CS171 – Miderm Review

Fall 2016

Overview

- Basics
  - Agents, environments, ...
- Search
  - Blind, heuristic, local, …
- Adversarial search
  - Minimax, alpha-beta pruning, …
- Constraint satisfaction problems
  - Setup, types; backtracking search; search orders; forward checking & arc consistency; local search

Please review past exams for practice. Note that some quarters the material is offered in a different order, so in those quarters there may be irrelevant problems in the midterm; you may find relevant ones in the final exam.

Review Agents
Chapter 2.1-2.3

- Agent definition (2.1)
- Rational Agent definition (2.2)
  - Performance measure
- Task environment definition (2.3)
  - PEAS acronym
- Properties of Task Environments
  - Fully vs. partially observable; single vs. multi agent; deterministic vs. stochastic; episodic vs. sequential; static vs. dynamic; discrete vs. continuous; known vs. unknown
- Basic Definitions
  - Percept, percept sequence, agent function, agent program

Agents

- An agent is anything that can be viewed as perceiving its environment through sensors and acting upon that environment through actuators

- Human agent:

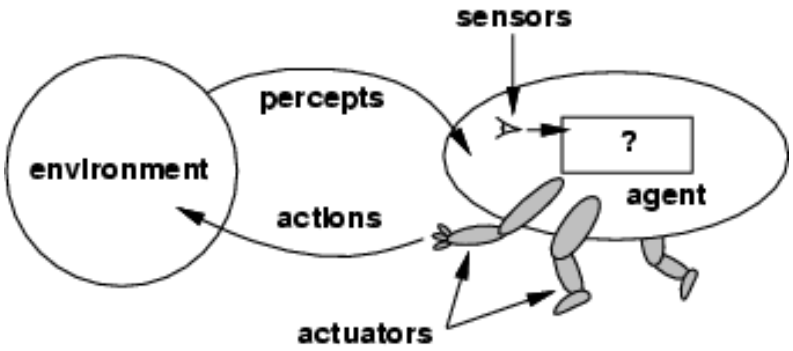
  eyes, ears, and other organs for sensors;

  hands, legs, mouth, and other body parts for actuators

- Robotic agent:

  cameras and infrared range finders for sensors; various motors for actuators

Agents and environments



- Percept: agent's perceptual inputs at an instant

- The agent function maps from percept sequences to actions:

$$[f: \mathcal{P}^{\star} \rightarrow \mathcal{A}]$$

- The agent program runs on the physical architecture to produce *f*

- agent = architecture + program

Rational agents

- Rational Agent: For each possible percept sequence, a rational agent should select an action that is *expected* to maximize its performance measure, based on the evidence provided by the percept sequence and whatever built-in knowledge the agent has.

- Performance measure: An objective criterion for success of an agent's behavior

- E.g., performance measure of a vacuum-cleaner agent could be amount of dirt cleaned up, amount of time taken, amount of electricity consumed, amount of noise generated, etc.

Task Environment

- Before we design an intelligent agent, we must specify its “task environment”:

PEAS:

Performance measure

Environment

Actuators

Sensors

Environment types

- Fully observable (vs. partially observable): An agent's sensors give it access to the complete state of the environment at each point in time.

- Deterministic (vs. stochastic): The next state of the environment is completely determined by the current state and the action executed by the agent. (If the environment is deterministic except for the actions of other agents, then the environment is strategic)

- Episodic (vs. sequential): An agent's action is divided into atomic episodes. Decisions do not depend on previous decisions/actions.

- Known (vs. unknown): An environment is considered to be "known" if the agent understands the laws that govern the environment's behavior.

Environment types

- Static (vs. dynamic): The environment is unchanged while an agent is deliberating. (The environment is semidynamic if the environment itself does not change with the passage of time but the agent's performance score does)

- Discrete (vs. continuous): A limited number of distinct, clearly defined percepts and actions.

  How do we represent or abstract or model the world?

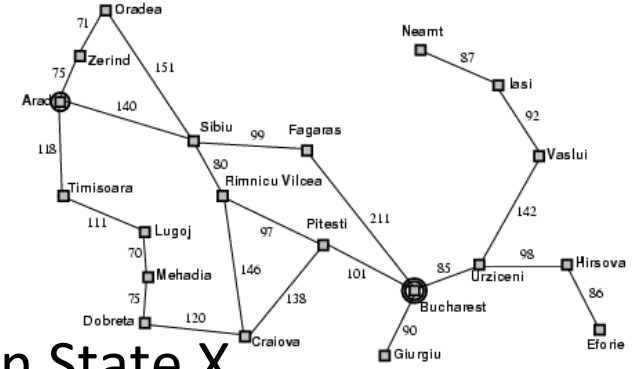
- Single agent (vs. multi-agent): An agent operating by itself in an environment. Does the other agent interfere with my performance measure?

Review State Space Search Chapters 3-4

- Problem Formulation (3.1, 3.3)
- Blind (Uninformed) Search (3.4)
  - Depth-First, Breadth-First, Iterative Deepening
  - Uniform-Cost, Bidirectional (if applicable)
  - Time? Space? Complete? Optimal?
- Heuristic Search (3.5)
  - A*, Greedy-Best-First
- Local Search (4.1, 4.2)
  - Hill-climbing, Simulated Annealing, Genetic Algorithms
  - Gradient descent

Problem Formulation

A problem is defined by five items:



initial state e.g., "at Arad"

actions

- Actions(X) = set of actions available in State X

transition model

- Result(S,A) = state resulting from doing action A in state S

goal test, e.g., *x* = "at Bucharest", *Checkmate(x)*

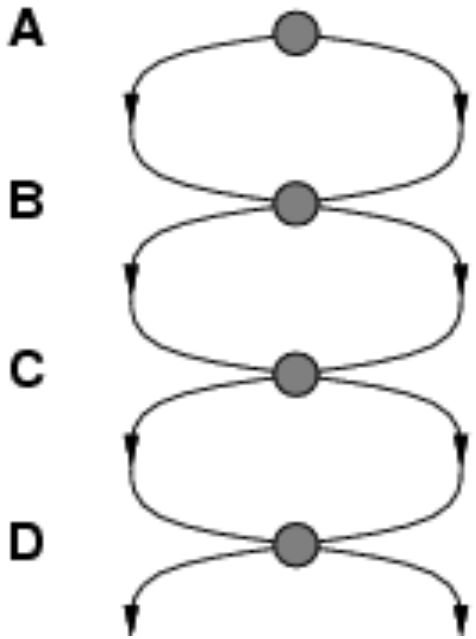
path cost (additive, i.e., the sum of the step costs)

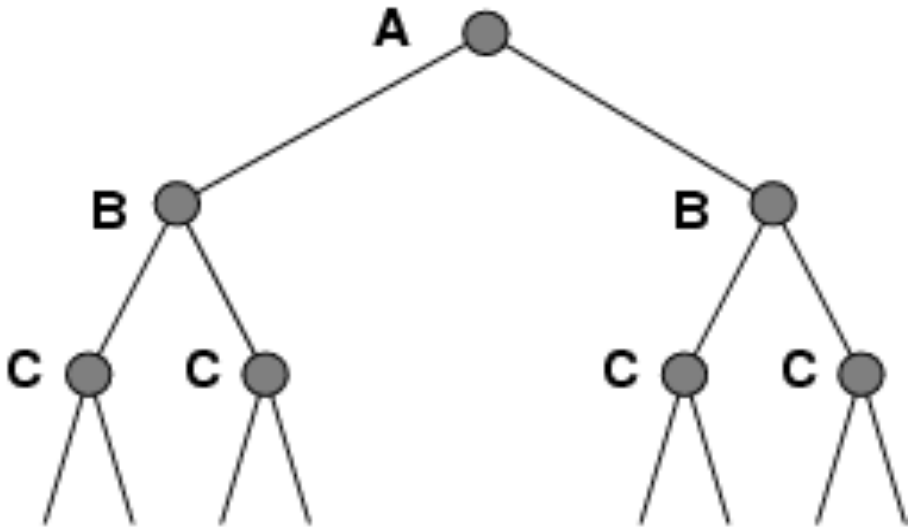
- *c(x,a,y)* = step cost of action *a* in state *x* to reach state *y*
  - assumed to be ≥ 0

A solution is a sequence of actions leading from the initial state to a goal state

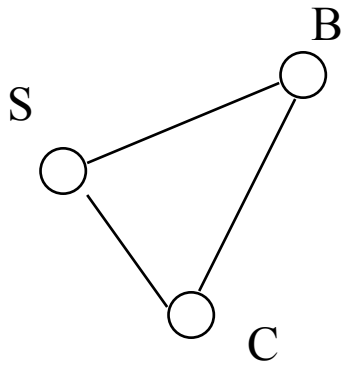
Tree search vs. Graph search
Review Fig. 3.7, p. 77

- Failure to detect repeated states can turn a linear problem into an exponential one!
- Test is often implemented as a hash table.

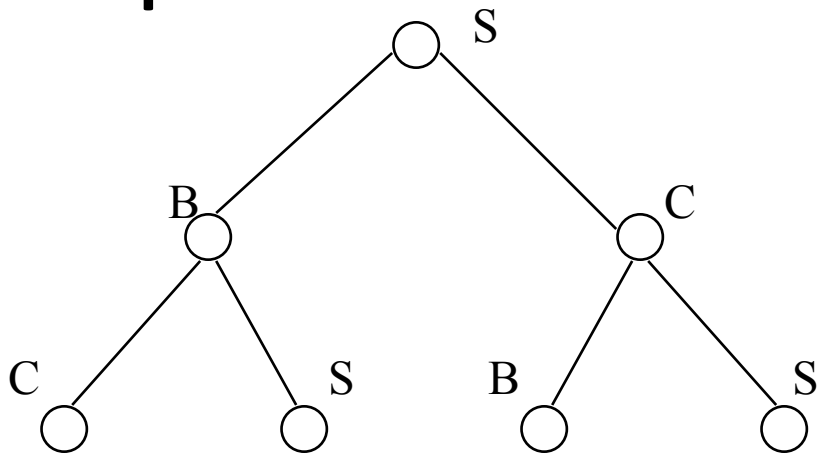




Solutions to Repeated States



State Space



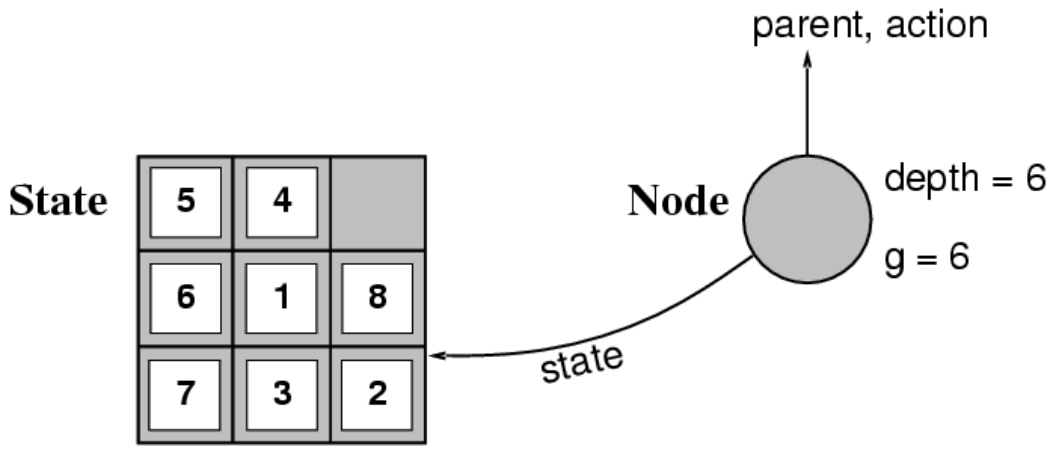
Example of a Search Tree

- Graph search ← faster, but memory inefficient
  - never generate a state generated before
    - must keep track of all possible states (uses a lot of memory)
    - e.g., 8-puzzle problem, we have 9! = 362,880 states
    - approximation for DFS/DLS: only avoid states in its (limited) memory: avoid infinite loops by checking path back to root.
  - "visited?" test usually implemented as a hash table

Implementation: states vs. nodes

- A state is a (representation of) a physical configuration

- A node is a data structure constituting part of a search tree
- A node contains info such as:
  - state, parent node, action, path cost *g(x)*, depth, etc.



- The `Expand` function creates new nodes, filling in the various fields using the `Actions(S)` and `Result(S,A)` functions associated with the problem.

General tree search

```
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](STATE[node]) then return SOLUTION(node)
        fringe ← INSERTALL(EXPAND(node, problem), fringe)

function EXPAND( node, problem) returns a set of nodes
    successors ← the empty set
    for each action, result in SUCCESSOR-FN[problem](STATE[node]) do
        s ← a new NODE
        PARENT-NODE[s] ← node;  ACTION[s] ← action;  STATE[s] ← result
        PATH-COST[s] ← PATH-COST[node] + STEP-COST(node, action, s)
        DEPTH[s] ← DEPTH[node] + 1
        add s to successors
    return successors
```

Goal test after pop

General graph search

```
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 SOLUTION(node)
        if STATE[node] is not in closed then
            add STATE[node] to closed
            fringe ← INSERTALL(EXPAND(node, problem), fringe)
```

Goal test after pop

Breadth-first graph search

```
function BREADTH-FIRST-SEARCH(problem) returns a solution, or failure
    node ← a node with STATE = problem.INITIAL-STATE, PATH-COST = 0 if
    problem.GOAL-TEST(node.STATE) then return SOLUTION(node) frontier ←
    a FIFO queue with node as the only element
    explored ← an empty set
    loop do
        if EMPTY?(frontier) then return failure
        node ← POP(frontier)   /* chooses the shallowest node in frontier */
        add node.STATE to explored
        for each action in problem.ACTIONS(node.STATE) do
            child ← CHILD-NODE(problem, node, action)
            if child.STATE is not in explored or frontier then
                if problem.GOAL-TEST(child.STATE) then return SOLUTION(child)
                frontier ← INSERT(child, frontier)
```

Goal test before push

Figure 3.11 Breadth-first search on a graph.

Uniform cost search: sort by *g*

A* is identical but uses *f=g+h*

Greedy best-first is identical but uses *h*

```
function UNIFORM-COST-SEARCH(problem) returns a solution, or failure
    node ← a node with STATE = problem.INITIAL-STATE, PATH-COST = 0
    frontier ← a priority queue ordered by PATH-COST, with node as the only element
    explored ← an empty set
    loop do
        if EMPTY?(frontier) then return failure
        node ← POP(frontier)   /* chooses the lowest-cost node in frontier */
        if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
        add node.STATE to explored
        for each action in problem.ACTIONS(node.STATE) do
            child ← CHILD-NODE(problem, node, action)
            if child.STATE is not in explored or frontier then
                frontier ← INSERT(child, frontier)
            else if child.STATE is in frontier with higher PATH-COST then
                replace that frontier node with child
```

Goal test after pop

Figure 3.14 Uniform-cost search on a graph. The algorithm is identical to the general graph search algorithm in Figure 3.7, except for the use of a priority queue and the addition of an extra check in case a shorter path to a frontier state is discovered. The data structure for frontier needs to support efficient membership testing, so it should combine the capabilities of a priority queue and a hash table.

Depth-limited search & IDS

```
function DEPTH-LIMITED-SEARCH( problem, limit) returns soln/fail/cutoff
  RECURSIVE-DLS(MAKE-NODE(INITIAL-STATE[problem]), problem, limit)

function RECURSIVE-DLS(node, problem, limit) returns soln/fail/cutoff
  cutoff-occurred? ← false
  if GOAL-TEST[problem](STATE[node]) then return SOLUTION(node)
  else if DEPTH[node] = limit then return cutoff
  else for each successor in EXPAND(node, problem) do
      result ← RECURSIVE-DLS(successor, problem, limit)
      if result = cutoff then cutoff-occurred? ← true
      else if result ≠ failure then return result
  if cutoff-occurred? then return cutoff else return failure
```

Goal test before push

```
function ITERATIVE-DEEPENING-SEARCH( problem) returns a solution, or fail-
ure
    inputs: problem, a problem
    for depth ← 0 to ∞ do
      result ← DEPTH-LIMITED-SEARCH( problem, depth)
      if result ≠ cutoff then return result
```

When to do Goal-Test? Summary

- For DFS, BFS, DLS, and IDS, the goal test is done when the child node is generated.
  - These are not optimal searches in the general case.
  - BFS and IDS are optimal if cost is a function of depth only; then, optimal goals are also shallowest goals and so will be found first

- For GBFS the behavior is the same whether the goal test is done when the node is generated or when it is removed
  - h(goal)=0 so any goal will be at the front of the queue anyway.

- For UCS and A* the goal test is done when the node is removed from the queue.
  - This precaution avoids finding a short expensive path before a long cheap path.

Blind Search Strategies (3.4)

- Depth-first: Add successors to front of queue
- Breadth-first: Add successors to back of queue
- Uniform-cost: Sort queue by path cost g(n)
- Depth-limited: Depth-first, cut off at limit *l*
- Iterated-deepening: Depth-limited, increasing *l*
- Bidirectional: Breadth-first from goal, too.

- **Review example Uniform-cost search**
  - Slides 25-34, Lecture on "Uninformed Search"

Search strategy evaluation

- A search strategy is defined by the order of node expansion

- Strategies are evaluated along the following dimensions:
  - completeness: does it always find a solution if one exists?
  - time complexity: number of nodes generated
  - space complexity: maximum number of nodes in memory
  - optimality: does it always find a least-cost solution?

- Time and space complexity are measured in terms of
  - *b:* maximum branching factor of the search tree
  - *d:* depth of the least-cost solution
  - *m*: maximum depth of the state space (may be ∞)
  - (for UCS: C*: true cost to optimal goal; $\varepsilon > 0$: minimum step cost)

Summary of algorithms
Fig. 3.21, p. 91

Criterion	Breadth-First	Uniform-Cost	Depth-First	Depth-Limited	Iterative Deepening DLS	Bidirectional (if applicable)
Complete?	Yes[a]	Yes[a,b]	No	No	Yes[a]	Yes[a,d]
Time	$O(b^d)$	$O(b^{\lfloor 1+C^*/\varepsilon \rfloor})$	$O(b^m)$	$O(b^l)$	$O(b^d)$	$O(b^{d/2})$
Space	$O(b^d)$	$O(b^{\lfloor 1+C^*/\varepsilon \rfloor})$	$O(bm)$	$O(bl)$	$O(bd)$	$O(b^{d/2})$
Optimal?	Yes[c]	Yes	No	No	Yes[c]	Yes[c,d]

There are a number of footnotes, caveats, and assumptions.
See Fig. 3.21, p. 91.

Generally the preferred uninformed search strategy

[a] complete if b is finite
[b] complete if step costs $\geq \varepsilon > 0$
[c] optimal if step costs are all identical
(also if path cost non-decreasing function of depth only)
[d] if both directions use breadth-first search
(also if both directions use uniform-cost search with step costs $\geq \varepsilon > 0$)

Heuristic function (3.5)

- Heuristic:
  - Definition: a commonsense rule (or set of rules) intended to increase the probability of solving some problem
  - "using rules of thumb to find answers"

- Heuristic function h(n)
  - Estimate of (optimal) cost from n to goal
  - Defined using only the *state* of node *n*
  - h(n) = 0 if n is a goal node
  - Example: straight line distance from n to Bucharest
    - Note that this is not the true state-space distance
    - It is an estimate – actual state-space distance can be higher

  - Provides problem-specific knowledge to the search algorithm

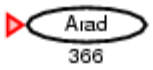
Greedy best-first search
(often called just "best-first")

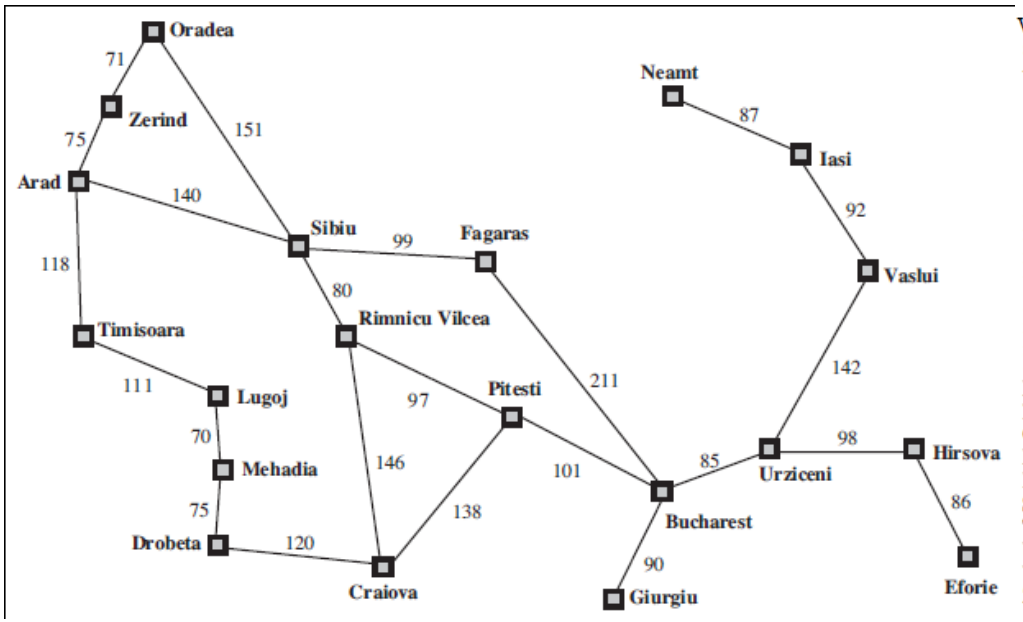
- *h(n)* = estimate of cost from *n* to *goal*
  - e.g., *h(n)* = straight-line distance from *n* to Bucharest

- Greedy best-first search expands the node that appears to be closest to goal.
  - Sort queue by *h(n)*

- Not an optimal search strategy
  - May perform well in practice

Greedy best-first search example

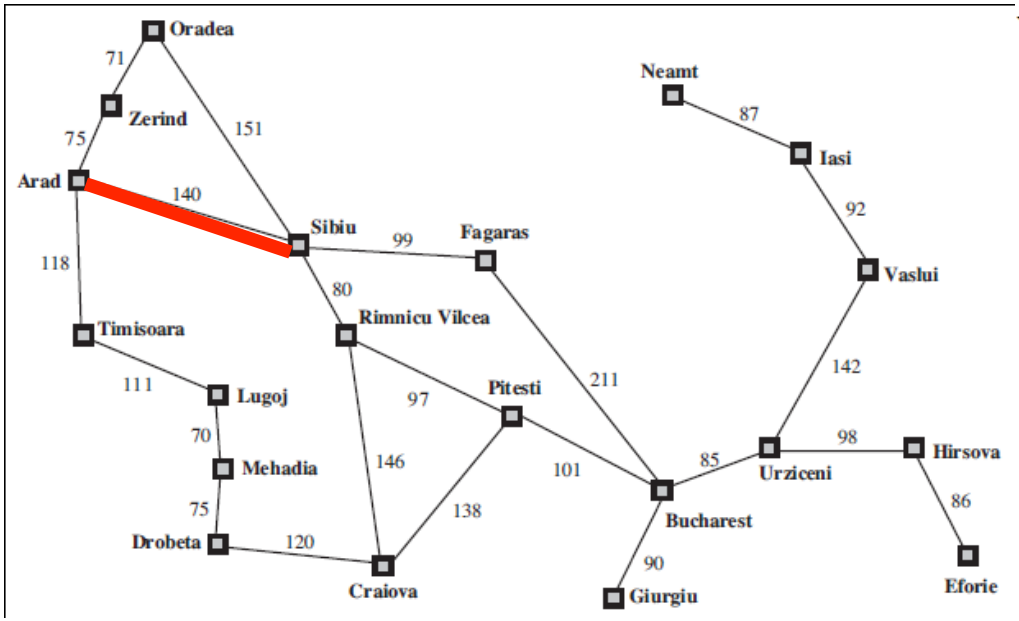




Values of h_{SLD}—straight-line distances to Bucharest.

Arad	366
Bucharest	0
Craiova	160
Drobeta	242
Eforie	161
Fagaras	176
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

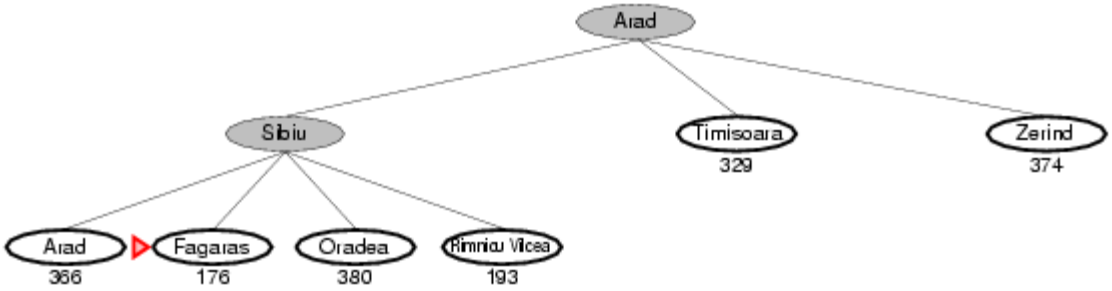
Greedy best-first search example

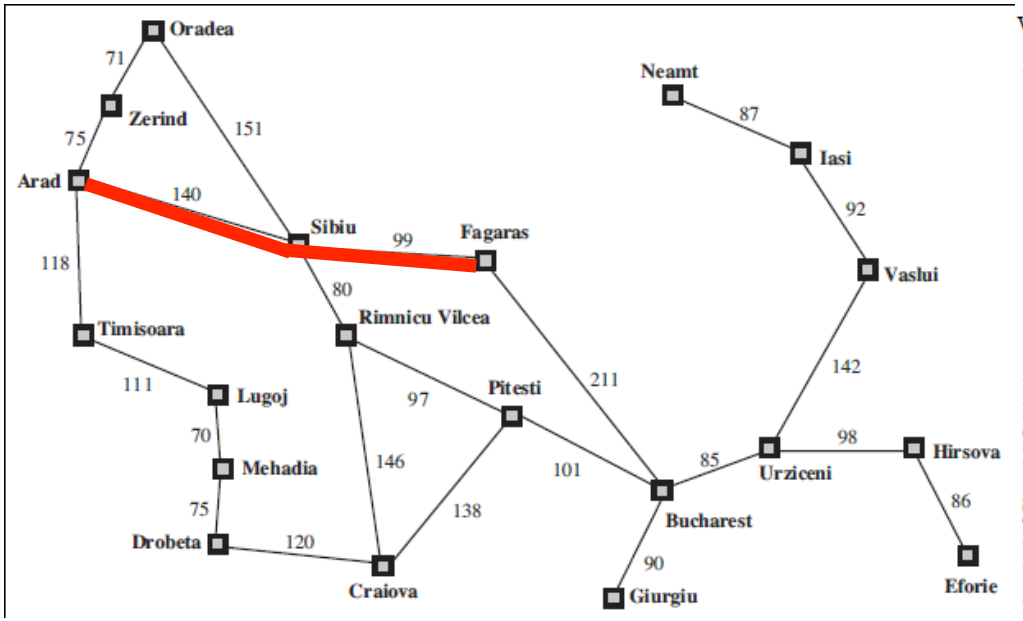


Values of h_{SLD}—straight-line distances to Bucharest.

Arad	366
Bucharest	0
Craiova	160
Drobeta	242
Eforie	161
Fagaras	176
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 best-first search example

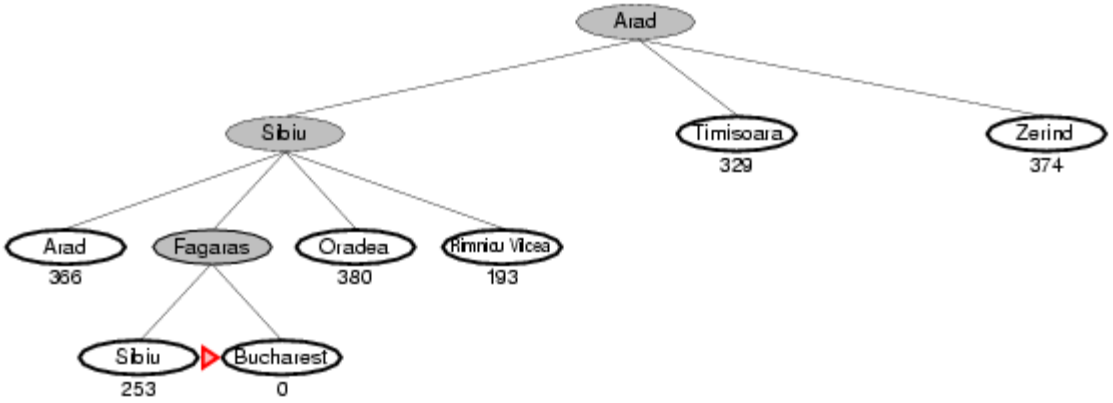


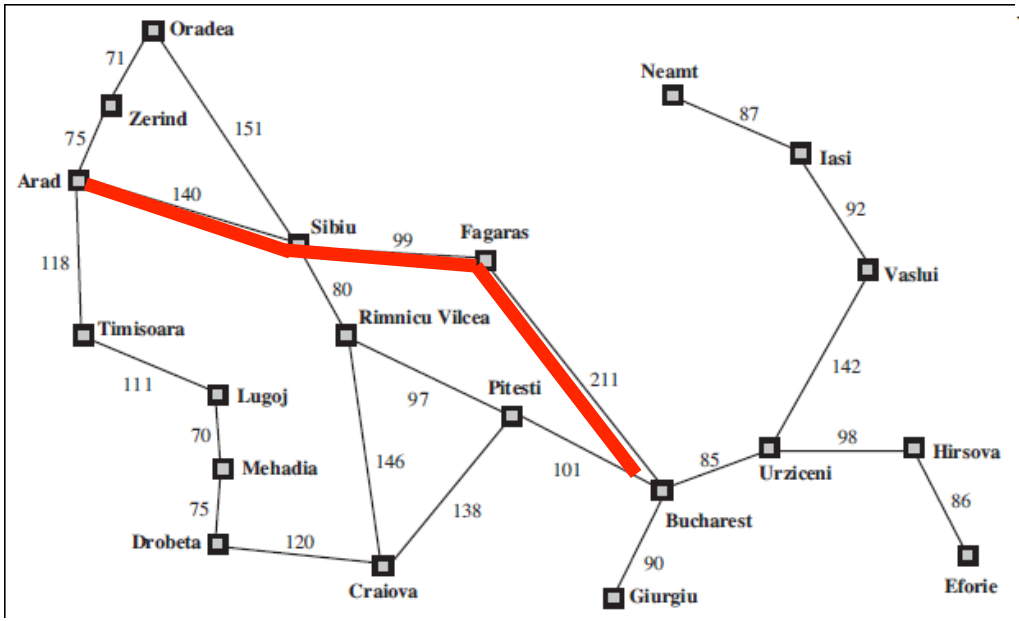


Values of h_{SLD}—straight-line distances to Bucharest.

Arad	366
Bucharest	0
Craiova	160
Drobeta	242
Eforie	161
Fagaras	176
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 best-first search example

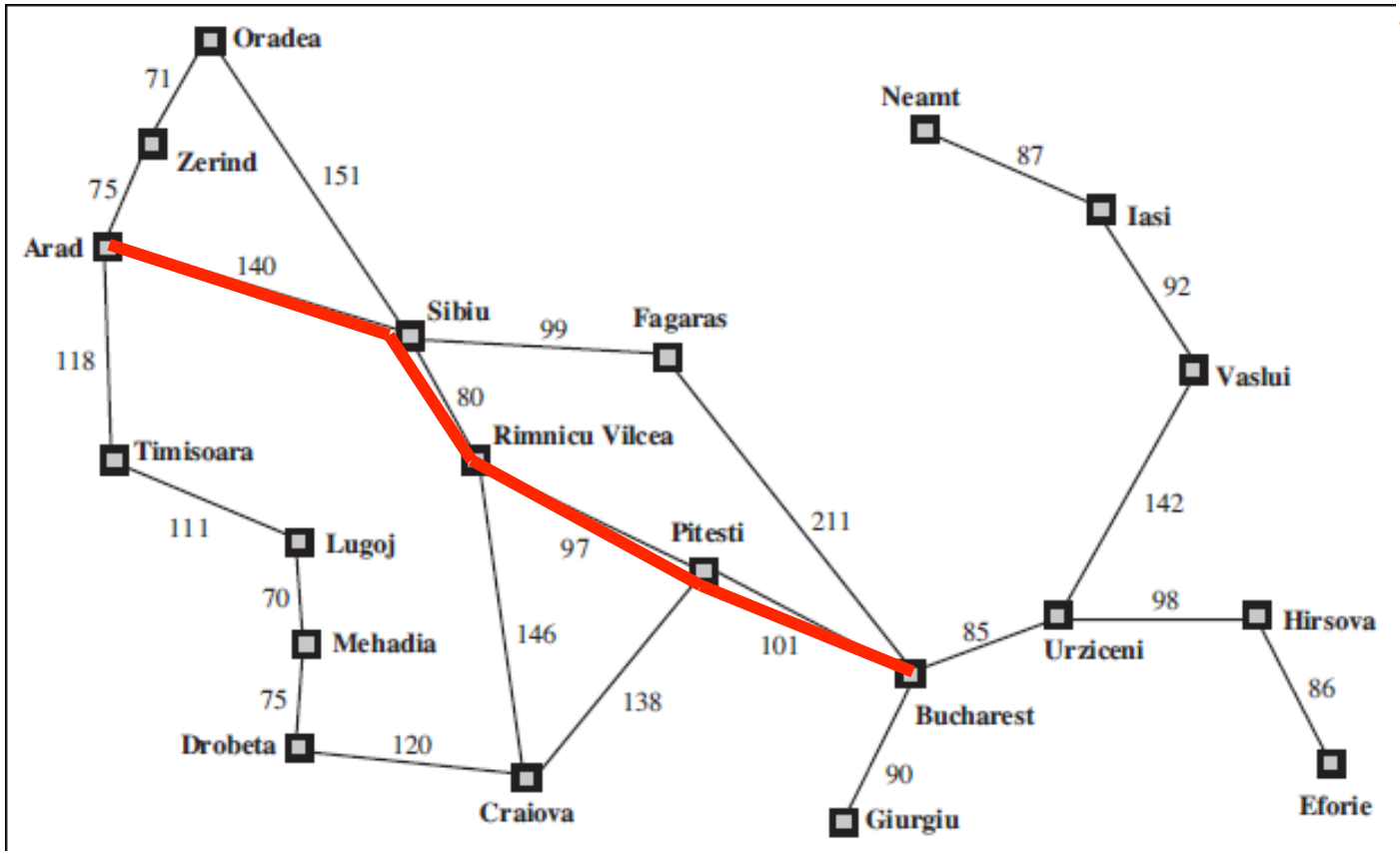




Values of h_{SLD}—straight-line distances to Bucharest.

Arad	366
Bucharest	0
Craiova	160
Drobeta	242
Eforie	161
Fagaras	176
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

Optimal Path



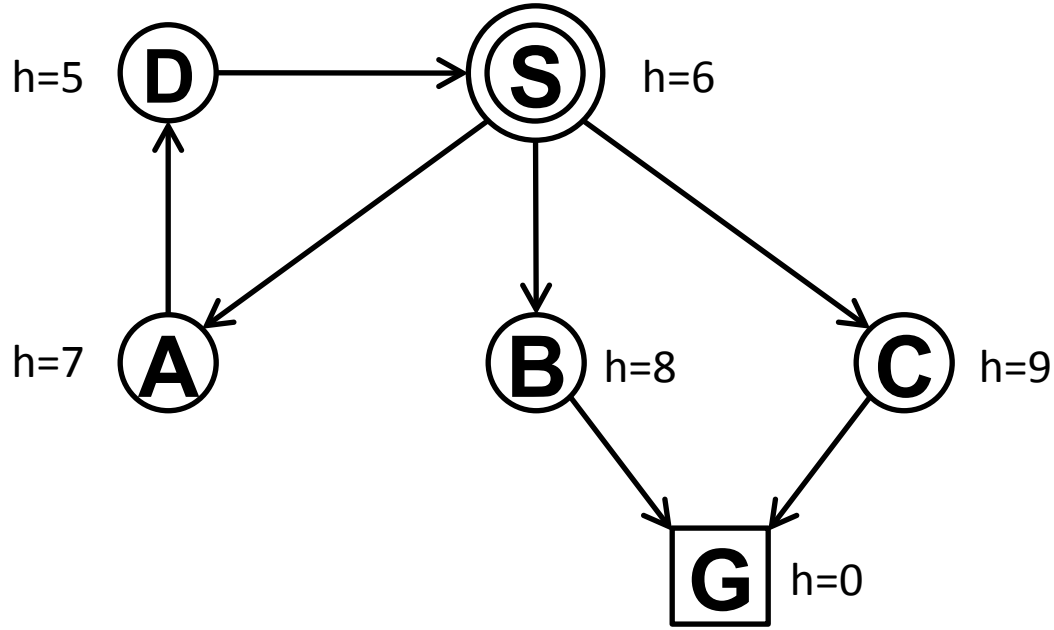
Values of h_{SLD}—straight-line distances to Bucharest.

Arad	366
Bucharest	0
Craiova	160
Drobeta	242
Eforie	161
Fagaras	176
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 Best-first Search

With tree search, will become stuck in this loop

Order of node expansion: S A D S A D S A D. . . .
Path found: none Cost of path found: none .



Properties of greedy best-first search

- Complete?
  - Tree version can get stuck in loops.
  - Graph version is complete in finite spaces.
- Time? $O(b^m)$
  - A good heuristic can give **dramatic** improvement
- Space? $O(1)$ tree search, $O(b^m)$ graph search
  - Graph search keeps all nodes in memory
  - A good heuristic can give **dramatic** improvement
- Optimal? No
  - E.g., Arad → Sibiu → Rimnicu Vilcea → Pitesti → Bucharest is shorter!

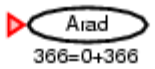
A* search

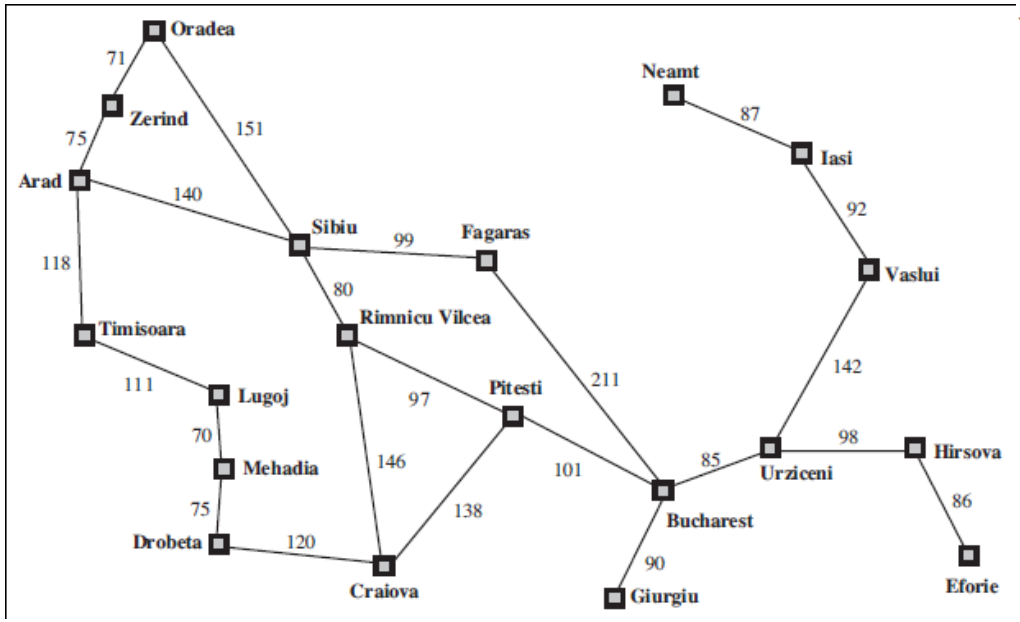
- Idea: avoid paths that are already expensive
  - Generally the preferred simple heuristic search
  - Optimal if heuristic is:

    admissible (tree search)/consistent (graph search)
- Evaluation function *f(n) = g(n) + h(n)*
  - g(n) = known path cost so far to node n.
  - h(n) = estimate of (optimal) cost to goal from node n.
  - f(n) = g(n)+h(n)

    = estimate of total cost to goal through node n.
- *Priority queue sort function = f(n)*

A* tree search example





Values of h_{SLD}—straight-line distances to Bucharest.

Arad	366
Bucharest	0
Craiova	160
Drobeta	242
Eforie	161
Fagaras	176
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* tree search example: Simulated queue. City/f=g+h

- Next:
- Children:
- Expanded:
- Frontier: Arad/366=0+366

A* tree search example: Simulated queue. City/f=g+h

Arad/
366=0+366

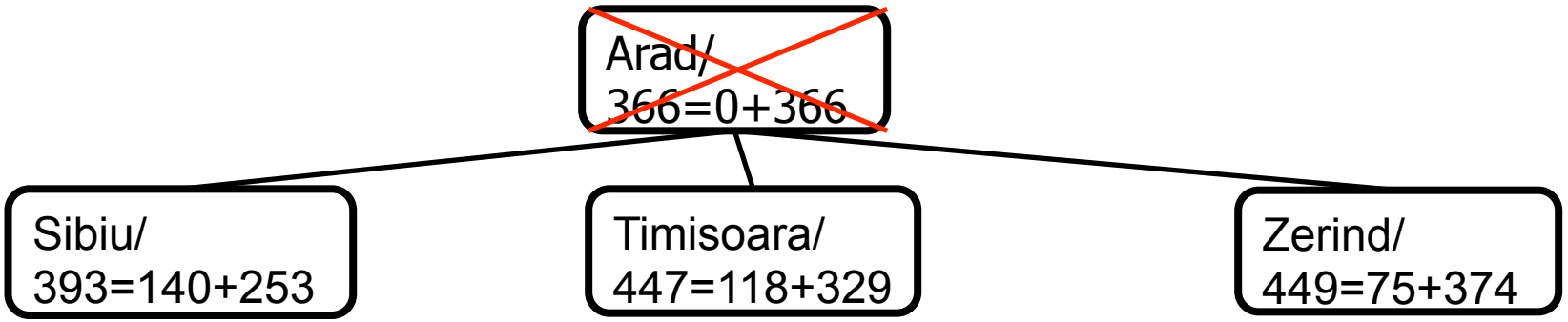
A* tree search example: Simulated queue. City/f=g+h

Arad/
366=0+366

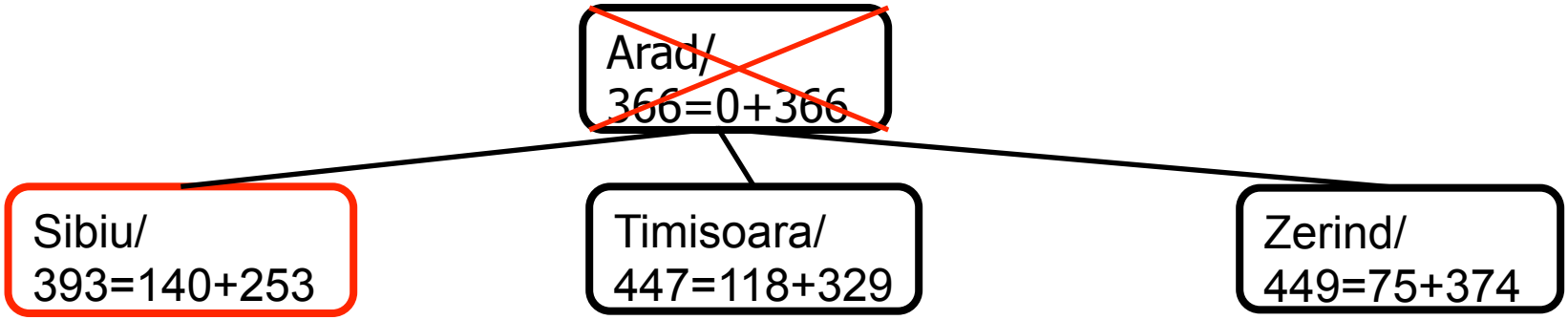
A* tree search example: Simulated queue. City/f=g+h

- Next: Arad/366=0+366
- Children: Sibiu/393=140+253, Timisoara/447=118+329, Zerind/449=75+374
- Expanded: Arad/366=0+366
- Frontier: ~~Arad/366=0+366~~, Sibiu/393=140+253, Timisoara/447=118+329, Zerind/449=75+374

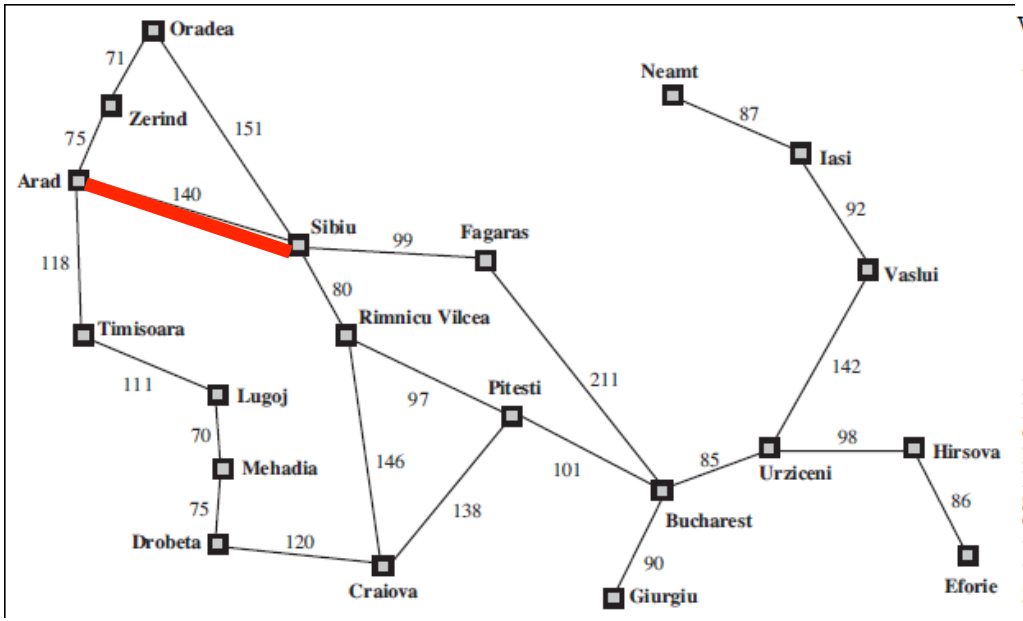
A* tree search example: Simulated queue. City/f=g+h



A* tree search example: Simulated queue. City/f=g+h



A* tree search example



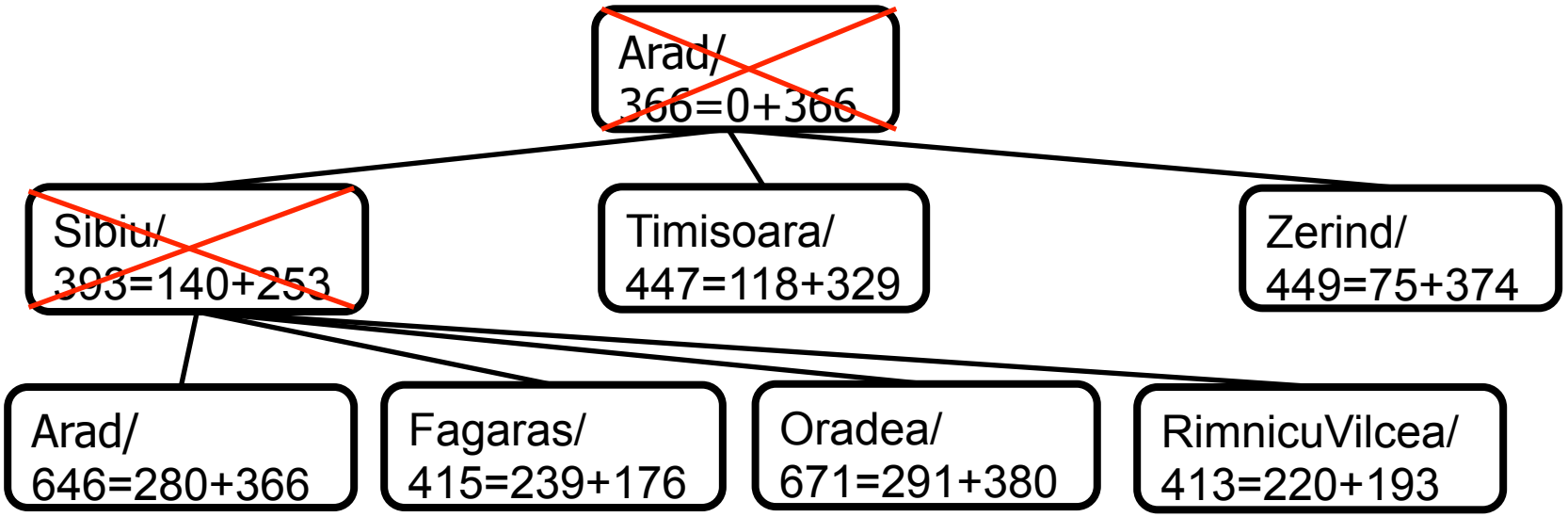
Values of h_{SLD}—straight-line distances to Bucharest.

Arad	366
Bucharest	0
Craiova	160
Drobeta	242
Eforie	161
Fagaras	176
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

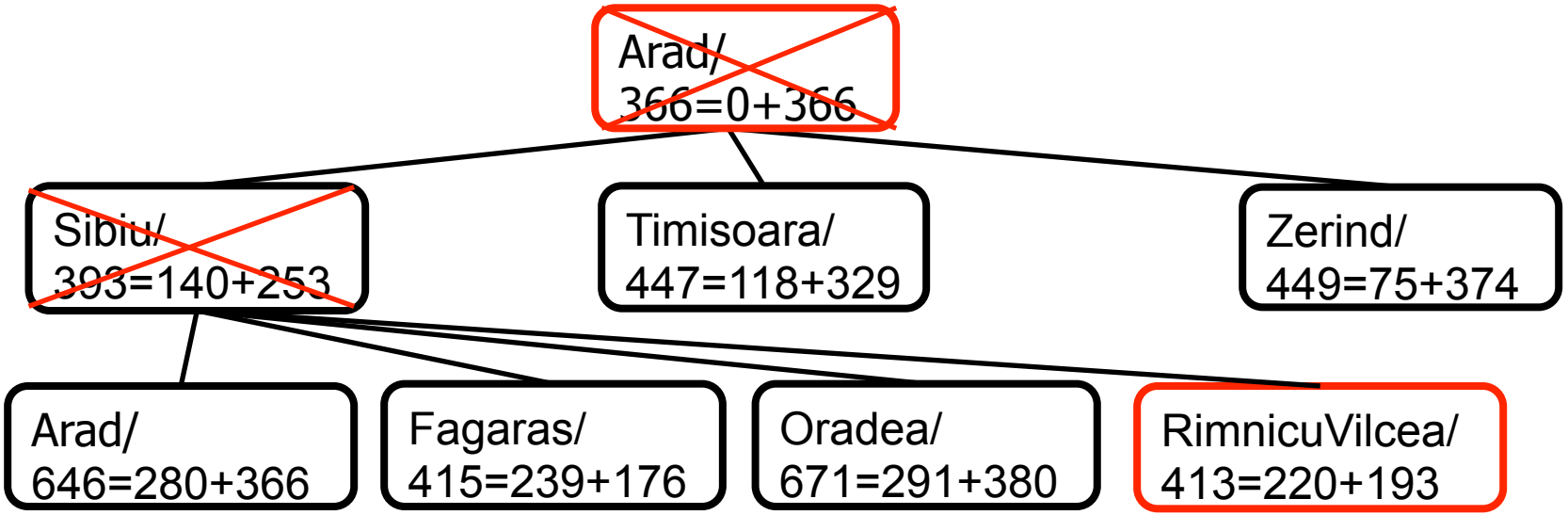
A* tree search example: Simulated queue. City/f=g+h

- Next: Sibiu/393=140+253
- Children: Arad/646=280+366, Fagaras/415=239+176, Oradea/671=291+380, RimnicuVilcea/413=220+193
- Expanded: Arad/366=0+366, Sibiu/393=140+253
- Frontier: ~~Arad/366=0+366, Sibiu/393=140+253,~~ Timisoara/447=118+329, Zerind/449=75+374, Arad/646=280+366, Fagaras/415=239+176, Oradea/671=291+380, RimnicuVilcea/413=220+193

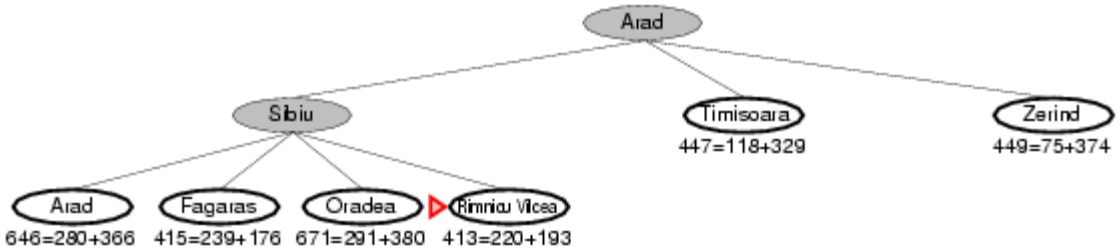
A* tree search example: Simulated queue. City/f=g+h

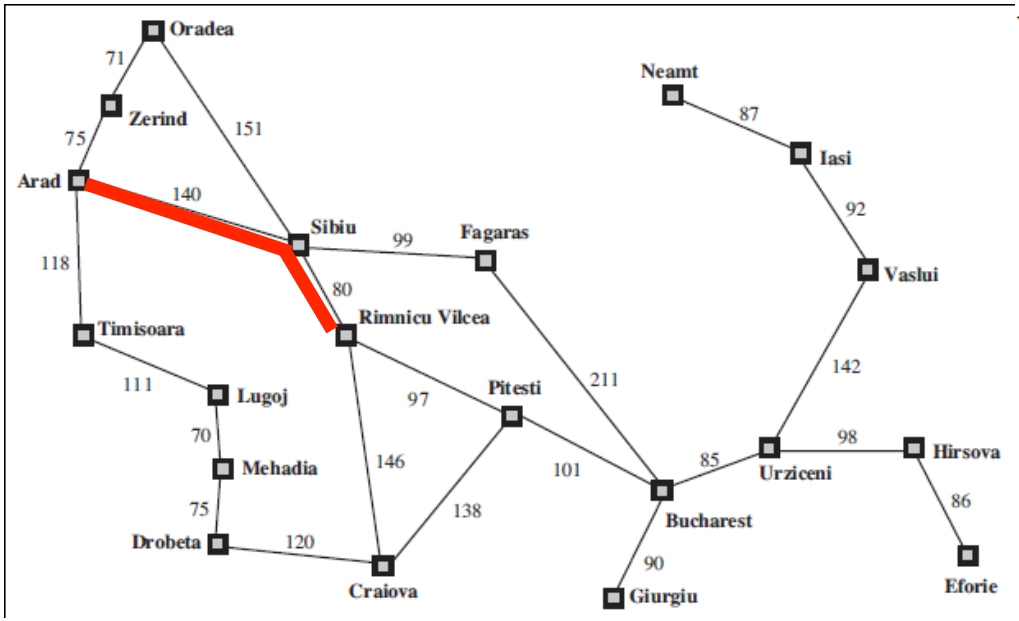


A* tree search example: Simulated queue. City/f=g+h



A* tree search example





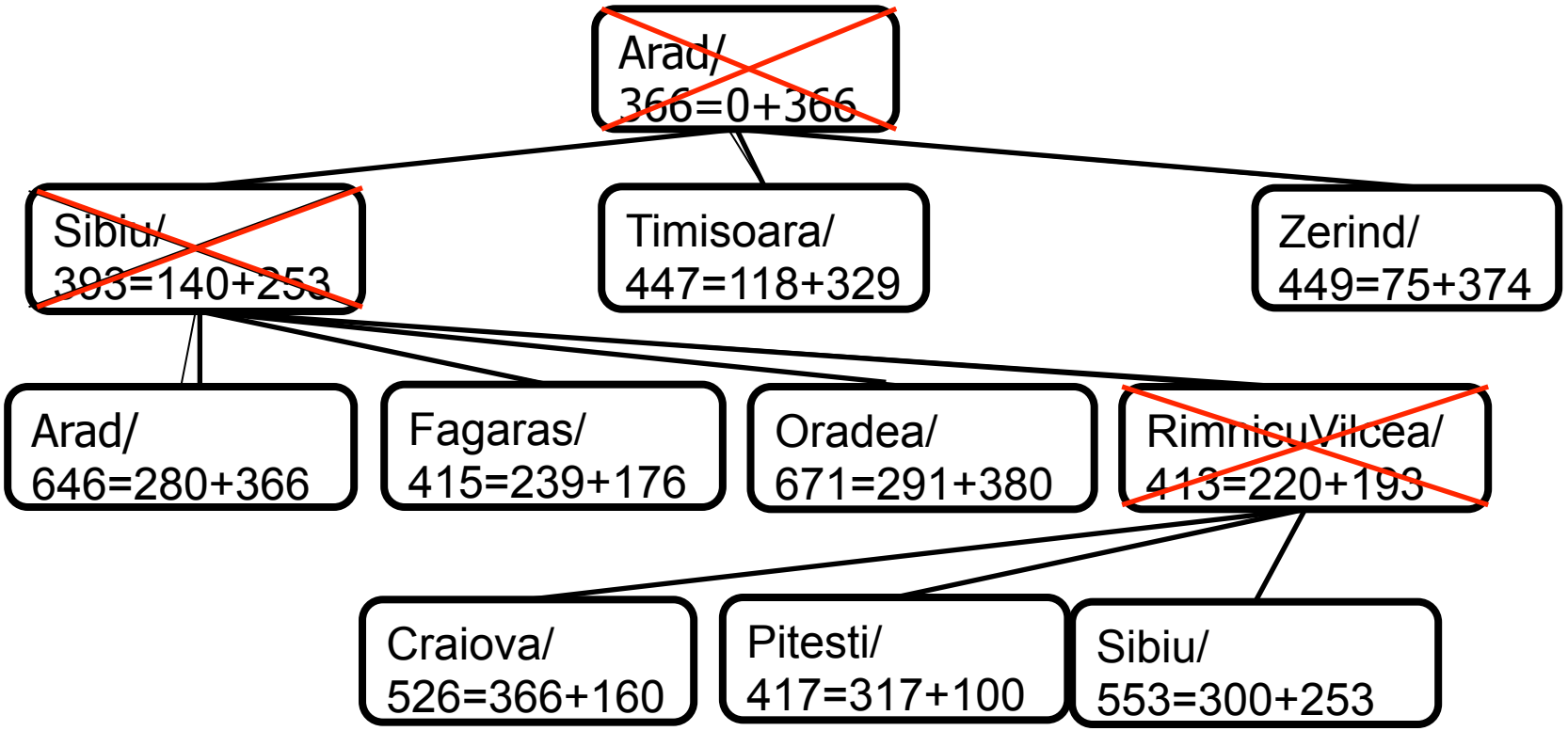
Values of h_{SLD}—straight-line distances to Bucharest.

Arad	366
Bucharest	0
Craiova	160
Drobeta	242
Eforie	161
Fagaras	176
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

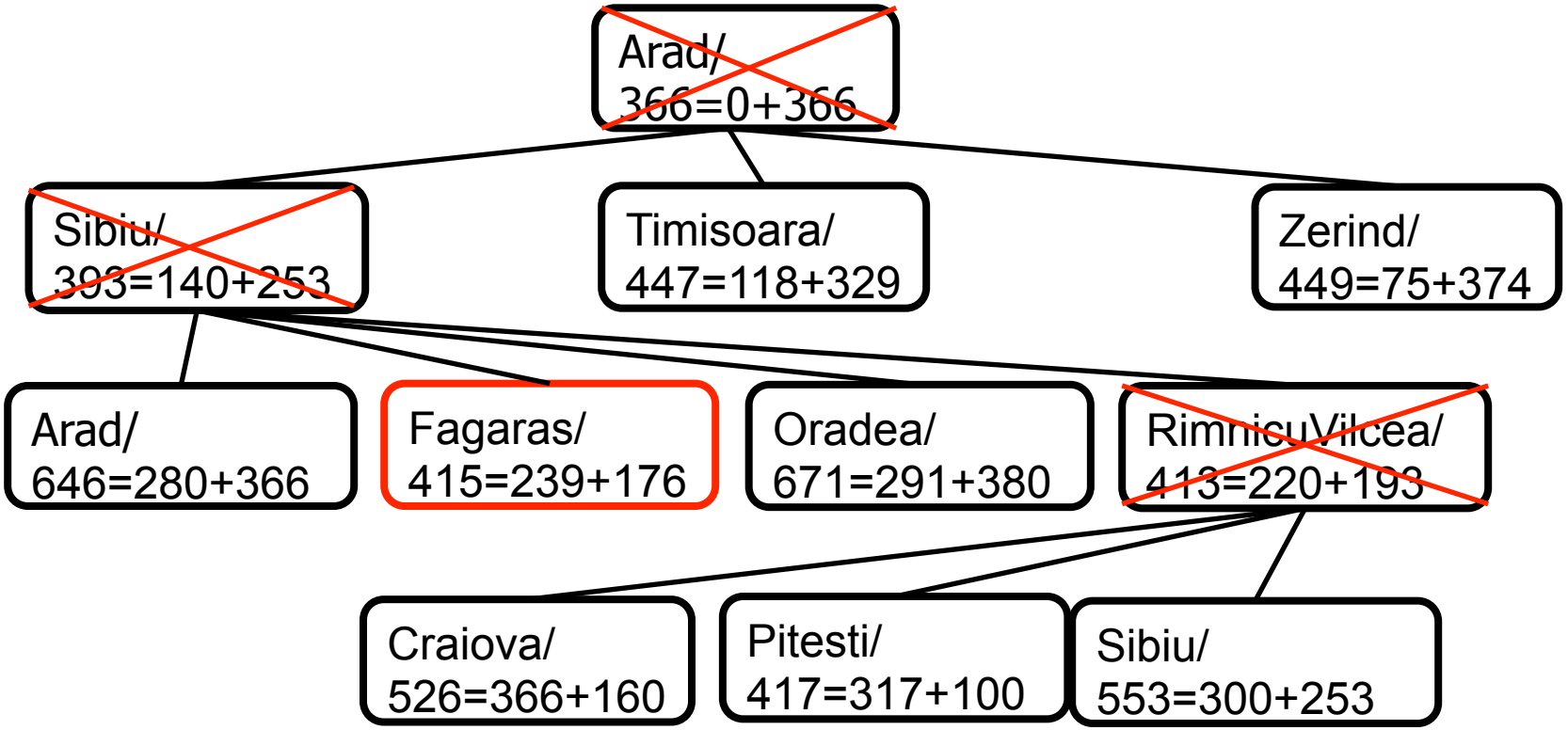
A* tree search example: Simulated queue. City/f=g+h

- Next: RimnicuVilcea/413=220+193
- Children: Craiova/526=366+160, Pitesti/417=317+100, Sibiu/553=300+253
- Expanded: Arad/366=0+366, Sibiu/393=140+253, RimnicuVilcea/413=220+193
- Frontier: ~~Arad/366=0+366, Sibiu/393=140+253~~, Timisoara/447=118+329, Zerind/449=75+374, Arad/646=280+366, Fagaras/415=239+176, Oradea/671=291+380, RimnicuVilcea/413=220+193, Craiova/~~526=366+160~~, ~~Pitesti/417=317~~+100, Sibiu/553=300+253

A* tree search example: Simulated queue. City/f=g+h

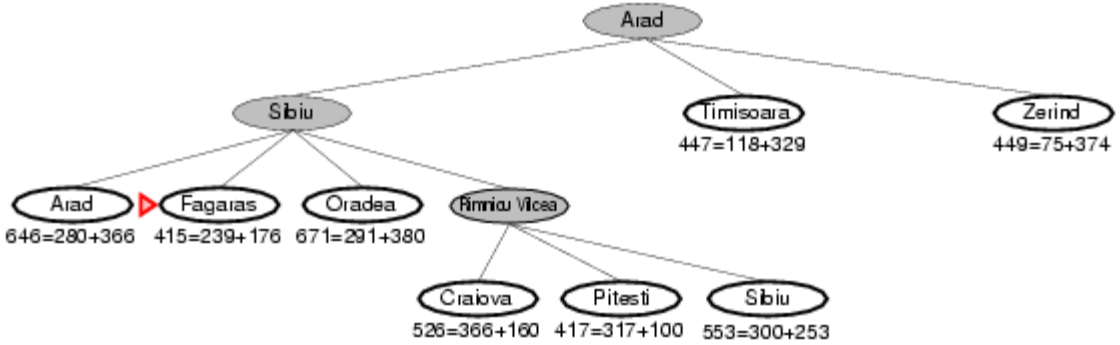


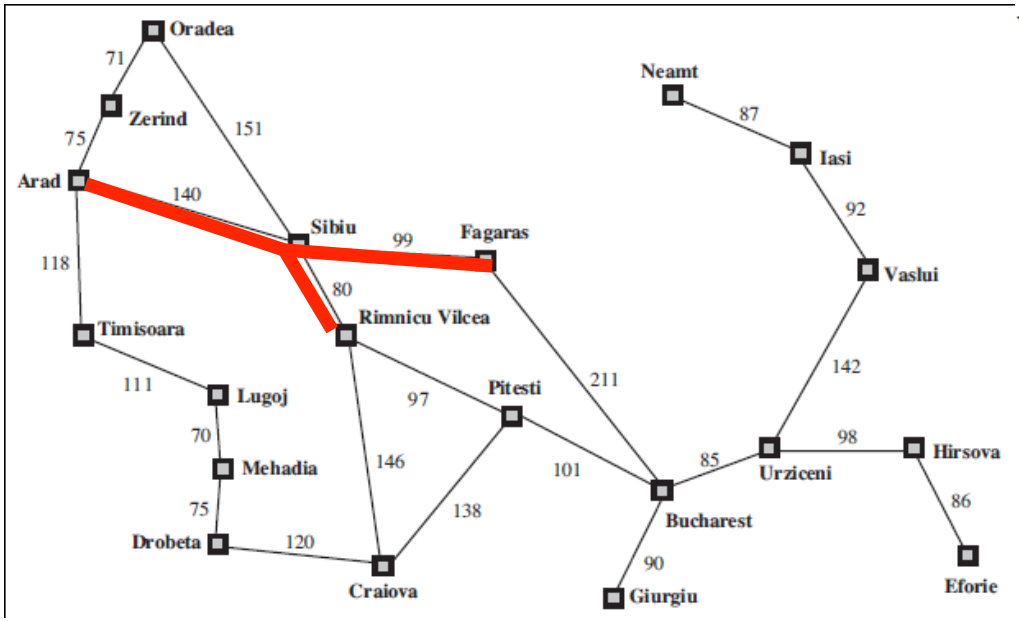
A* search example: Simulated queue. City/f=g+h



A* tree search example

Note: The search below did not "back track." Rather, both arms are being pursued in parallel on the queue.





Values of h_{SLD}—straight-line distances to Bucharest.

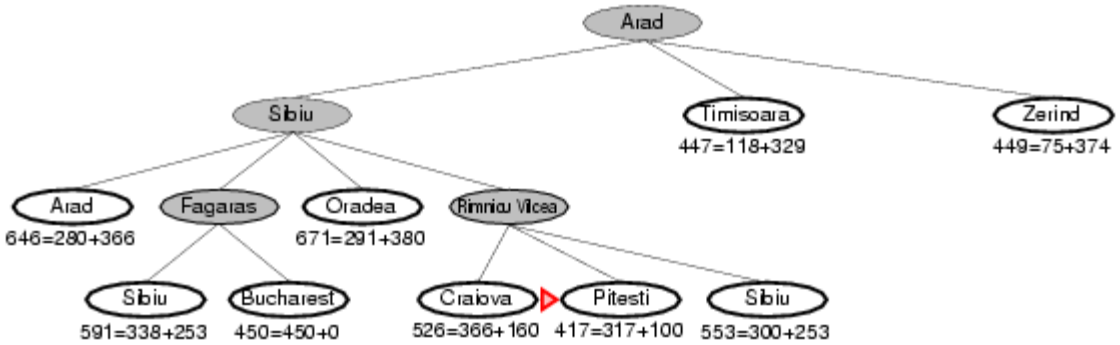
Arad	366
Bucharest	0
Craiova	160
Drobeta	242
Eforie	161
Fagaras	176
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	100
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

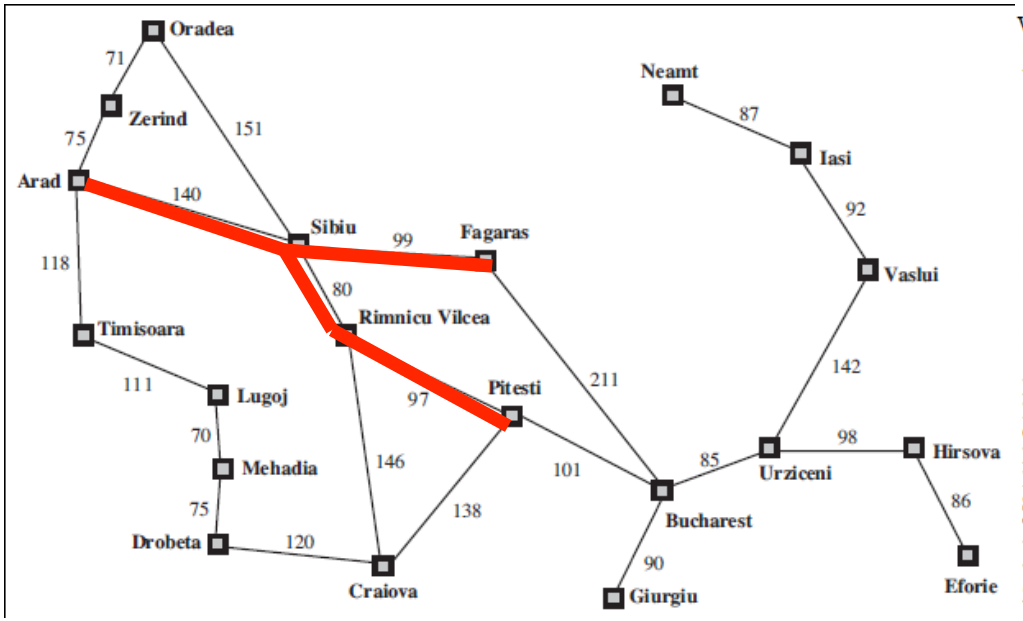
A* tree search example: Simulated queue. City/f=g+h

- Next: Fagaras/415=239+176
- Children: Bucharest/450=450+0, Sibiu/591=338+253
- Expanded: Arad/366=0+366, Sibiu/393=140+253, RimnicuVilcea/413=220+193, Fagaras/415=239+176
- Frontier: ~~Arad/366=0+366, Sibiu/393=140+253~~, Timisoara/447=118+329, Zerind/449=75+374, Arad/646=280+366, Fagaras/415=239+1~~76, Oradea/671=291+38~~0, RimnicuVilcea/413=220+193, Craiova~~/526=366+160, Pitesti/417=317~~+100, Sibiu/553=300+253, Bucharest/450=450+0, Sibiu/591=338+253

A* tree search example

Note: The search below did not "back track." Rather, both arms are being pursued in parallel on the queue.





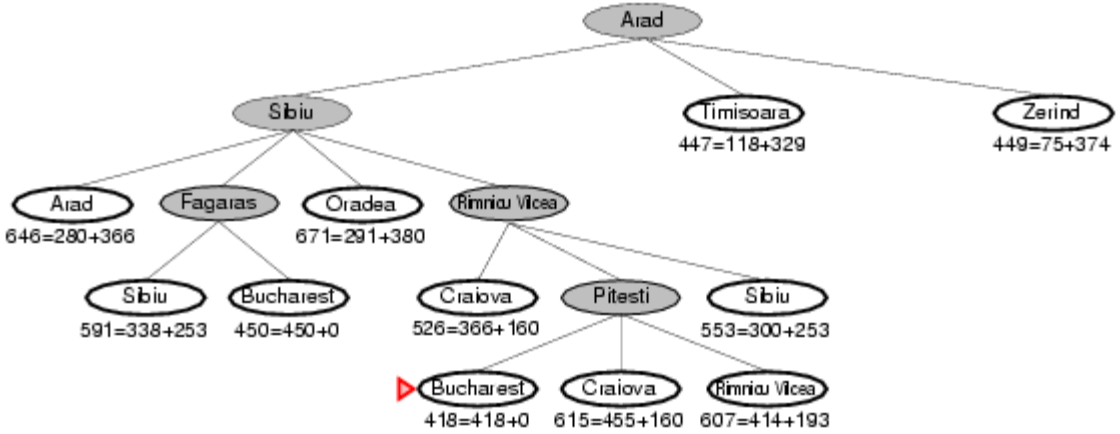
Values of h_{SLD}—straight-line distances to Bucharest.

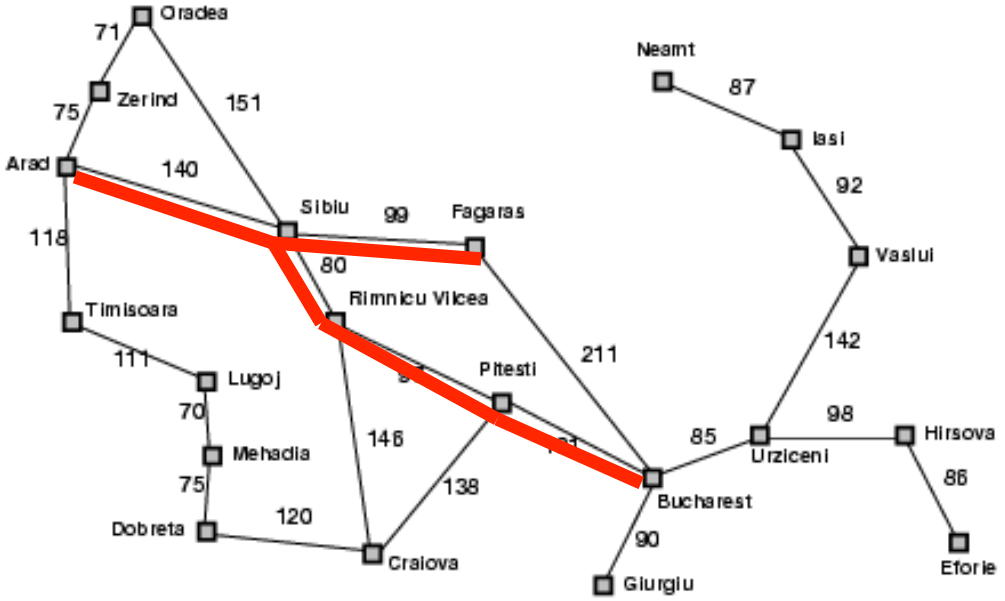
Arad	366
Bucharest	0
Craiova	160
Drobeta	242
Eforie	161
Fagaras	176
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	100
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

A* tree search example: Simulated queue. City/f=g+h

- Next: Pitesti/417=317+100
- Children: Bucharest/418=418+0, Craiova/615=455+160, RimnicuVilcea/607=414+193
- Expanded: Arad/366=0+366, Sibiu/393=140+253, RimnicuVilcea/413=220+193, Fagaras/415=239+176, Pitesti/417=317+100
- Frontier: ~~Arad/366=0+366, Sibiu/393=140+253~~, Timisoara/447=118+329, Zerind/449=75+374, Arad/646=280+366, Fagaras/415=239+1~~76, Oradea/671=291+~~380, RimnicuVilcea/413=220+193, Craiova~~/526=366+160, Pitesti/417=317~~+100, Sibiu/553=300+253, B~~ucharest/450=450+0,~~ Sibiu/591=338+253, Bucharest/418=418+0, Craiova/615=455+160, RimnicuVilcea/607=414+193

A* tree search example





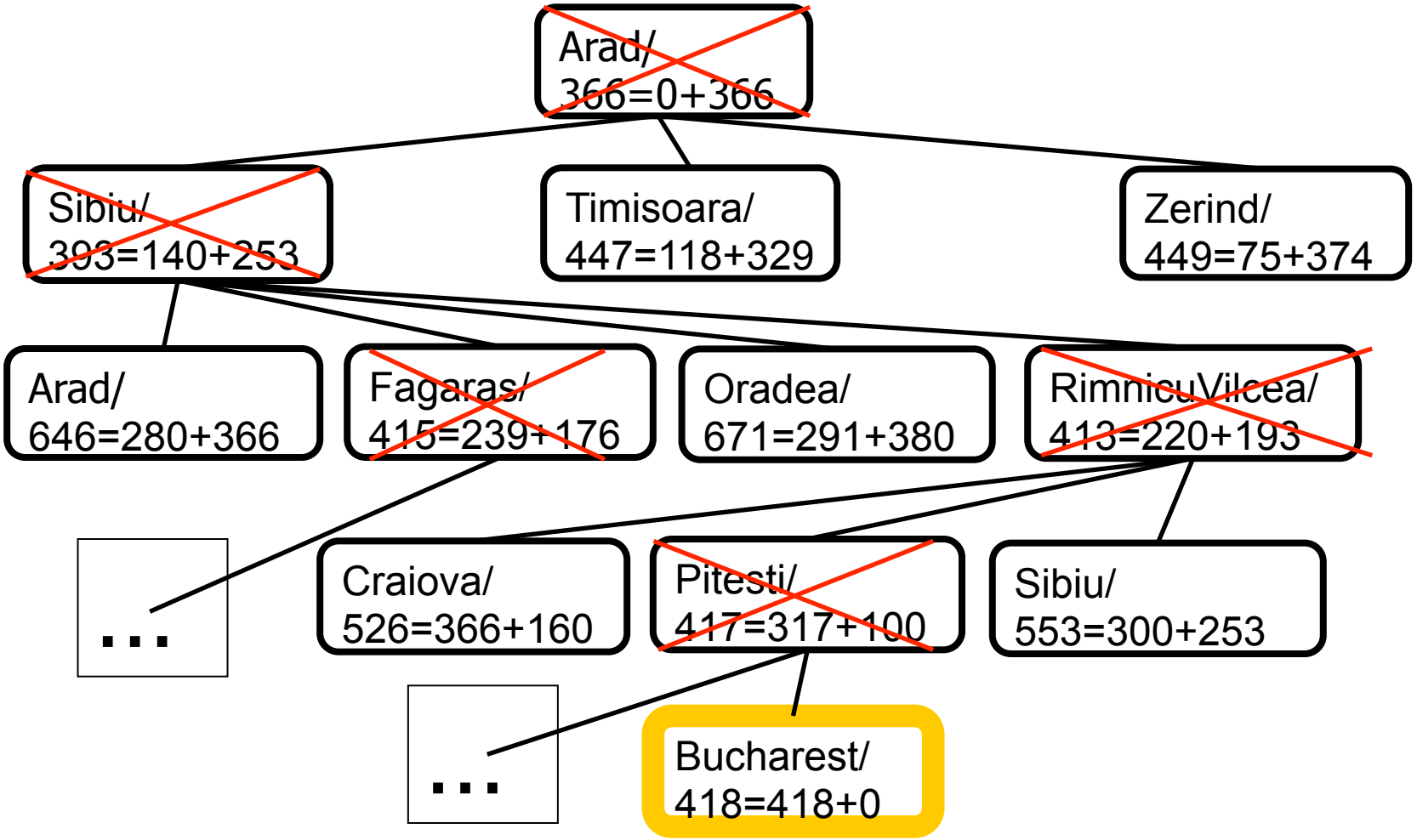
Straight–line distance to Bucharest	
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	176
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	10
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

A* tree search example: Simulated queue. City/f=g+h

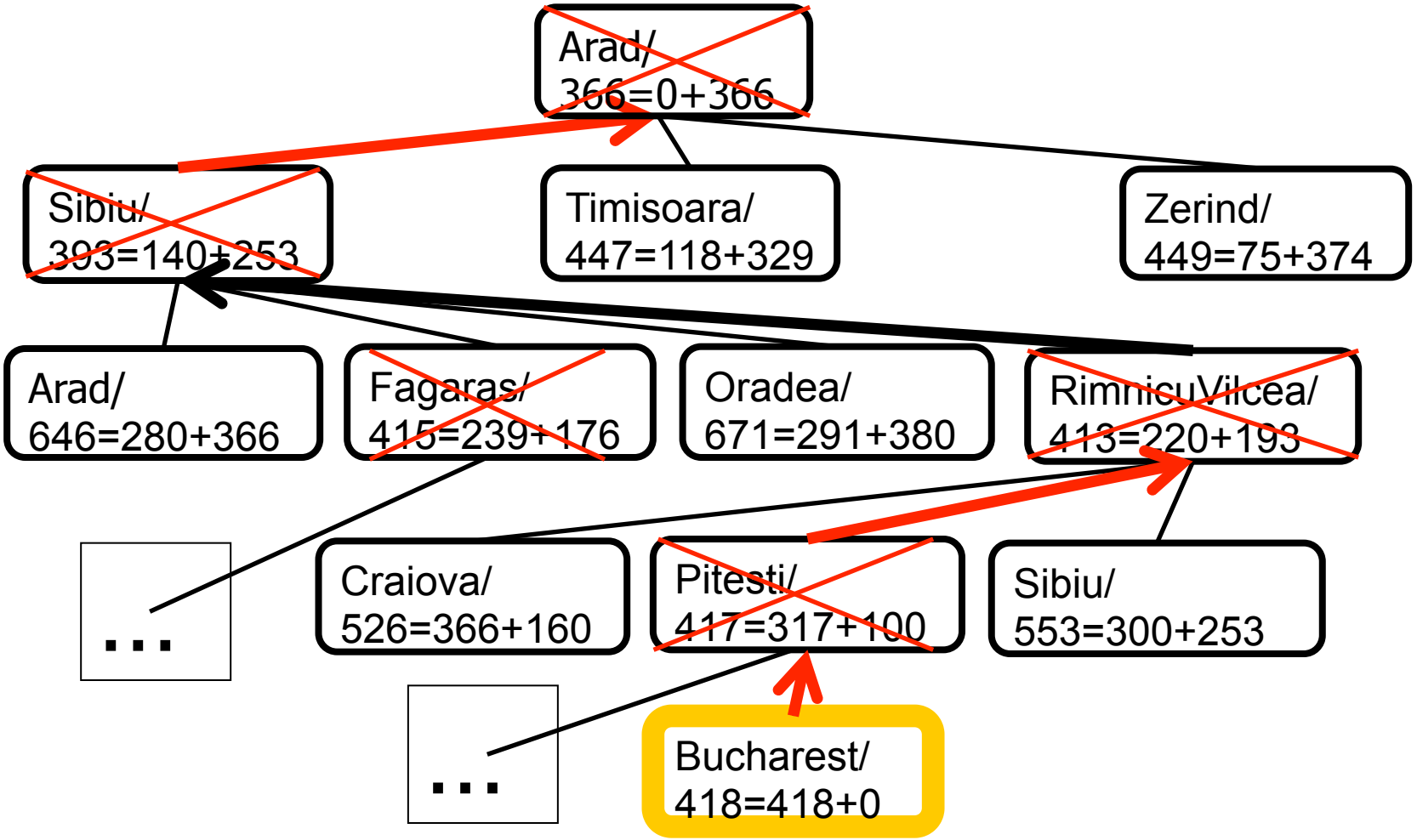
- Next: Bucharest/418=418+0
- Children: **None; goal test succeeds.**
- Expanded: Arad/366=0+366, Sibiu/393=140+253, RimnicuVilcea/413=220+193, Fagaras/415=239+176, Pitesti/417=317+100, Bucharest/418=418+0
- Frontier: Arad/366=0+366, Sibiu/393=140+253, Timisoara/447=118+329, Zerind/449=75+374, Arad/646=280+366, Fagaras/415=239+176, Oradea/671=291+380, RimnicuVilcea/413=220+193, Craiova/526=366+160, Pitesti/417=317+100, Sibiu/553=300+253, Bucharest/450=450+0, Sibiu/591=338+253, Bucharest/418=418+0, Craiova/615=455+160, RimnicuVilcea/607=414+193

Note that the short expensive path stays on the queue. The long cheap path is found and returned.

A* tree search example: Simulated queue. City/f=g+h



A* tree search example: Simulated queue. City/f=g+h



Properties of A*

- Complete? Yes

  (unless there are infinitely many nodes with $f \leq f(G)$;

  can't happen if step-cost $\geq \varepsilon > 0$)

- Time/Space? Exponential $O(b^d)$

  except if: $|h(n) - h^*(n)| \leq O(\log h^*(n))$

- Optimal?

  (with: Tree-Search, admissible heuristic;

  Graph-Search, consistent heuristic)

- *Optimally Efficient?*

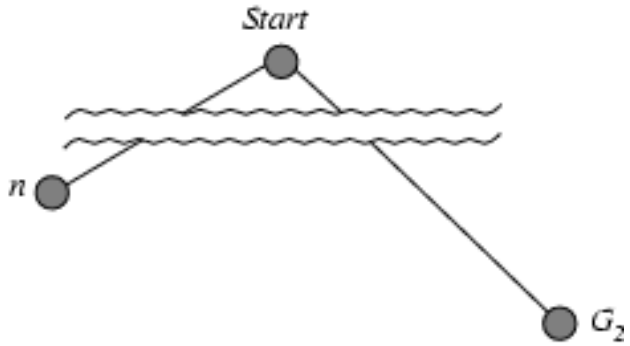
  (no optimal algorithm with same heuristic is guaranteed to expand fewer nodes)

Optimality of A* (proof)

Tree Search, where *h(n)* is admissible

- Suppose some suboptimal goal G_2 has been generated and is in the frontier. Let *n* be an unexpanded node in the frontier such that *n* is on a shortest path to an optimal goal *G*.

We want to prove:
f(n) < f(G2)
(then A* will expand n before G2)



- $f(G_2) = g(G_2)$ since $h(G_2) = 0$
- $f(G) = g(G)$ since $h(G) = 0$
- $g(G_2) > g(G)$ since G_2 is suboptimal
- $f(G_2) > f(G)$ from above, with h=0
- $h(n) \leq h^*(n)$ since h is admissible (*under*-estimate)
- $g(n) + h(n) \leq g(n) + h^*(n)$ from above
- $f(n) \leq f(G)$ since g(n)+h(n)=f(n) & g(n)+h*(n)=f(G)
- $f(n) < f(G2)$ from above

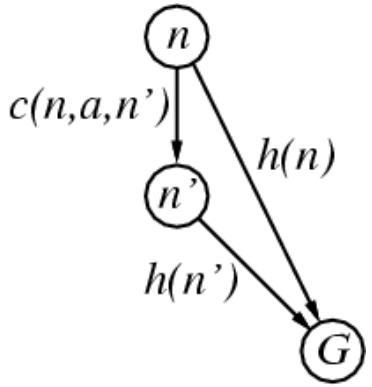
Admissible heuristics

- A heuristic $h(n)$ is admissible if for every node n,
  $h(n) \leq h^*(n)$, where $h^*(n)$ is the true cost to reach the goal state from n.
- An admissible heuristic never overestimates the cost to reach the goal, i.e., it is optimistic
- Example: $h_{SLD}(n)$ (never overestimates the actual road distance)
- Theorem: If $h(n)$ is admissible, A* using `TREE-SEARCH` is optimal

Consistent heuristics (consistent => admissible)

- A heuristic is consistent if for every node *n*, every successor *n'* of *n* generated by any action *a*,

$$h(n) \leq c(n,a,n') + h(n')$$



- If *h* is consistent, we have

f(n') = g(n') + h(n') (by def.)
= g(n) + c(n,a,n') + h(n') (g(n')=g(n)+c(n.a.n'))
≥ g(n) + h(n) = f(n) (consistency)
f(n') ≥ f(n)

It's the triangle inequality !

- i.e., *f(n)* is non-decreasing along any path.

- Theorem:
  If *h(n)* is consistent, A* using GRAPH−SEARCH is optimal

keeps all checked nodes in memory to avoid repeated states

Dominance

- IF $h_2(n) \geq h_1(n)$ for all n
  THEN h_2 dominates h_1
  - h_2 is almost always better for search than h_1
  - h_2 guarantees to expand no more nodes than does h_1
  - h_2 almost always expands fewer nodes than does h_1
  - Not useful unless both h_1 & h_2 are admissible/consistent

- Typical 8-puzzle search costs
  (average number of nodes expanded):
  - *d=12* IDS = 3,644,035 nodes
    $A^*(h_1)$ = 227 nodes
    $A^*(h_2)$ = 73 nodes
  - *d=24* IDS = too many nodes
    $A^*(h_1)$ = 39,135 nodes
    $A^*(h_2)$ = 1,641 nodes

Local search algorithms (4.1, 4.2)

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

- State space = set of "complete" configurations
- Find configuration satisfying constraints, e.g., n-queens
- In such cases, we can use local search algorithms
- keep a single "current" state, try to improve it.
- Very memory efficient (only remember current state)

Random Restart Wrapper

- These are stochastic local search methods
  - Different solution for each trial and initial state
- Almost every trial hits difficulties (see below)
  - Most trials will not yield a good result (sadly)
- Many random restarts improve your chances
  - Many "shots at goal" may, finally, get a good one
- Restart a random initial state; many times
  - Report the best result found; across many trials

Random Restart Wrapper

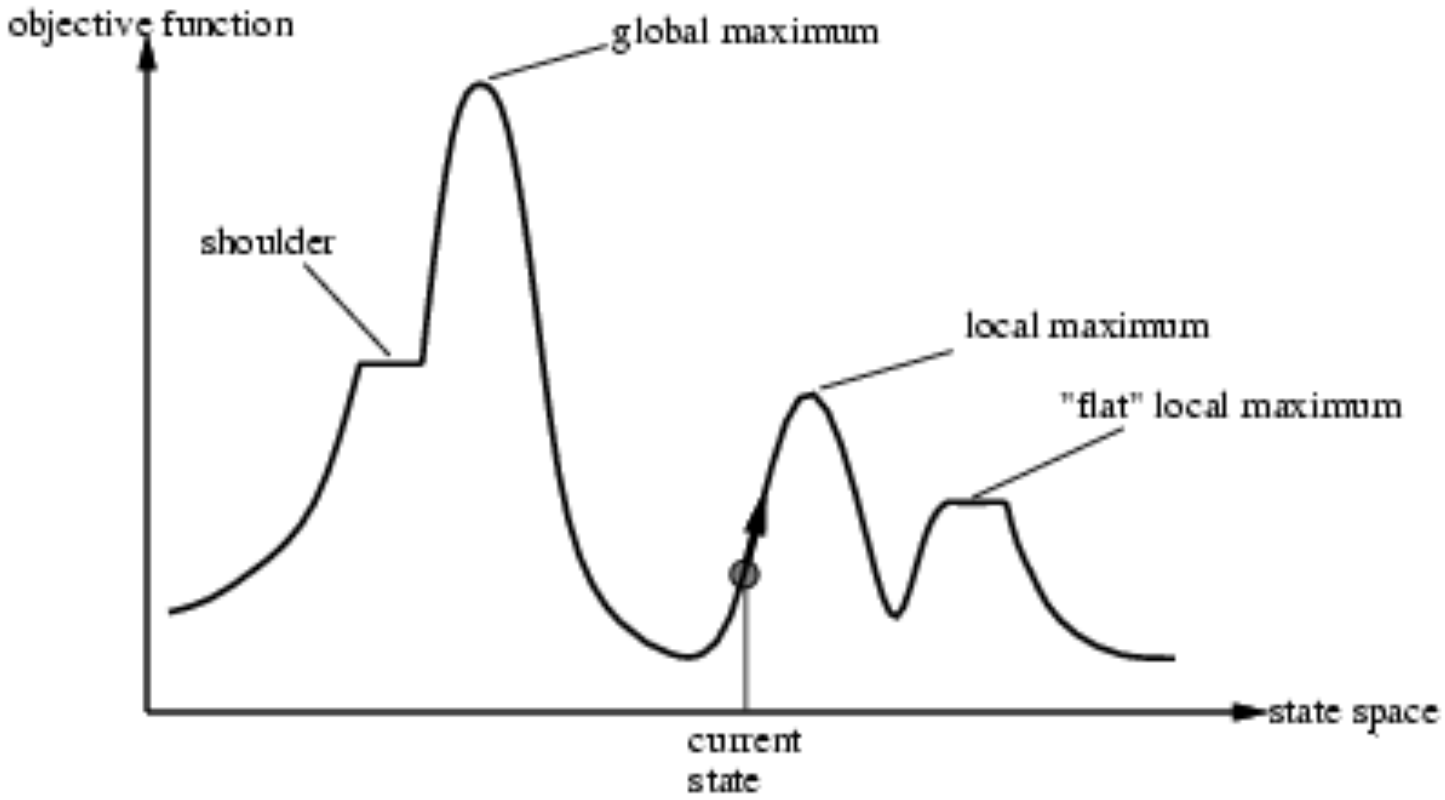
```
BestResultFoundSoFar <- infinitely bad;
UNTIL ( you are tired of doing it ) DO {
    Result <- ( Local search from random initial state );
    IF ( Result is better than BestResultFoundSoFar )
        THEN ( Set BestResultFoundSoFar to Result );
    }
RETURN BestResultFoundSoFar;
```

Typically, “you are tired of doing it” means that some resource limit is exceeded, e.g., number of iterations, wall clock time, CPU time, etc. It may also mean that Result improvements are small and infrequent, e.g., less than 0.1% Result improvement in the last week of run time.

Local Search Difficulties

These difficulties apply to ALL local search algorithms, and become MUCH more difficult as the dimensionality of the search space increases to high dimensions.

- Problems: depending on state, can get stuck in local maxima
  - Many other problems also endanger your success!!



Hill-climbing search

- "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
```

Simulated annealing search

- Idea: escape local maxima by allowing some "bad" moves but gradually decrease their frequency

```
function SIMULATED-ANNEALING( problem, schedule) returns a solution state
   inputs: problem, a problem
           schedule, a mapping from time to "temperature"
   local variables: current, a node
                    next, a node
                    T, a "temperature" controlling prob. of downward steps

   current ← MAKE-NODE(INITIAL-STATE[problem])
   for t ← 1 to ∞ do
       T ← schedule[t]
       if T = 0 then return current
       next ← a randomly selected successor of current
       ΔE ← VALUE[next] – VALUE[current]
       if ΔE > 0 then current ← next
       else current ← next only with probability e^(Δ E/T)
```

Improvement: Track the BestResultFoundSoFar. Here, this slide follows Fig. 4.5 of the textbook, which is simplified.

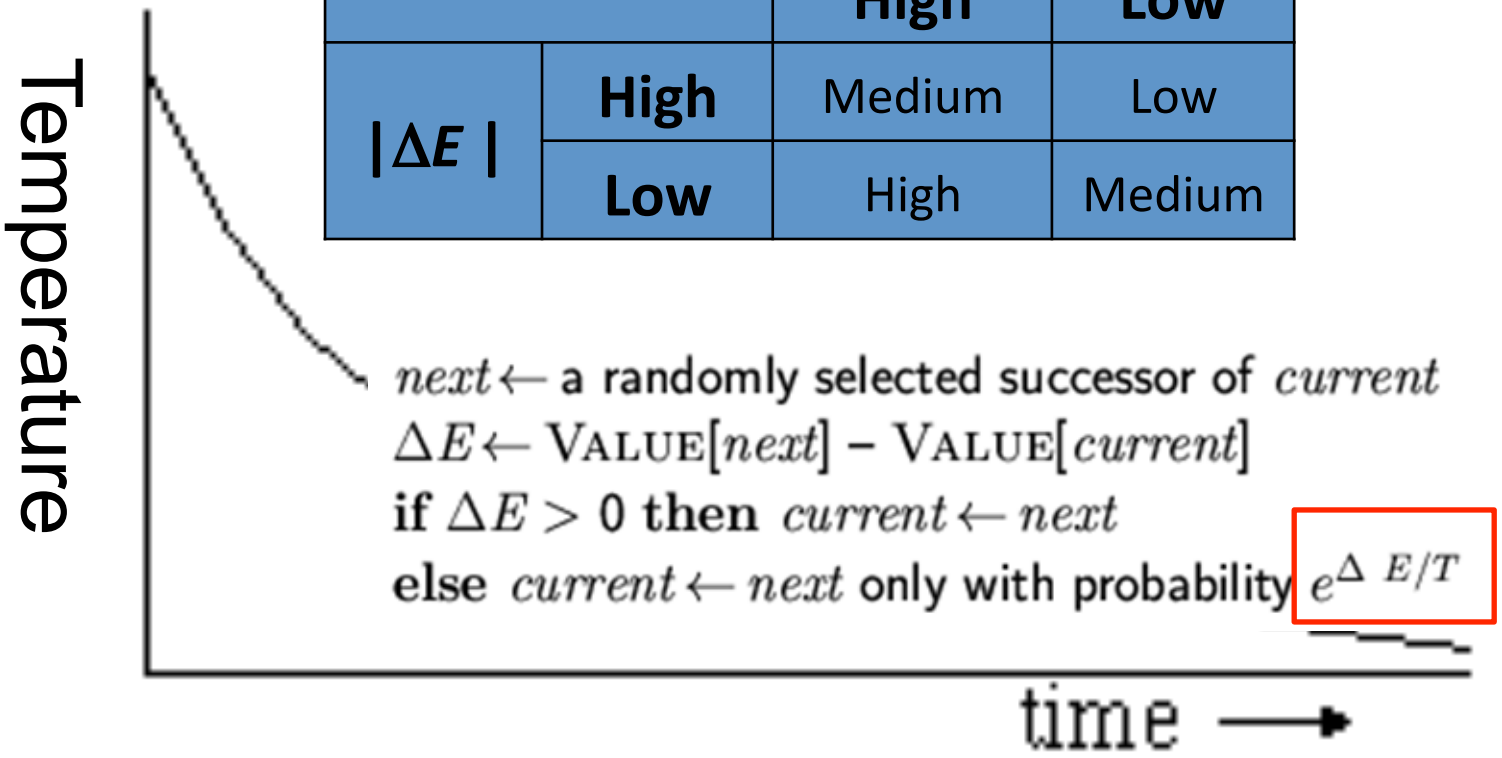
P(accepting a worse successor)

Decreases as Temperature T decreases

Increases as | ΔE | decreases

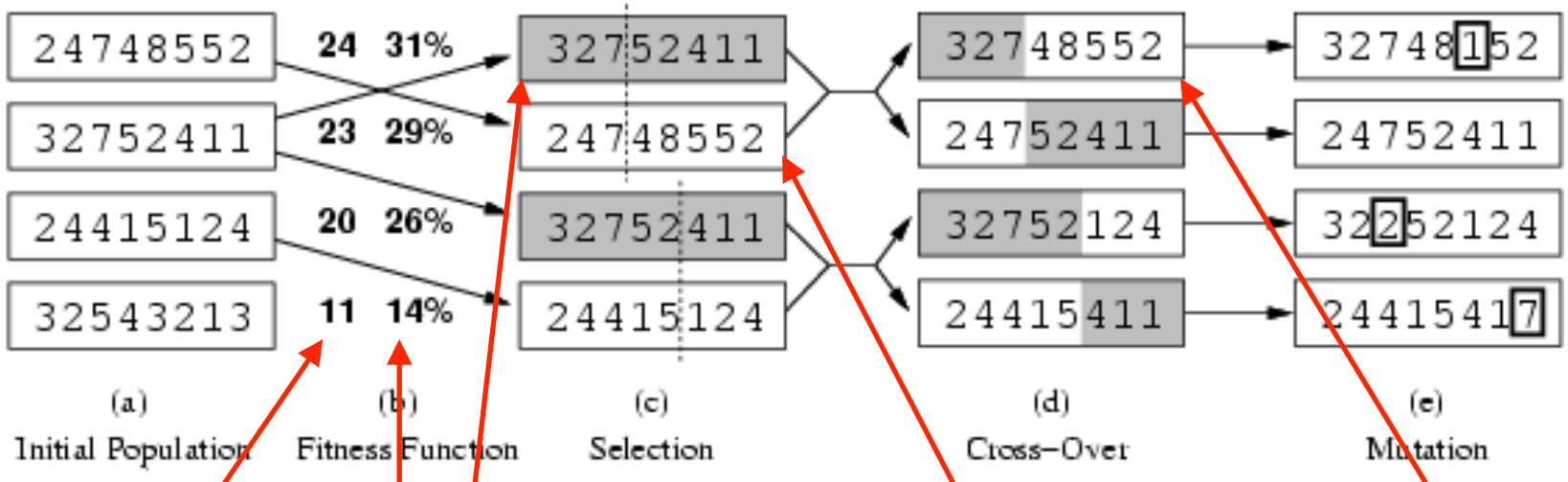
(Sometimes step size also decreases with T)

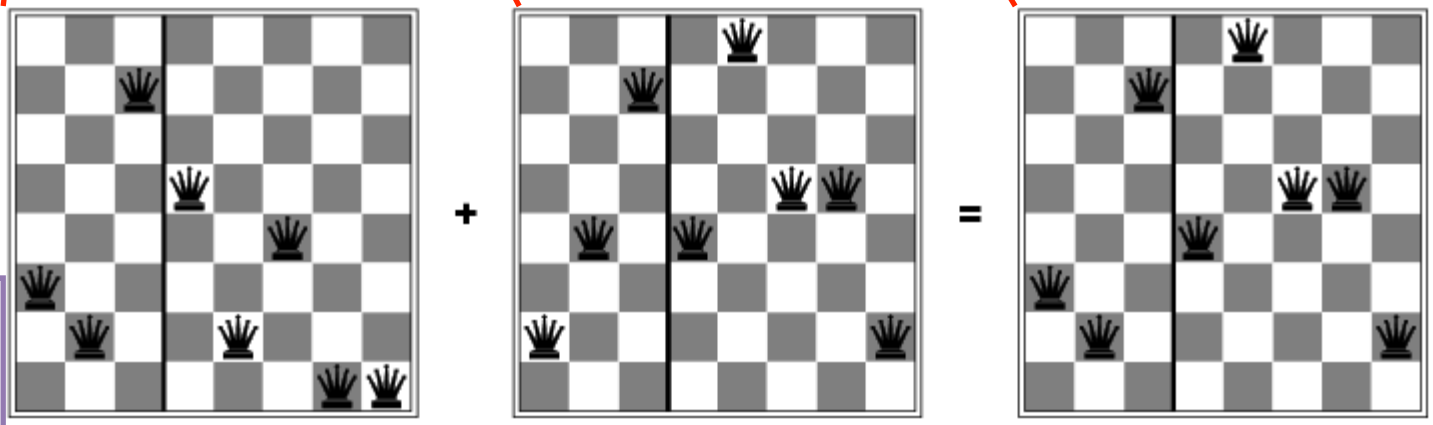
e^(ΔE / T)		Temperature T	
		High	**Low**
|ΔE |	**High**	Medium	Low
	Low	High	Medium



Genetic algorithms (Darwin!!)

- A state = a string over a finite alphabet (an individual)
- Start with *k* randomly generated states (a population)
- Fitness function (= our heuristic objective function).
  - Higher fitness values for better states.
- Select individuals for next generation based on fitness
  - P(individual in next gen.) = individual fitness/Σ population fitness
- Crossover fit parents to yield next generation (off-spring)
- Mutate the offspring randomly with some low probability





fitness = #non-attacking queens

probability of being in next generation = fitness/(Σ_i fitness_i)

How to convert a fitness value into a probability of being in the next generation.

- Fitness function: #non-attacking queen pairs
  - min = 0, max = 8 × 7/2 = 28
- Σ_i fitness_i = 24+23+20+11 = 78
- P(child_1 in next gen.) = fitness_1/(Σ_i fitness_i) = 24/78 = 31%
- P(child_2 in next gen.) = fitness_2/(Σ_i fitness_i) = 23/78 = 29%; etc

You Will Be Expected to Know

- Basic definitions (section 5.1)
- Minimax optimal game search (5.2)
- Evaluation functions (5.4.1)
- Cutting off search (5.4.2)
- Optional: Sections 5.4.3-4; 5.8

Games as Search

- Two players: MAX and MIN
- MAX moves first and they take turns until the game is over
  - Winner gets reward, loser gets penalty.
  - "Zero sum" means the sum of the reward and the penalty is a constant.

- Formal definition as a search problem:
  - **Initial state:** Set-up specified by the rules, e.g., initial board configuration of chess.
  - **Player(s):** Defines which player has the move in a state.
  - **Actions(s):** Returns the set of legal moves in a state.
  - **Result(s,a):** Transition model defines the result of a move.
  - (**2nd ed.: Successor function:** list of (move,state) pairs specifying legal moves.)
  - **Terminal-Test(s):** Is the game finished? True if finished, false otherwise.
  - **Utility function(s,p):** Gives numerical value of terminal state s for player p.
    - E.g., win (+1), lose (-1), and draw (0) in tic-tac-toe.
    - E.g., win (+1), lose (0), and draw (1/2) in chess.

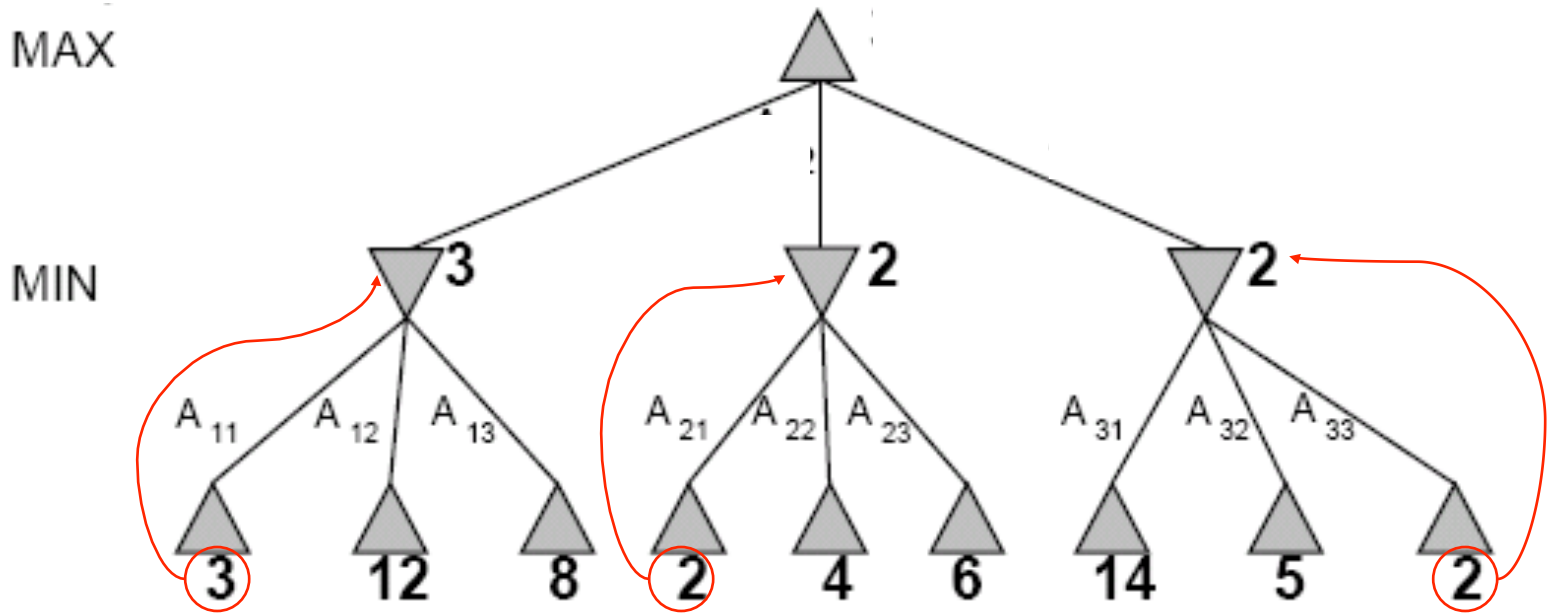
- MAX uses search tree to determine "best" next move.

An optimal procedure: The Min-Max method

Will find the optimal strategy and best next move for Max:

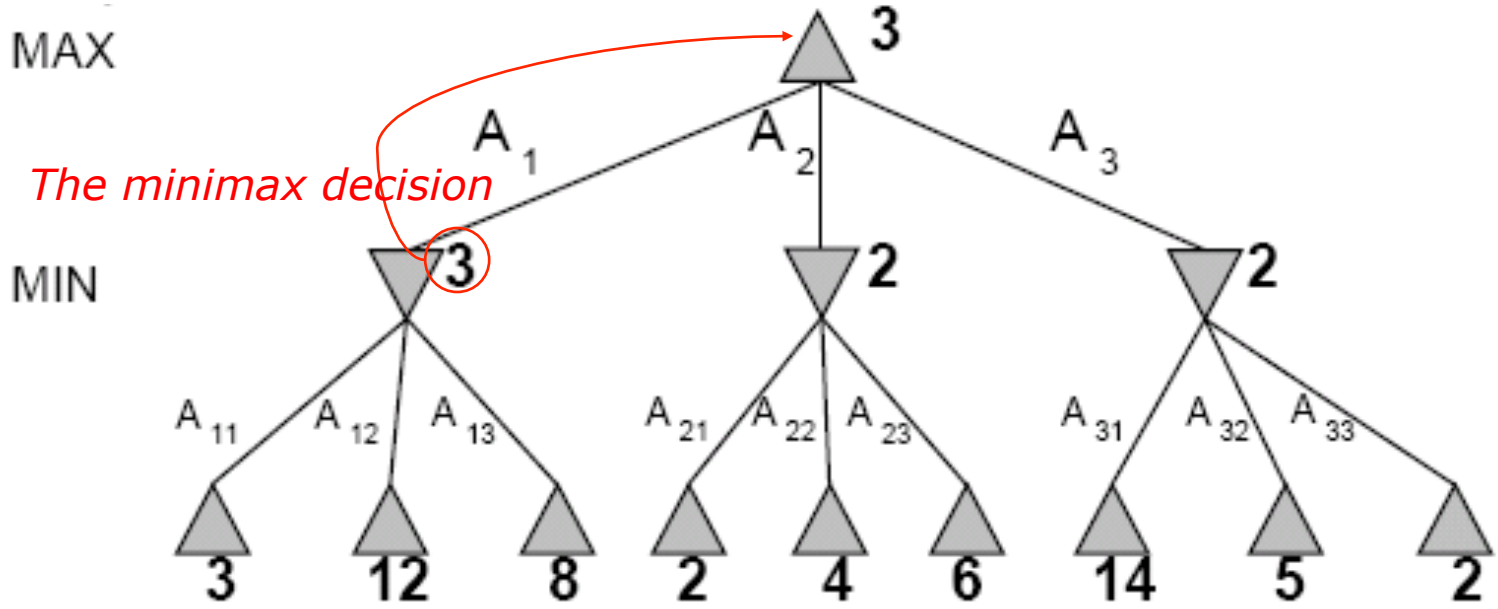
- 1. Generate the whole game tree, down to the leaves.
- 2. Apply utility (payoff) function to each leaf.
- 3. Back-up values from leaves through branch nodes:
  - a Max node computes the Max of its child values
  - a Min node computes the Min of its child values
- 4. At root: Choose move leading to the child of highest value.

Two-Ply Game Tree



Two-Ply Game Tree

Minimax maximizes the utility of the worst-case outcome for Max



Pseudocode for Minimax Algorithm

function MINIMAX-DECISION(*state*) **returns** *an action*
  inputs: *state*, current state in game
return arg $\max_{a \in \text{ACTIONS}(state)}$ MIN-VALUE(Result(*state*,*a*))

---

function MAX-VALUE(*state*) **returns** *a utility value*
  if TERMINAL-TEST(*state*) **then return** UTILITY(*state*)
  $v \leftarrow -\infty$
  for *a* in ACTIONS(*state*) **do**
    $v \leftarrow$ MAX(*v*,MIN-VALUE(Result(*state*,*a*)))
  return *v*

---

function MIN-VALUE(*state*) **returns** *a utility value*
  if TERMINAL-TEST(*state*) **then return** UTILITY(*state*)
  $v \leftarrow +\infty$
  for *a* in ACTIONS(*state*) **do**
    $v \leftarrow$ MIN(*v*,MAX-VALUE(Result(*state*,*a*)))
  return *v*

Properties of minimax

- Complete?
  - Yes (if tree is finite).
- Optimal?
  - Yes (against an optimal opponent).
  - Can it be beaten by an opponent playing sub-optimally?
    - No. (Why not?)
- Time complexity?
  - $O(b^m)$
- Space complexity?
  - O(bm) (depth-first search, generate all actions at once)
  - O(m) (backtracking search, generate actions one at a time)

Cutting off search

MinimaxCutoff is identical to MinimaxValue except

1. Terminal? is replaced by Cutoff?
2. Utility is replaced by Eval

Does it work in practice?

$$b^m = 10^6, \quad b = 35 \quad \Rightarrow \quad m = 4$$

4-ply lookahead is a hopeless chess player!

4-ply $\approx$ human novice
8-ply $\approx$ typical PC, human master
12-ply $\approx$ Deep Blue, Kasparov

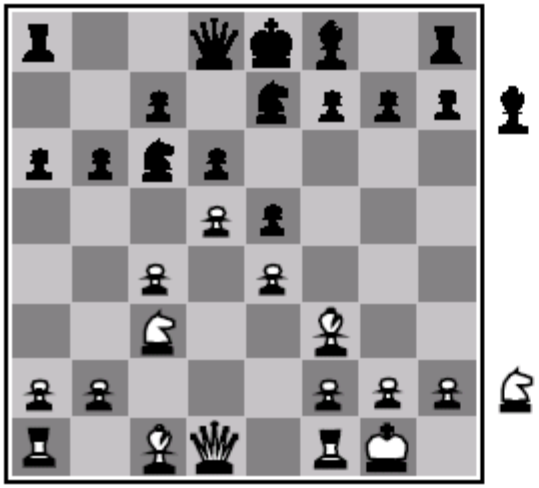
Static (Heuristic) Evaluation Functions

- An Evaluation Function:
  - Estimates how good the current board configuration is for a player.
  - Typically, evaluate how good it is for the player, how good it is for the opponent, then subtract the opponent's score from the player's.
  - Othello: Number of white pieces - Number of black pieces
  - Chess: Value of all white pieces - Value of all black pieces

- Typical values from -infinity (loss) to +infinity (win) or [-1, +1].

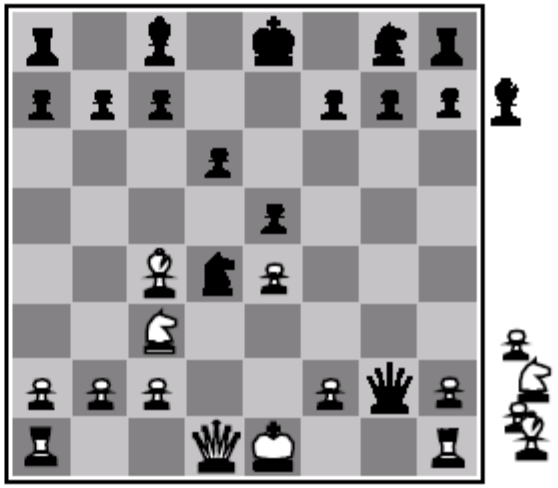
- If the board evaluation is X for a player, it's -X for the opponent
  - "Zero-sum game"

Evaluation functions



Black to move

White slightly better



White to move

Black winning

For chess, typically *linear* weighted sum of features

$$Eval(s) = w_1 f_1(s) + w_2 f_2(s) + \ldots + w_n f_n(s)$$

e.g., $w_1 = 9$ with
$f_1(s) =$ (number of white queens) – (number of black queens), etc.

Iterative (Progressive) Deepening

- In real games, there is usually a time limit T to make a move

- How do we take this into account?
- Using MiniMax we cannot use “partial” results with any confidence unless the full tree has been searched
  - So, we could be conservative and set a conservative depth-limit which guarantees that we will find a move in time < T
    - Disadvantage: we may finish early, could do more search

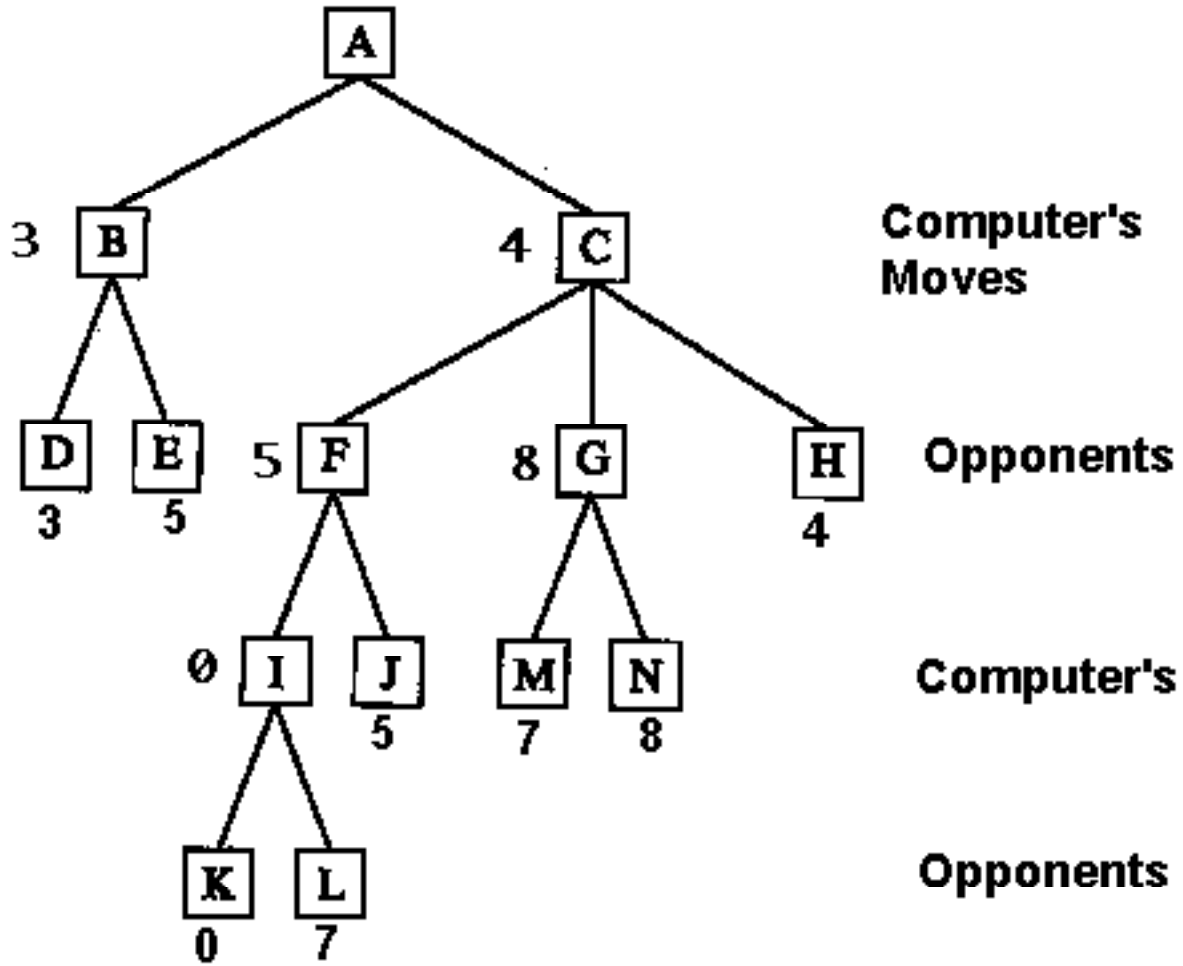
- In practice, Iterative Deepening Search (IDS) is used
  - IDS runs depth-first search with an increasing depth-limit
  - When the clock runs out we use the solution found at the previous depth limit
  - With alpha-beta pruning (next lecture), we can sort the nodes based on values found in previous depth limit to maximize pruning during the next depth limit => search deeper

Heuristics and Game Tree Search:
Limited horizon

- The Horizon Effect
  - Sometimes there's a major "effect" (e.g., a piece is captured) which is "just below" the depth to which the tree has been expanded
  - The computer cannot see that this major event could happen because it has a "limited horizon" --- i.e., when search stops
  - There are heuristics to try to follow certain branches more deeply to detect such important events, and so avoid stupid play
  - This helps to avoid catastrophic losses due to "short-sightedness"

- Heuristics for Tree Exploration
  - Often better to explore some branches more deeply in allotted time
  - Various heuristics exist to identify "promising" branches
  - Stop at "quiescent" positions --- all battles are over, things are quiet
  - Continue when things are in violent flux --- the middle of a battle

Selectively Deeper Game Trees



Eliminate Redundant Nodes

- On average, each board position appears in the search tree approximately ~10^{150} / ~10^{40} ≈ 10^{100} times.

  => Vastly redundant search effort.

- Can't remember all nodes (too many).

  => Can't eliminate all redundant nodes.

- However, some short move sequences provably lead to a redundant position.

  – These can be deleted dynamically with no memory cost

- Example:

  1. P-QR4 P-QR4; 2. P-KR4 P-KR4

  leads to the same position as

  1. P-QR4 P-KR4; 2. P-KR4 P-QR4

Alpha-beta Algorithm

- Depth first search
  - only considers nodes along a single path from root at any time

α = highest-value choice found at any choice point of path for MAX
(initially, α = −infinity)

β = lowest-value choice found at any choice point of path for MIN
(initially, β = +infinity)

- Pass current values of α and β down to child nodes during search.
- Update values of α and β during search:
  - MAX updates α at MAX nodes
  - MIN updates β at MIN nodes
- Prune remaining branches at a node when $\alpha \geq \beta$

Pseudocode for Alpha-Beta Algorithm

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 ACTIONS(*state*) with value *v*

---

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

(MIN-VALUE is defined analogously)

When to Prune?

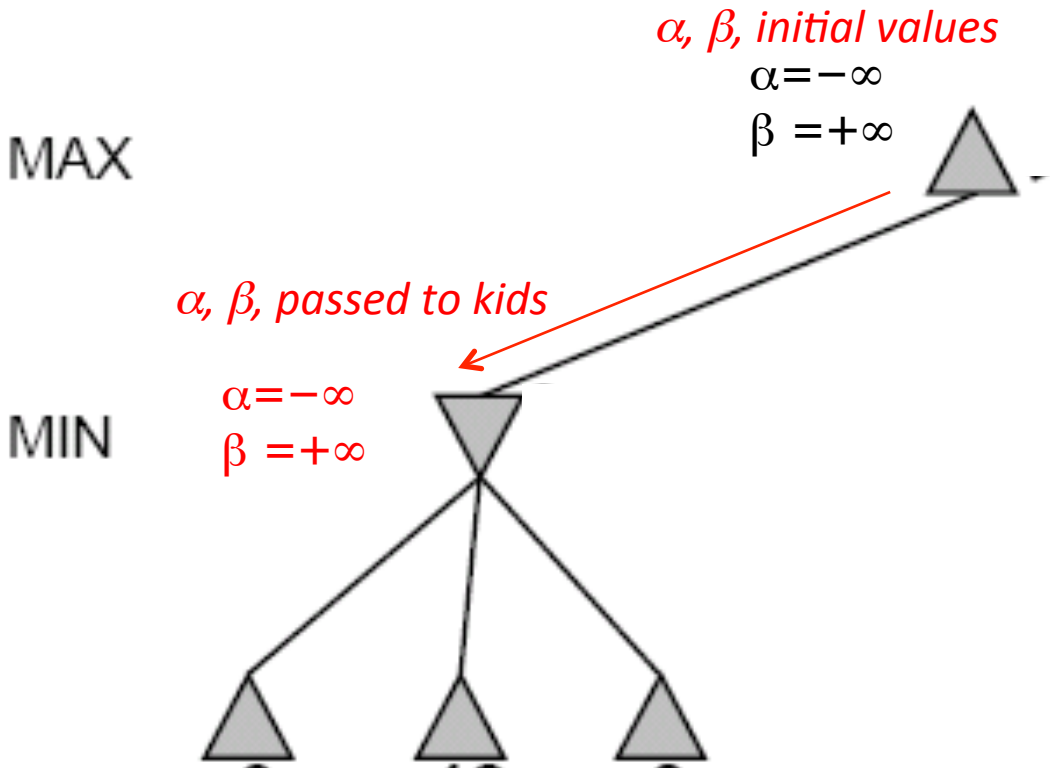
- **Prune whenever $\alpha \geq \beta$.**
  - Prune below a Max node whose alpha value becomes greater than or equal to the beta value of its ancestors.
    - **Max nodes update alpha** based on children's returned values.
  - Prune below a Min node whose beta value becomes less than or equal to the alpha value of its ancestors.
    - **Min nodes update beta** based on children's returned values.

α/β Pruning vs. Returned Node Value

- Some students are confused about the use of α/β pruning vs. the returned value of a node
- α/β are used **ONLY FOR PRUNING**
  - α/β have no effect on anything other than pruning
  - IF (α >= β) THEN prune & return current node value
- Returned node value = "best" child seen so far
  - Maximum child value seen so far for MAX nodes
  - Minimum child value seen so far for MIN nodes
  - If you prune, return to parent "best" child so far
- Returned node value is received by parent

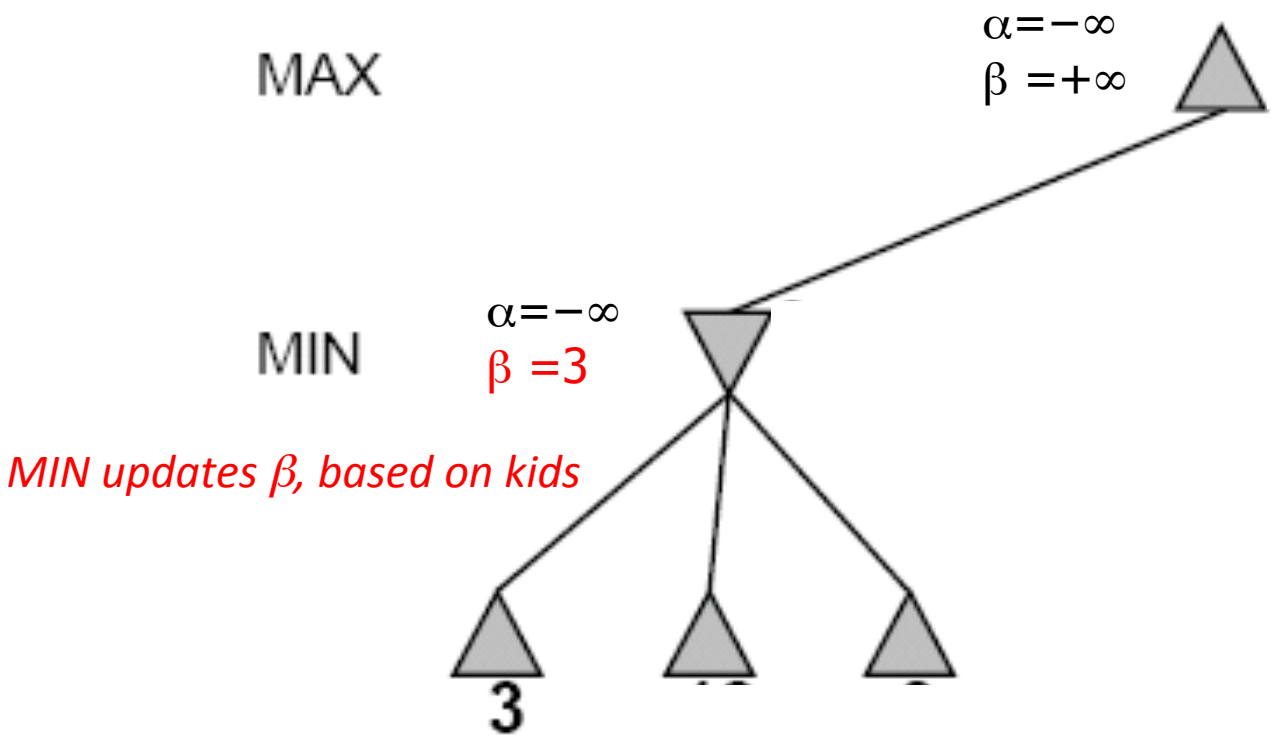
Alpha-Beta Example Revisited

Do DF-search until first leaf

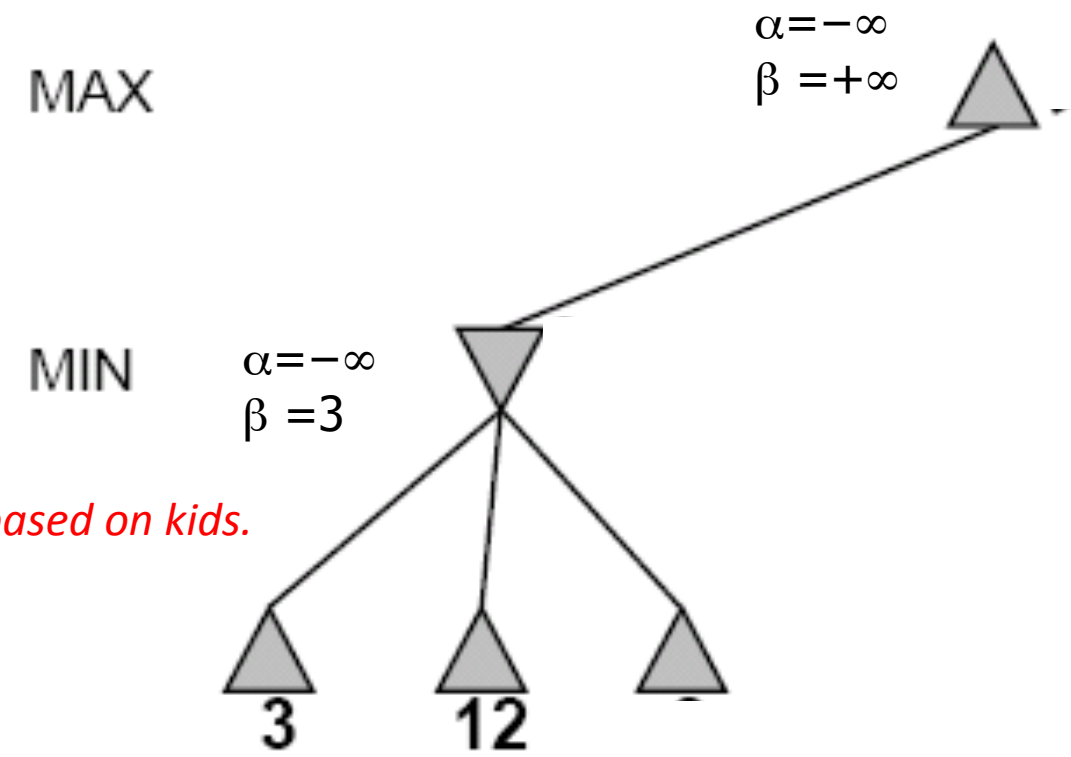


Review Detailed Example of Alpha-Beta Pruning in lecture slides.

Alpha-Beta Example (continued)

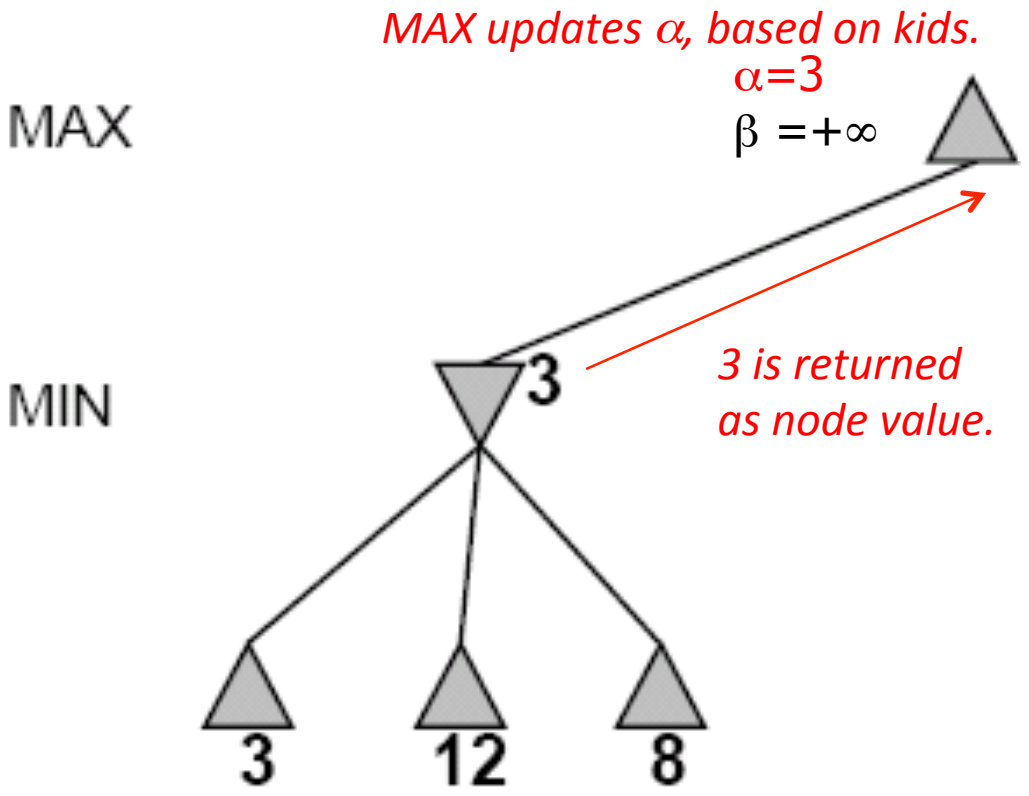


Alpha-Beta Example (continued)

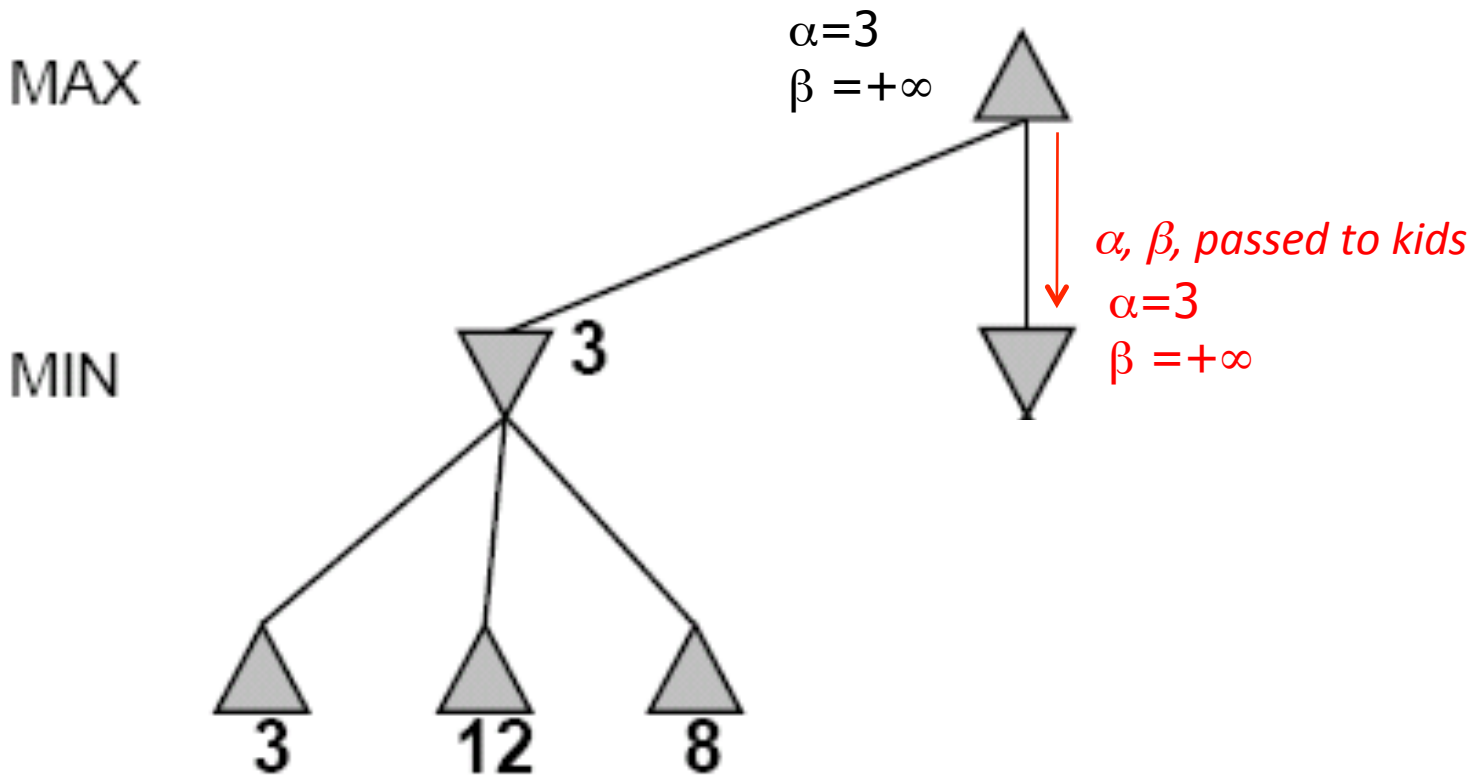


MIN updates β, based on kids.
No change.

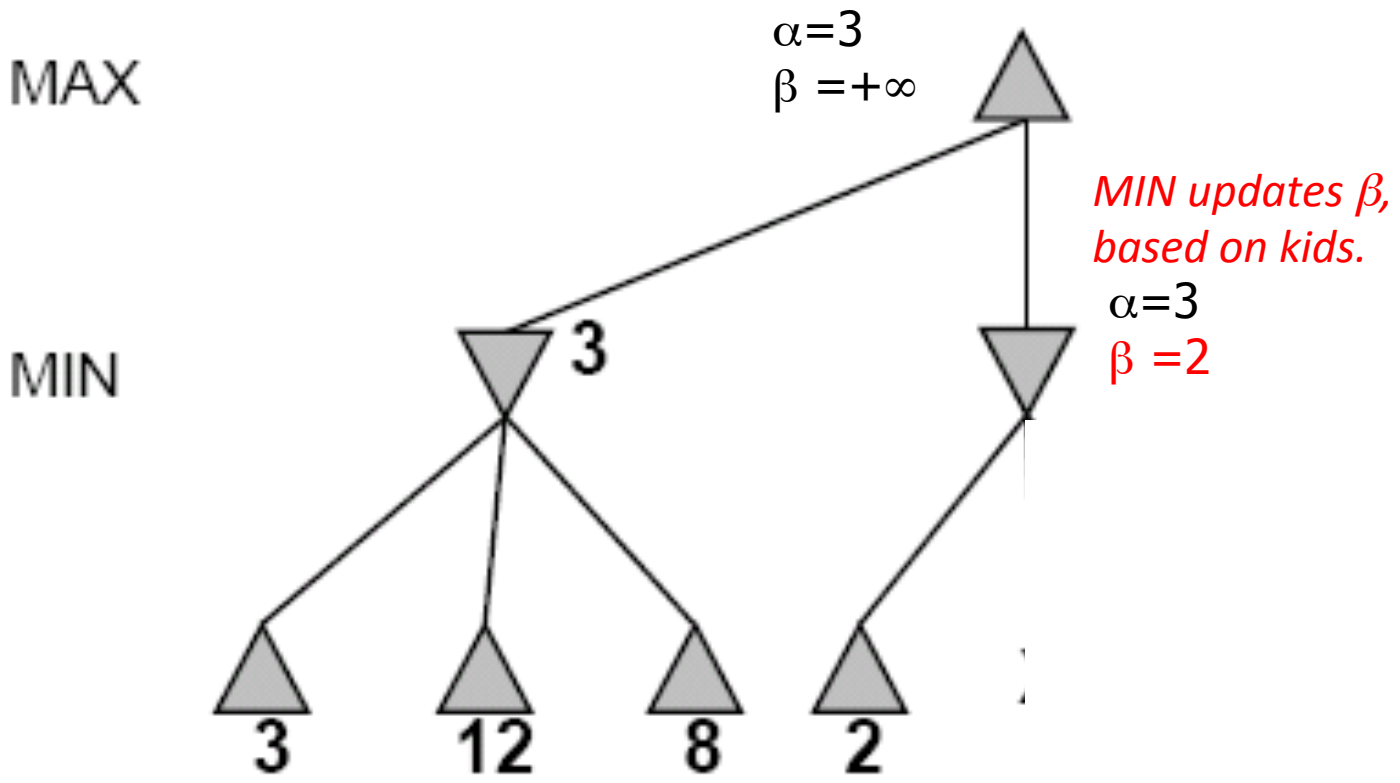
Alpha-Beta Example (continued)



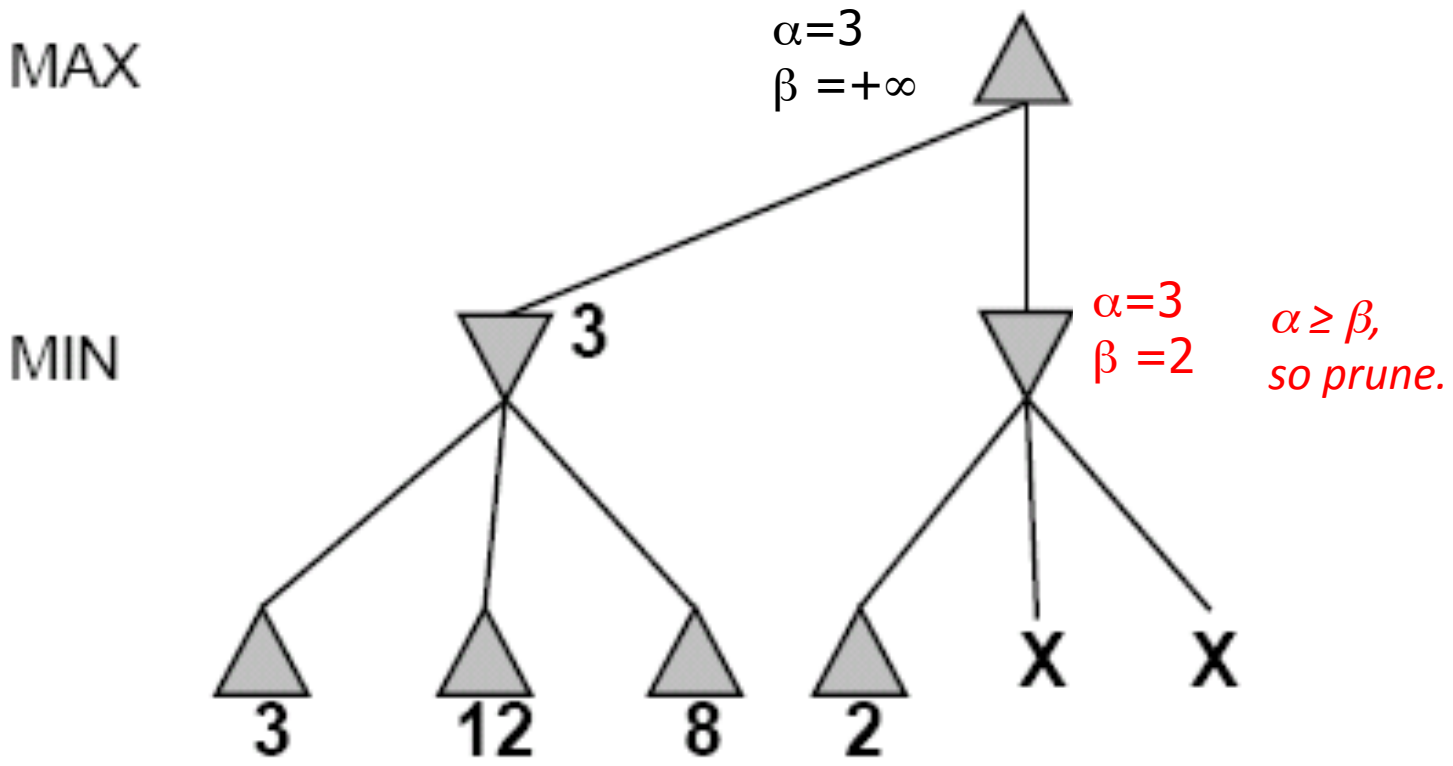
Alpha-Beta Example (continued)



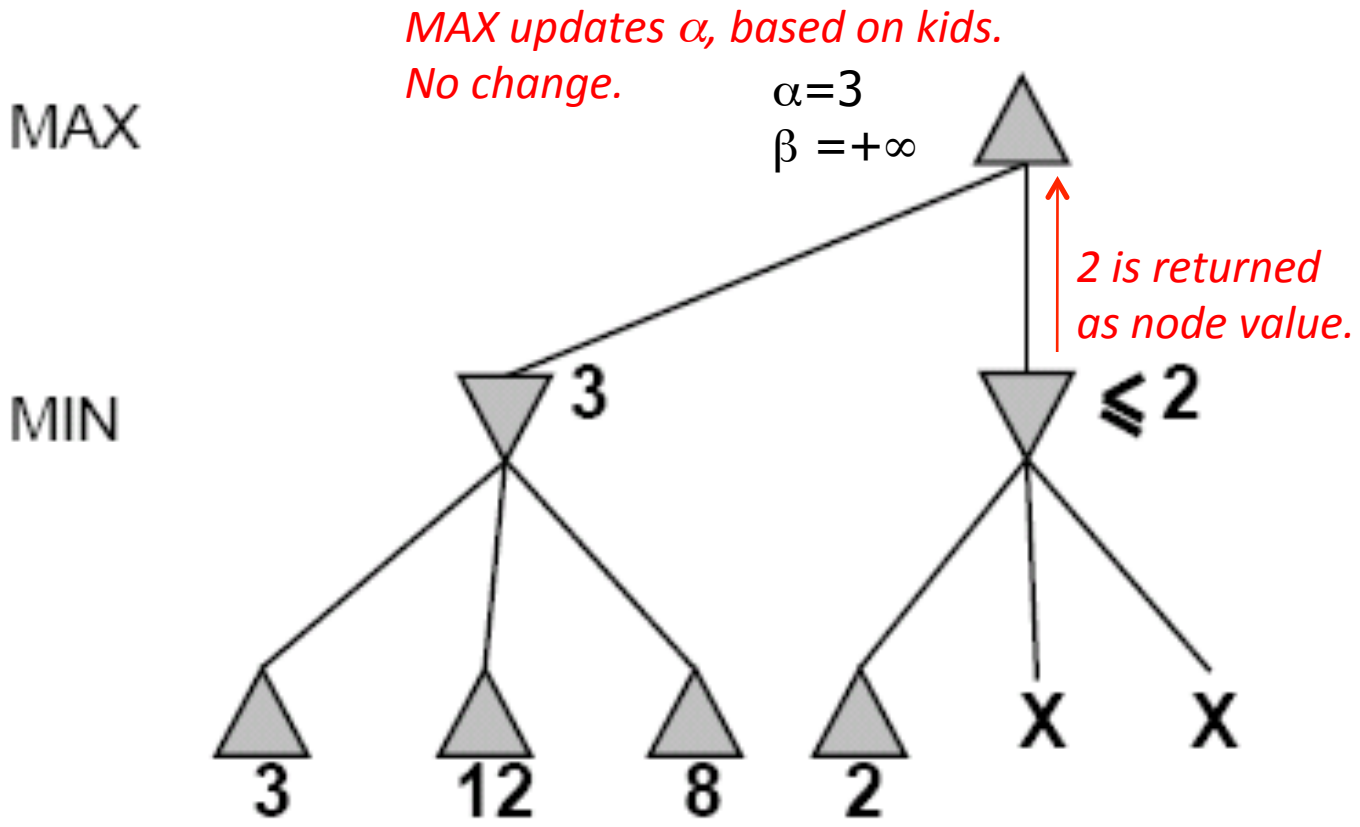
Alpha-Beta Example (continued)



Alpha-Beta Example (continued)



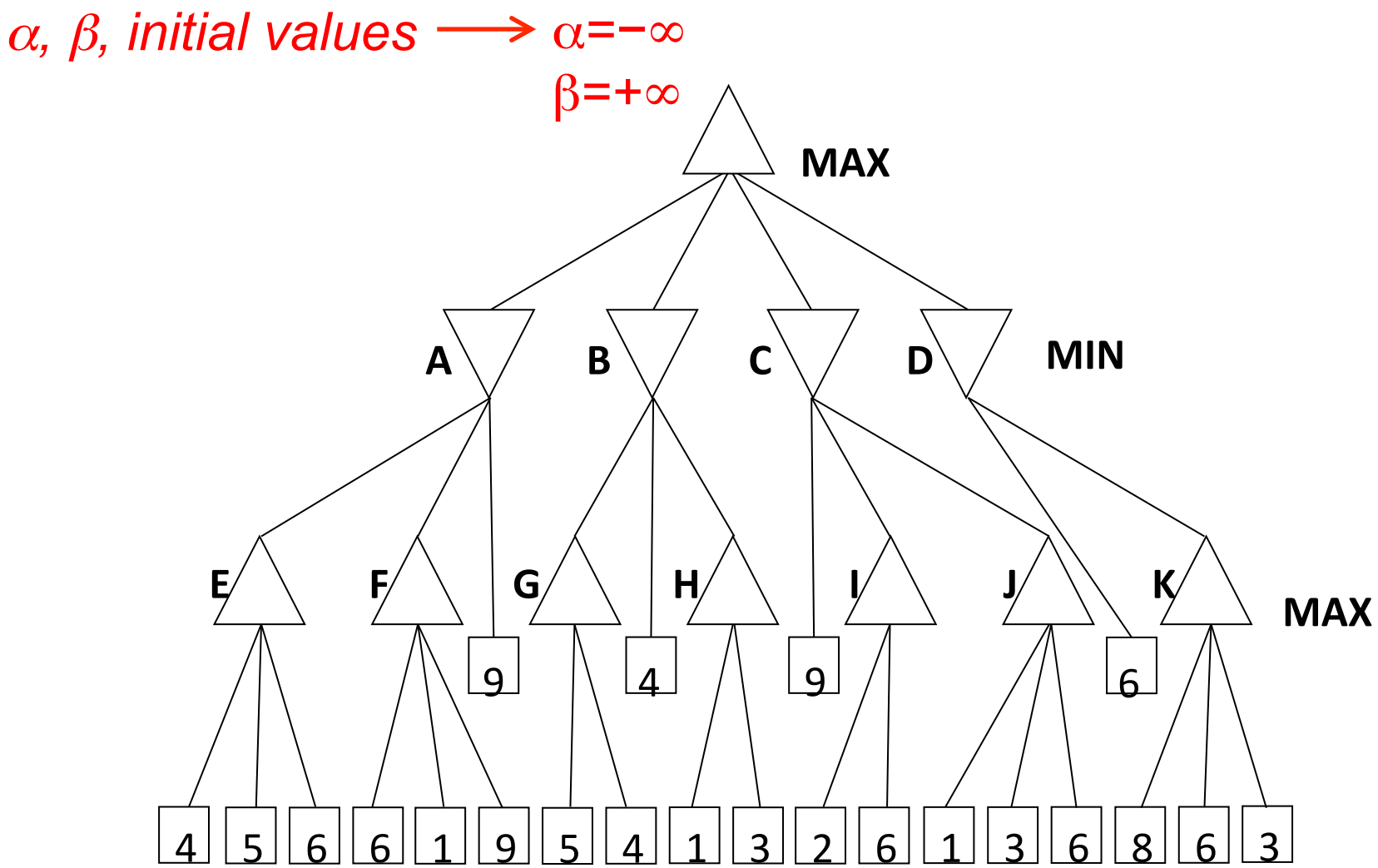
Alpha-Beta Example (continued)



Review Detailed Example of Alpha-Beta Pruning in lecture slides.

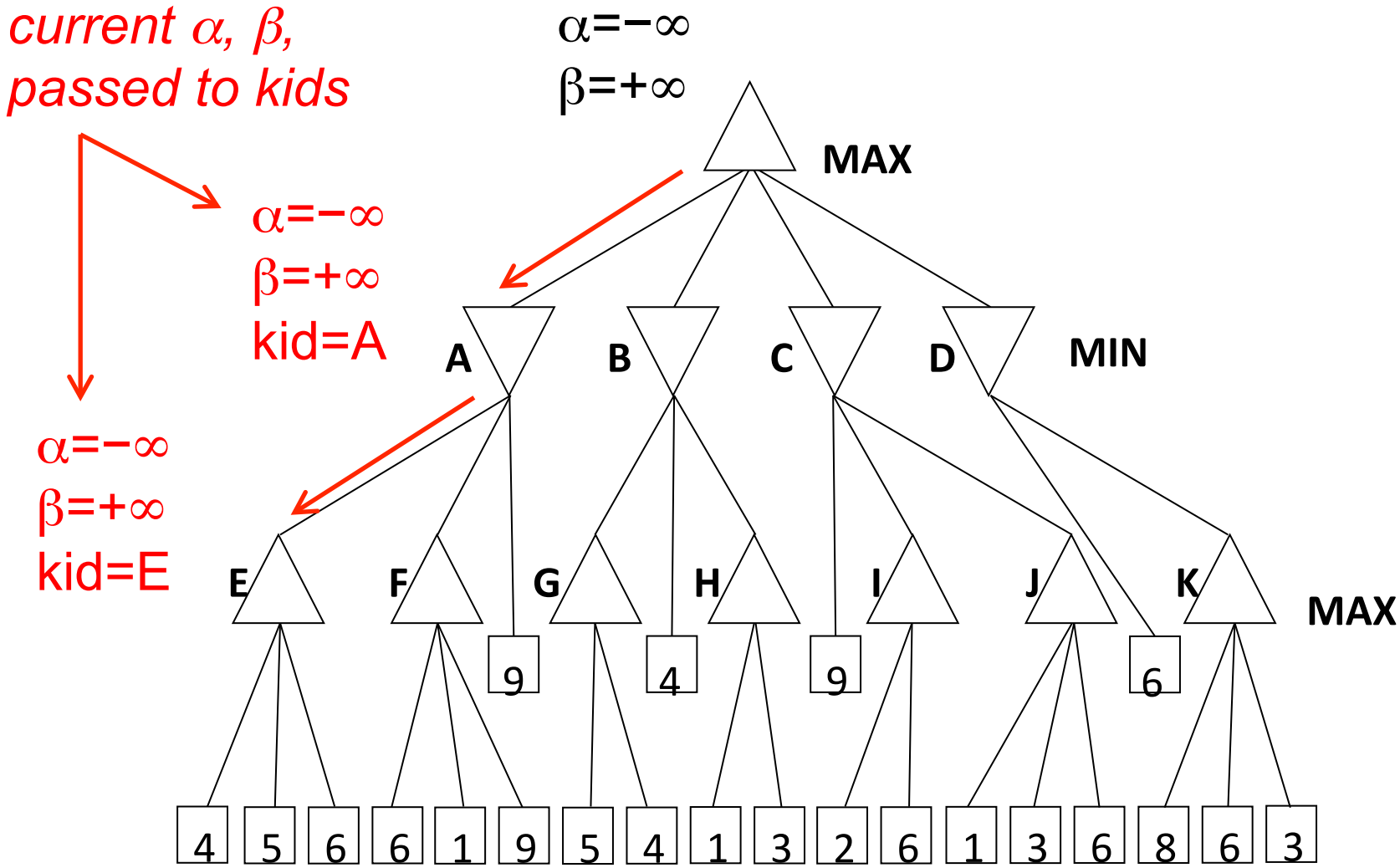
Long Detailed Alpha-Beta Example

Branch nodes are labeled A-K for easy discussion

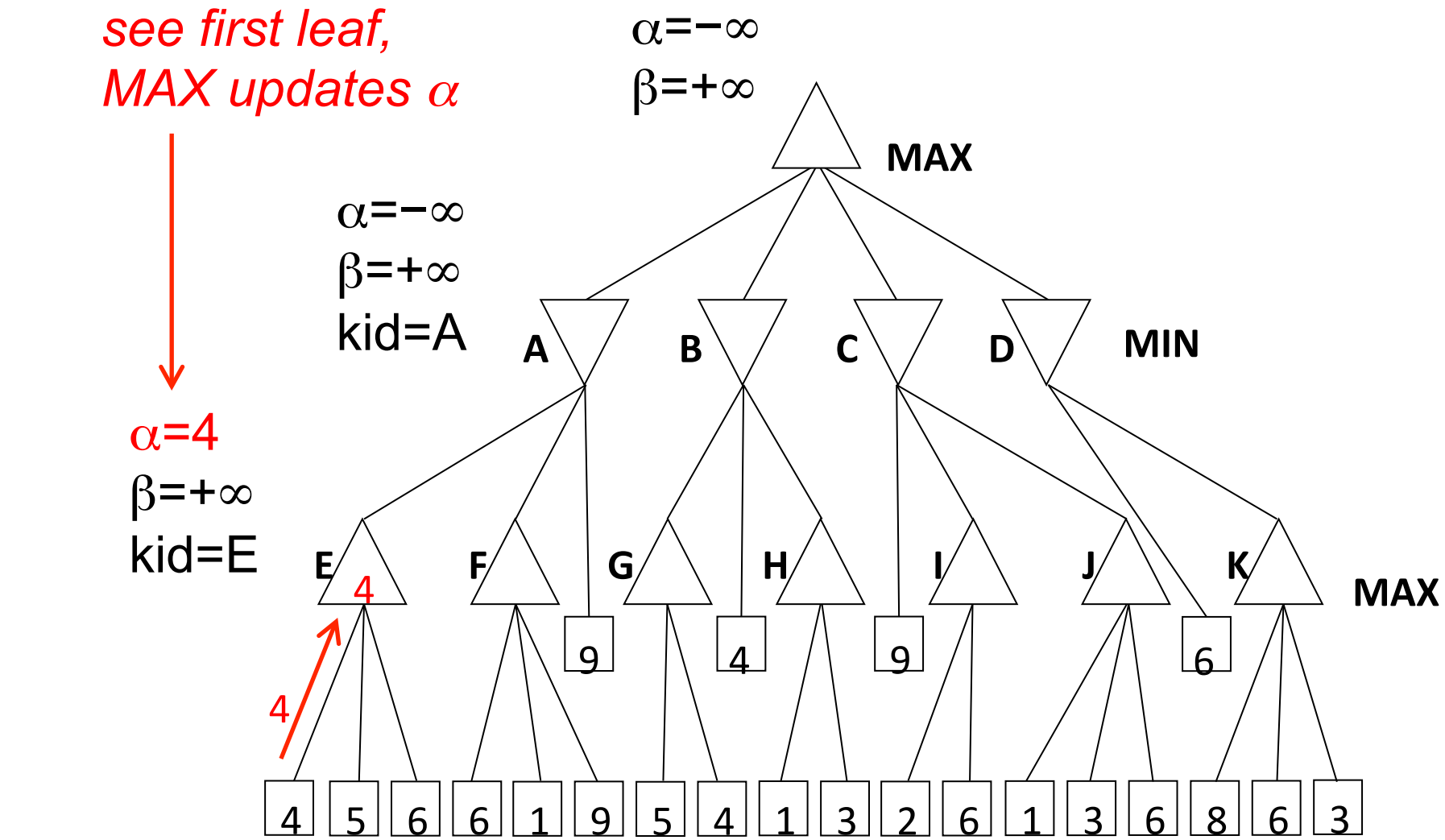


Long Detailed Alpha-Beta Example

Note that search cut-off occurs at different depths

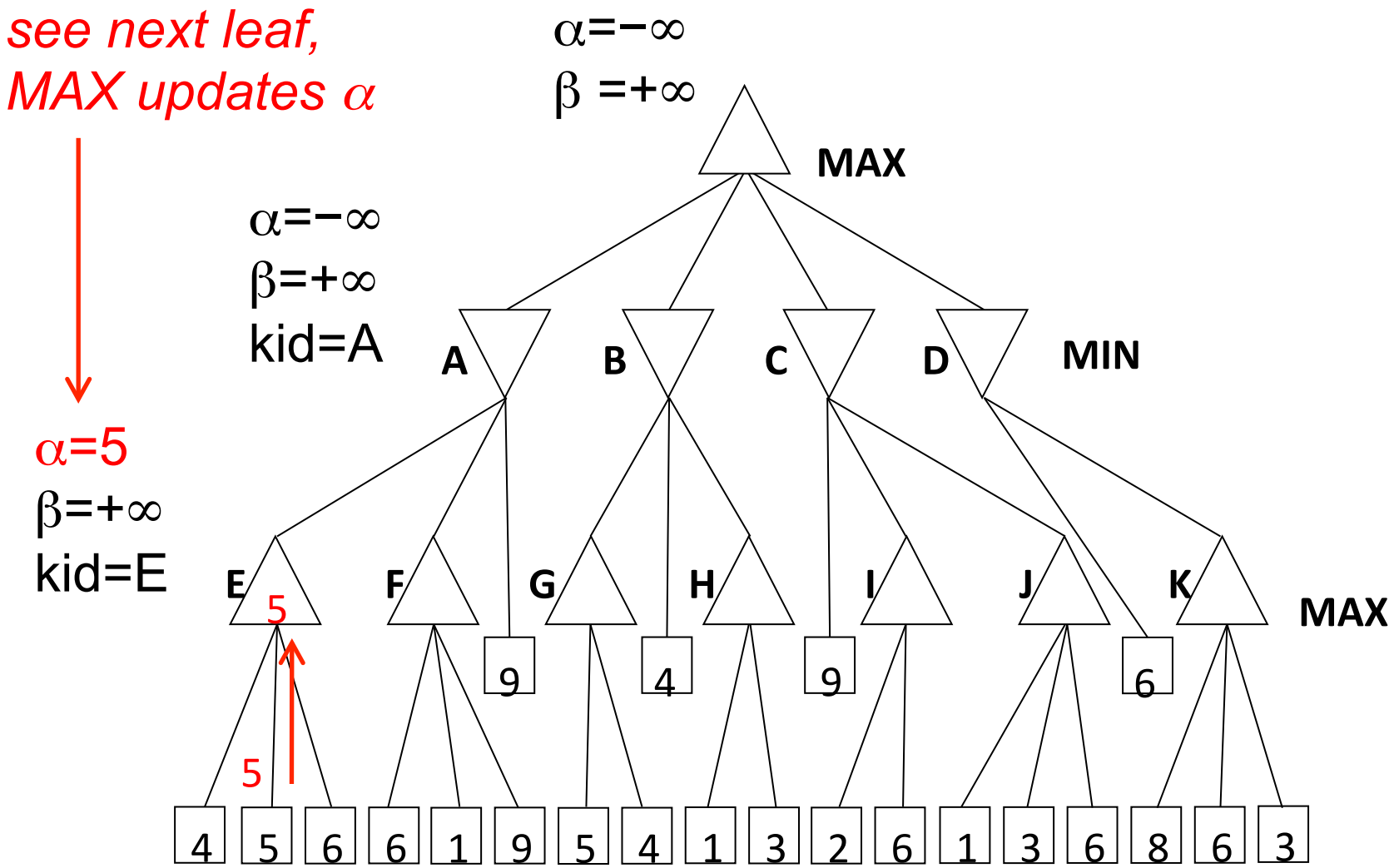


Long Detailed Alpha-Beta Example

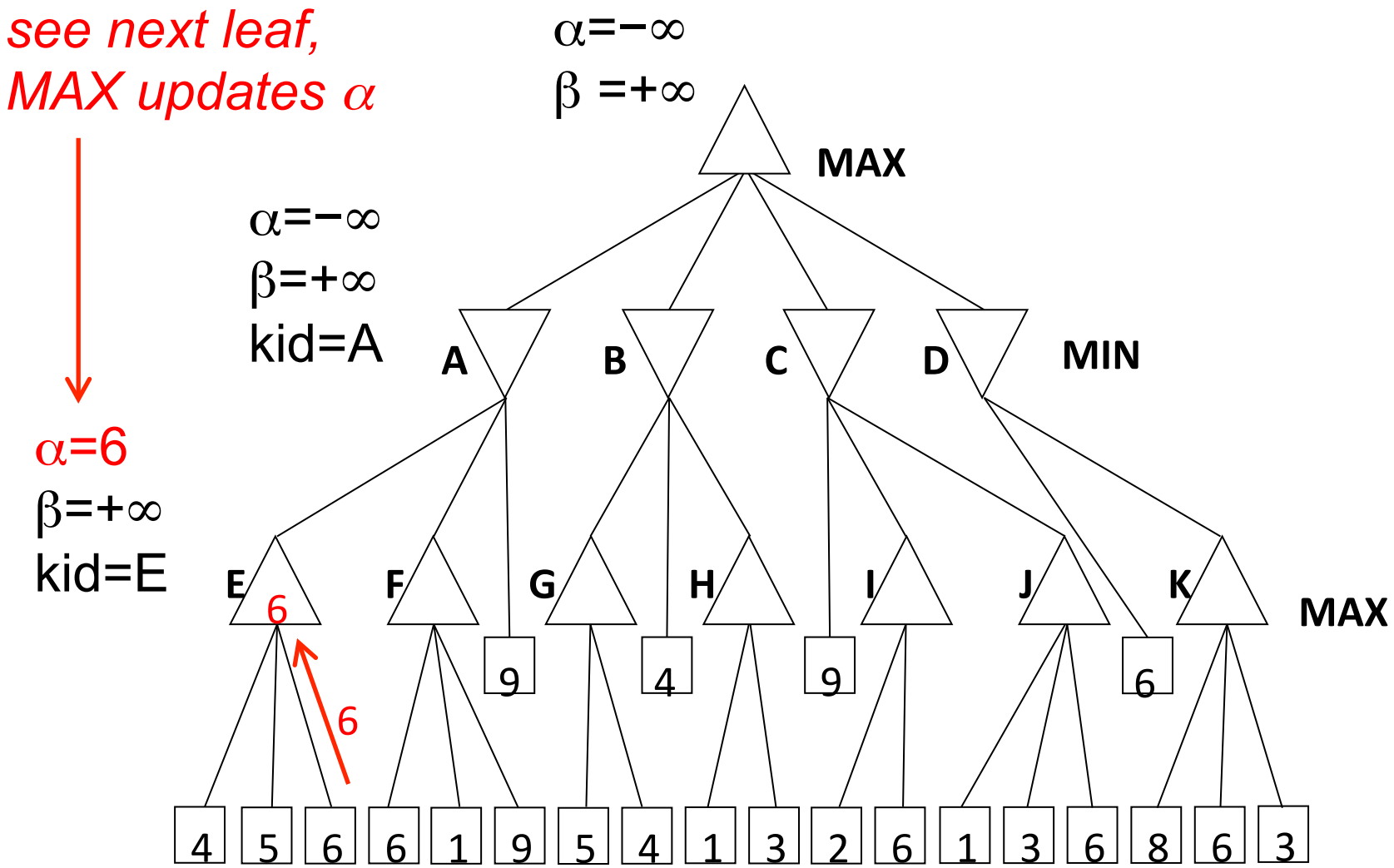


We also are running MiniMax search and recording node values within the triangles, without explicit comment.

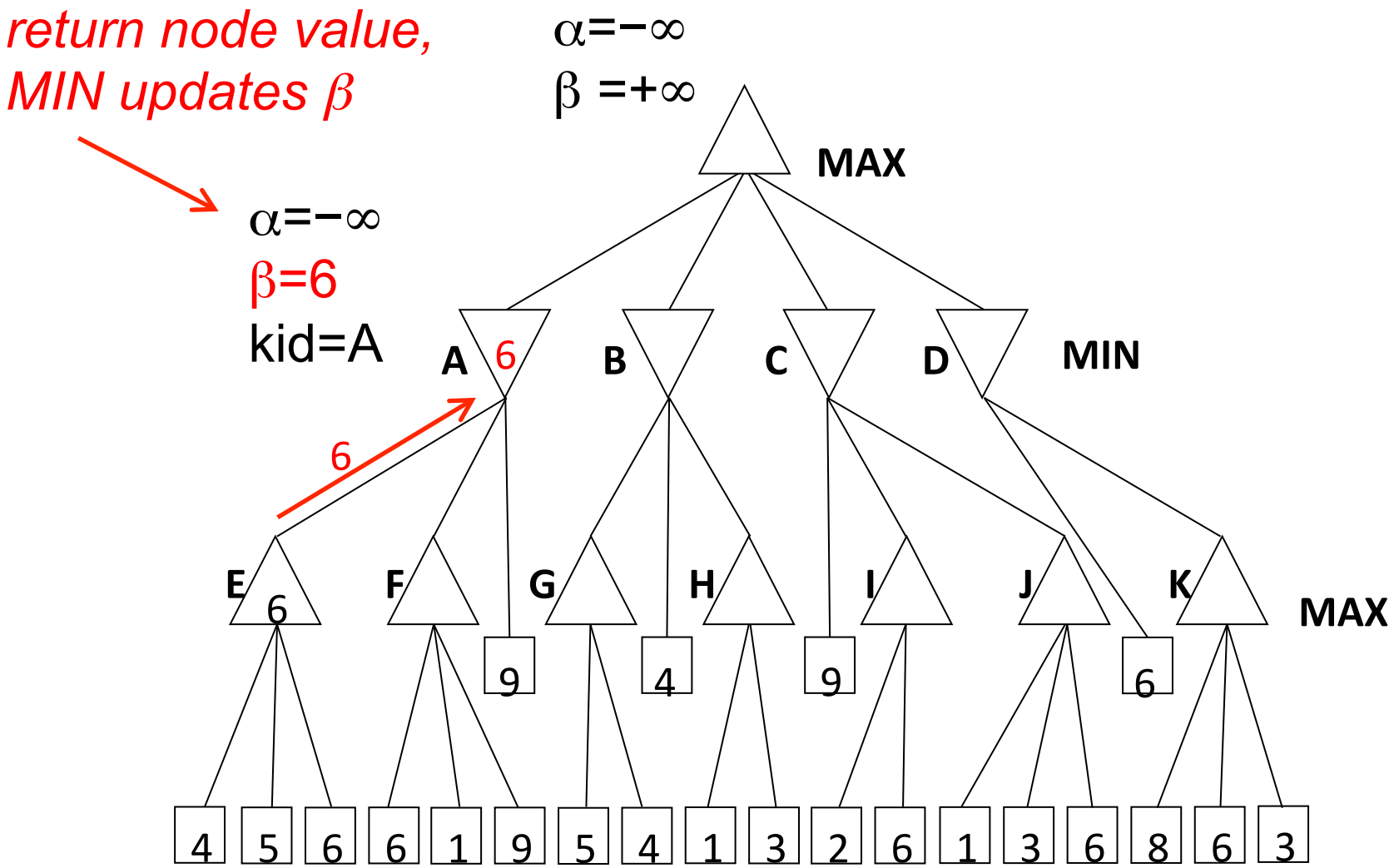
Long Detailed Alpha-Beta Example



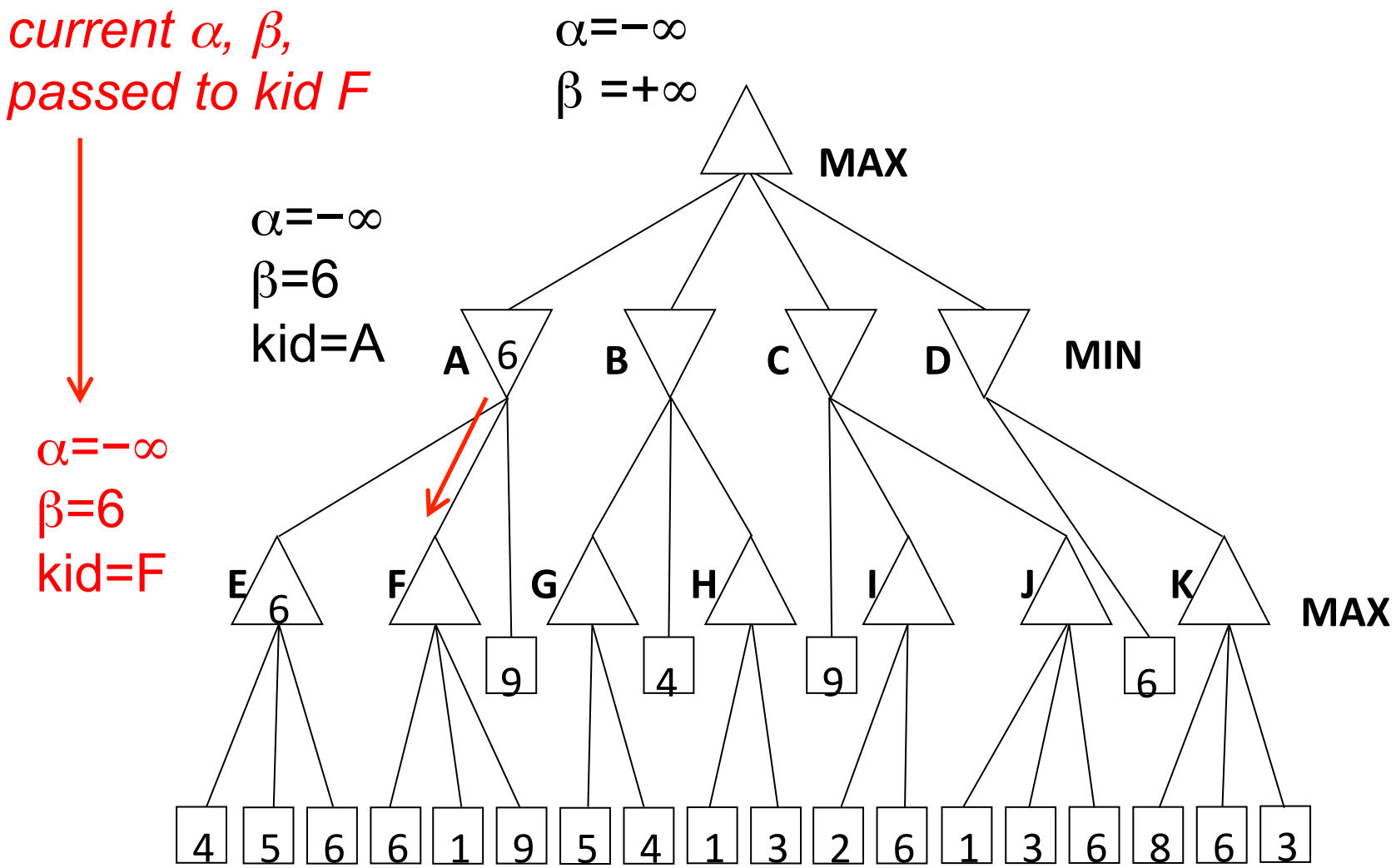
Long Detailed Alpha-Beta Example



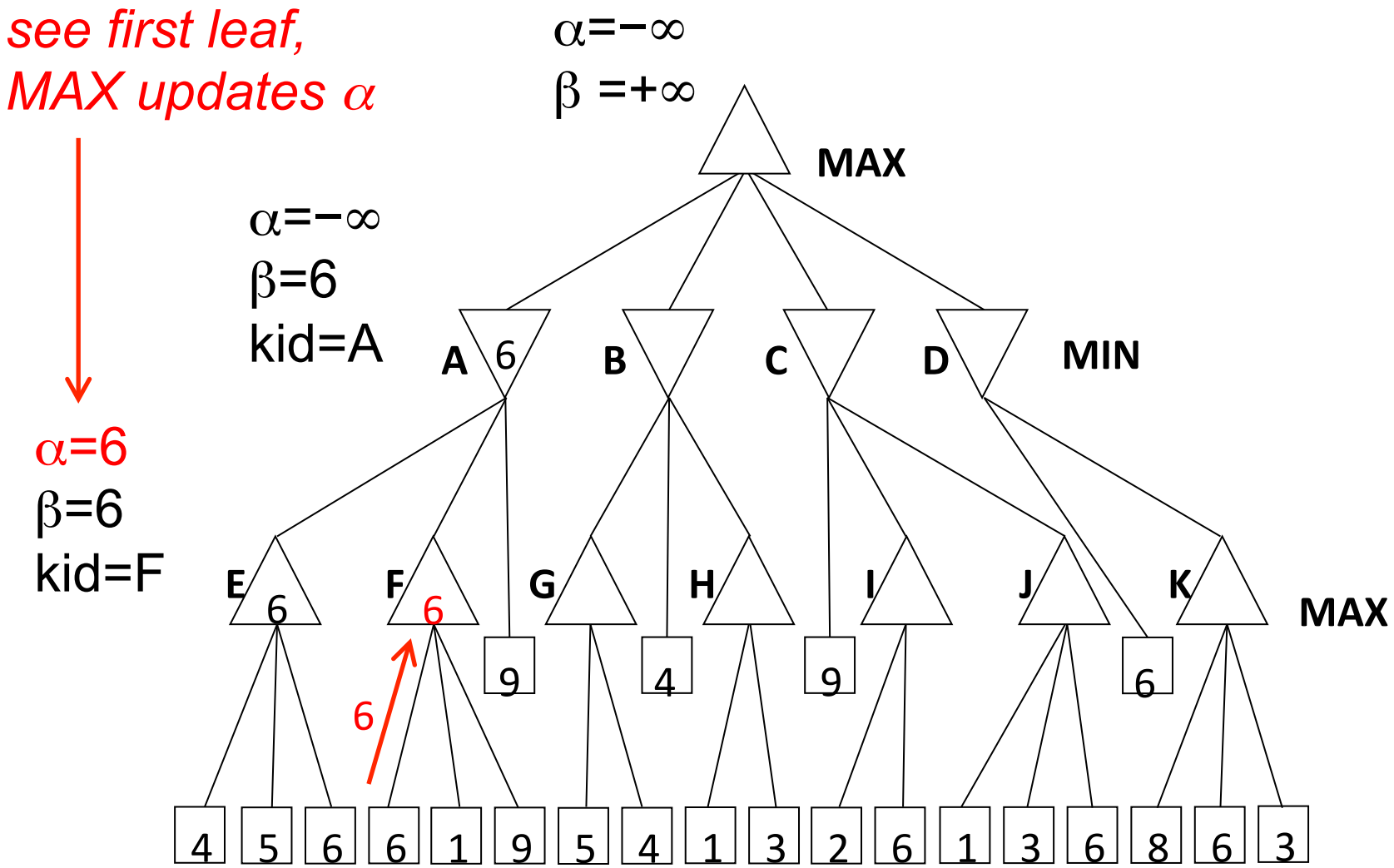
Long Detailed Alpha-Beta Example



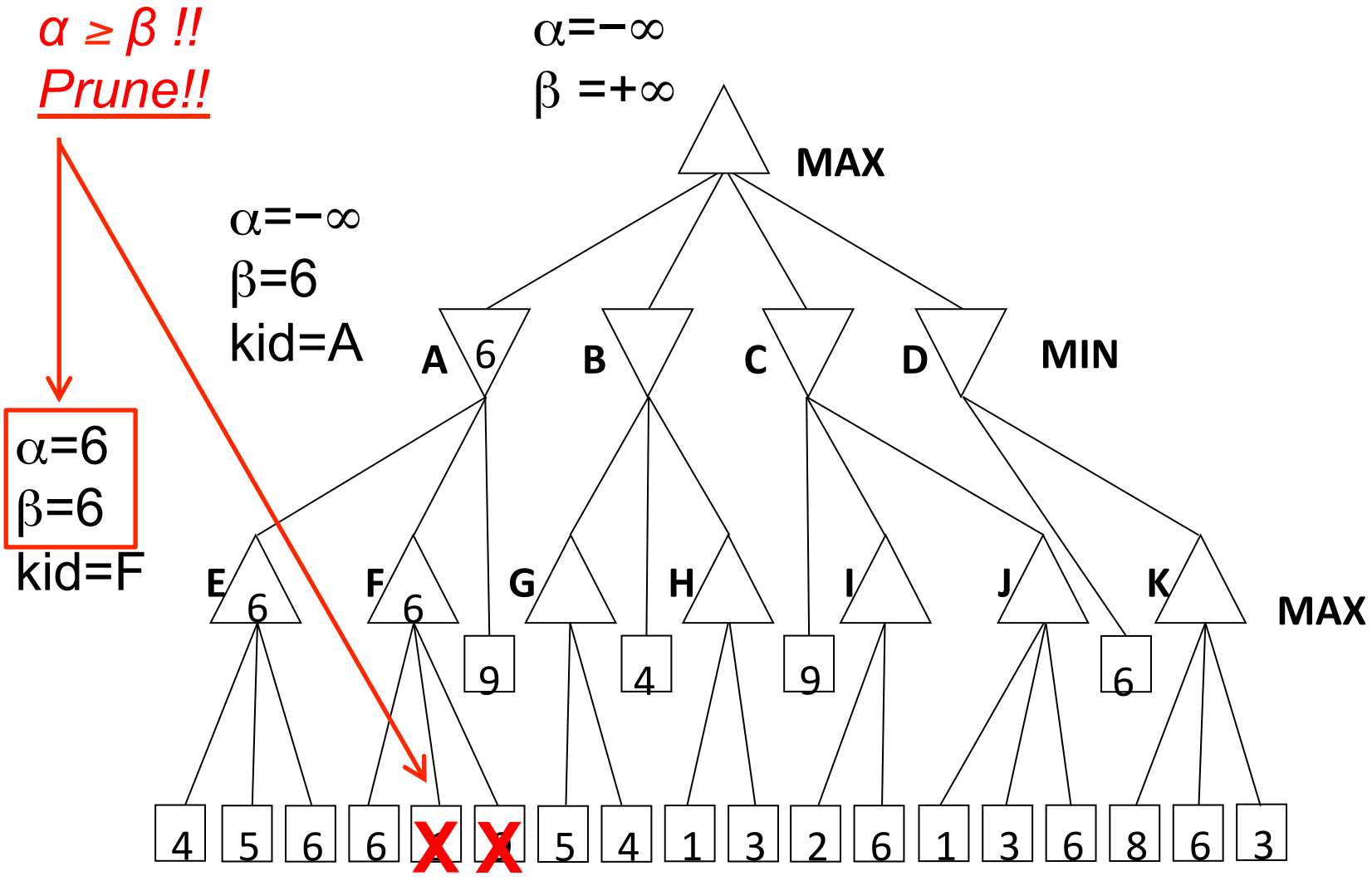
Long Detailed Alpha-Beta Example



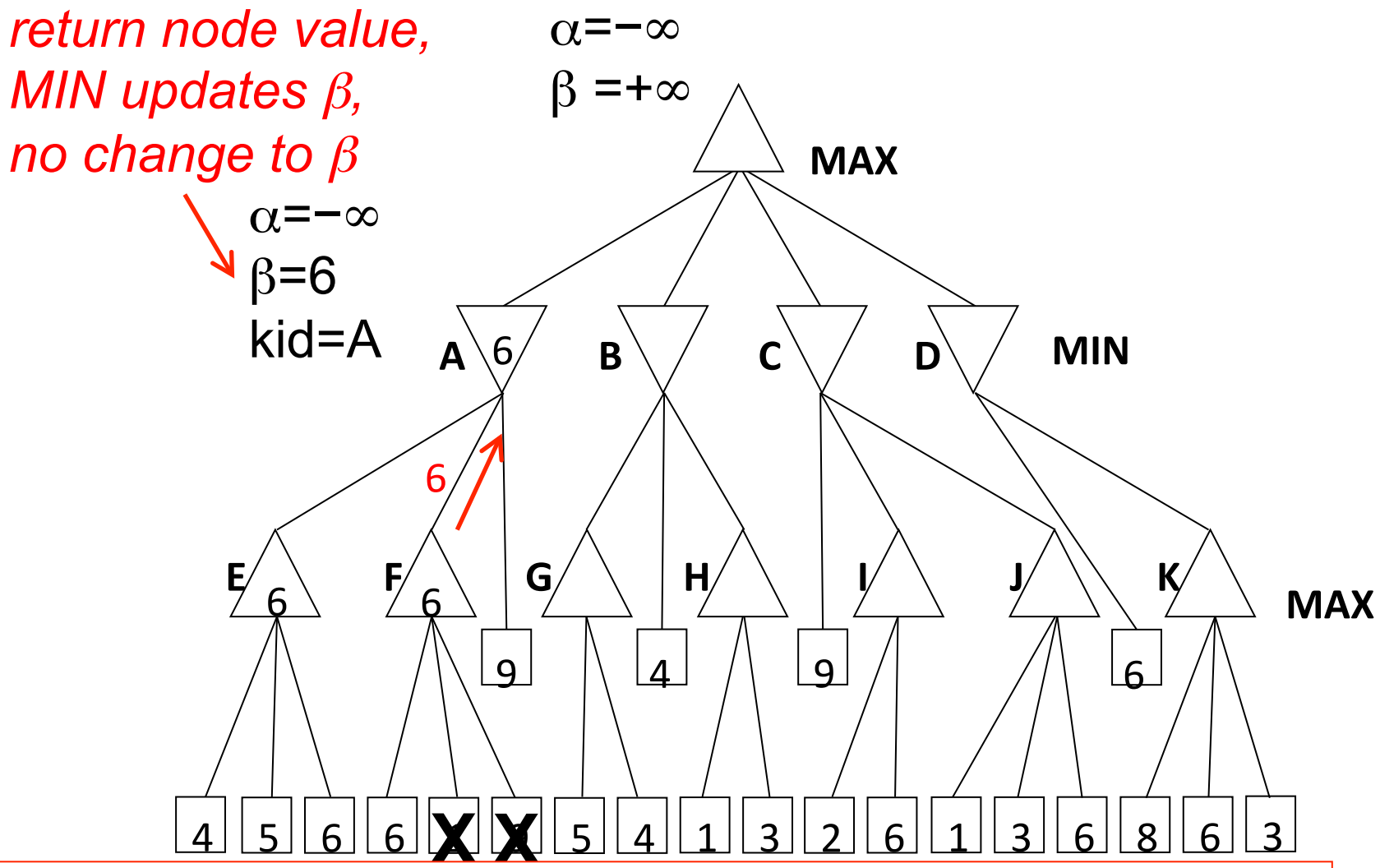
Long Detailed Alpha-Beta Example



Long Detailed Alpha-Beta Example

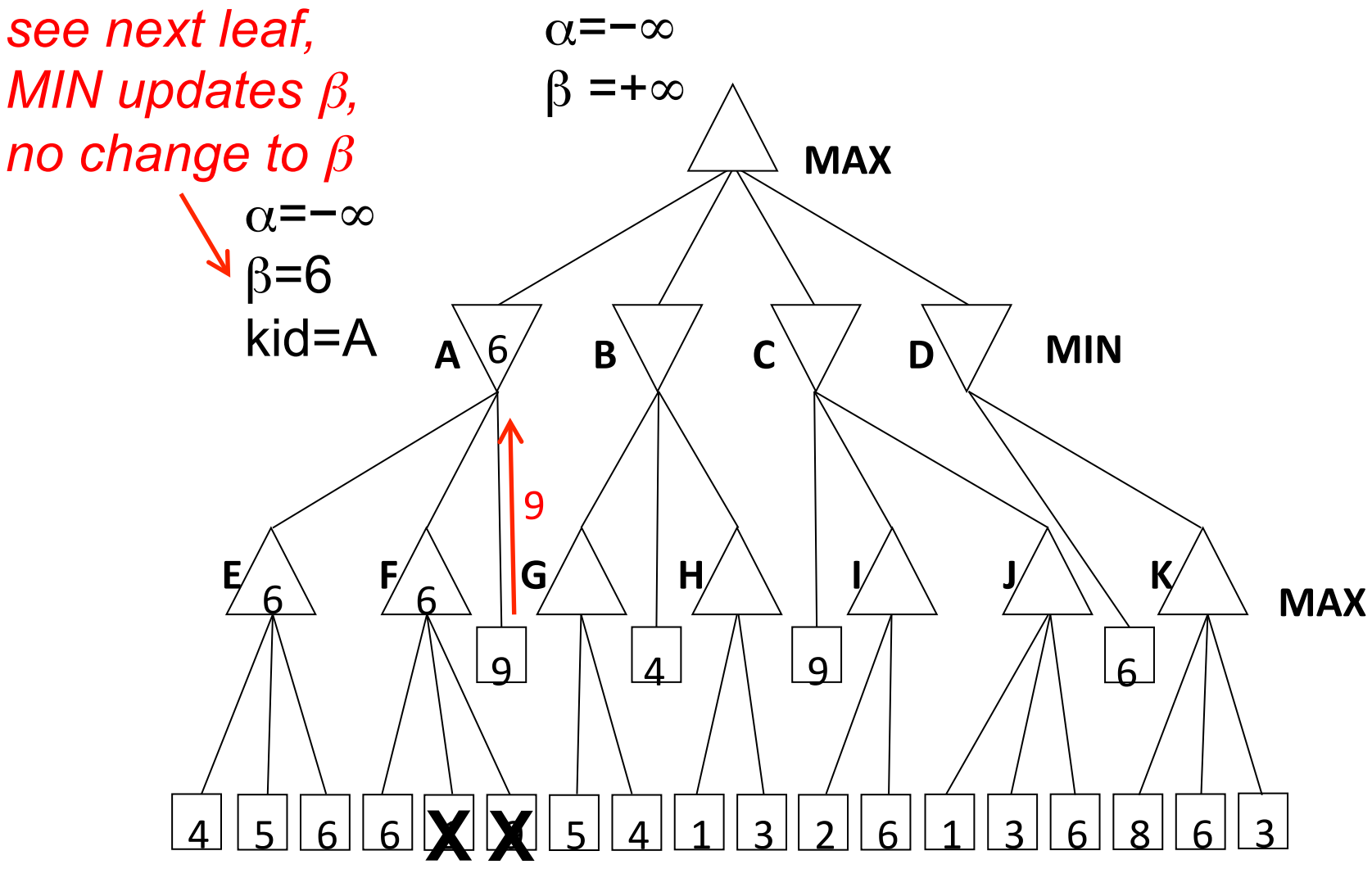


Long Detailed Alpha-Beta Example

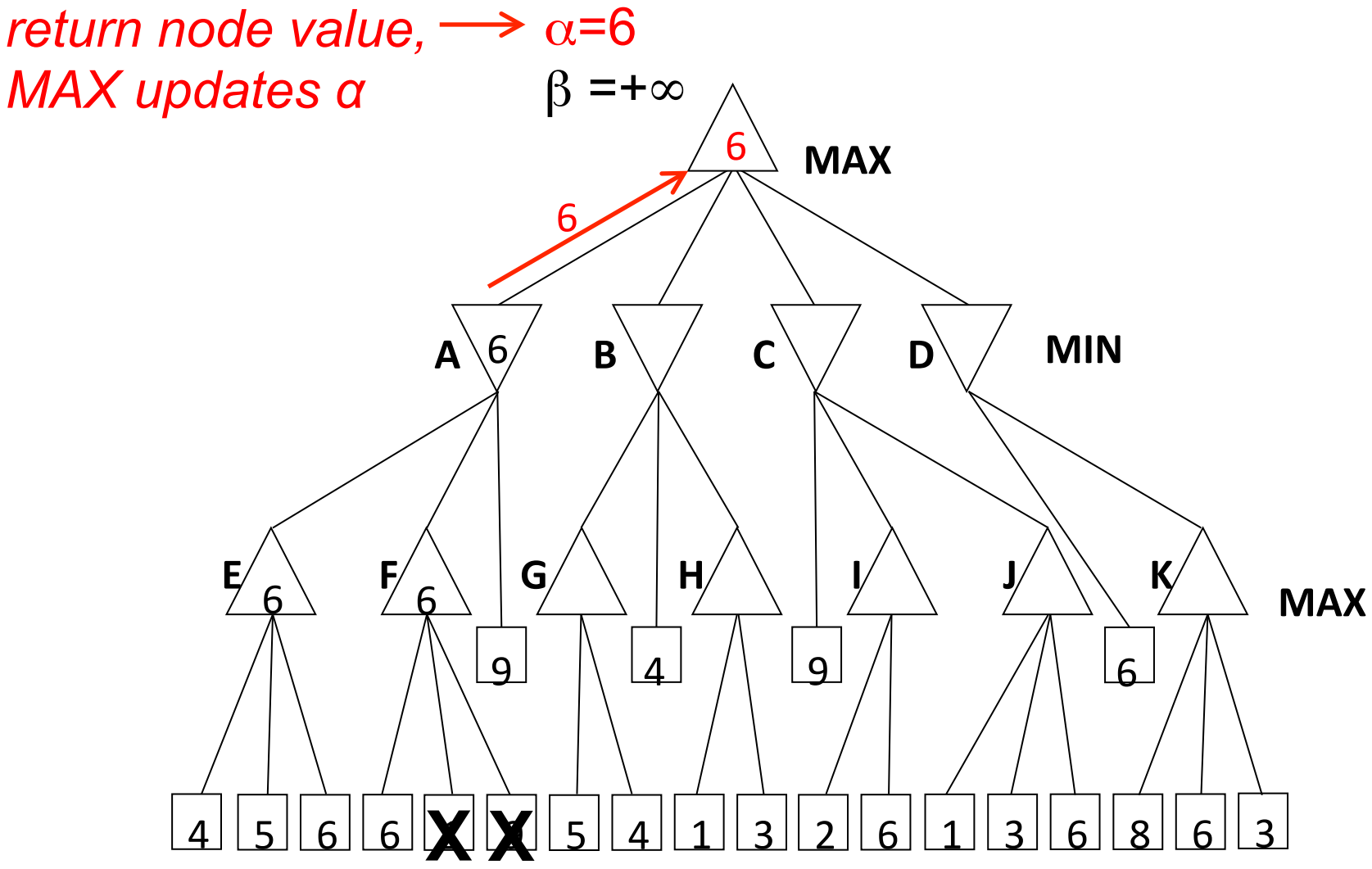


If we had continued searching at node F, we would see the 9 from its third leaf. Our returned value would be 9 instead of 6. But at A, MIN would choose E(=6) instead of F(=9). Internal values may change; root values do not.

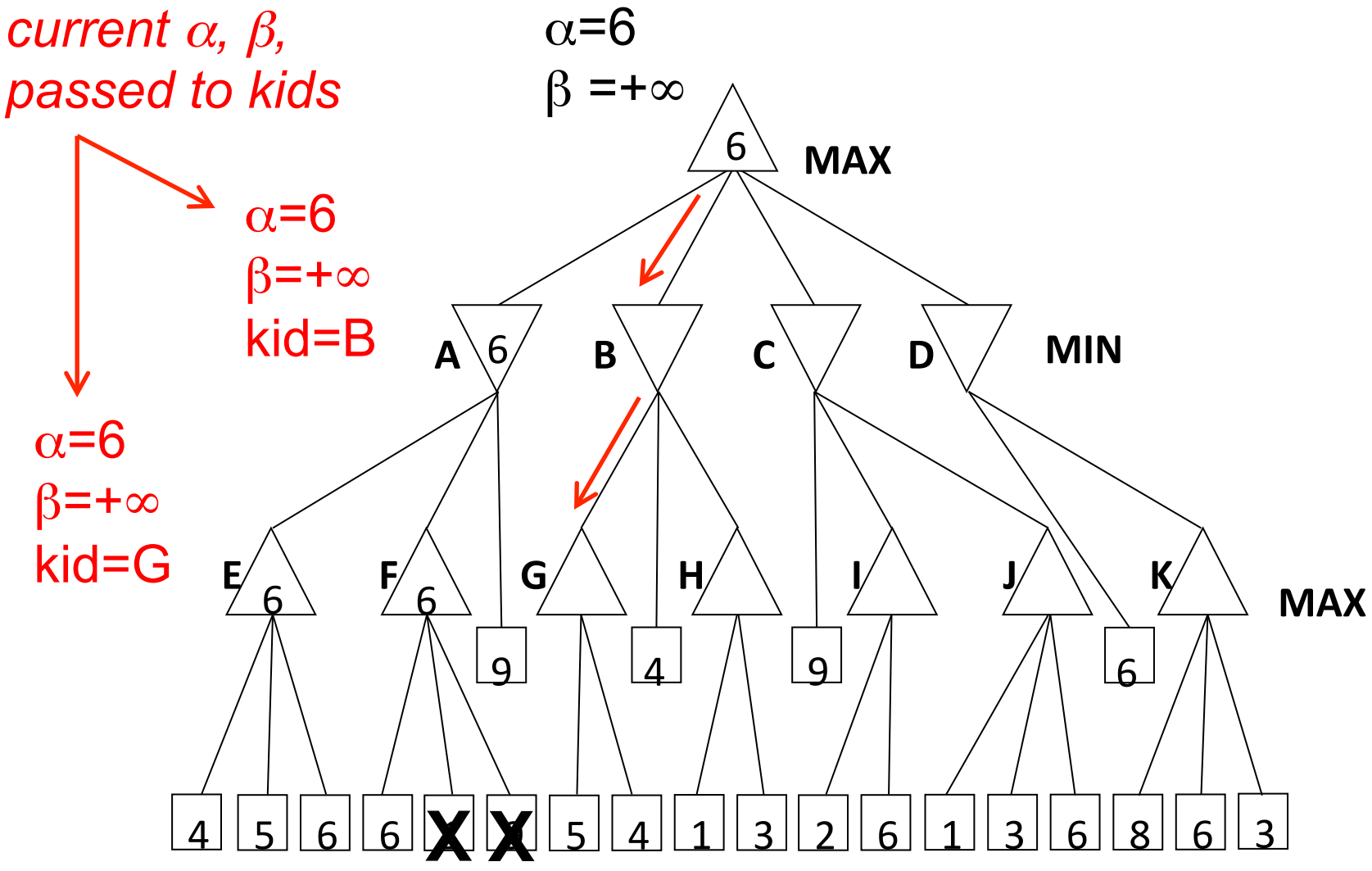
Long Detailed Alpha-Beta Example



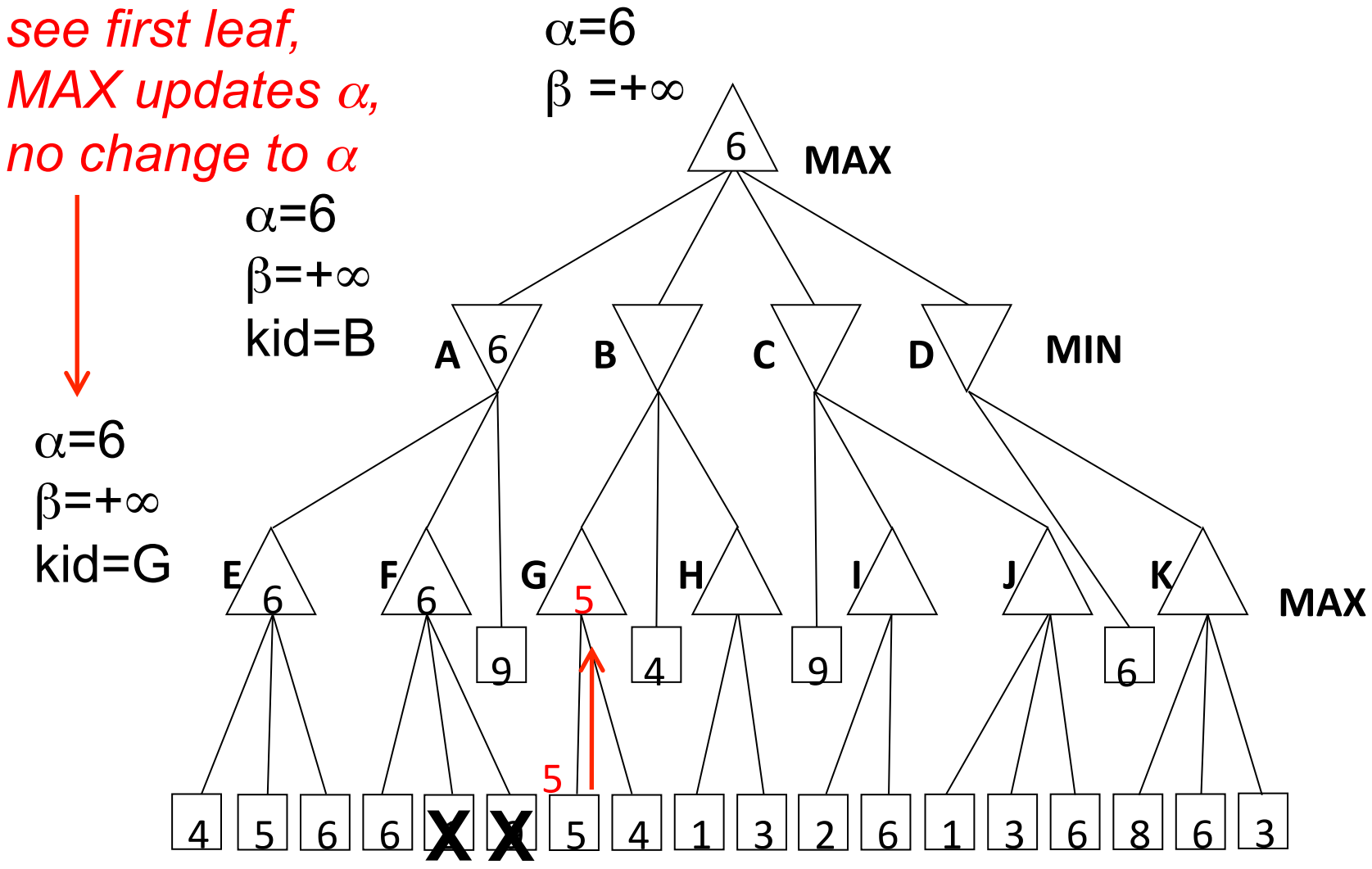
Long Detailed Alpha-Beta Example



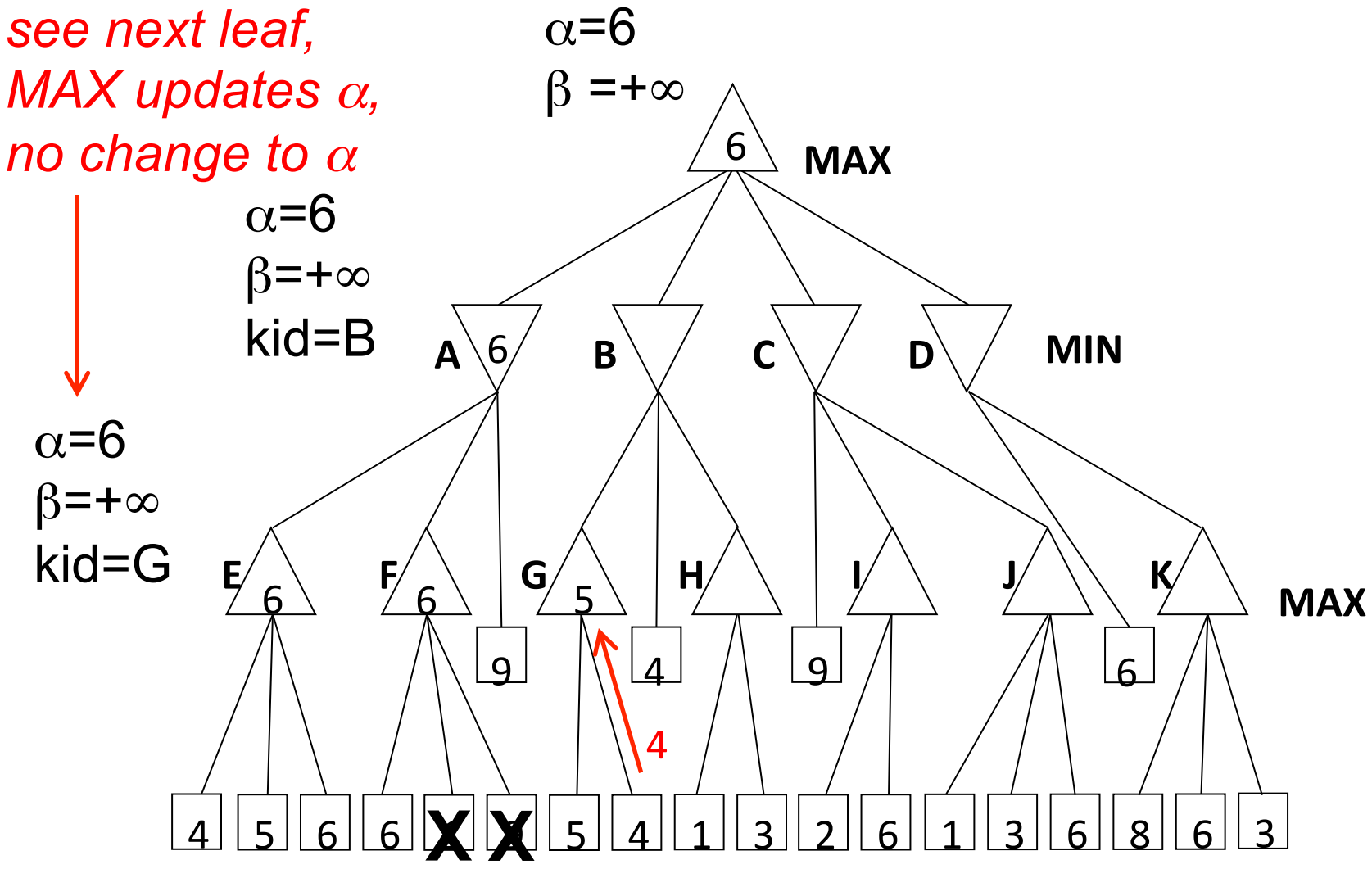
Long Detailed Alpha-Beta Example



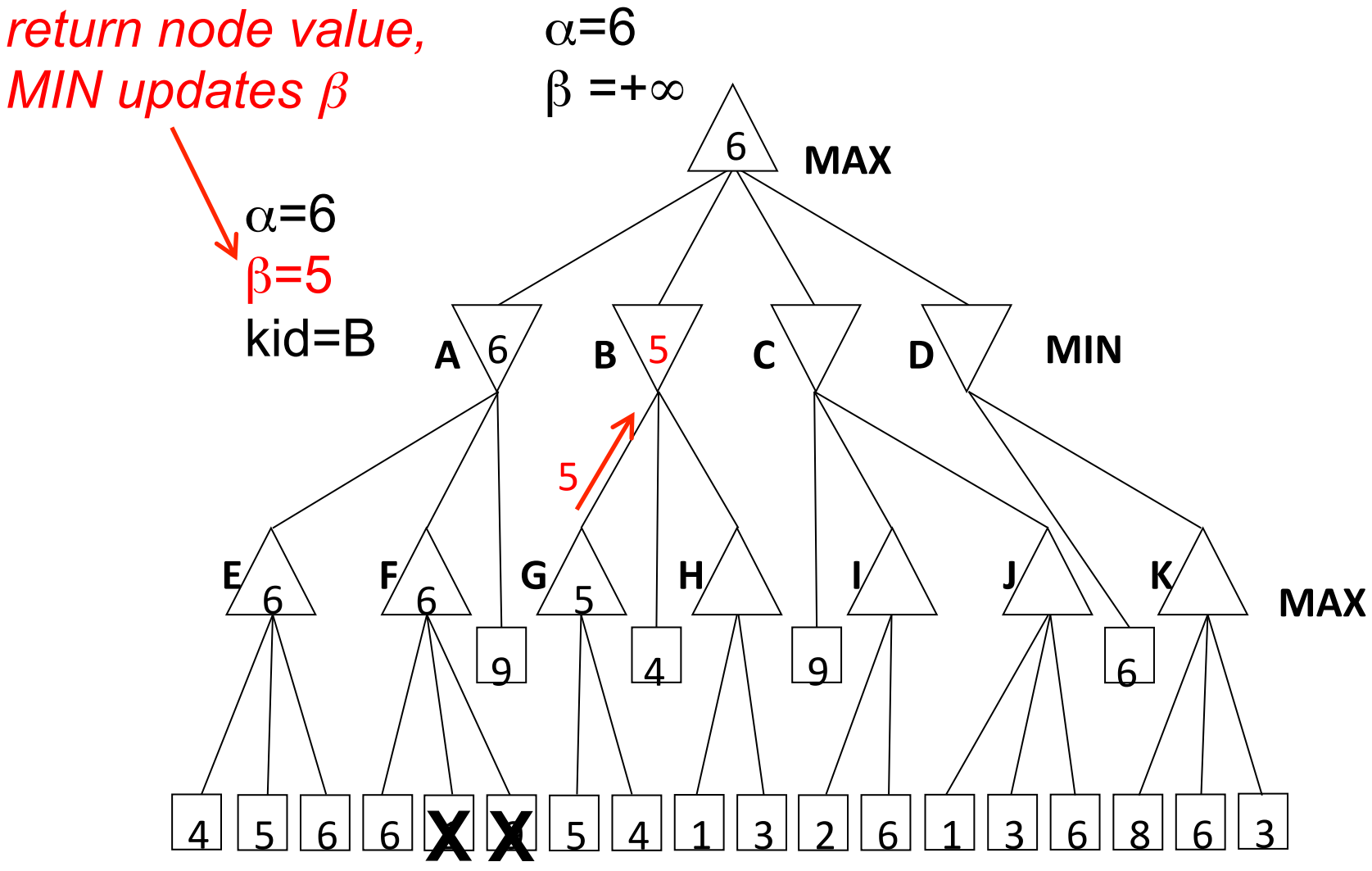
Long Detailed Alpha-Beta Example



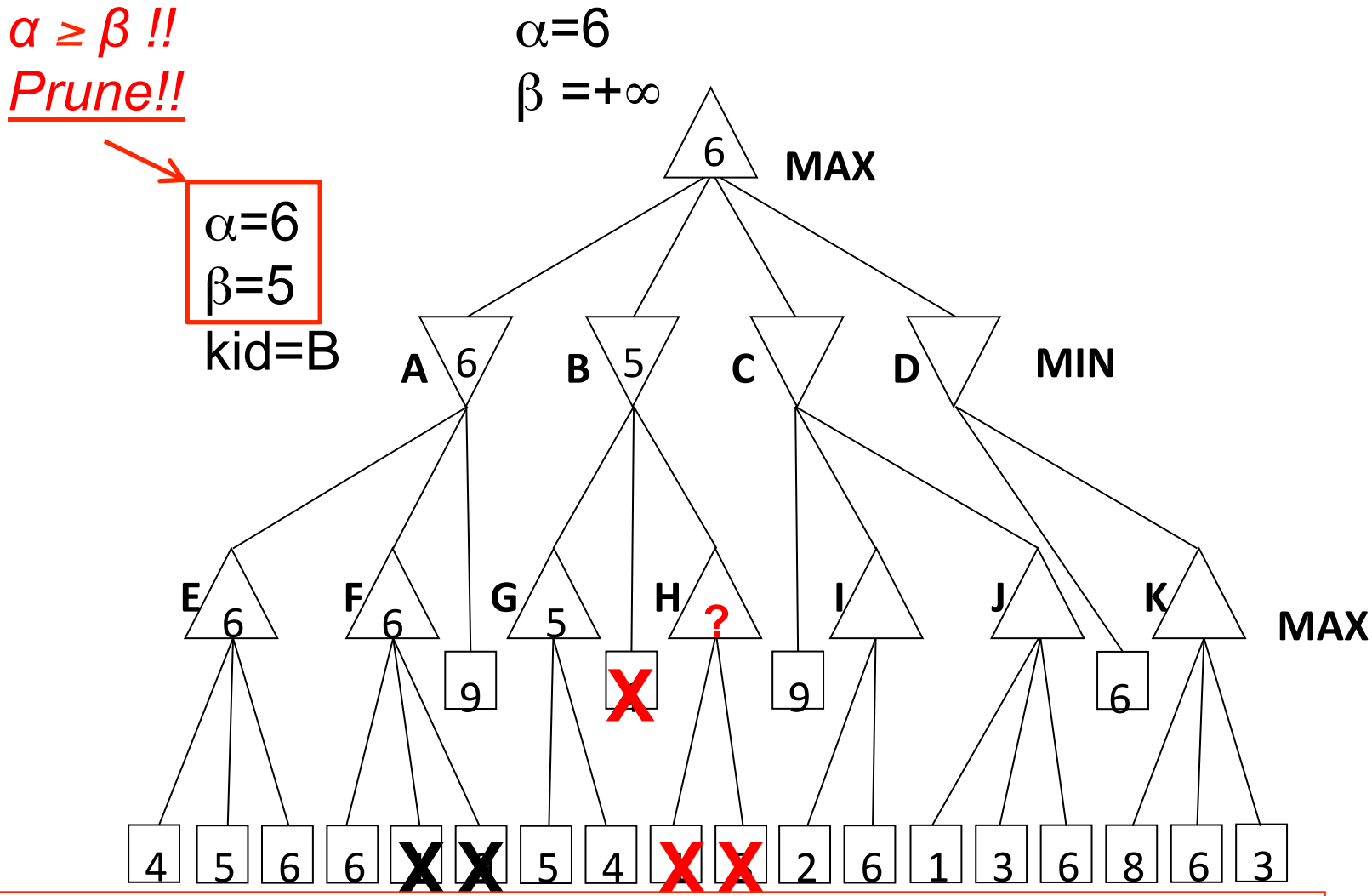
Long Detailed Alpha-Beta Example



Long Detailed Alpha-Beta Example



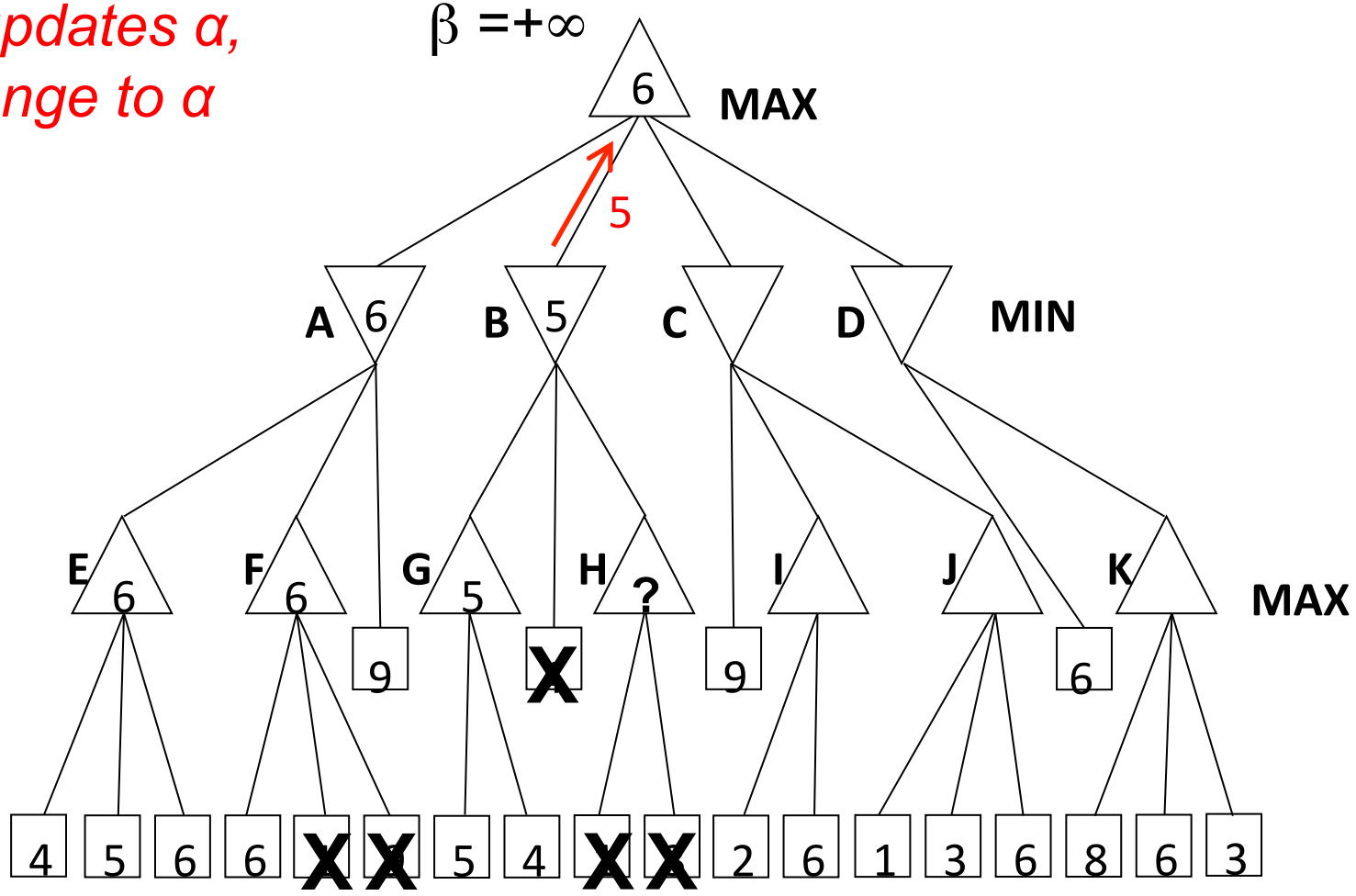
Long Detailed Alpha-Beta Example



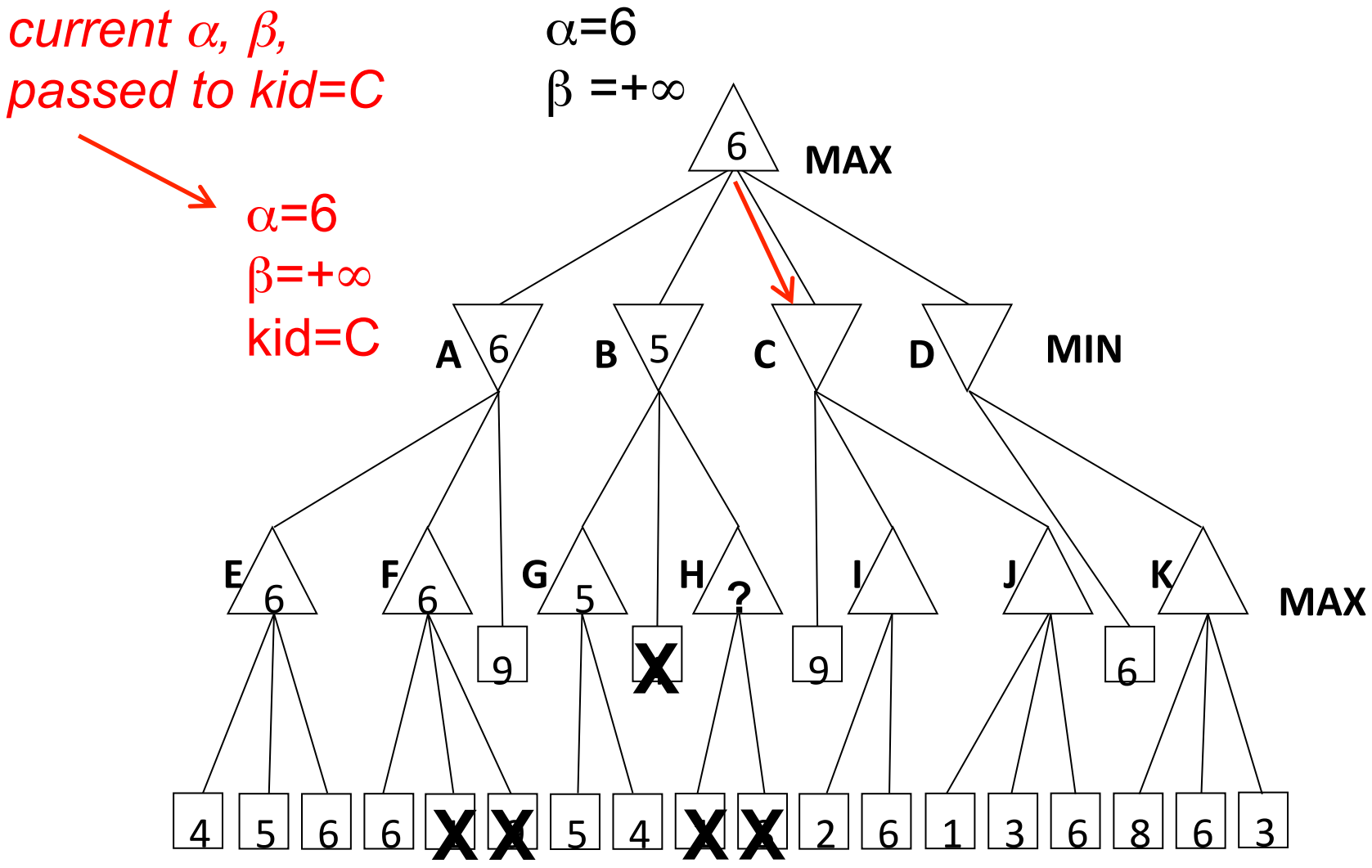
Note that we never find out, what is the node value of H? But we have proven it doesn't matter, so we don't care.

Long Detailed Alpha-Beta Example

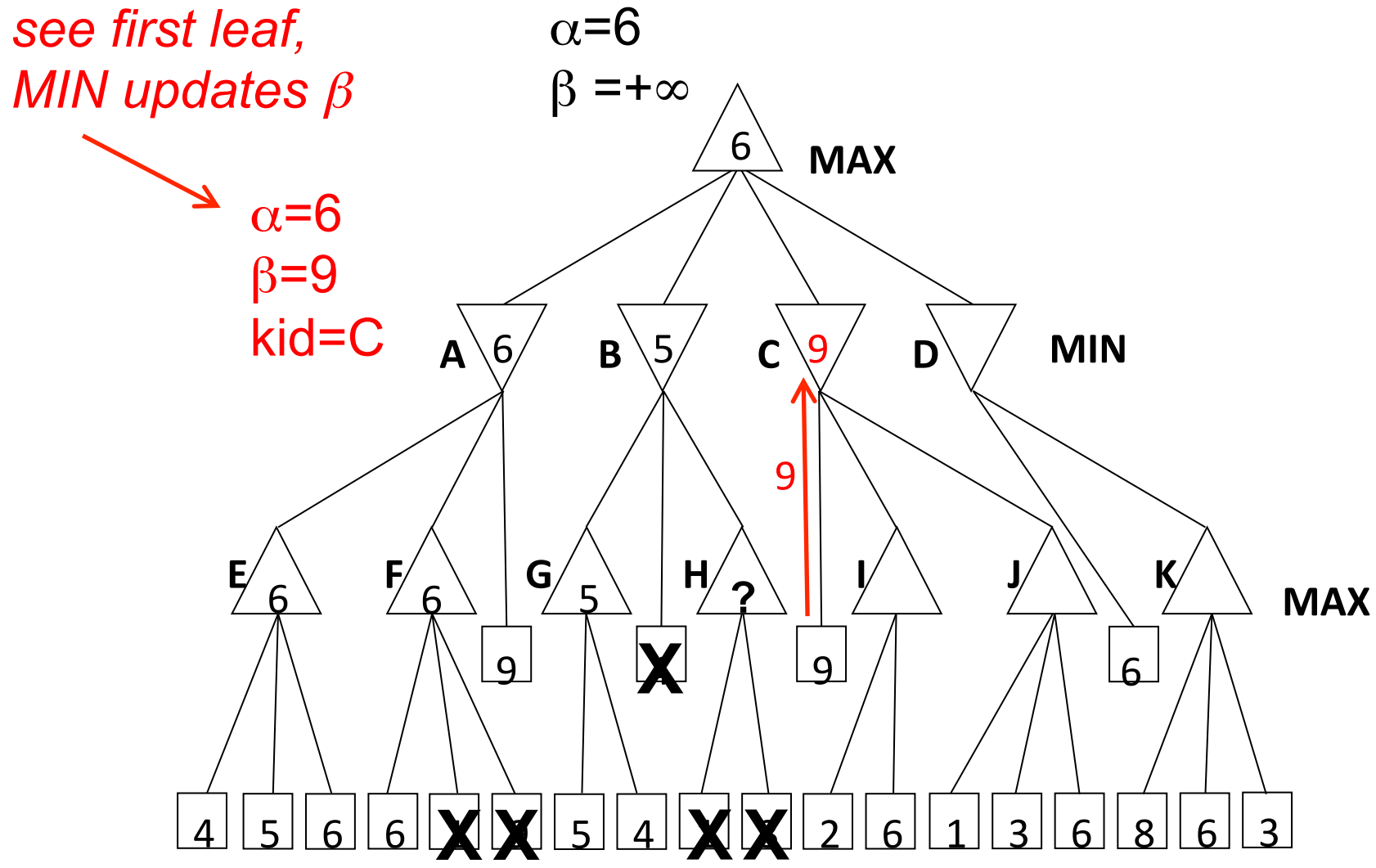
return node value, → α=6
MAX updates α, β =+∞
no change to α



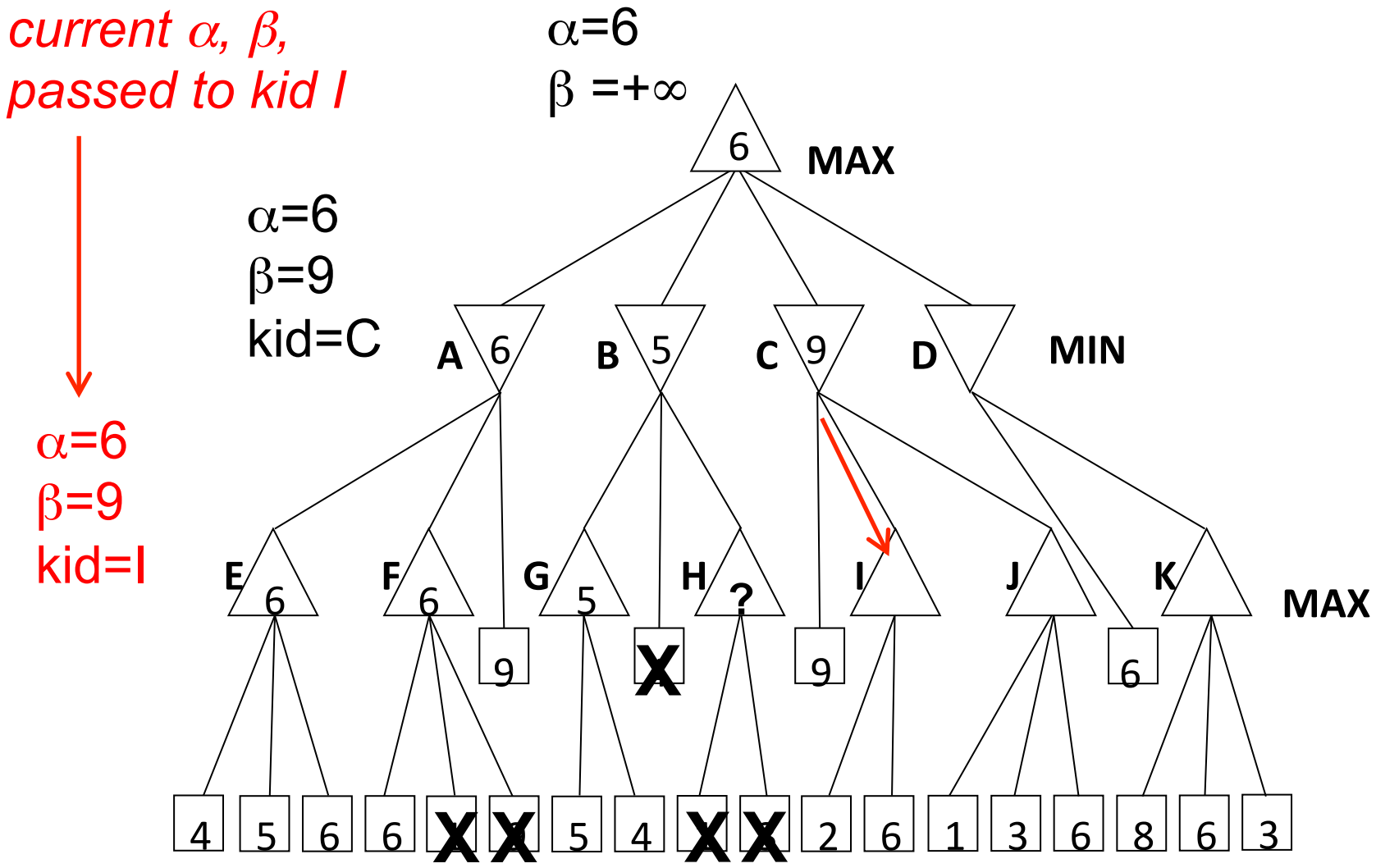
Long Detailed Alpha-Beta Example



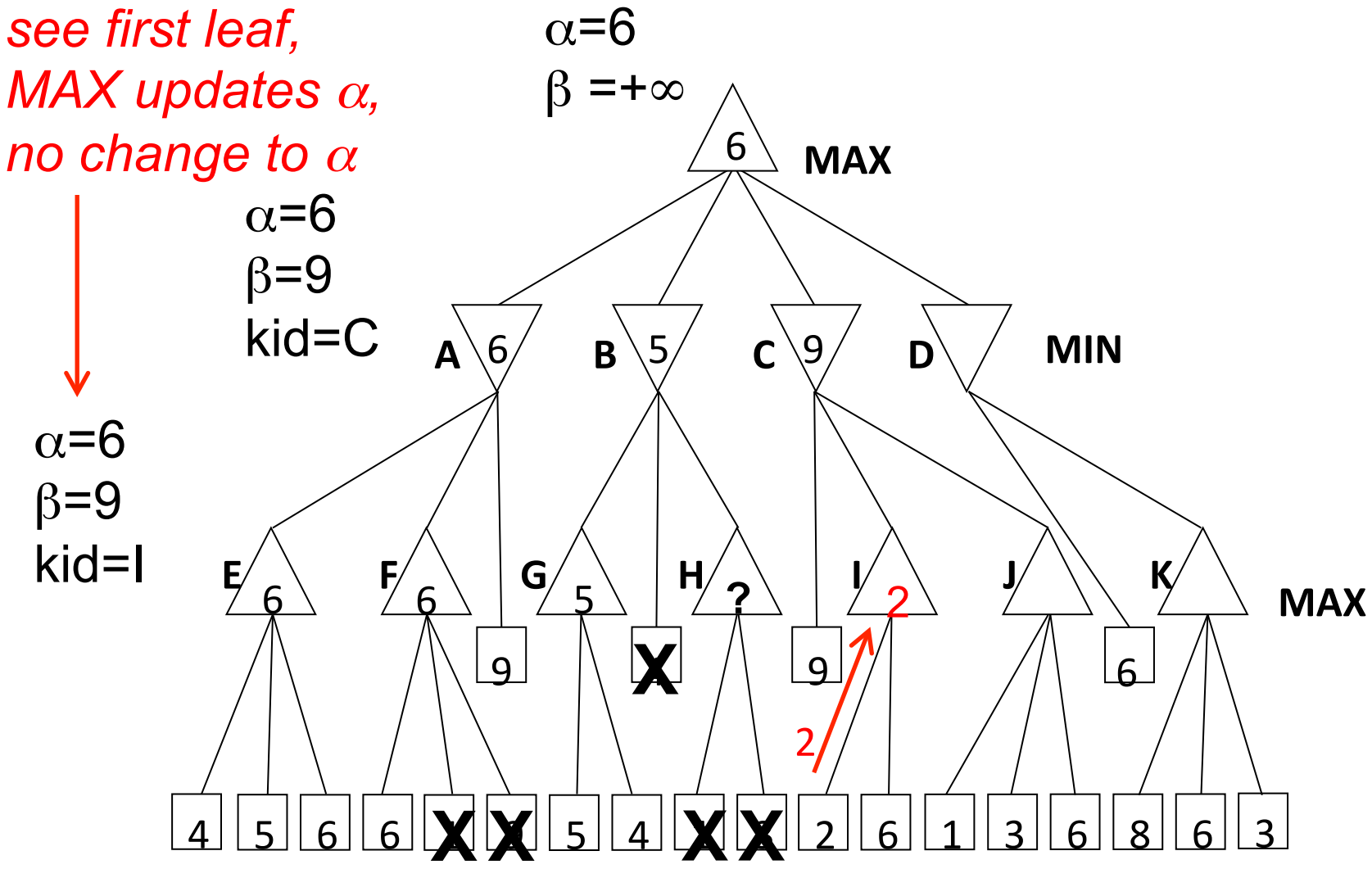
Long Detailed Alpha-Beta Example



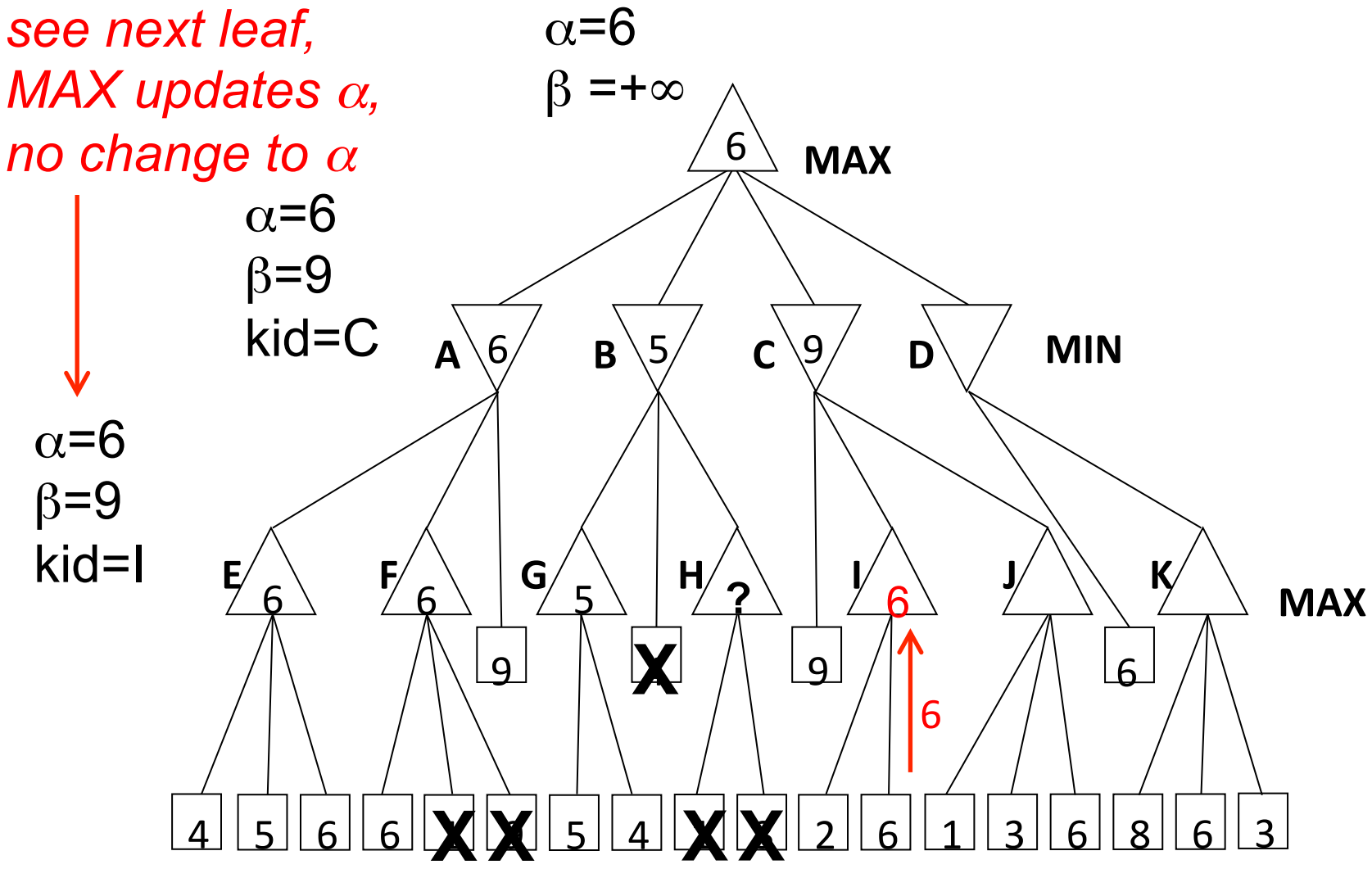
Long Detailed Alpha-Beta Example



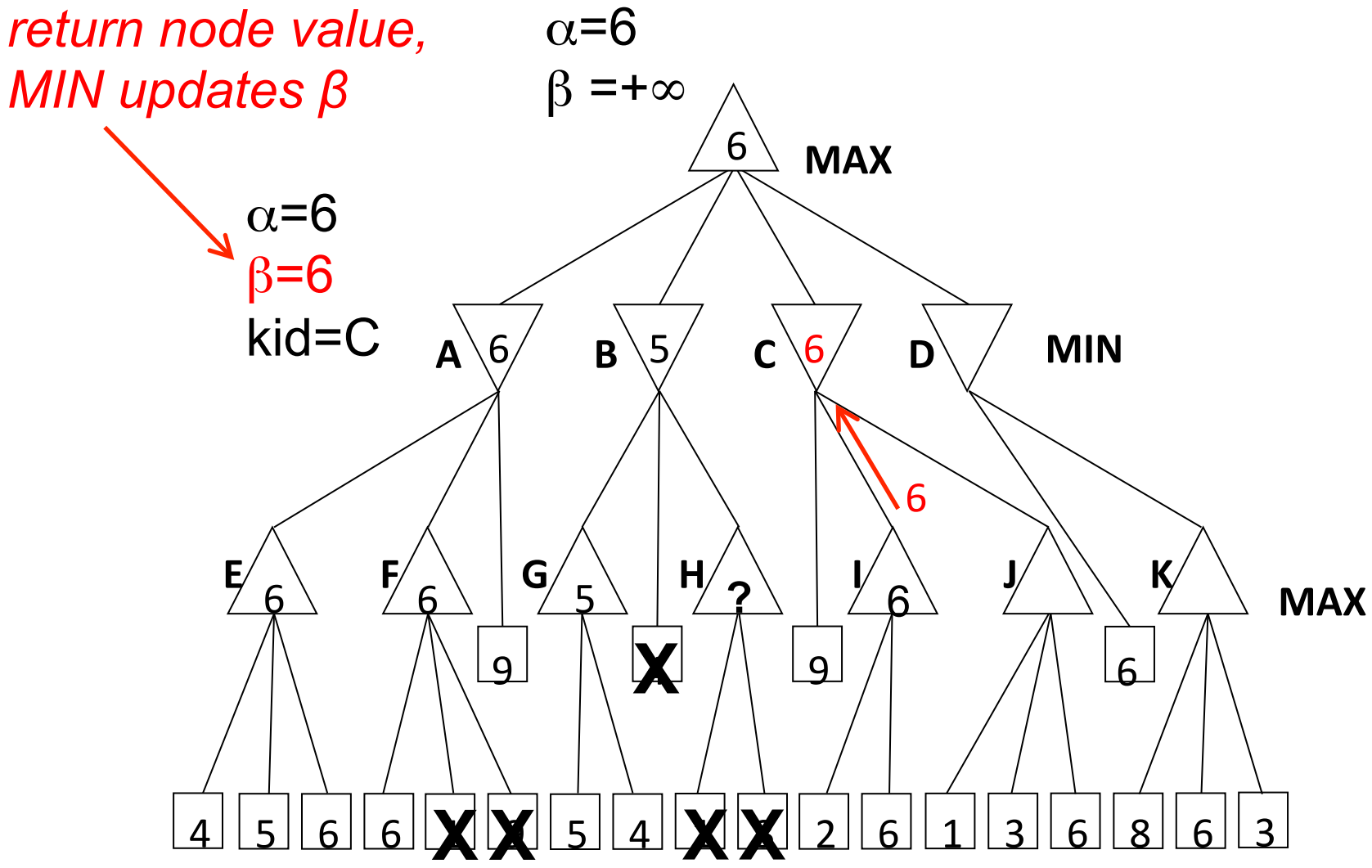
Long Detailed Alpha-Beta Example



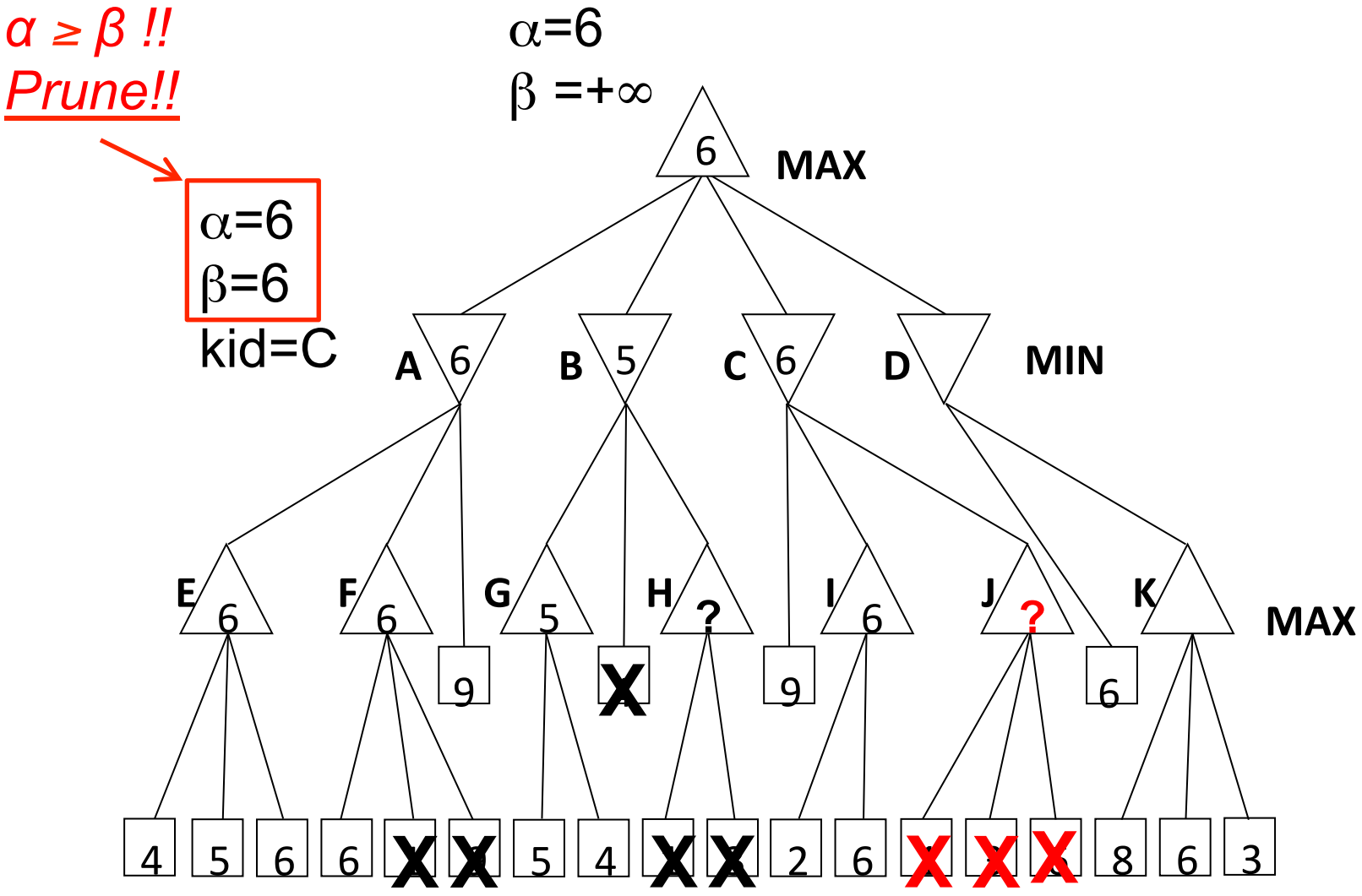
Long Detailed Alpha-Beta Example



Long Detailed Alpha-Beta Example

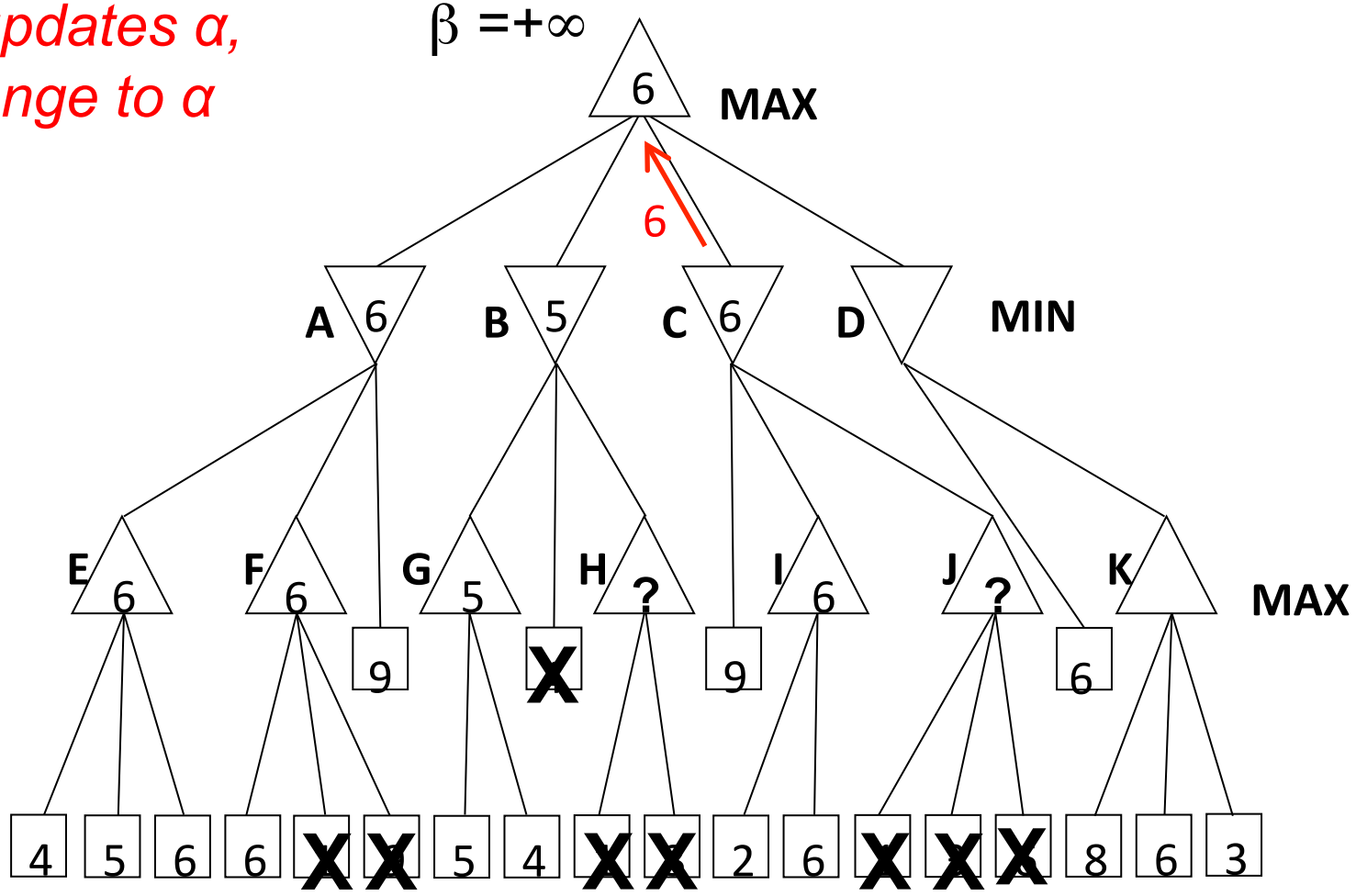


Long Detailed Alpha-Beta Example

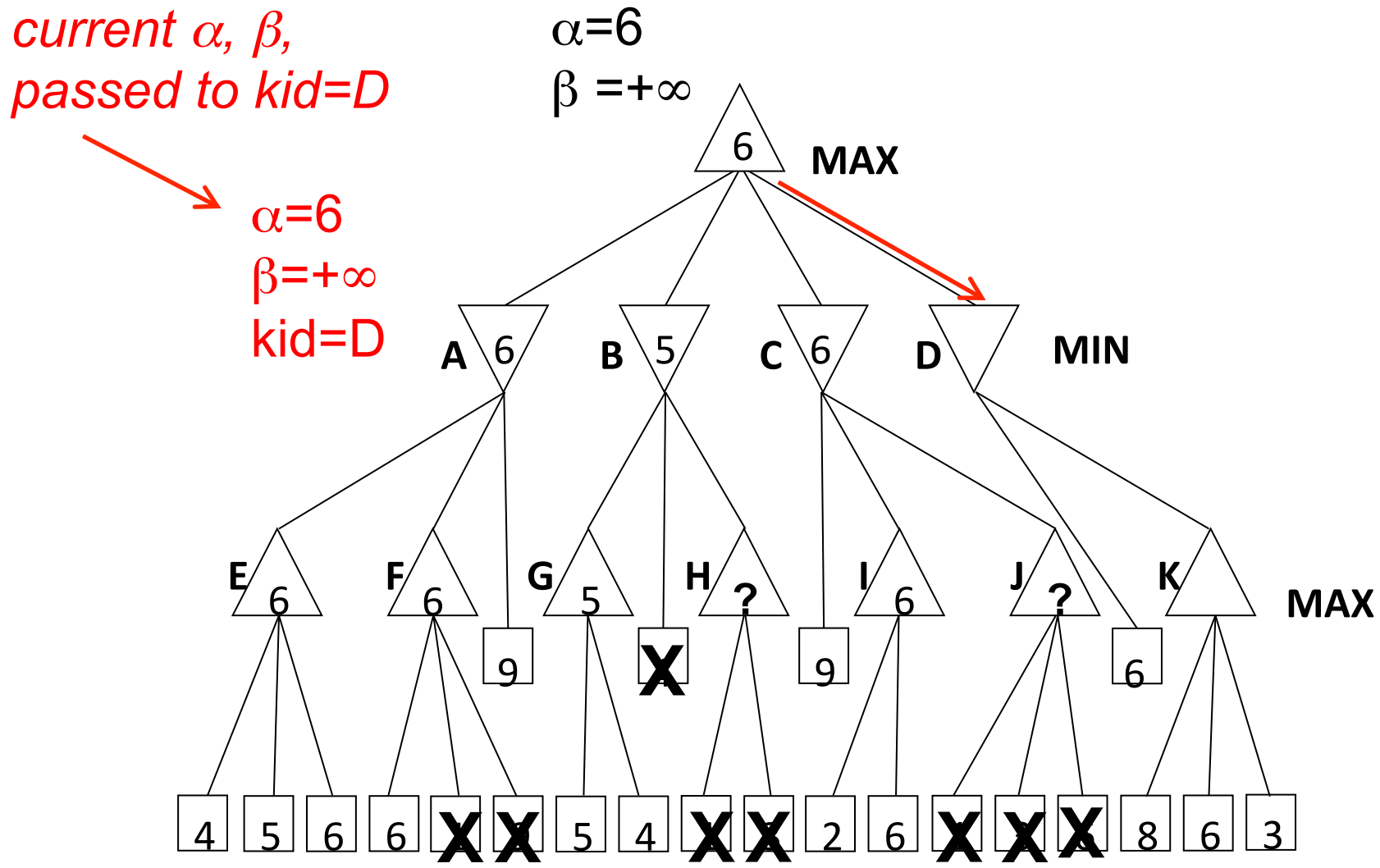


Long Detailed Alpha-Beta Example

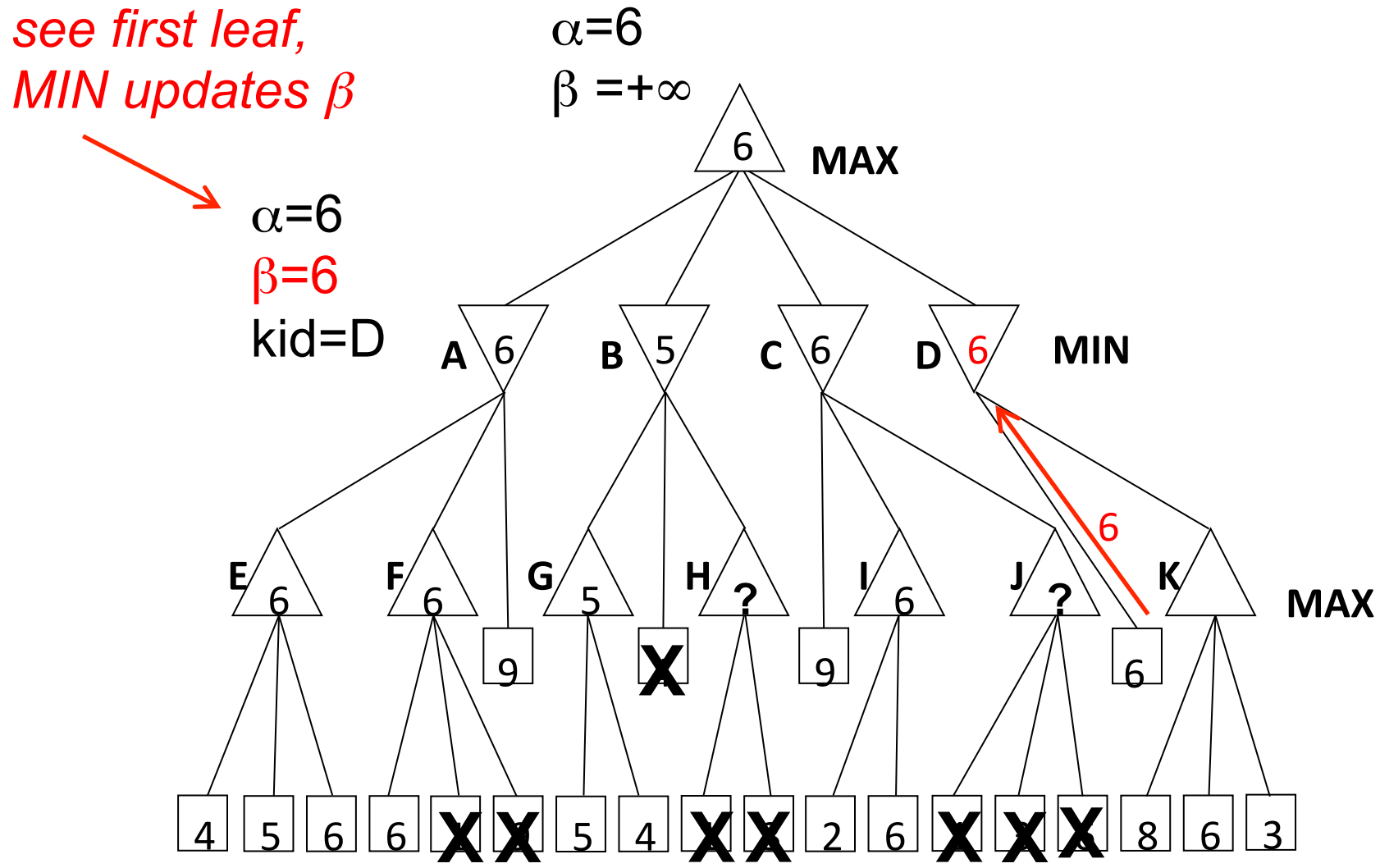
return node value, → α=6
MAX updates α, β =+∞
no change to α



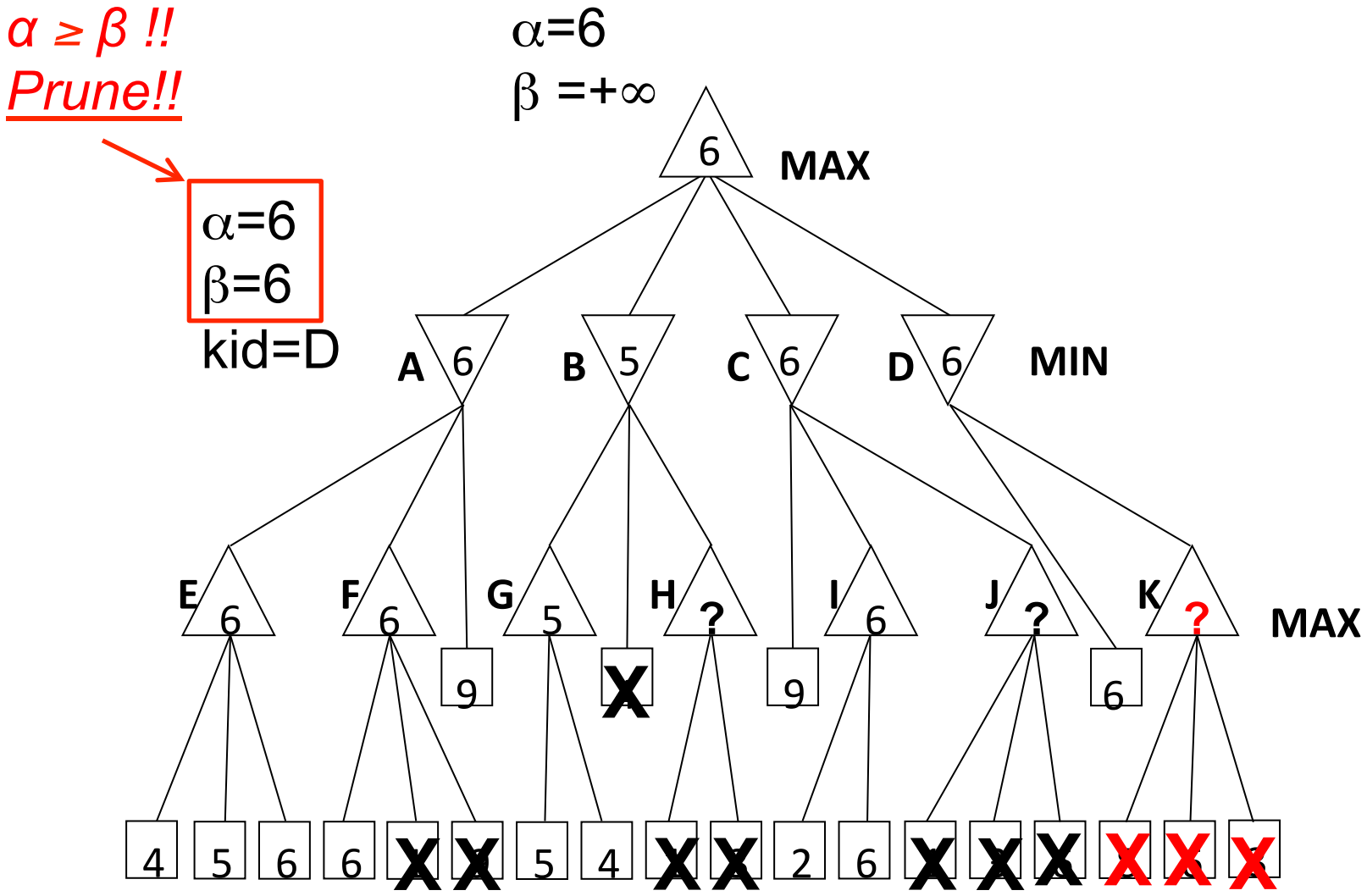
Long Detailed Alpha-Beta Example



Long Detailed Alpha-Beta Example

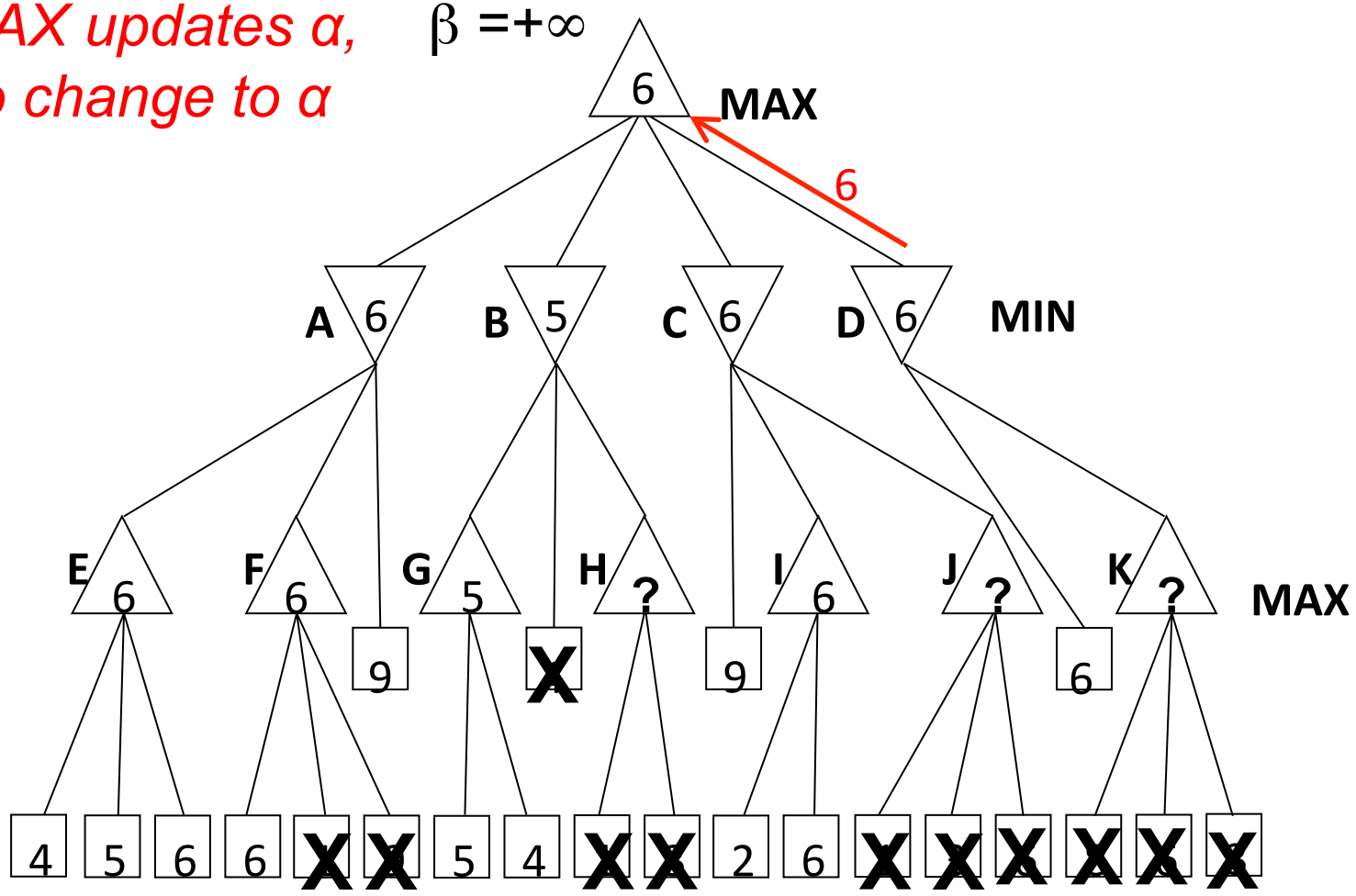


Long Detailed Alpha-Beta Example



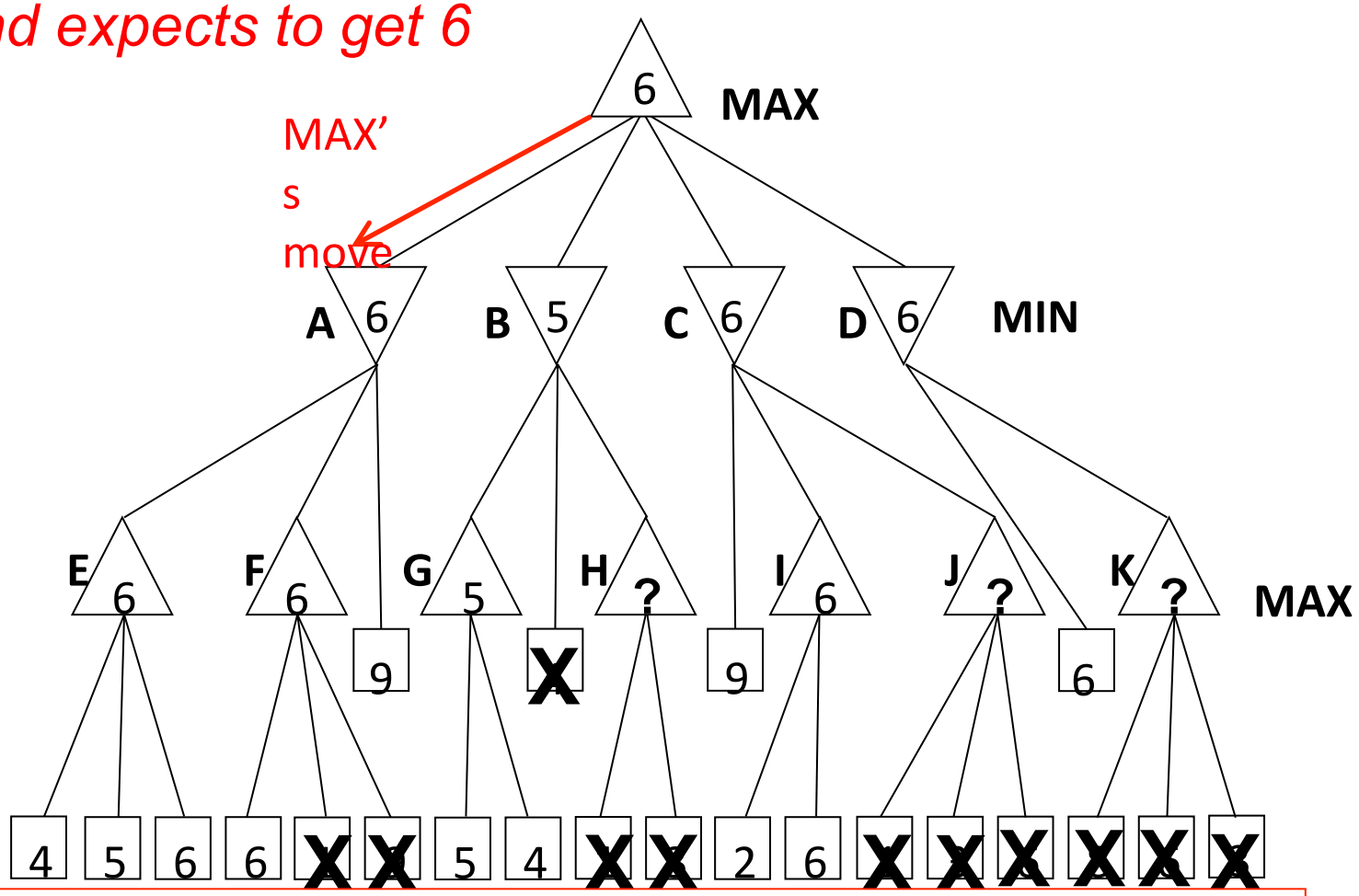
Long Detailed Alpha-Beta Example

return node value, α=6
MAX updates α, β =+∞
no change to α



Long Detailed Alpha-Beta Example

MAX moves to A,
and expects to get 6



Although we may have changed some internal branch node return values, the final root action and expected outcome are identical to if we had not done alpha-beta pruning. Internal values may change; root values do not.

You Will Be Expected to Know

- Basic definitions (section 6.1)
  - What is a CSP?
- Backtracking search for CSPs (6.3)
- Variable ordering or selection (6.3.1)
  - Minimum Remaining Values (MRV) heuristic
  - Degree Heuristic (DH) (to unassigned variables)
- Value ordering or selection (6.3.1)
  - Least constraining value (LCV) heuristic

Constraint Satisfaction Problems

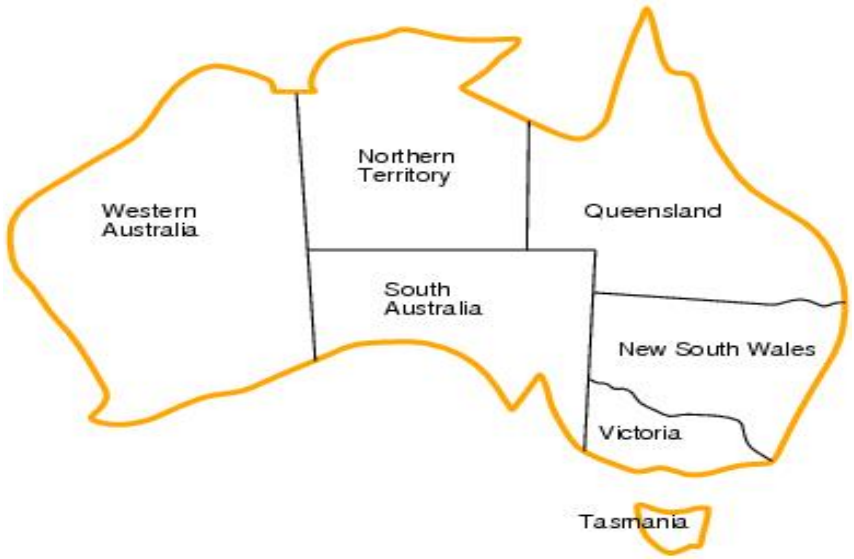
- What is a CSP?
  - Finite set of variables $X_1, X_2, \ldots, X_n$
  - Nonempty domain of possible values for each variable $D_1, D_2, \ldots, D_n$
  - Finite set of constraints $C_1, C_2, \ldots, C_m$
    - Each constraint C_i limits the values that variables can take,
    - e.g., $X_1 \neq X_2$
  - Each constraint C_i is a pair <scope, relation>
    - Scope = Tuple of variables that participate in the constraint.
    - Relation = List of allowed combinations of variable values.
      May be an explicit list of allowed combinations.
      May be an abstract relation allowing membership testing and listing.

- CSP benefits
  - Standard representation pattern
  - Generic goal and successor functions
  - Generic heuristics (no domain specific expertise).

CSPs --- What is a solution?

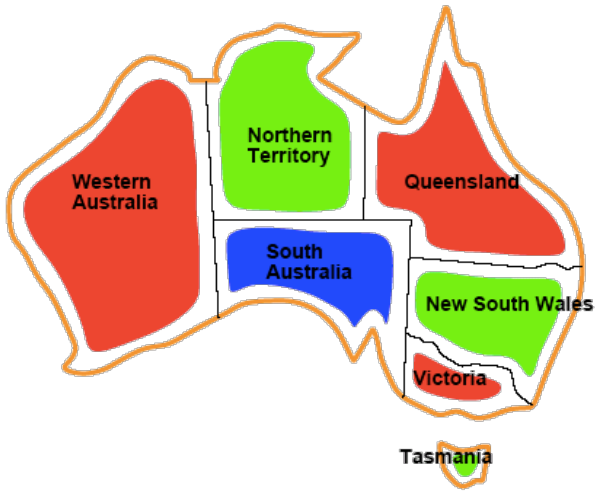
- A *state* is an *assignment* of values to some or all variables.
  - An assignment is *complete* when every variable has an assigned value.
  - An assignment is *partial* when one or more variables have no assigned value.
- ***Consistent assignment:***
  - An assignment that does not violate the constraints.
- A ***solution*** to a CSP is a complete and consistent assignment.
  - All variables are assigned, and none of the assignments violate the constraints.
- CSPs may require a solution that maximizes an *objective function*.
  - For simple linear cases , an optimal solution can be obtained by Linear Programming.
- Examples of Applications:
  - Scheduling the time of observations on the Hubble Space Telescope
  - Airline schedules
  - Cryptography
  - Computer vision, image interpretation

CSP 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*

CSP example: Map coloring solution

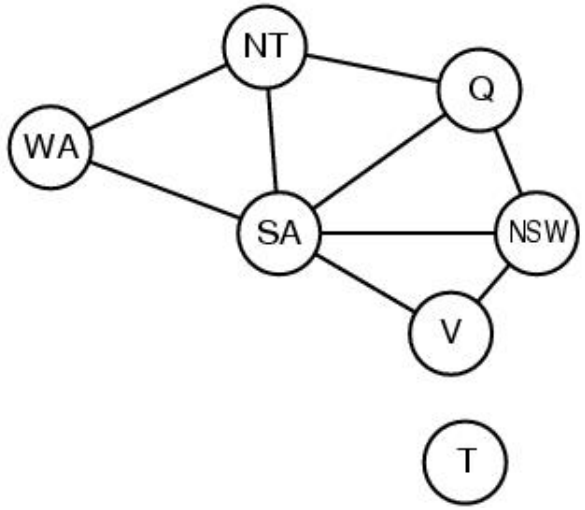


- A solution is:
  - A complete and consistent assignment.
  - All variables assigned, all constraints satisfied.

- E.g., *{WA=red, NT=green, Q=red, NSW=green, V=red, SA=blue, T=green}*

Constraint graphs

- Constraint graph:
  - nodes are variables
  - arcs are binary constraints



- Graph can be used to simplify search
  e.g. Tasmania is an independent subproblem

Backtracking search

- Similar to Depth-first search
  - At each level, picks a single variable to explore
  - Iterates over the domain values of that variable
- Generates kids one at a time, one per value
- Backtracks when a variable has no legal values left
- Uninformed algorithm
  - No good general performance

Backtracking search (Figure 6.5)

```
function BACKTRACKING-SEARCH(csp) return a solution or failure
    return RECURSIVE-BACKTRACKING({} , csp)

function RECURSIVE-BACKTRACKING(assignment, csp) return a solution or failure
    if assignment is complete then return assignment
    var ← SELECT-UNASSIGNED-VARIABLE(VARIABLES[csp],assignment,csp)
    for each value in ORDER-DOMAIN-VALUES(var, assignment, csp) do
        if value is consistent with assignment according to CONSTRAINTS[csp] then
            add {var=value} to assignment
            result ← RECURSIVE-BACTRACKING(assignment, csp)
            if result ≠ failure then return result
            remove {var=value} from assignment
    return failure
```

Backtracking search (Figure 6.5)

function BACKTRACKING-SEARCH(*csp*) **return** a solution or failure
    return RECURSIVE-BACKTRACKING(*{}* , *csp*)

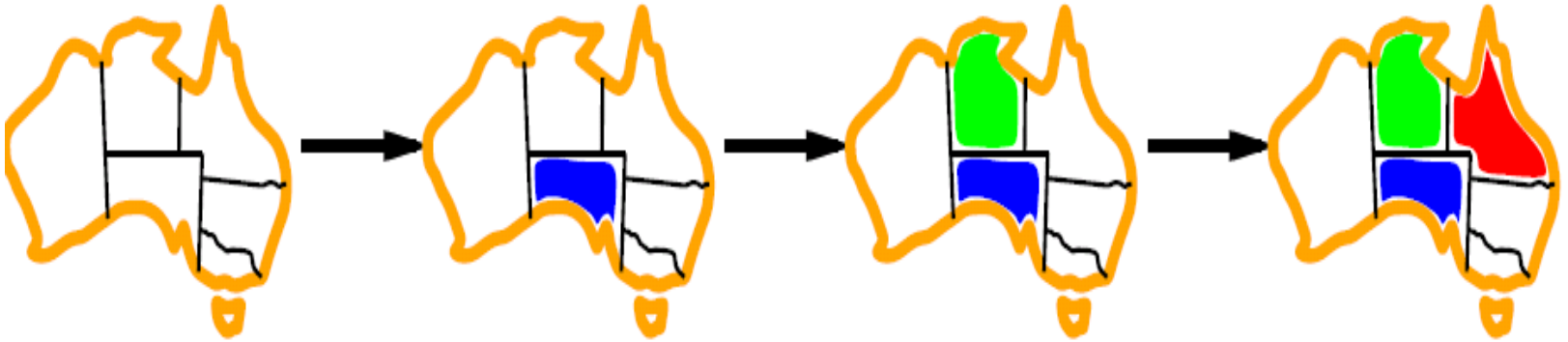
function RECURSIVE-BACKTRACKING(*assignment, csp*) **return** a solution or failure
    if *assignment* is complete **then return** *assignment*
    var ← SELECT-UNASSIGNED-VARIABLE(VARIABLES[*csp*],*assignment*,*csp*)
    for each *value* **in** ORDER-DOMAIN-VALUES(*var, assignment, csp*) **do**
        if *value* is consistent with *assignment* according to CONSTRAINTS[*csp*] **then**
            add *{var=value}* to assignment
            result ← RRECURSIVE-BACTRACKING(*assignment, csp*)
            if *result* ≠ *failure* **then return** *result*
            remove *{var=value}* from *assignment*
    return *failure*

Minimum remaining values (MRV)

var ← SELECT-UNASSIGNED-VARIABLE(VARIABLES[*csp*],*assignment*,*csp*)

- A.k.a. most constrained variable heuristic
- *Heuristic Rule*: choose variable with the fewest legal moves
  - e.g., will immediately detect failure if X has no legal values

Degree heuristic for the initial variable



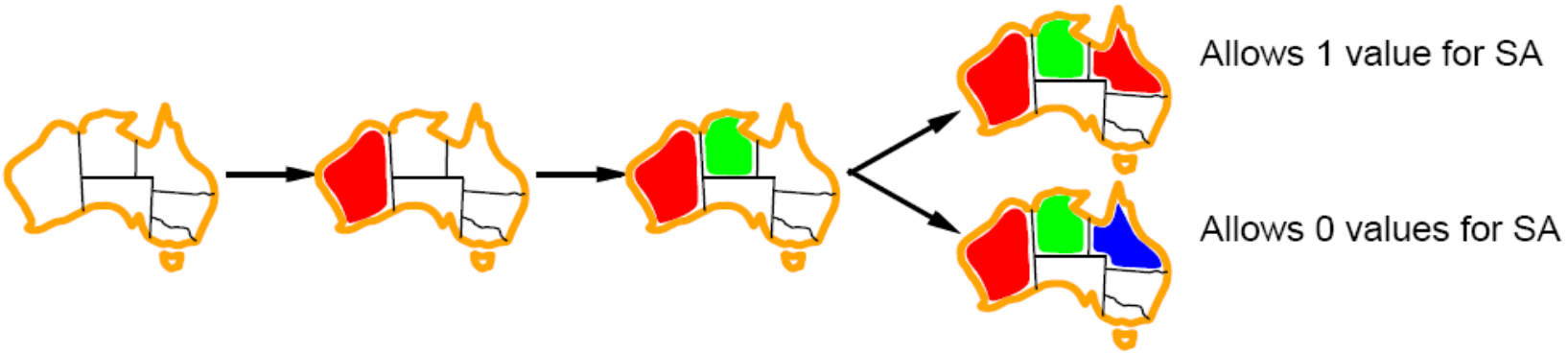
- *Heuristic Rule*: select variable that is involved in the largest number of constraints on other unassigned variables.
- Degree heuristic can be useful as a tie breaker.
- *In what order should a variable's values be tried?*

Backtracking search (Figure 6.5)

function BACKTRACKING-SEARCH(*csp*) **return** a solution or failure
    return RECURSIVE-BACKTRACKING(*{}* , *csp*)

function RECURSIVE-BACKTRACKING(*assignment, csp*) **return** a solution or failure
    if *assignment* is complete **then return** *assignment*
    var ← SELECT-UNASSIGNED-VARIABLE(VARIABLES[*csp*],*assignment*,*csp*)
    for each *value* **in** ORDER-DOMAIN-VALUES(*var, assignment, csp*) **do**
        if *value* is consistent with *assignment* according to CONSTRAINTS[*csp*] **then**
            add *{var=value}* to assignment
            result ← RRECURSIVE-BACTRACKING(*assignment, csp*)
            if *result* ≠ *failure* **then return** *result*
            remove *{var=value}* from *assignment*
    return *failure*

Least constraining value for value-ordering



- Least constraining value heuristic
- Heuristic Rule: given a variable choose the least constraining value
  - leaves the maximum flexibility for subsequent variable assignments

Minimum remaining values (MRV) vs. Least constraining value (LCV)

- Why do we want the MRV (minimum values, most constraining) for variable selection --- but the LCV (maximum values, least constraining) for value selection?

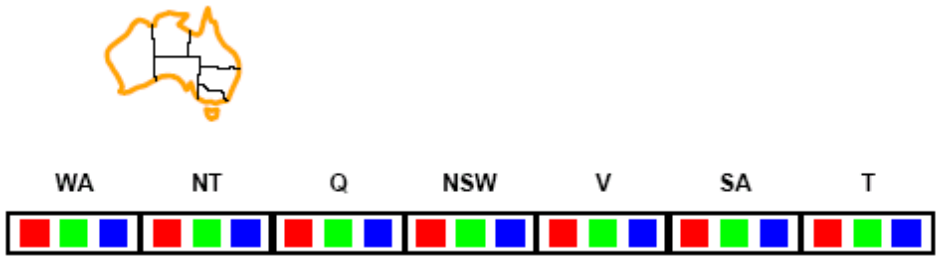
- Isn't there a contradiction here?

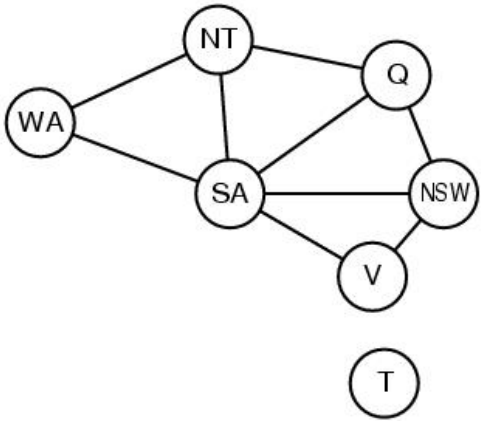
- MRV for variable selection to reduces the branching factor.
  - Smaller branching factors lead to faster search.
  - Hopefully, when we get to variables with currently many values, constraint propagation (next lecture) will have removed some of their values and they'll have small branching factors by then too.
- LCV for value selection increases the chance of early success.
  - If we are going to fail at this node, then we have to examine every value anyway, and their order makes no difference at all.
  - If we are going to succeed, then the earlier we succeed the sooner we can stop searching, so we want to succeed early.
  - LCV rules out the fewest possible solutions below this node, so we have the most chances for early success.

You Will Be Expected to Know

- Node consistency, arc consistency, path consistency, K-consistency (6.2)
- Forward checking (6.3.2)
- Local search for CSPs
  - Min-Conflict Heuristic (6.4)
- ~~The structure of problems (6.5)~~

Forward checking

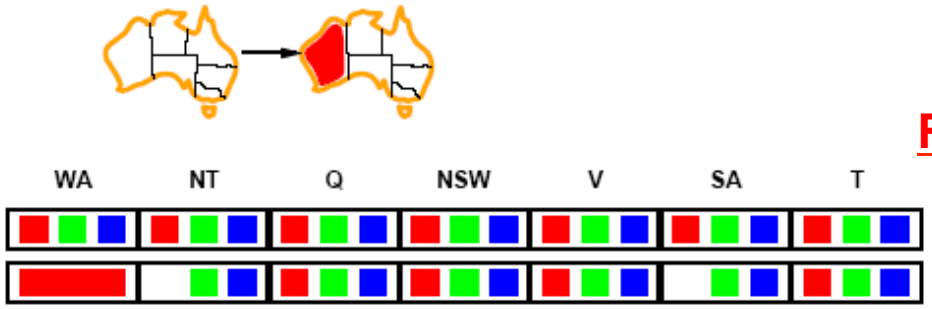


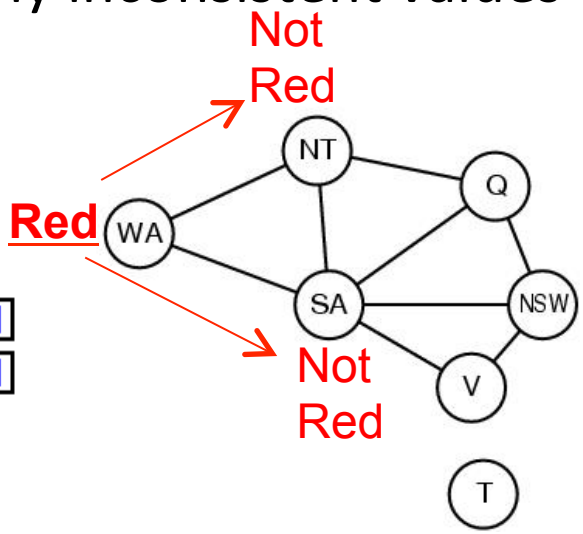


- Can we detect inevitable failure early?
  - *And avoid it later?*
- *Forward checking idea:* keep track of remaining legal values for unassigned variables.
- When a variable is assigned a value, update all neighbors in the constraint graph.
- **Forward checking stops after one step and does not go beyond immediate neighbors.**
- Terminate search when any variable has no legal values.

Forward checking

Check only neighbors; delete any inconsistent values



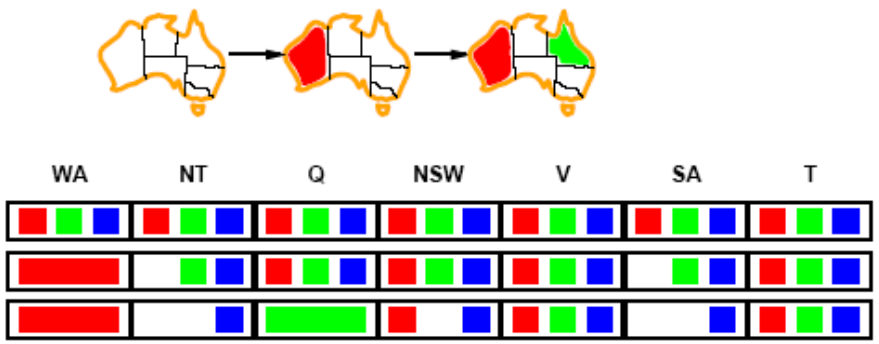


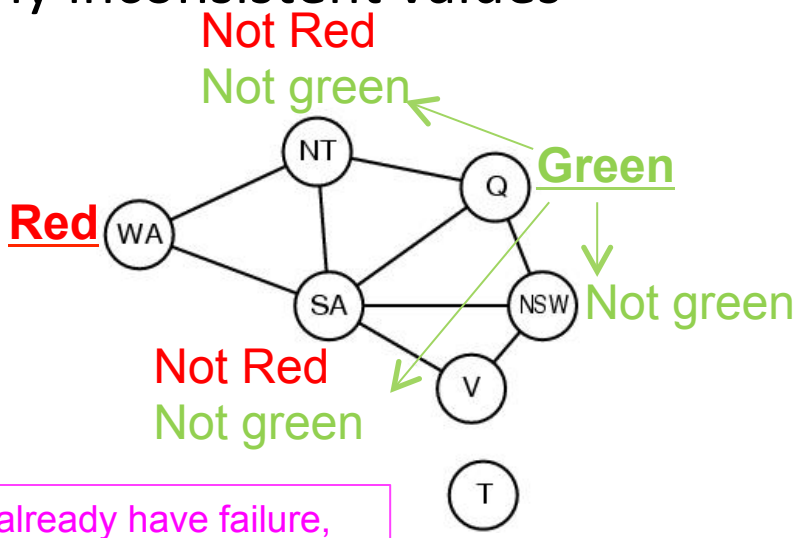
- Assign *{WA=red}*

- Effects on other variables connected by constraints to WA
  - *NT can no longer be red*
  - *SA can no longer be red*

Forward checking

Check only neighbors; delete any inconsistent values





- Assign *{Q=green}*

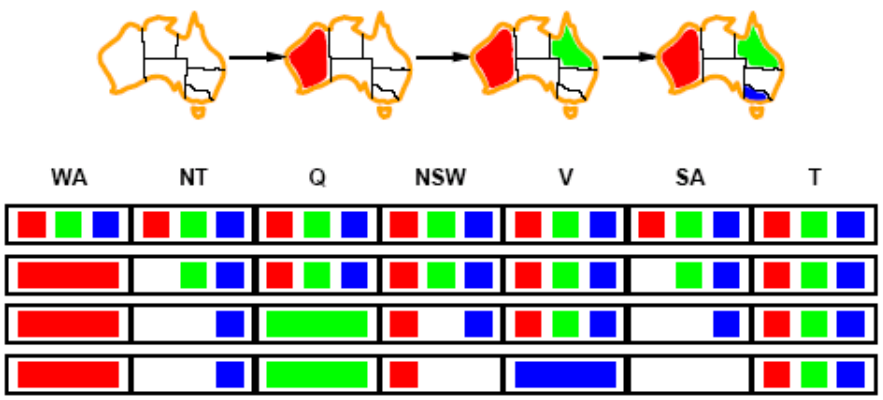
We already have failure, but Forward Checking is too simple to detect it now.

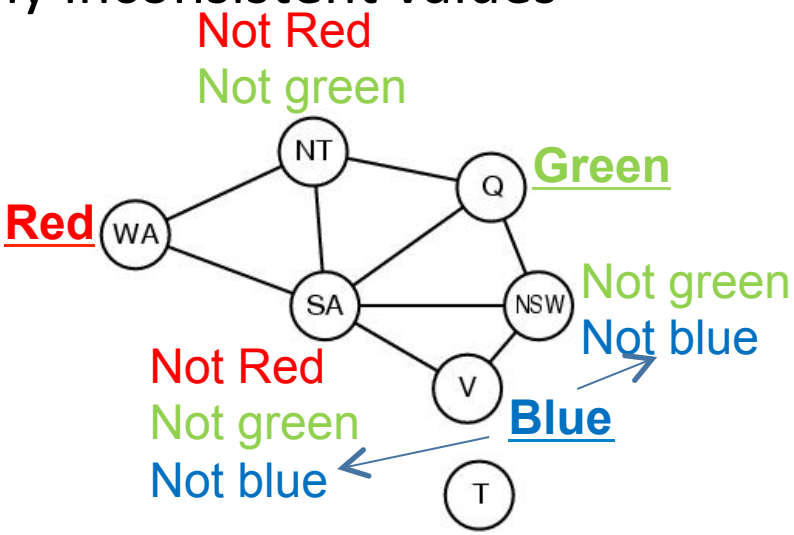
- Effects on other variables connected by constraints with WA
  - *NT can no longer be green*
  - *NSW can no longer be green*
  - *SA can no longer be green*

- *MRV heuristic* would automatically select NT or SA next

Forward checking

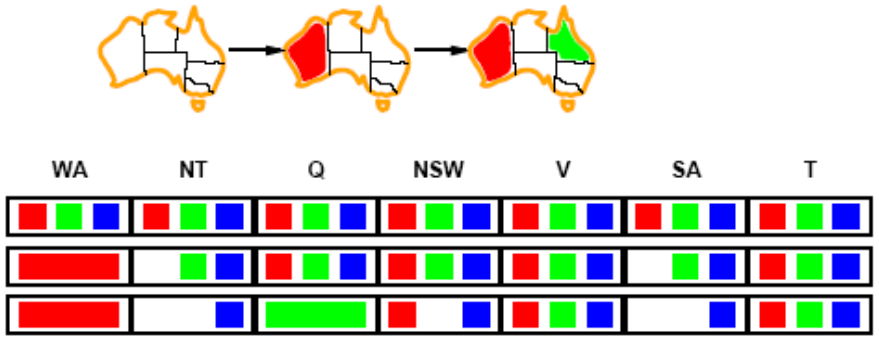
Check only neighbors; delete any inconsistent values

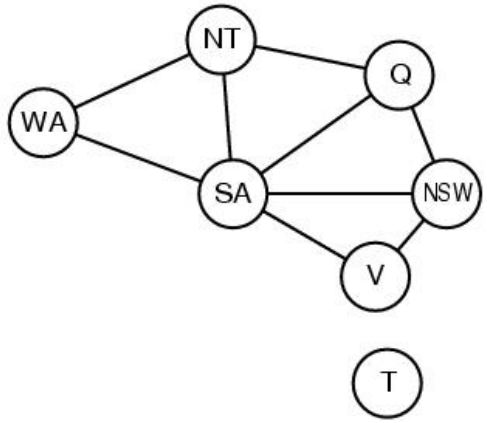




- If *V* is assigned *blue*
- Effects on other variables connected by constraints with WA
  - *NSW can no longer be blue*
  - *SA is empty*
- FC has detected that partial assignment is *inconsistent* with the constraints and backtracking can occur.

Constraint propagation





- **Solving CSPs with combination of heuristics plus forward checking is more efficient than either approach alone.**

- **Forward Checking does not detect all failures when they become obvious.**
  - E.g., NT and SA cannot both be blue in the example above, so failure.

- **Higher-order Constraint Propagation can detect early failure.**
  - However, to do so takes more computing time --- is it worth the extra effort??

Arc consistency algorithm (AC-3)

function AC-3(*csp*) **returns** false if inconsistency found, else true, may reduce *csp* 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*

/* *initial queue must contain both* (X_i, X_j) *and* (X_j, X_i) */

while queue is not empty **do**

$(X_i, X_j) \leftarrow$ REMOVE-FIRST(*queue*)

if REMOVE-INCONSISTENT-VALUES(X_i, X_j) **then**

if size of $D_i = 0$ **then return** false

for each X_k **in** NEIGHBORS[X_i] − {X_j} **do**

add (X_k, X_i) to queue if not already there

return true

function REMOVE-INCONSISTENT-VALUES(X_i, X_j) **returns** *true* iff we delete a value from the domain of X_i

removed ⇐ *false*

for each *x* **in** DOMAIN[X_i] **do**

if no value *y* in DOMAIN[X_j] allows (*x,y*) to satisfy the constraints between X_i and X_j

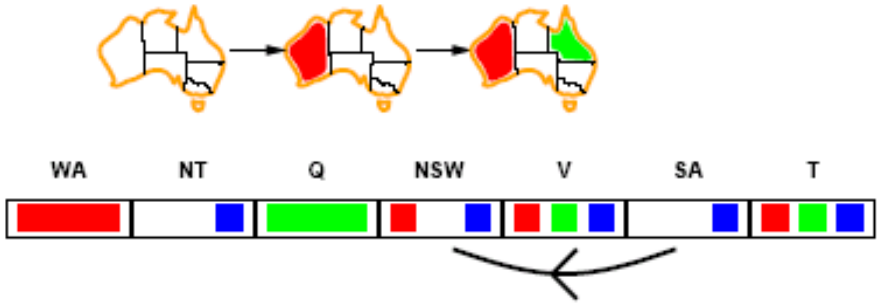
then delete *x* from DOMAIN[X_i]; *removed* ⇐ *true*

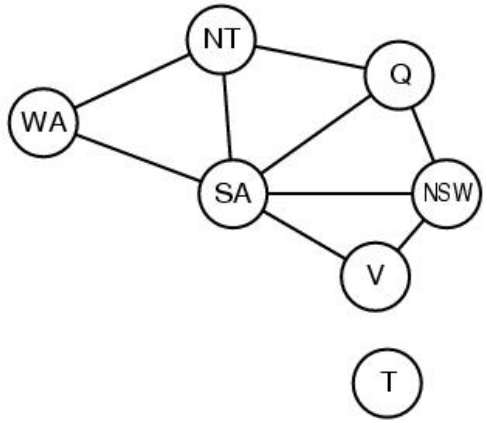
return *removed*

(from Mackworth, 1977)

Arc consistency (AC-3)

Like Forward Checking, but exhaustive until quiescence





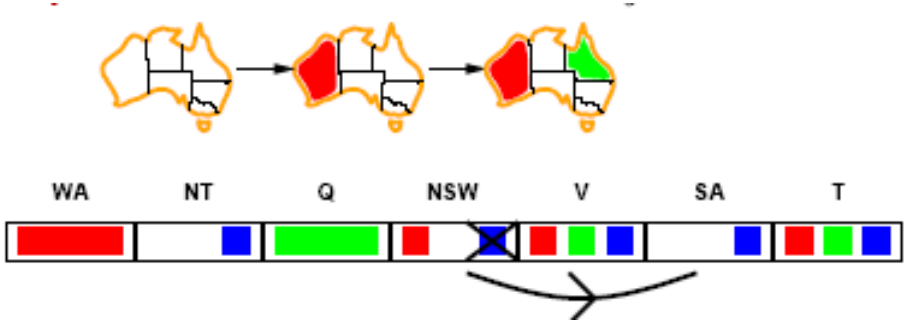
- *An Arc X → Y* is consistent if
  for *every* value *x* of *X* there is some value *y* of *Y* consistent with *x*
  (note that this is a directed property)

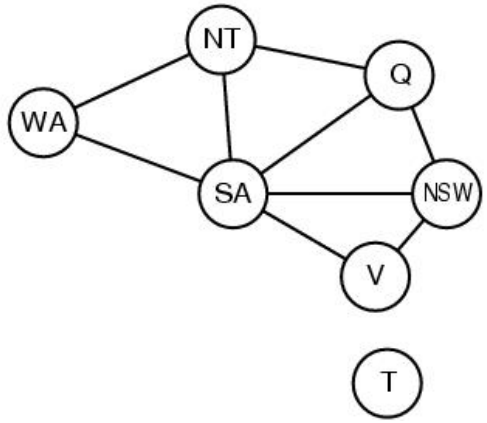
- Consider state of search after WA and Q are assigned & FC is done:

  SA → NSW is consistent if
  SA=blue and *NSW=red*

Arc consistency (AC-3)

Like Forward Checking, but exhaustive until quiescence





- *X* → *Y* is consistent if

  for *every* value *x* of *X* there is some value *y* of *Y* consistent with *x*

- *NSW* → *SA* is consistent if

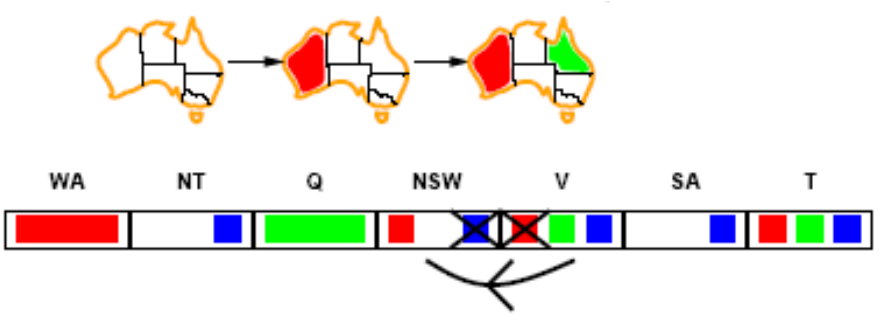
  NSW=red and *SA=blue*

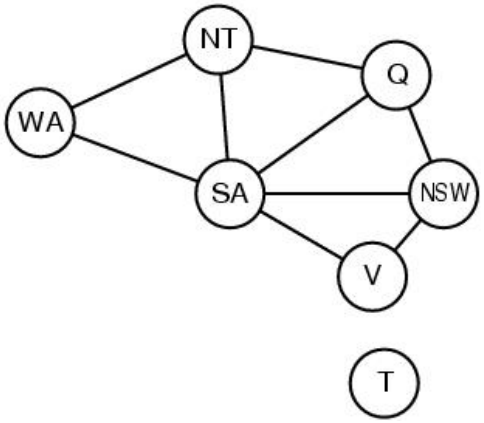
  NSW=blue and SA=???

- **NSW=blue can be pruned:** No current domain value of *SA* is consistent

Arc consistency (AC-3)

Like Forward Checking, but exhaustive until quiescence





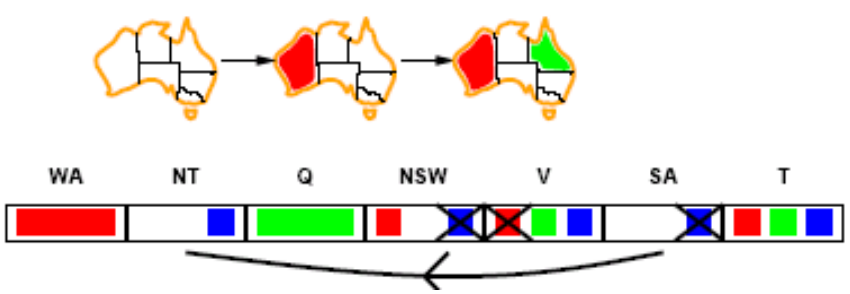
- Enforce arc-consistency:

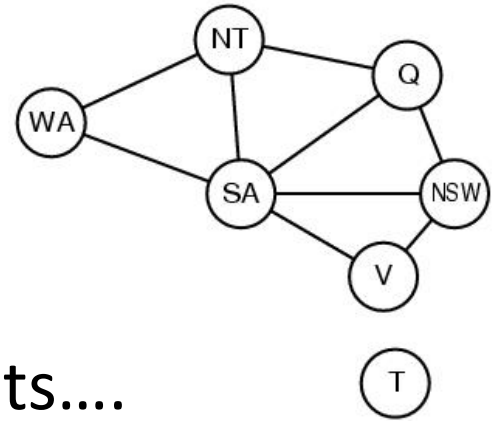
  Arc can be made consistent by removing *blue* from *NSW*

- Continue to propagate constraints….
  - Check *V* → *NSW*
  - Not consistent for V = red
  - Remove red from *V*

Arc consistency (AC-3)

Like Forward Checking, but exhaustive until quiescence





Continue to propagate constraints….

- *SA* → *NT* is not consistent
  - **and cannot be made consistent (Failure!)**
- Arc consistency detects failure earlier than FC
  - Requires more computation: Is it worth the effort??

Arc consistency checking

- Can be run as a preprocessor, or after each assignment
  - As preprocessor before search: Removes obvious inconsistencies
  - After each assignment: Reduces search cost but increases step cost

- **AC must be run repeatedly until no inconsistency remains**
  - **Like Forward Checking, but exhaustive until quiescence**

- Trade-off
  - Requires overhead to do; but usually better than direct search
  - In effect, it can successfully eliminate large (and inconsistent) parts of the state space more effectively than can direct search alone

- Need a systematic method for arc-checking
  - If *X* loses a value, neighbors of *X* need to be rechecked:

    I.e., incoming arcs can become inconsistent again (outgoing arcs will stay consistent).

Local search for CSPs

- Use complete-state representation
  - Initial state = all variables assigned values
  - Successor states = change 1 (or more) values

- For CSPs
  - allow states with unsatisfied constraints (unlike backtracking)
  - operators **reassign** variable values
  - hill-climbing with n-queens is an example

- Variable selection: randomly select any conflicted variable

- Value selection: *min-conflicts heuristic*
  - Select new value that results in a minimum number of conflicts with the other variables

Local search for CSP

function MIN-CONFLICTS(*csp, max_steps*) **return** solution or failure

inputs: *csp*, a constraint satisfaction problem

max_steps, the number of steps allowed before giving up

current ← an initial complete assignment for *csp*

for *i* = 1 to *max_steps* **do**

if *current* is a solution for *csp* then return *current*

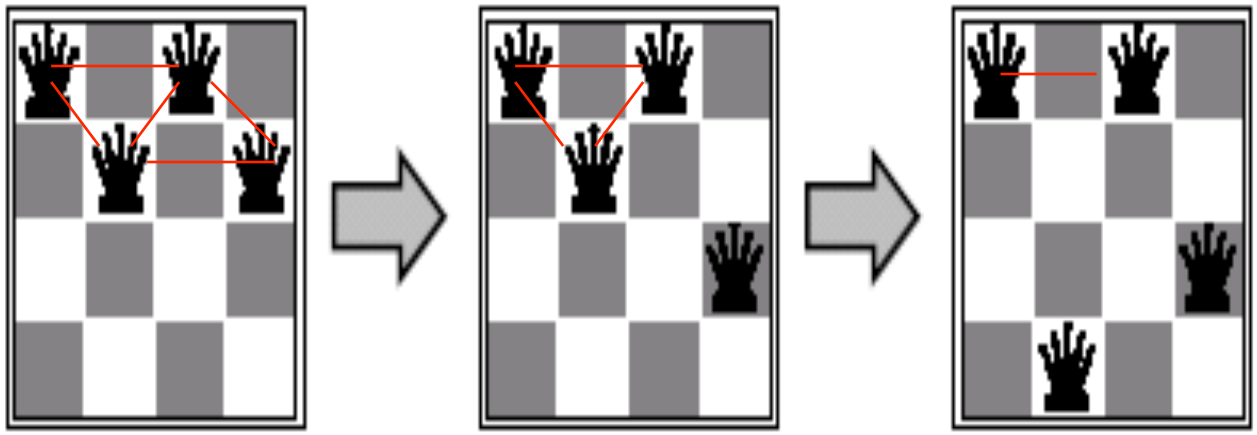
var ← a randomly chosen, conflicted variable from VARIABLES[*csp*]

value ← the value *v* for *var* that minimizes CONFLICTS(*var,v,current,csp*)

set *var = value* in *current*

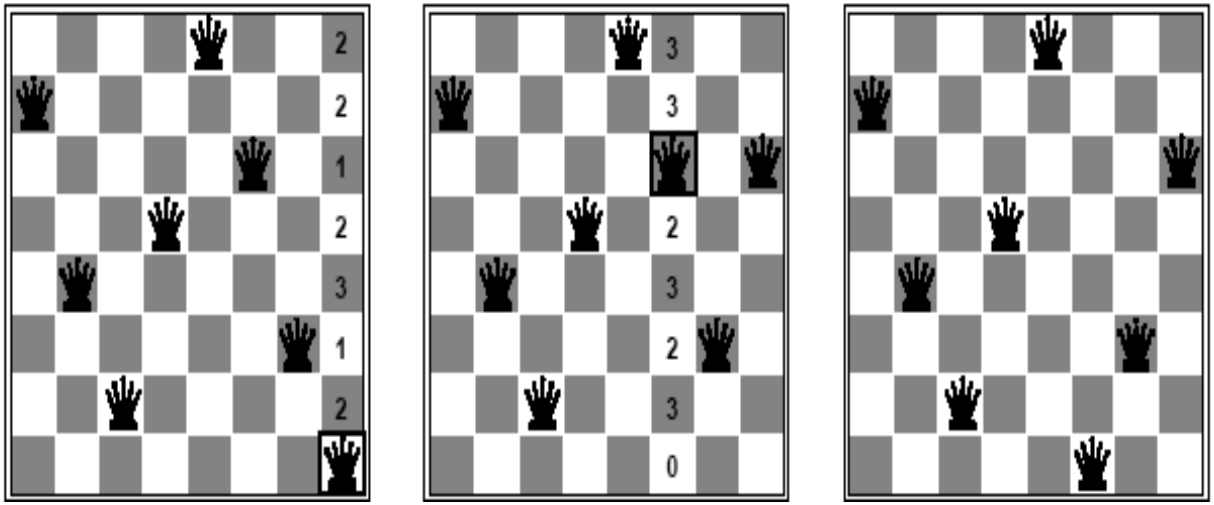
return *failure*

Min-conflicts example 1



Use of min-conflicts heuristic in hill-climbing.

Min-conflicts example 2



- A two-step solution for an 8-queens problem using min-conflicts heuristic
- At each stage a queen is chosen for reassignment in its column
- The algorithm moves the queen to the min-conflict square breaking ties randomly.

Advantages of local search

- Local search can be particularly useful in an online setting
  - Airline schedule example
    - E.g., mechanical problems require than 1 plane is taken out of service
    - Can locally search for another "close" solution in state-space
    - Much better (and faster) in practice than finding an entirely new schedule

- The runtime of min-conflicts is roughly independent of problem size.
  - Can solve the millions-queen problem in roughly 50 steps.

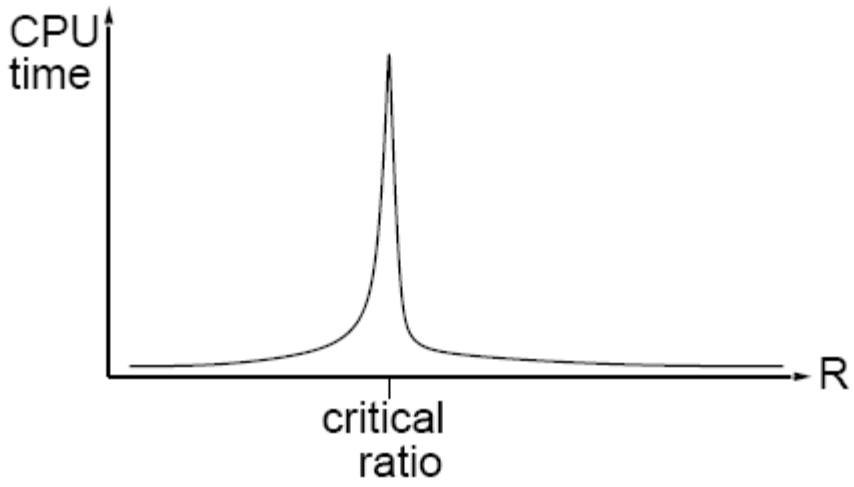
  - Why?
    - n-queens is easy for local search because of the relatively high density of solutions in state-space

Performance of min-conflicts

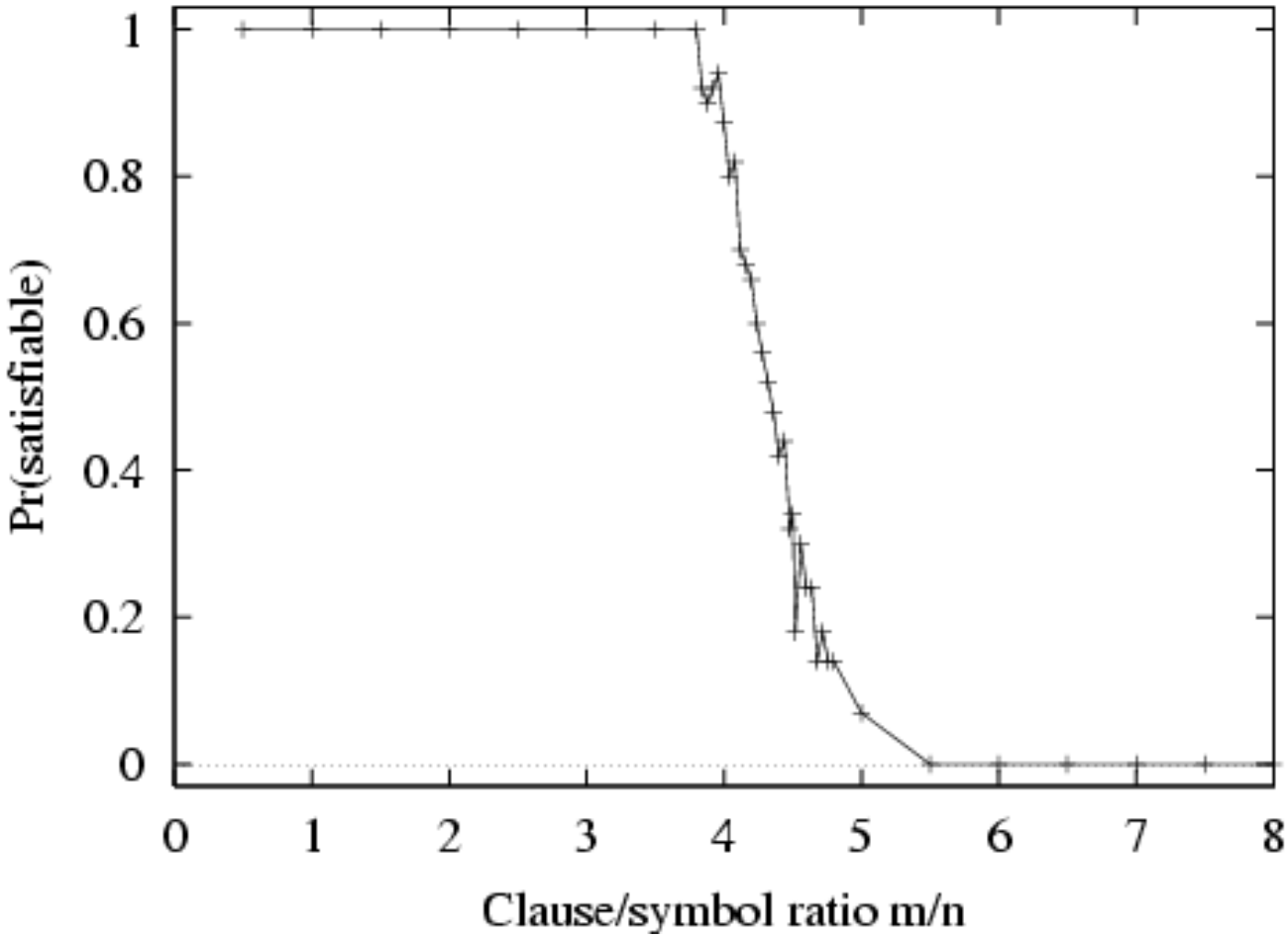
Given random initial state, can solve n-queens in almost constant time for arbitrary n with high probability (e.g., $n = 10{,}000{,}000$)

The same appears to be true for any randomly-generated CSP
except in a narrow range of the ratio

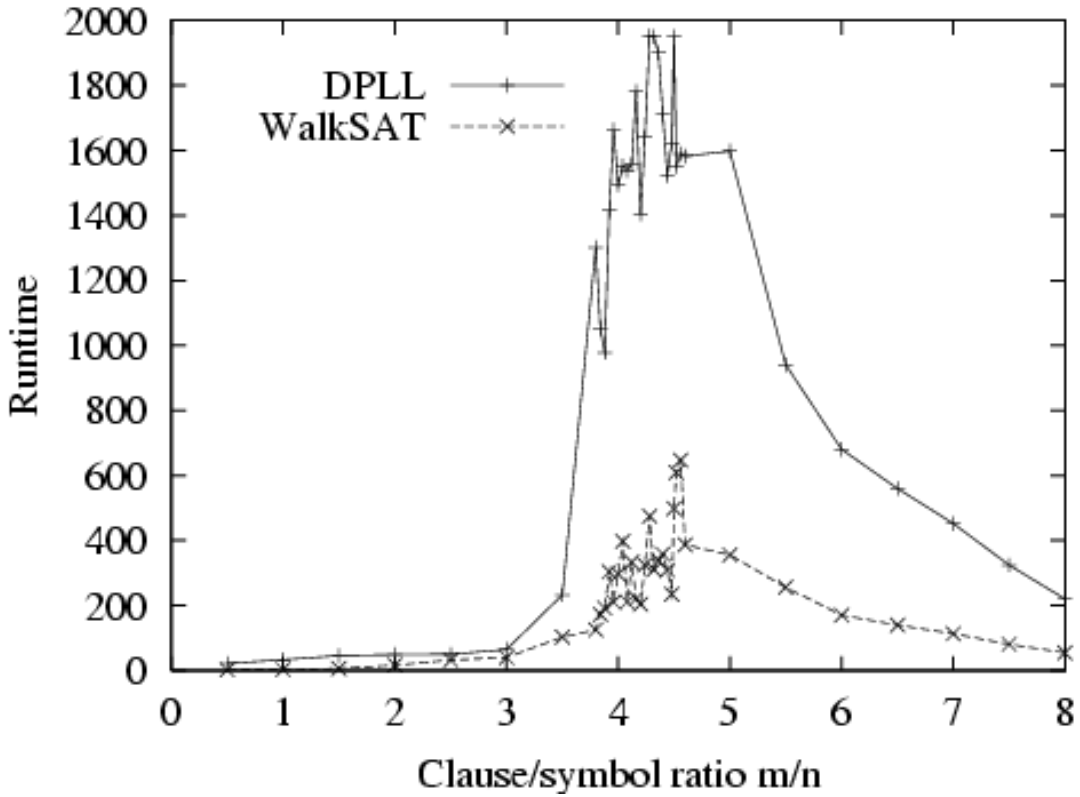
$$R = \frac{\text{number of constraints}}{\text{number of variables}}$$



Hard satisfiability problems



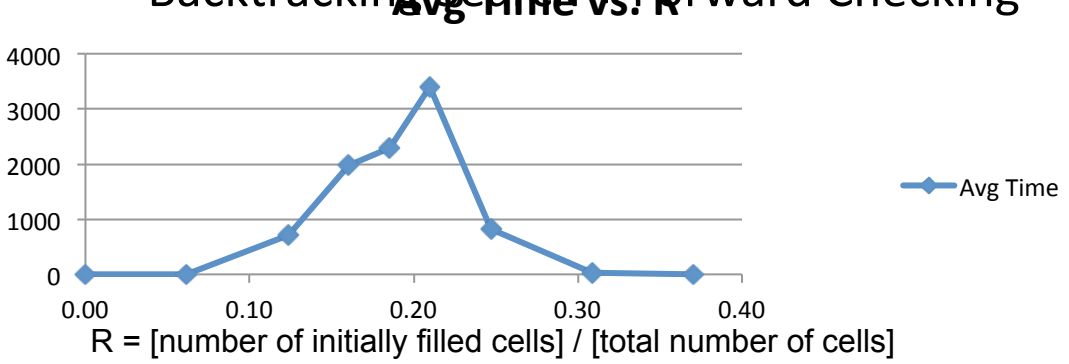
Hard satisfiability problems

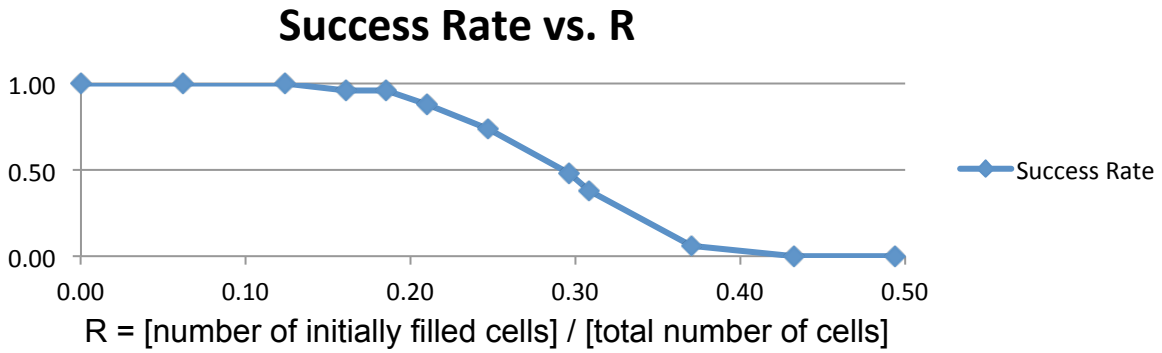


- Median runtime for 100 satisfiable random 3-CNF sentences, $n = 50$

Sudoku

Backtracking Search + Forward Checking





- R = [number of initially filled cells] / [total number of cells]
- Success Rate = P(random puzzle is solvable)
- [total number of cells] = 9x9 = 81
- [number of initially filled cells] = variable