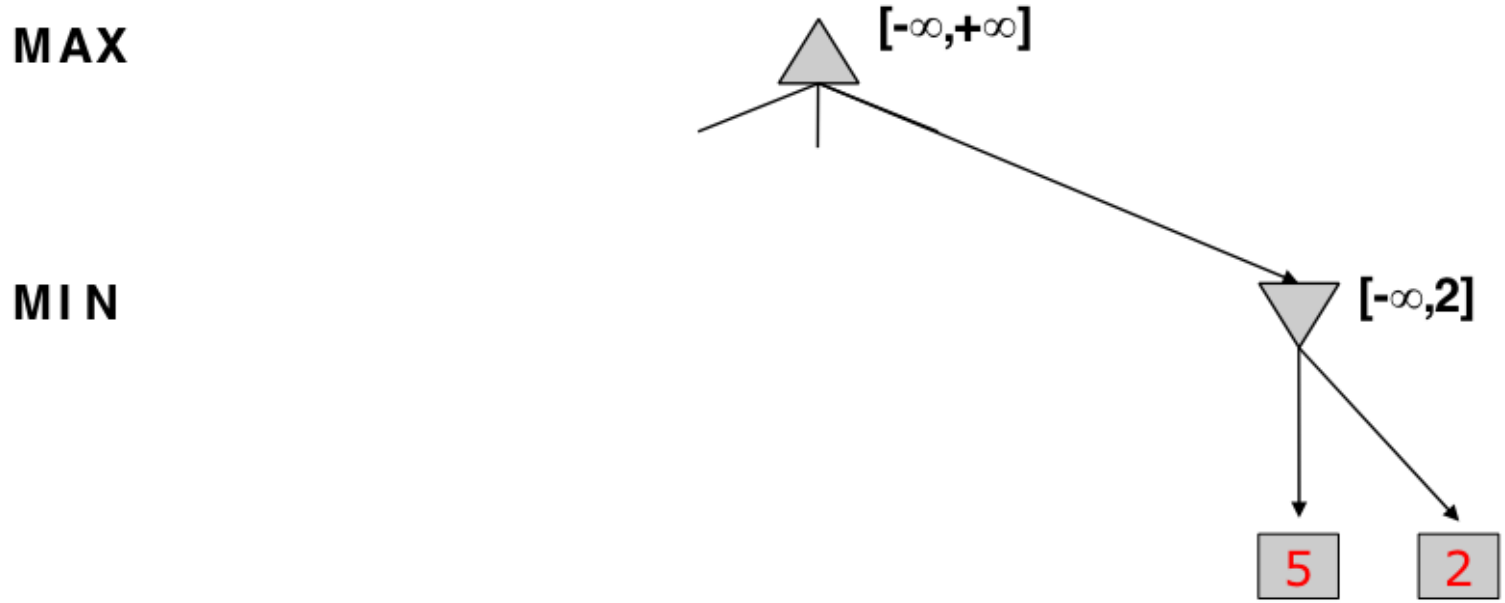


MIN

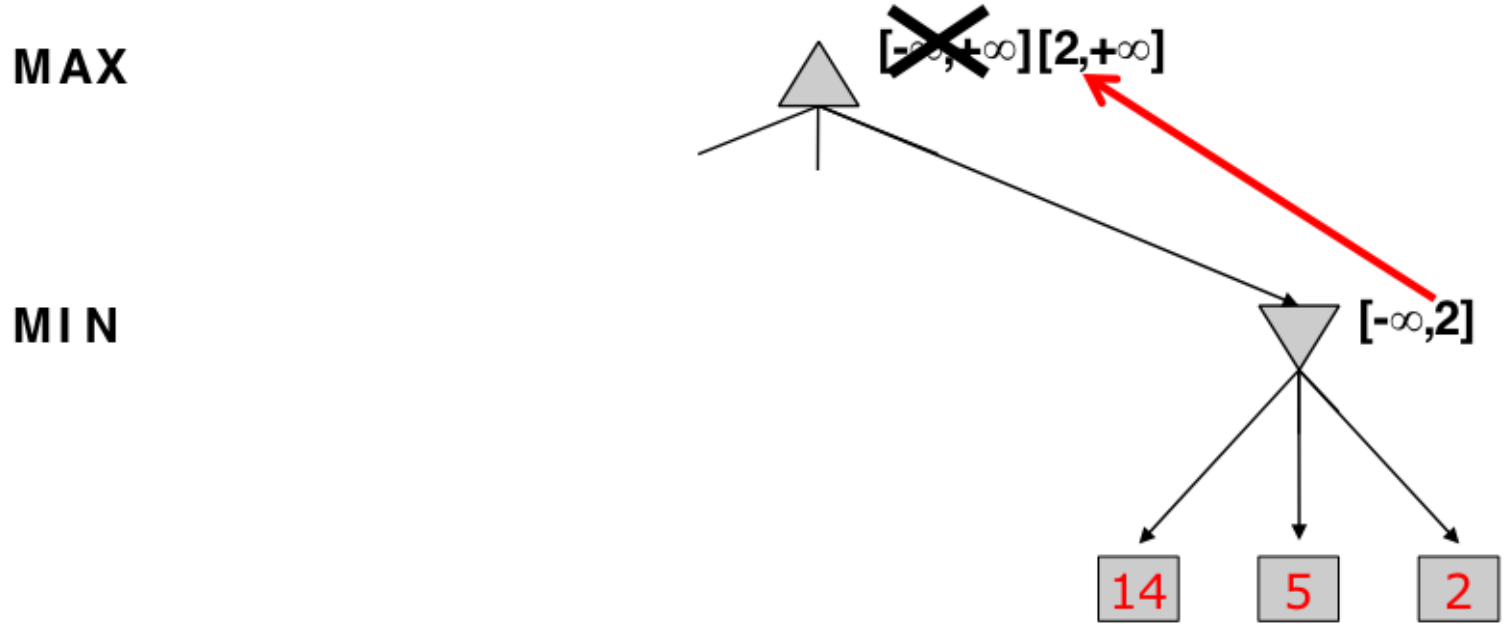
α-β pruning: example 2



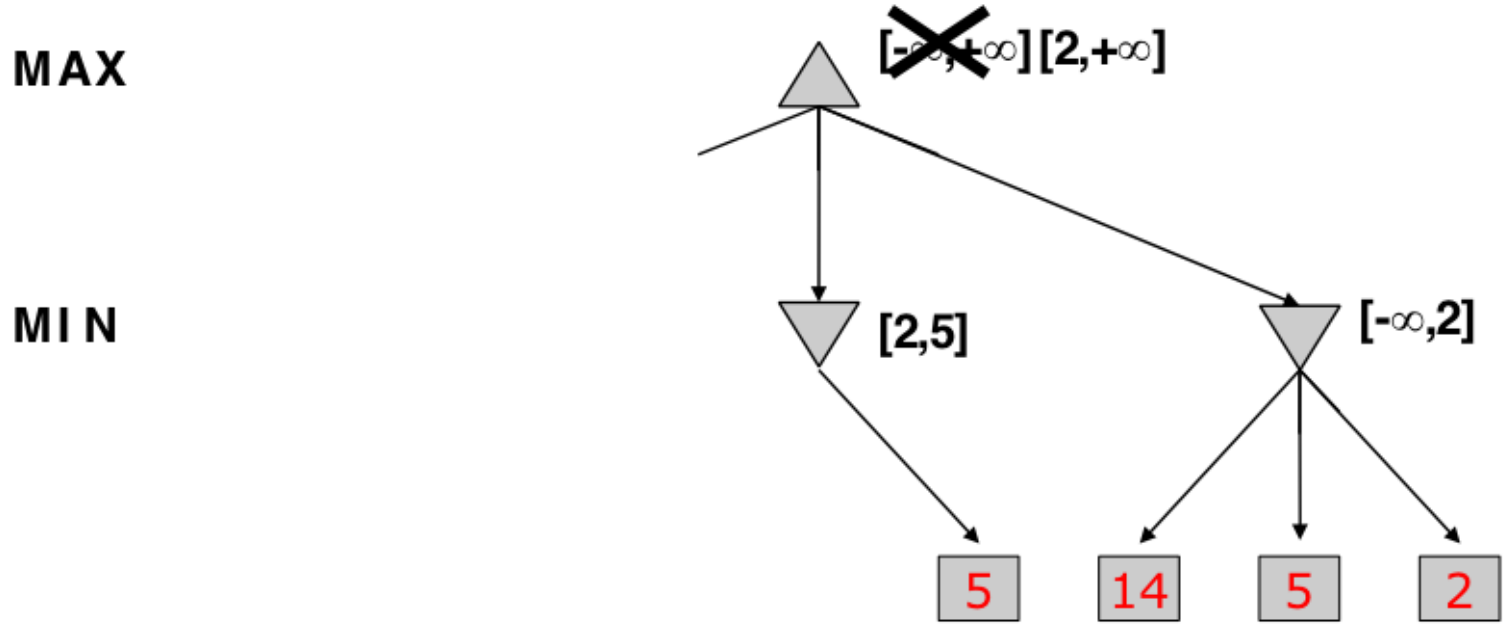
α-β pruning: example 2



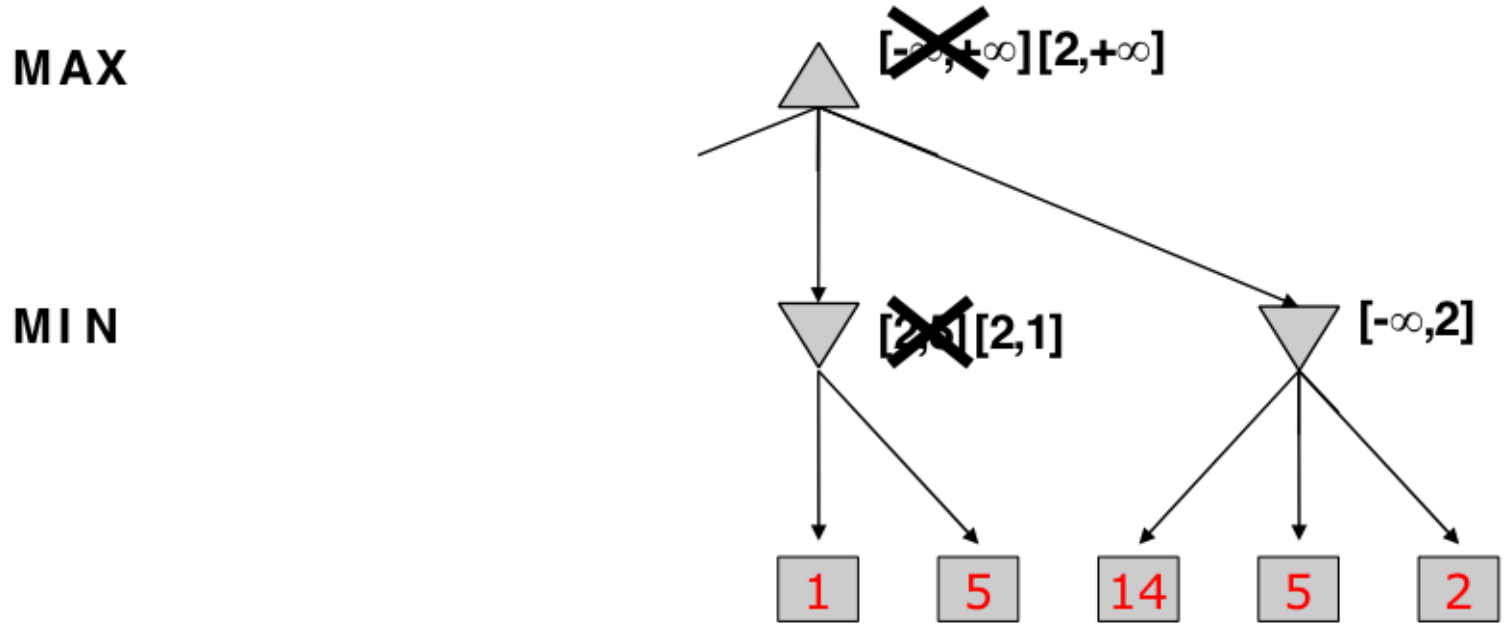
α-β pruning: example 2



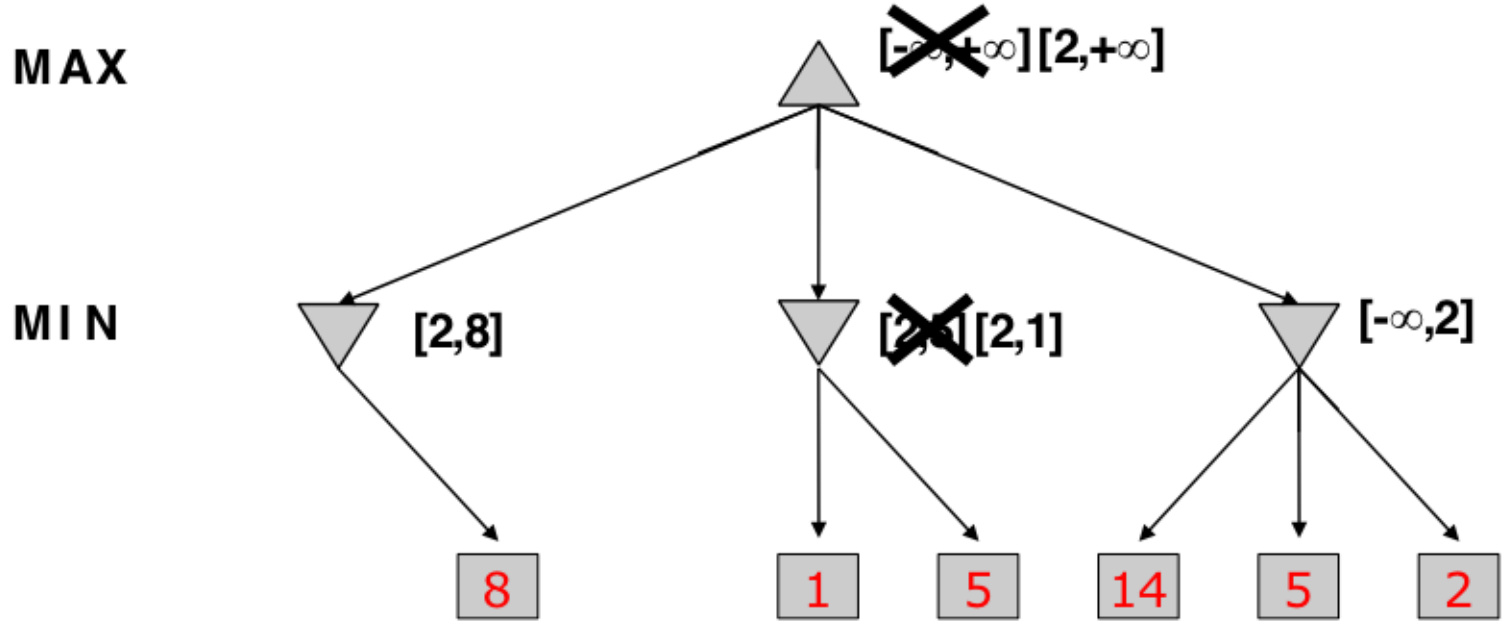
α-β pruning: example 2



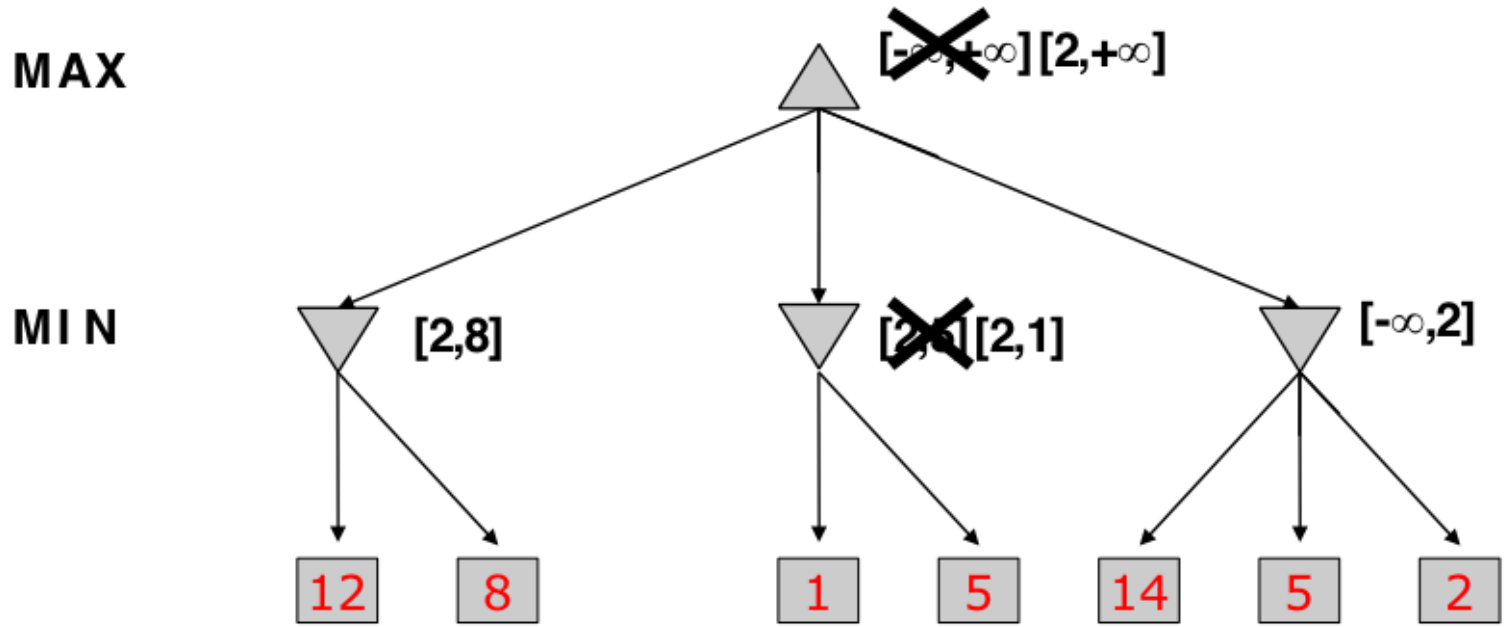
α-β pruning: example 2



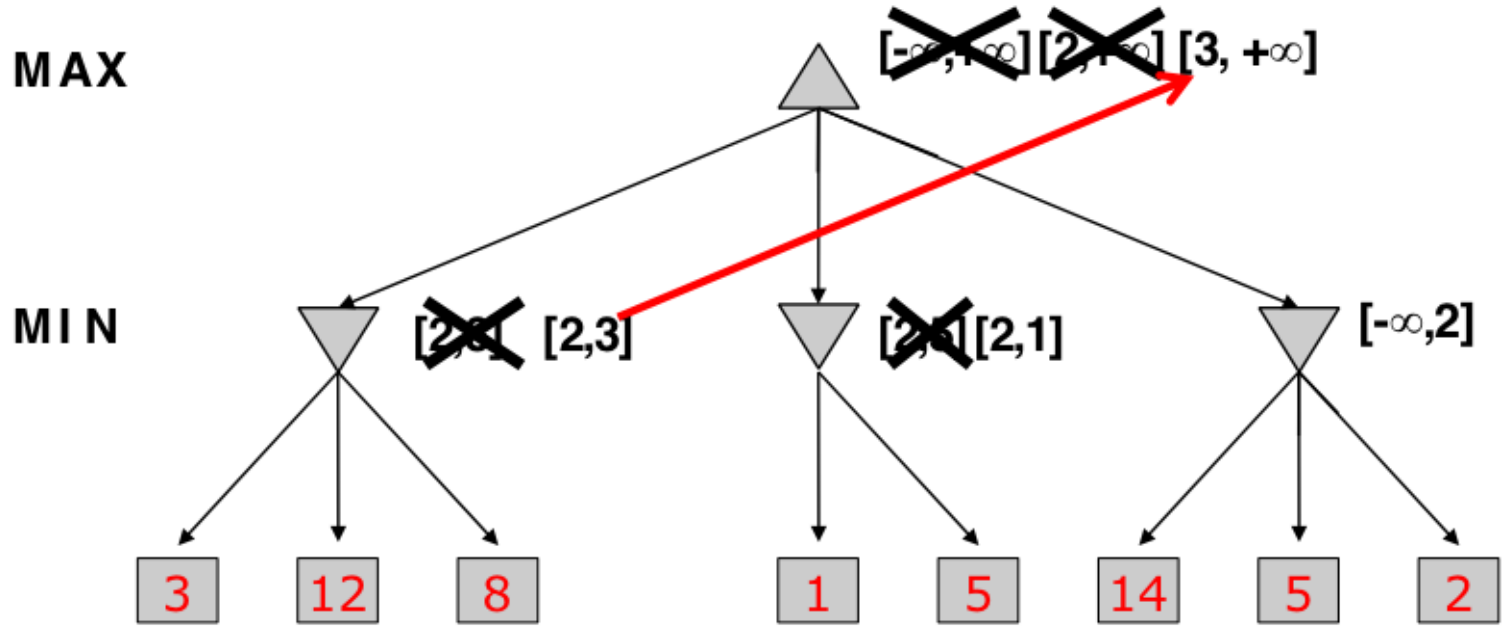
α-β pruning: example 2



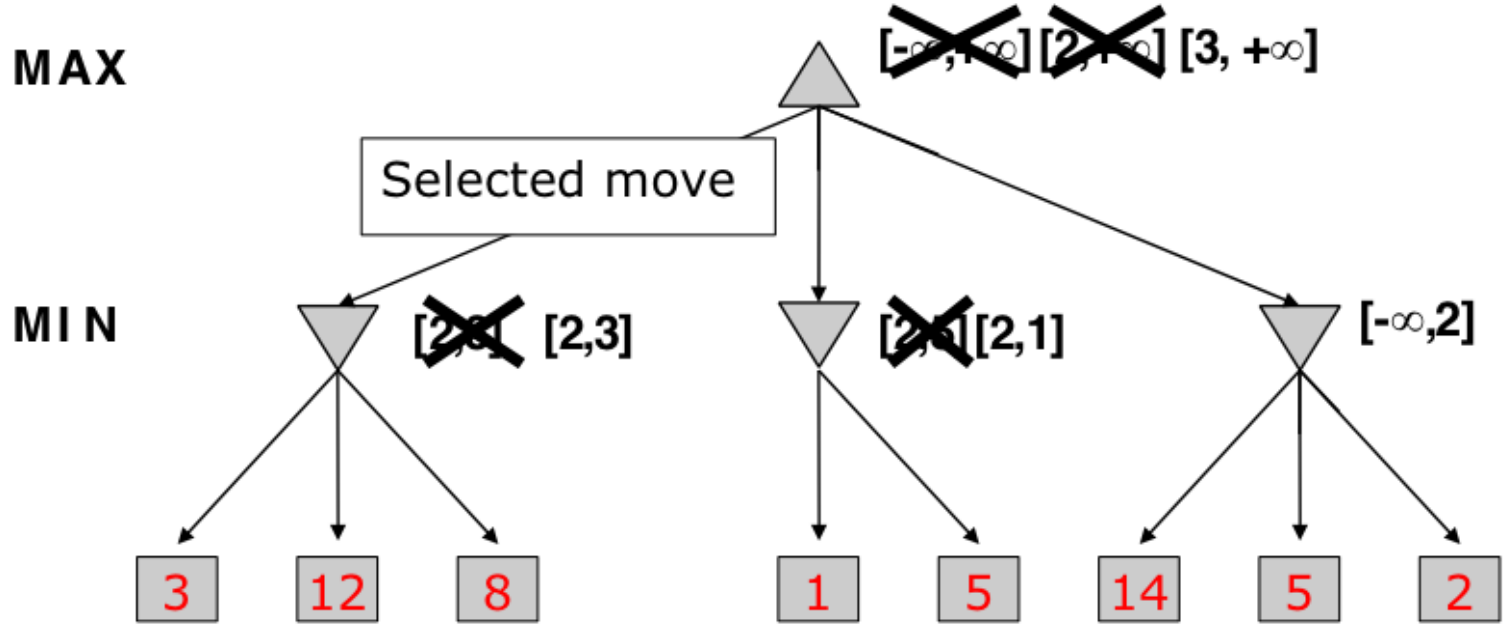
α-β pruning: example 2



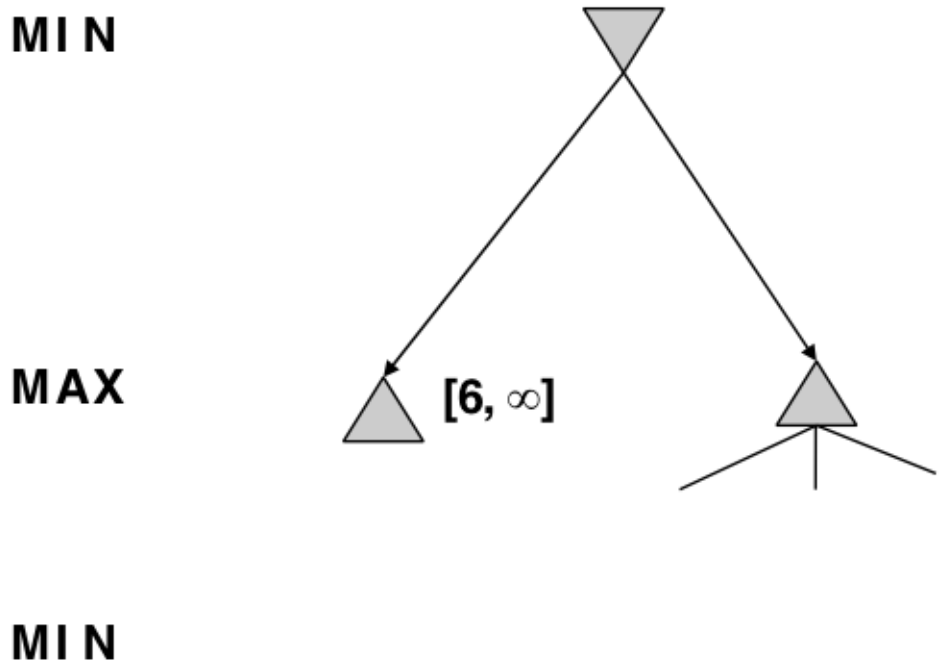
α-β pruning: example 2



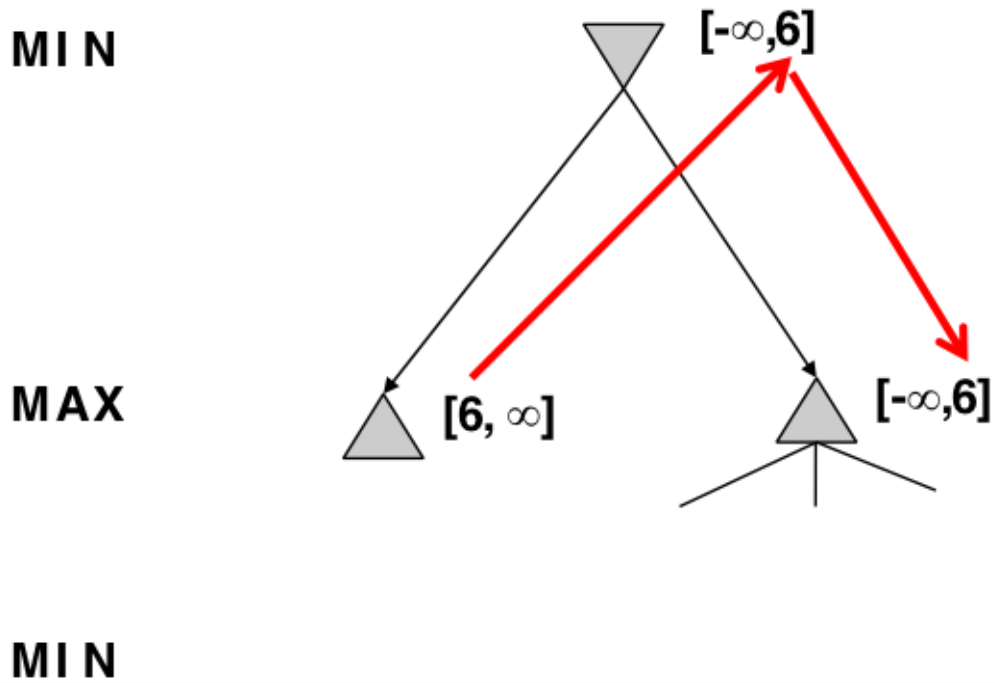
α-β pruning: example 2



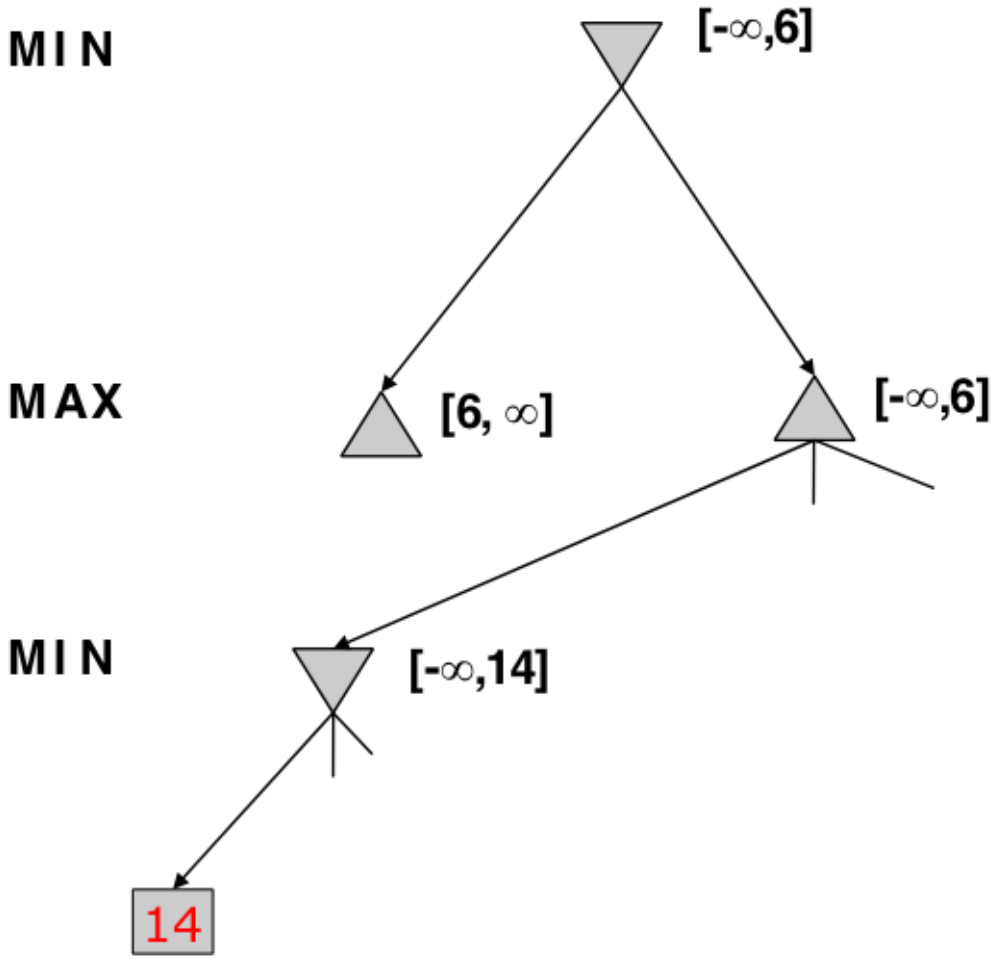
α-β pruning: example 3



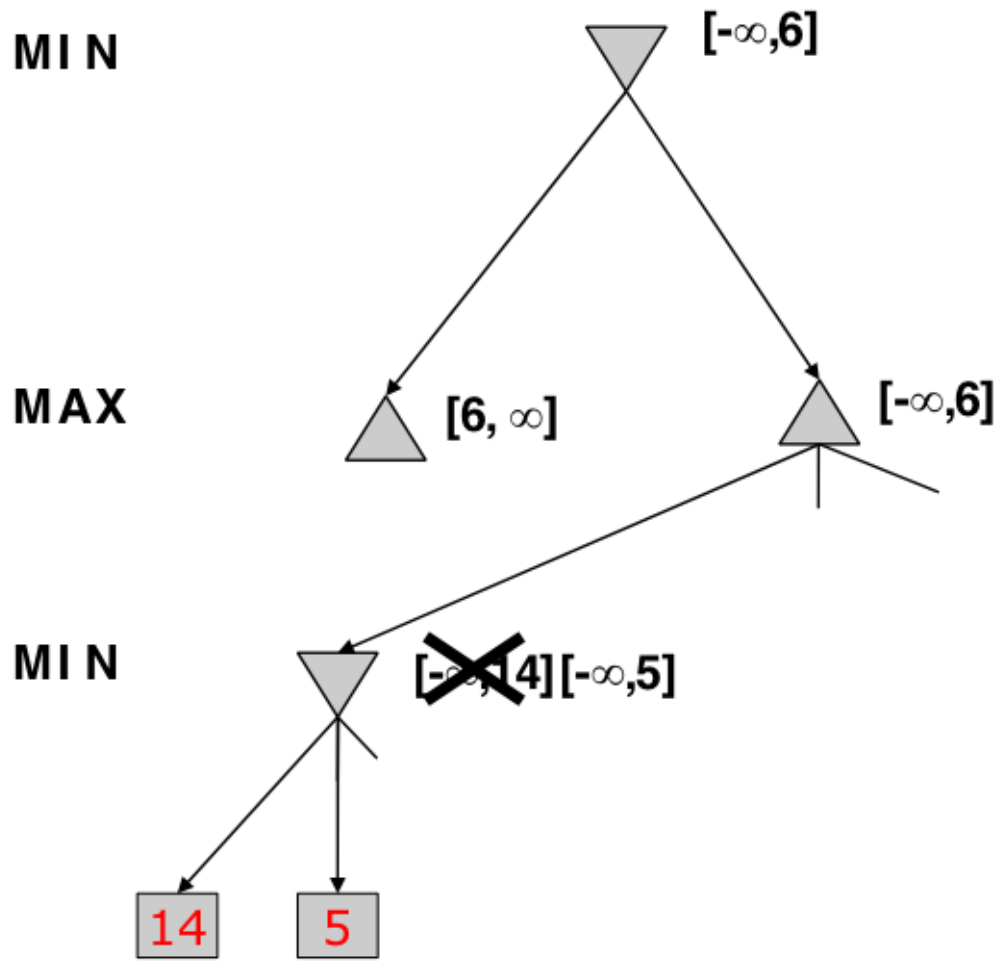
α-β pruning: example 3



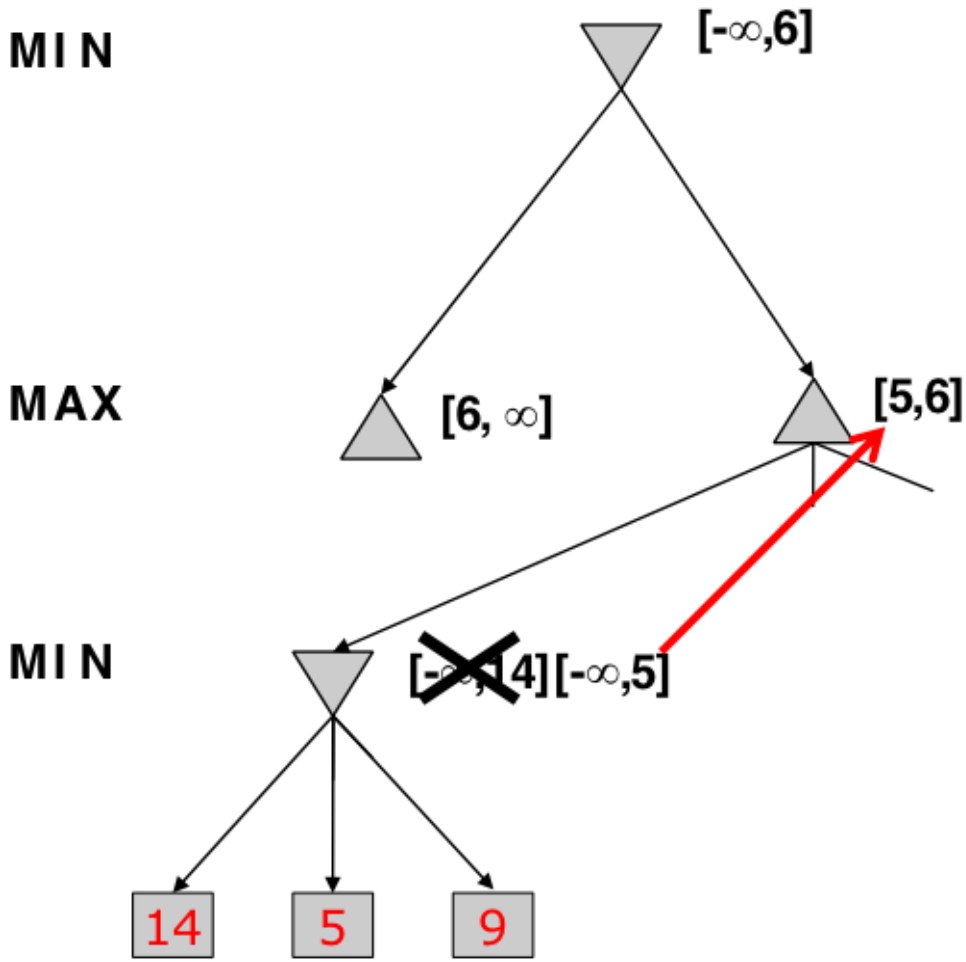
α-β pruning: example 3



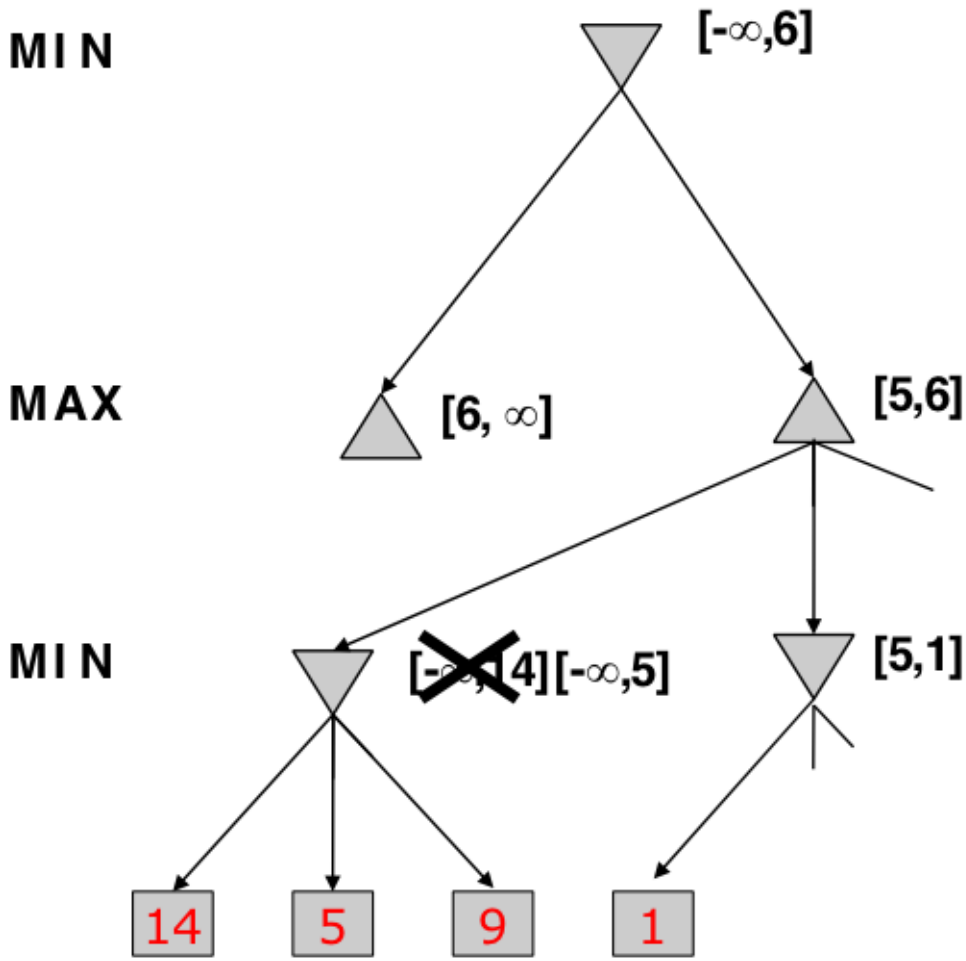
α-β pruning: example 3



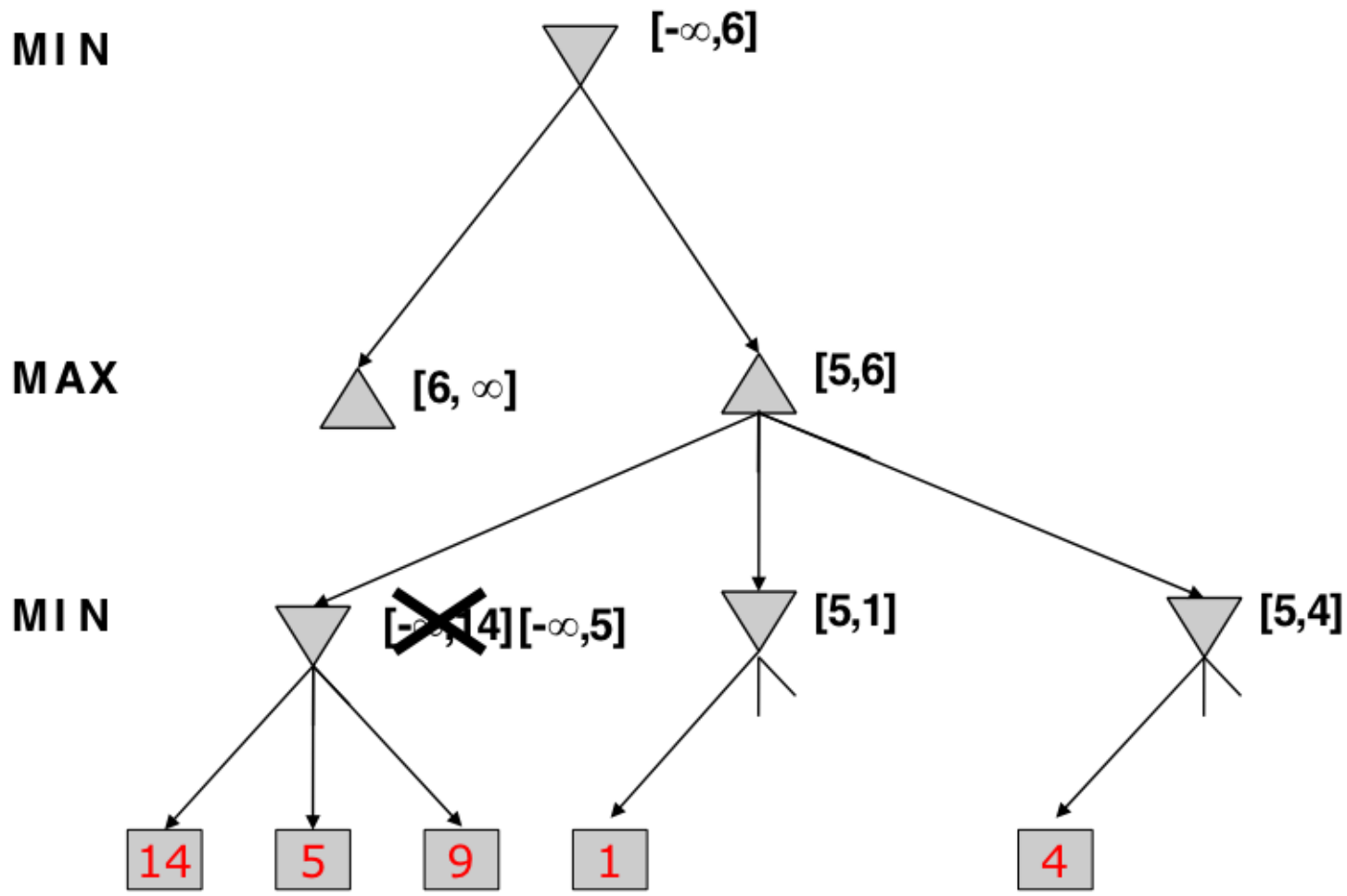
α-β pruning: example 3



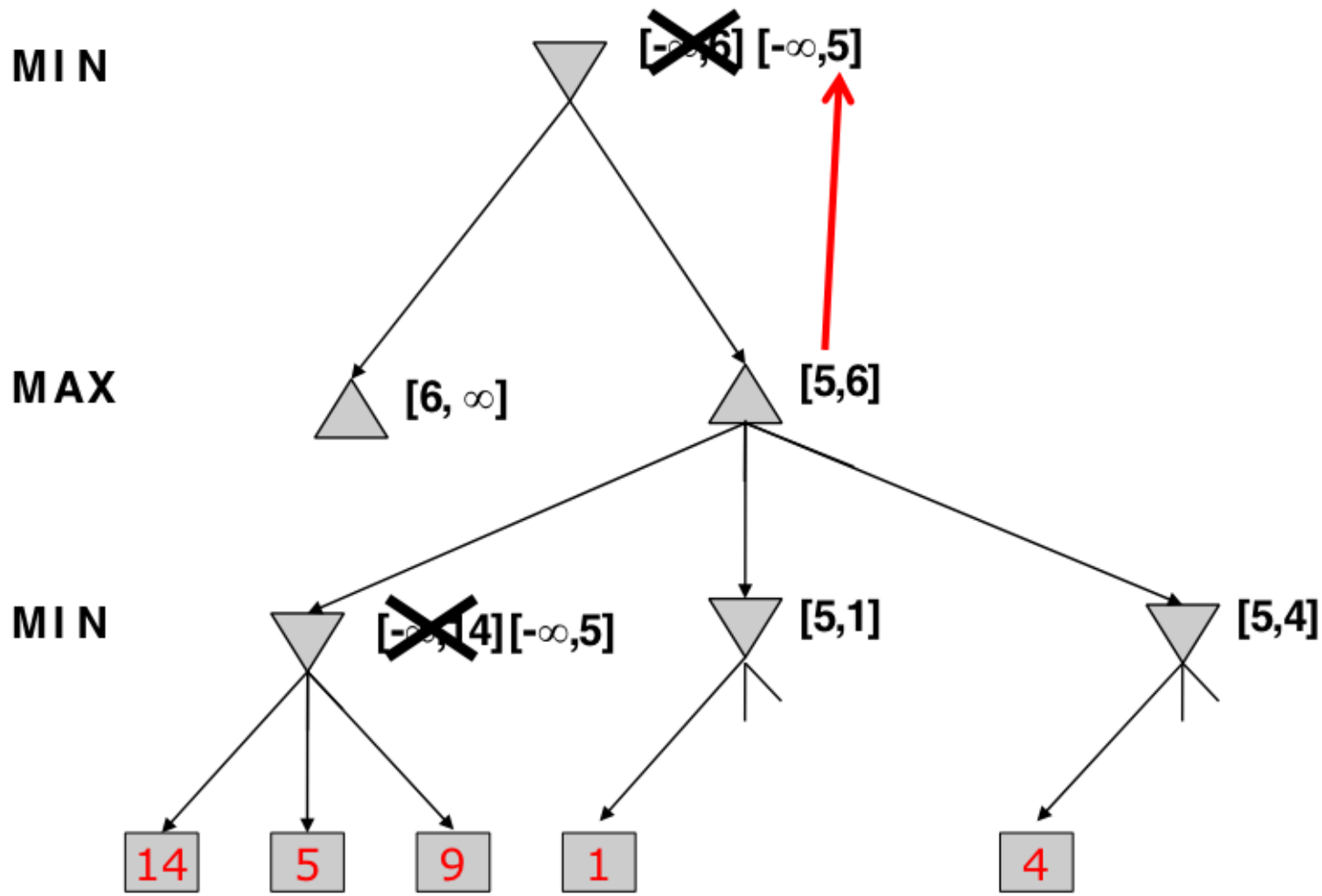
α-β pruning: example 3



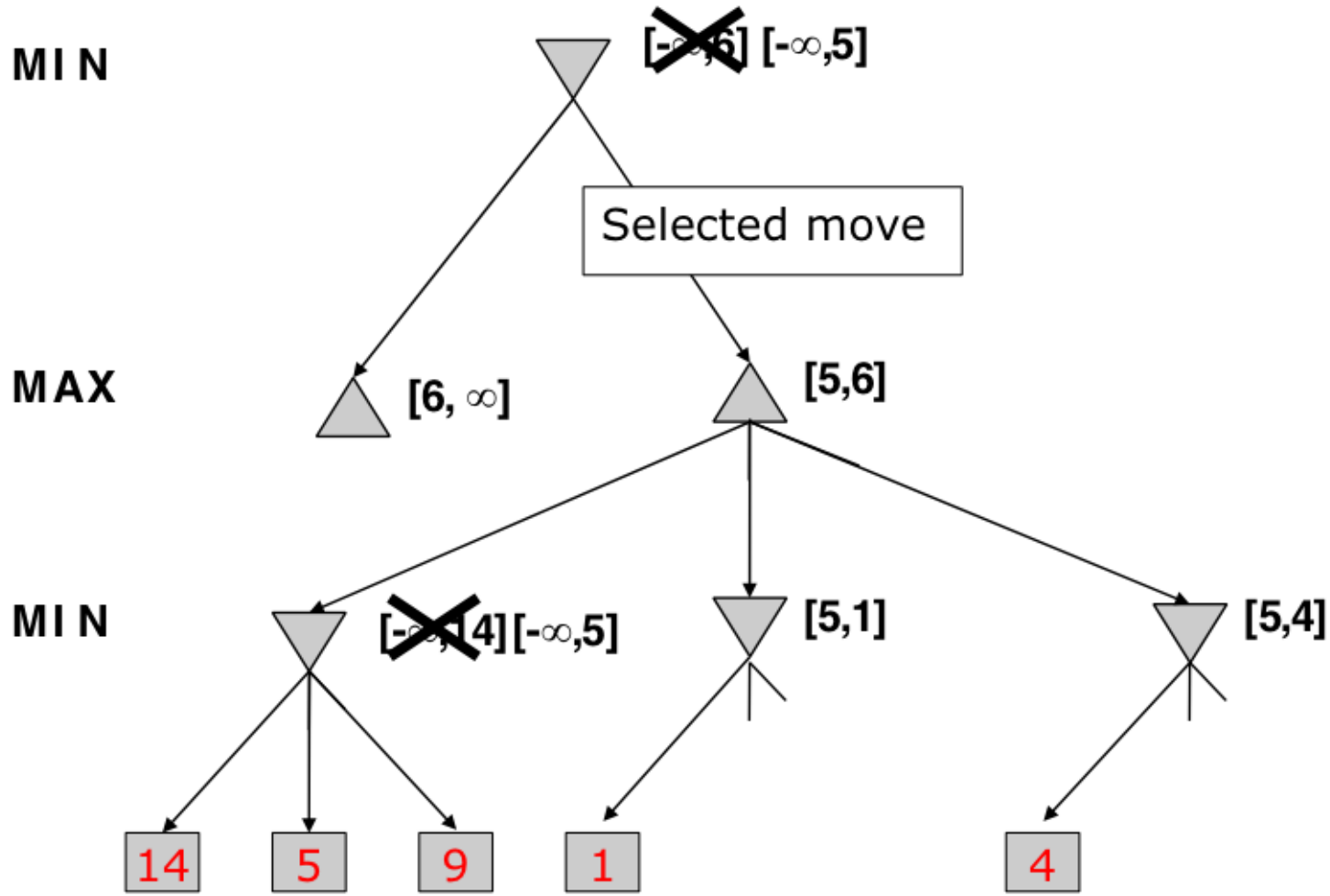
α-β pruning: example 3



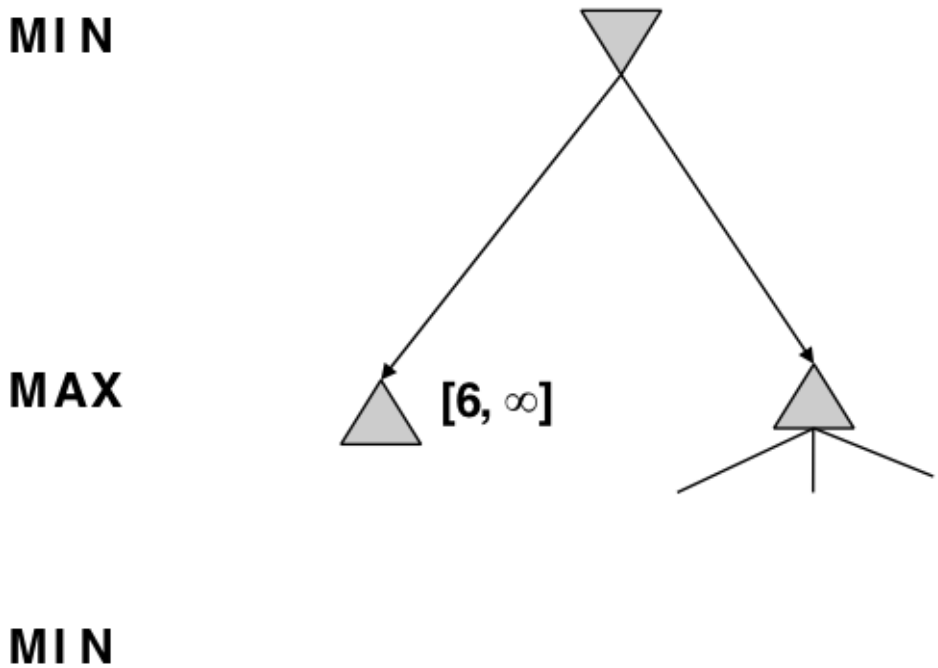
α-β pruning: example 3



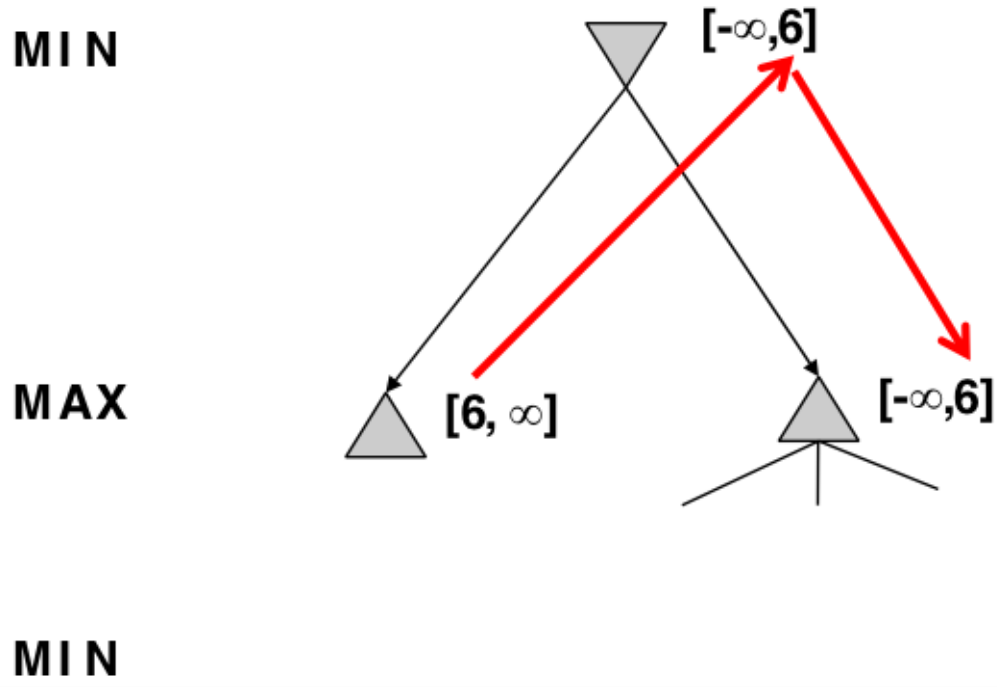
α-β pruning: example 3



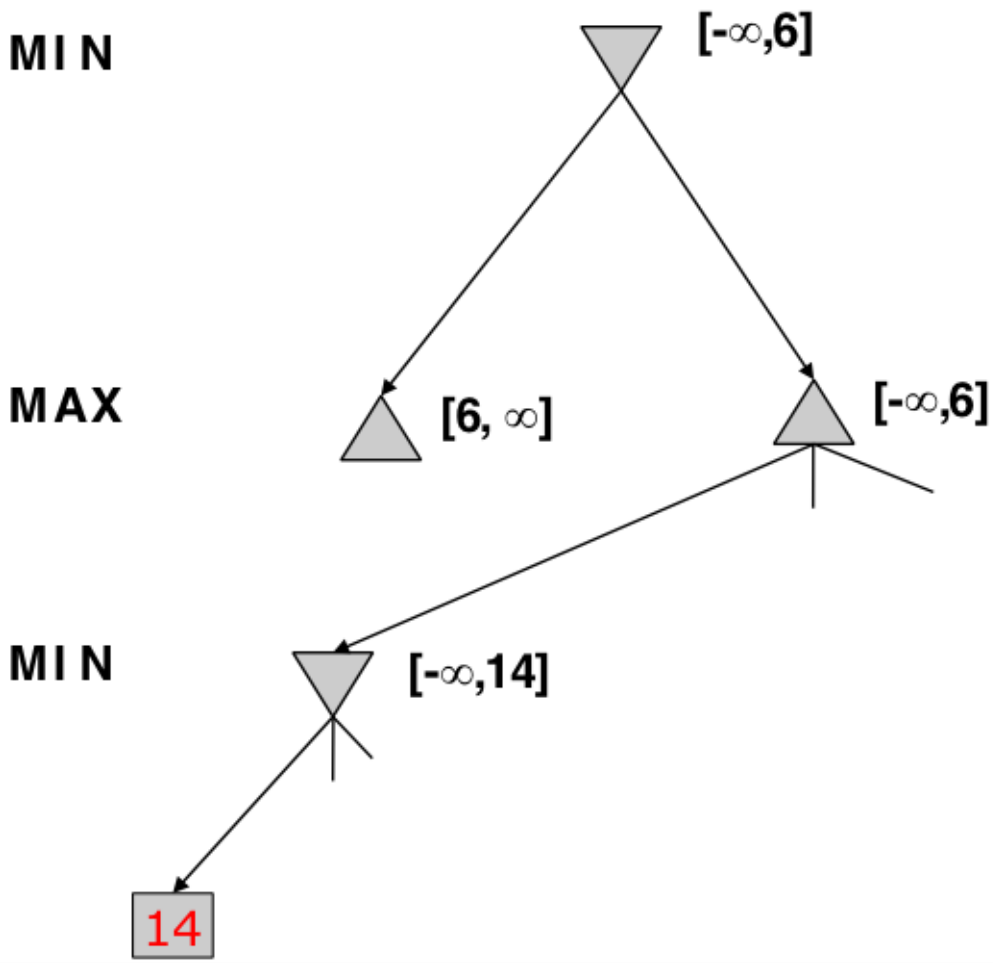
α-β pruning: example 4



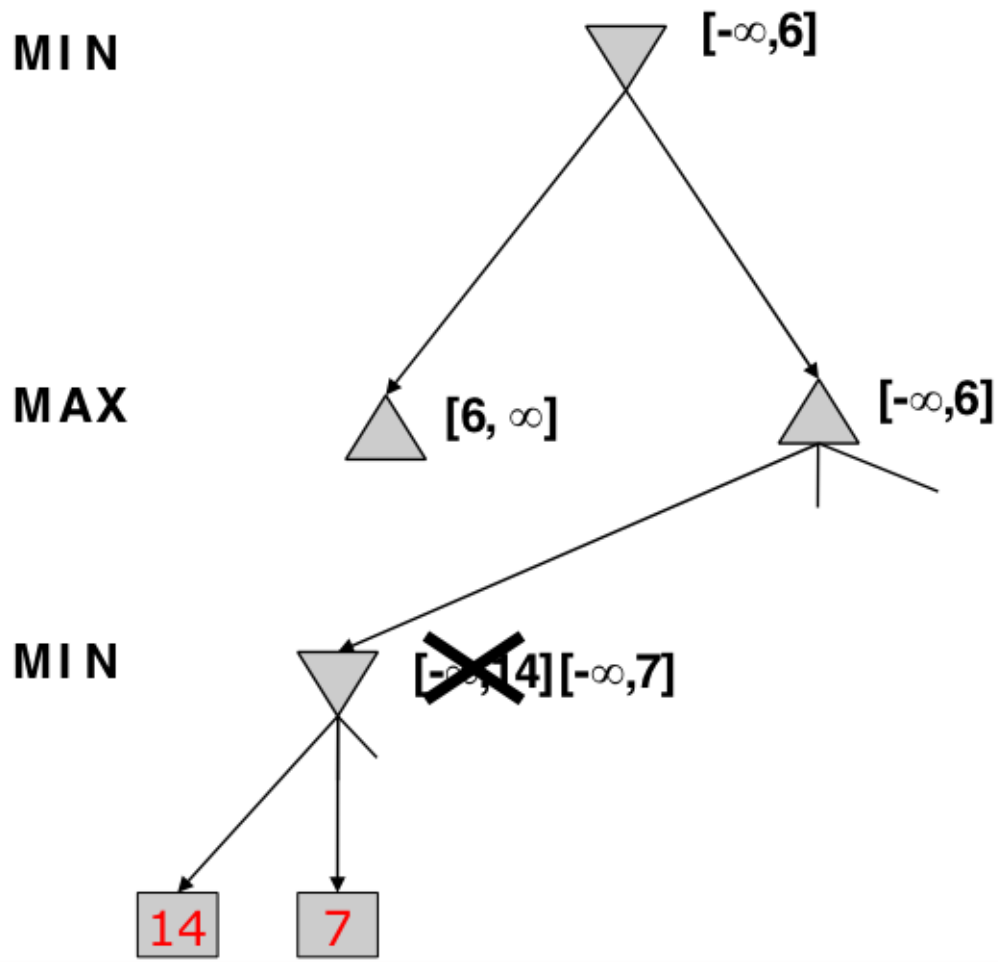
α-β pruning: example 4



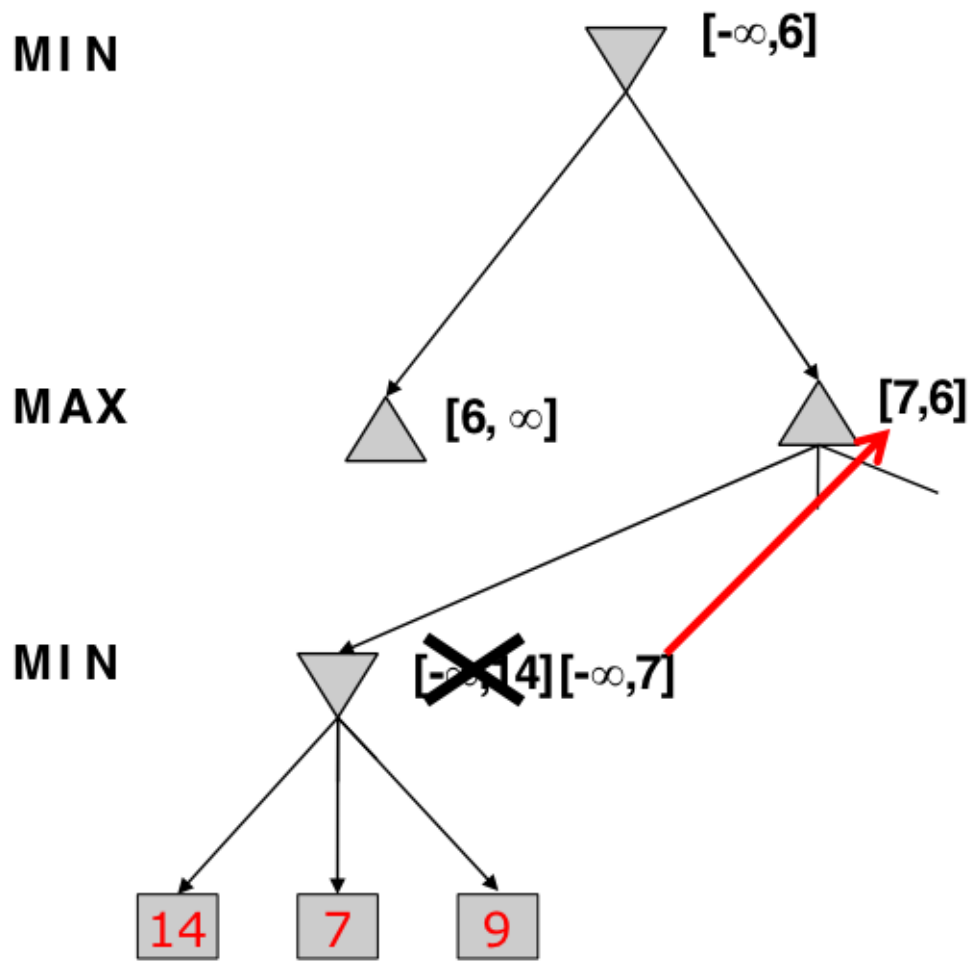
α-β pruning: example 4



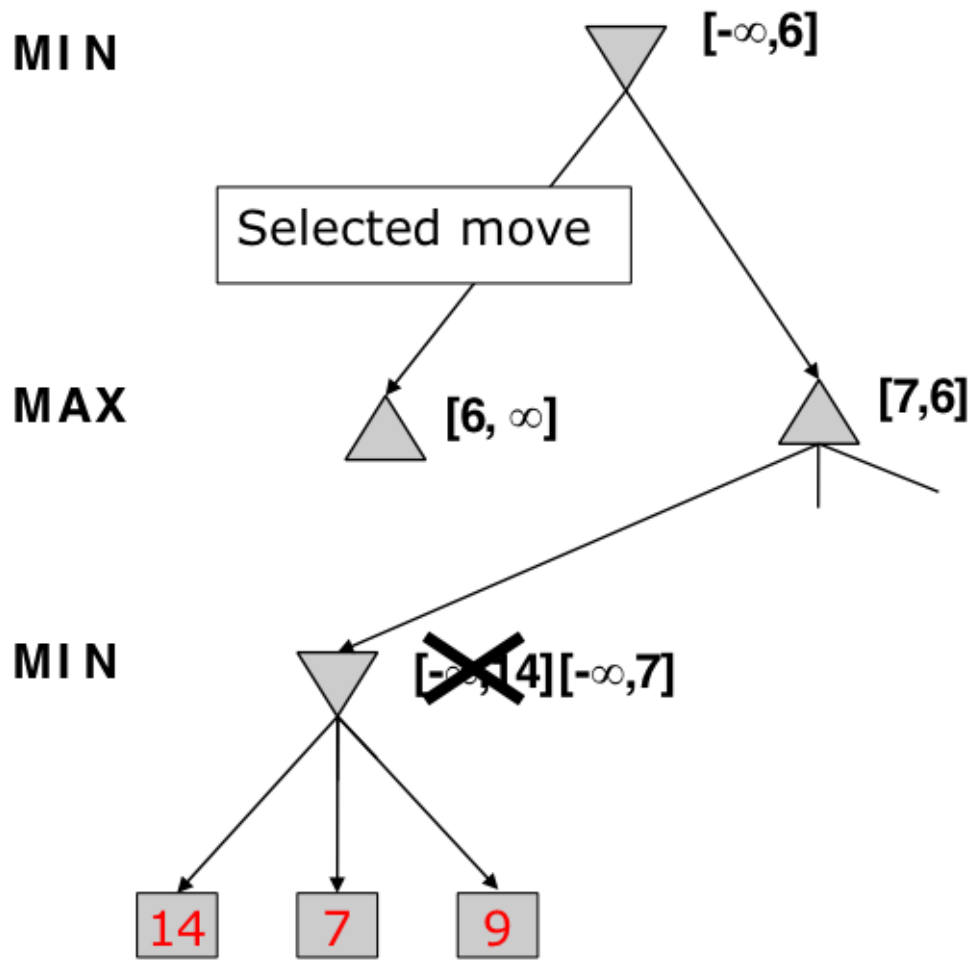
α-β pruning: example 4



α-β pruning: example 4



α-β pruning: example 4

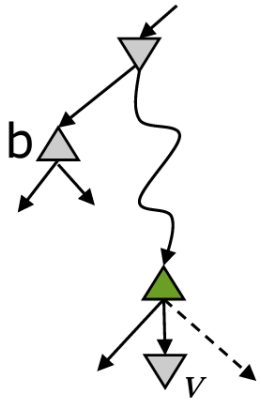


The α-β algorithm

```
function ALPHA-BETA-DECISION(state) returns an action
   return the a in ACTIONS(state) maximizing MIN-VALUE(RESULT(a, state))
```

```
function MAX-VALUE(state, α, β) returns a utility value
   inputs: state, current state in game
           α, the value of the best alternative for MAX along the path to state
           β, the value of the best alternative for MIN along the path to state

   if TERMINAL-TEST(state) then return UTILITY(state)
   v ← −∞
   for a, s in SUCCESSORS(state) do
      v ← MAX(v, MIN-VALUE(s, α, β))
      if v ≥ β then return v
      α ← MAX(α, v)
   return v
```



```
function MIN-VALUE(state, α, β) returns a utility value
   same as MAX-VALUE but with roles of α, β reversed
```

Properties of α-β pruning

Pruning **does not** affect final result

Good move ordering improves effectiveness of pruning

With "perfect ordering," time complexity $= O(b^{m/2})$
$\Rightarrow$ **doubles** solvable depth

A simple example of the value of reasoning about which computations are relevant (a form of metareasoning)

Unfortunately, 35^{50} is still impossible!