Artificial Intelligence (IT3160E)

Than Quang Khoat

khoattq@soict.hust.edu.vn

School of Information and Communication Technology
Hanoi University of Science and Technology

2022

Content:

- Introduction of Artificial Intelligence
- Intelligent agent
- **Problem solving: Search**, Constraint satisfaction
  - **Informed search**
- Logic and reasoning
- Knowledge representation
- Machine learning

Reminder: Tree-based 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] applied to STATE(node) succeeds return node
        fringe ← INSERTALL(EXPAND(node, problem), fringe)
```

- A search strategy (i.e., method)
  = A way of determining the order to examine the tree's nodes

Informed search

- Uninformed search strategies use only the information contained in the problem definition
  - Not suitable for many practical problems (due to high cost of time and memory)

- Informed search strategies use the *problem-specific knowledge* $\rightarrow$ The search process is more efficient
  - Best-first search algorithms (Greedy best-first, A*)
  - Local search algorithms (Hill-climbing, Simulated annealing, Local beam)

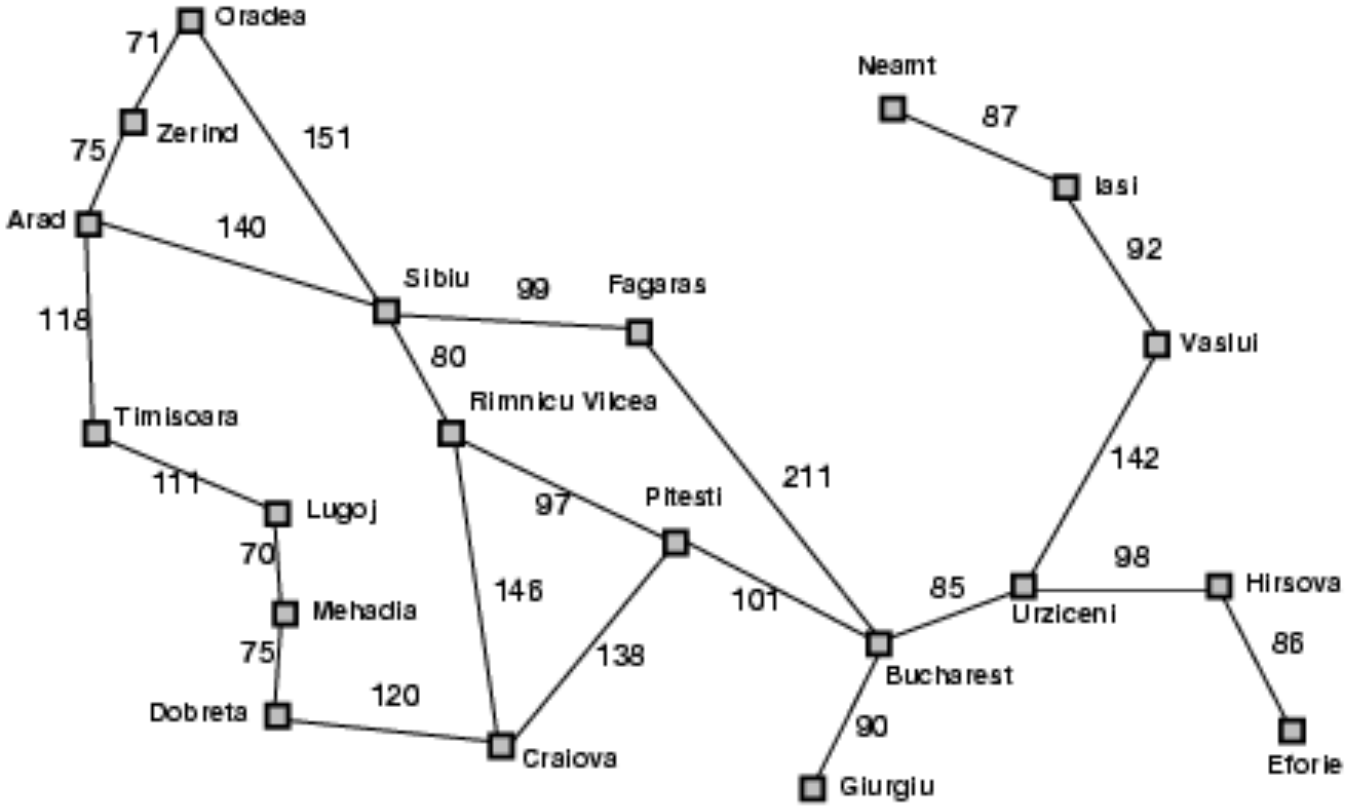
Best-first search

- Intuitive idea: use an *evaluation function f(n)* for every node of the search tree
  - To evaluate the "suitability" of that node
  - → During the search process, the nodes with the highest suitability are given priority
- Implementation
  - Order the nodes in *fringe* in descending order of suitability
- Best-first search algorithms
  - Greedy best-first search
  - A* search

Greedy best-first search

- The evaluation function *f(n)* is a *heuristic* function *h(n)*
- The heuristic function *h(n)* estimates the **cost from *n* to goal**
- Example: In the problem of finding a way from Arad to Bucharest: $h_{SLD}(n)$ = Estimated straight-line distance from *n* to Bucharest
- Greedy best-first search expands the node that **appears** to be closest to goal

Greedy best-first search – Example (1)

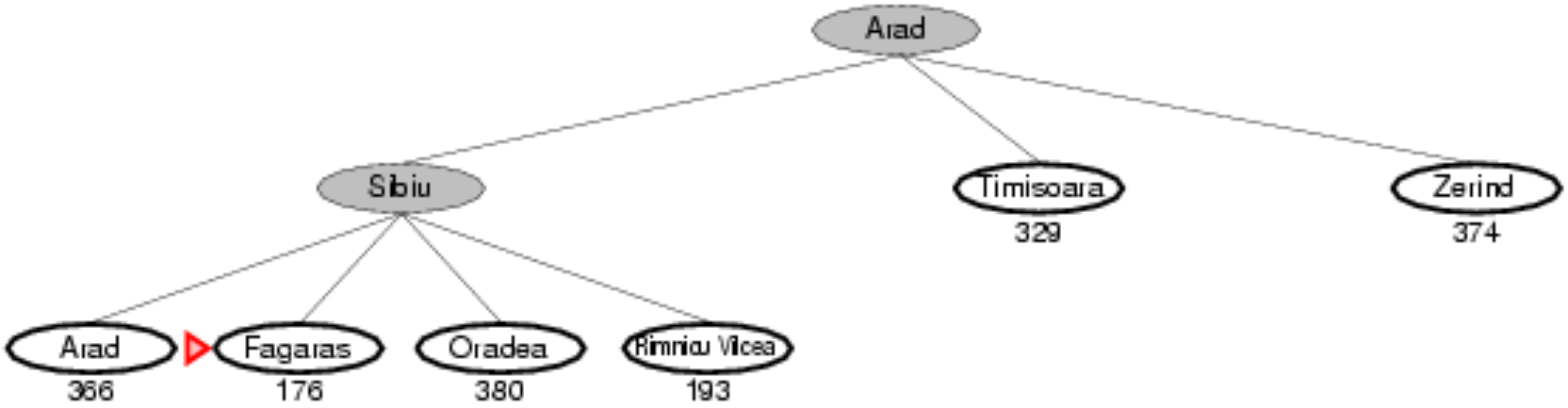


Straight–line distance to Bucharest	
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	176
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	10
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

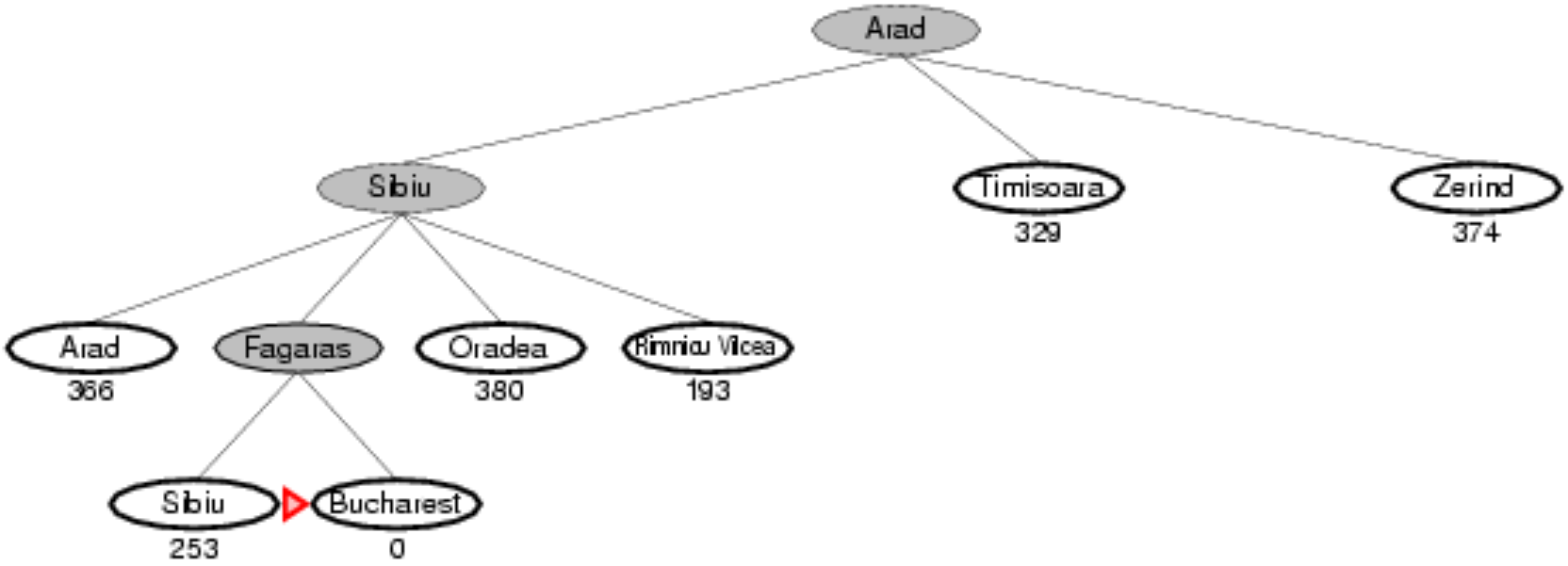
Greedy best-first search – Example (2)

Greedy best-first search – Example (3)

Greedy best-first search – Example (4)



Greedy best-first search – Example (5)



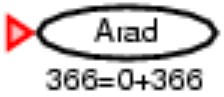
Properties of Greedy best-first search

- Complete?
  - No, because it can get stuck in loops, e.g., Iasi → Neamt → Iasi → Neamt →…
- Time?
  - $O(b^m)$
  - But a good heuristic function can give dramatic improvement
- Space?
  - $O(b^m)$ – Keeps all nodes in memory
- Optimal?
  - No

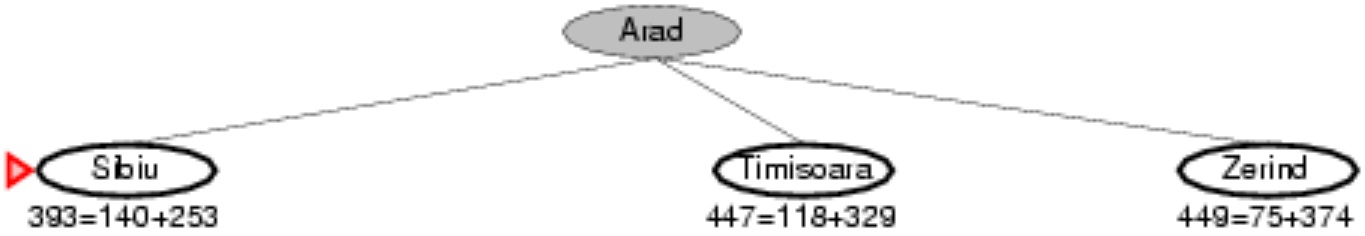
A* search

- Intuitive idea: Avoid expanding paths that are already (i.e., up to the current moment) determined expensive
- Evaluation function *f(n)* = *g(n)* + *h(n)*
  - *g(n)* = The cost from the root node to (the current one) *n*
  - *h(n)* = The estimated cost from *n* to goal
  - *f(n)* = The estimated total cost of path through *n* to goal

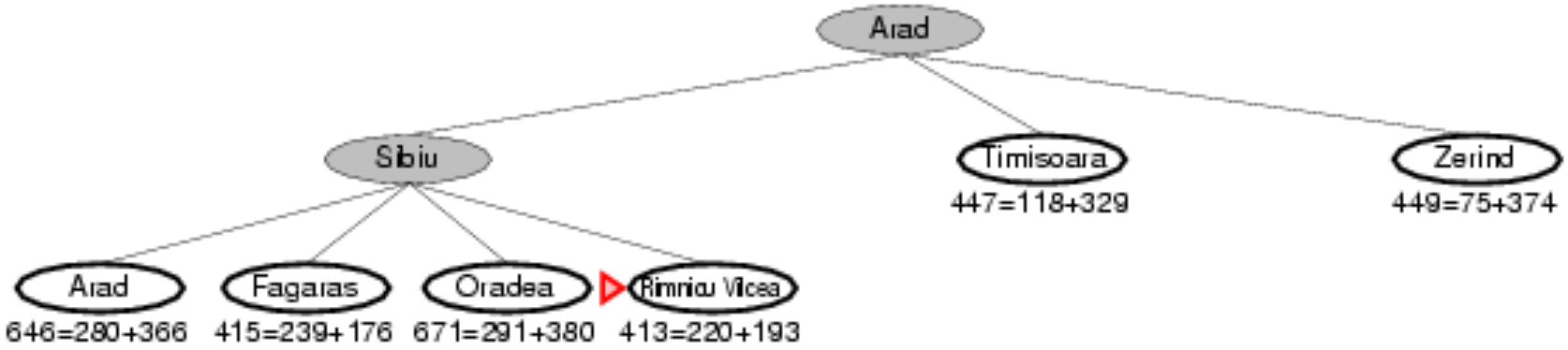
A* search: Example (1)



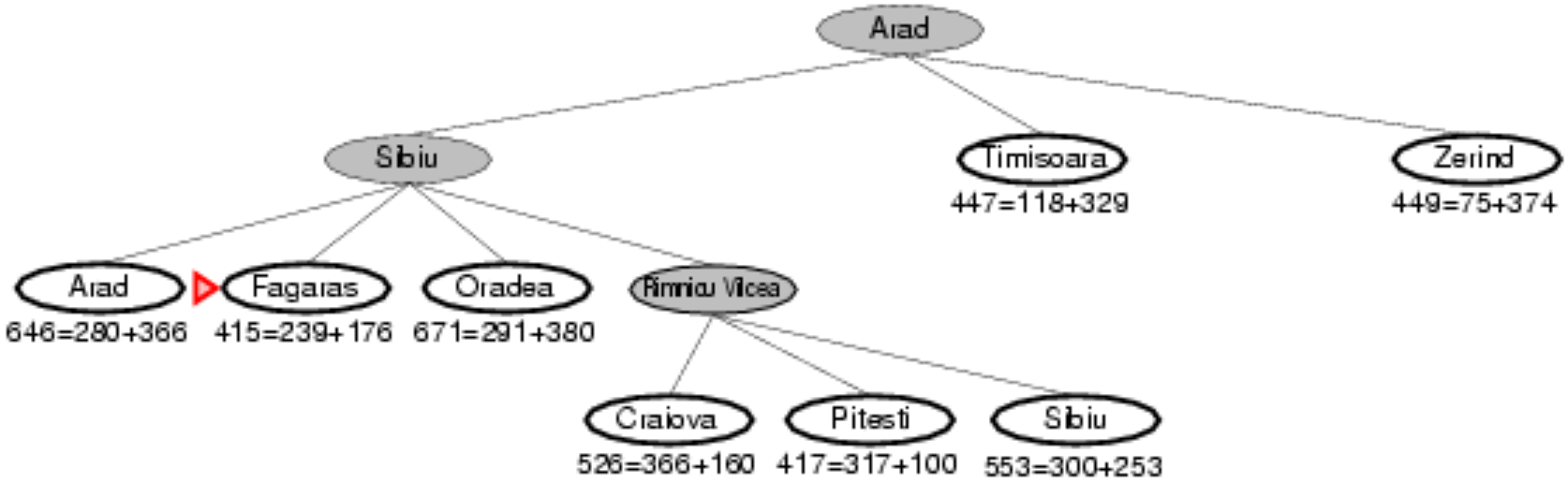
A* search: Example (2)



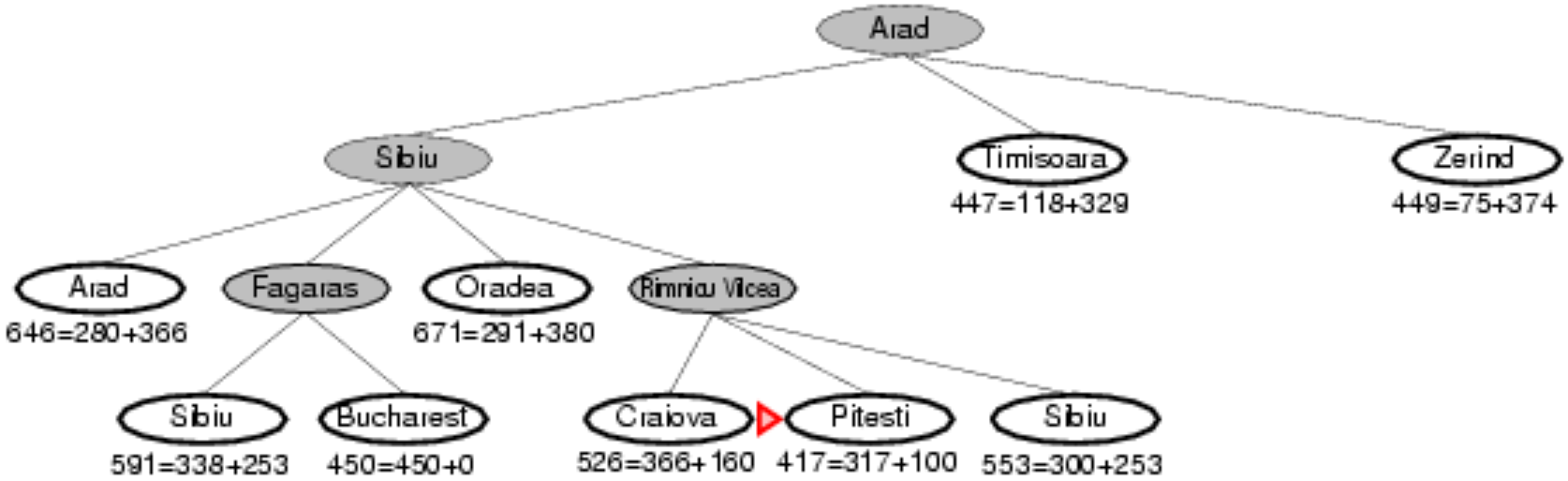
A* search: Example (3)



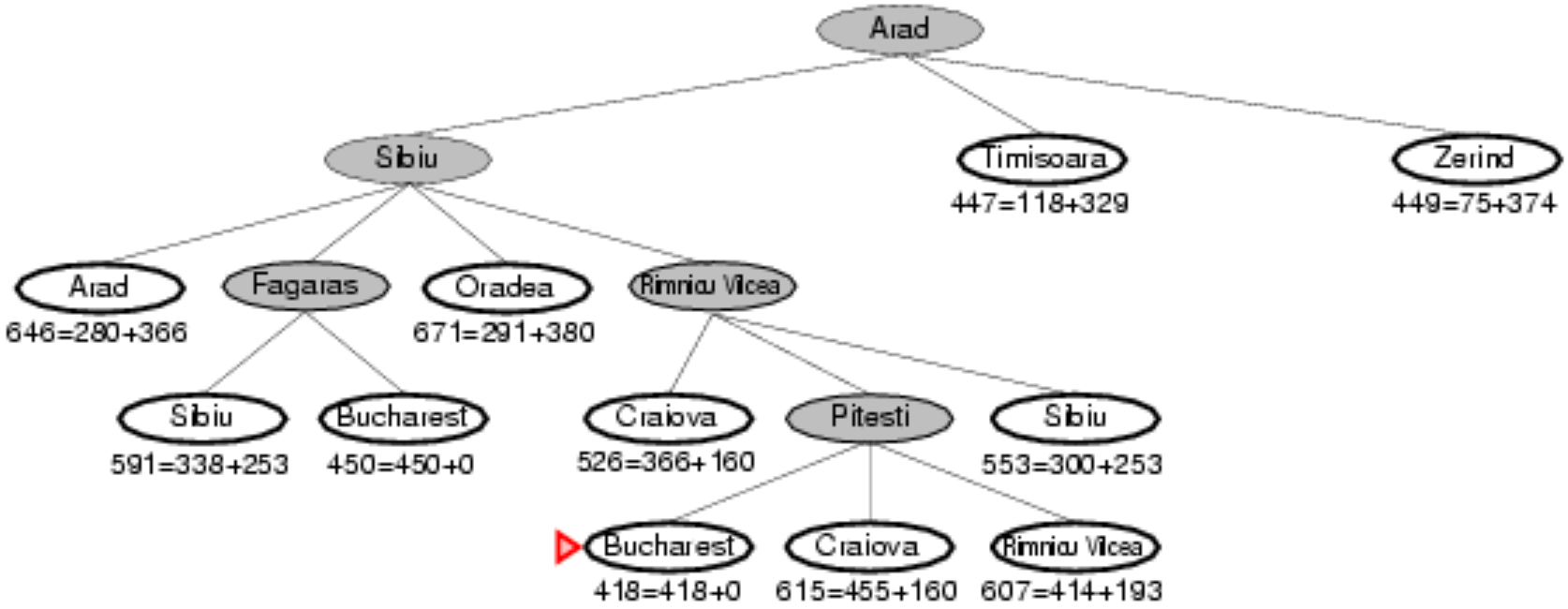
A* search: Example (4)



A* search: Example (5)



A* search: Example (6)

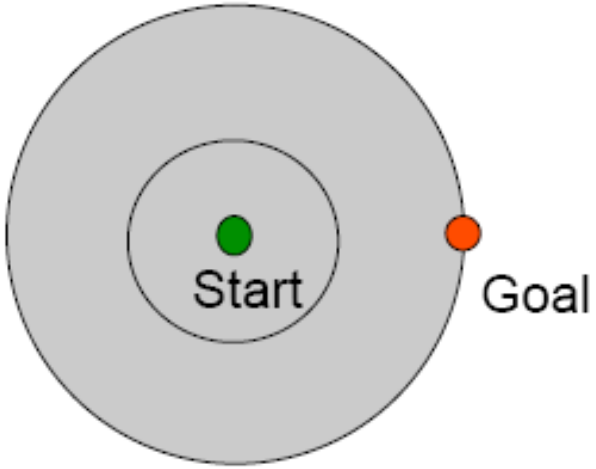


A* search: Properties

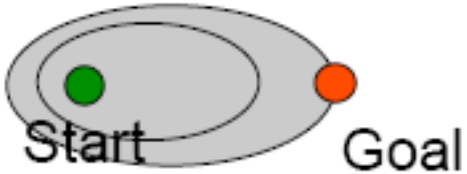
- If *the state space is finite* and there is *a solution to avoid repeating the states*, then the A* algorithm is complete (i.e., can find the solution), but the optimal is not guaranteed
- If *the state space is finite* and there is *no solution to avoid repeating the states*, then the A* algorithm is incomplete (i.e., no guarantee to find a solution)
- If *the state space is infinite*, then the A* algorithm is incomplete (i.e., no guarantee to find a solution)
- When is A* optimal?

A* vs. UCS

- Uniform-cost search (UCS) expands in all directions



- A* expands mainly towards the goal, but the optimal is guaranteed

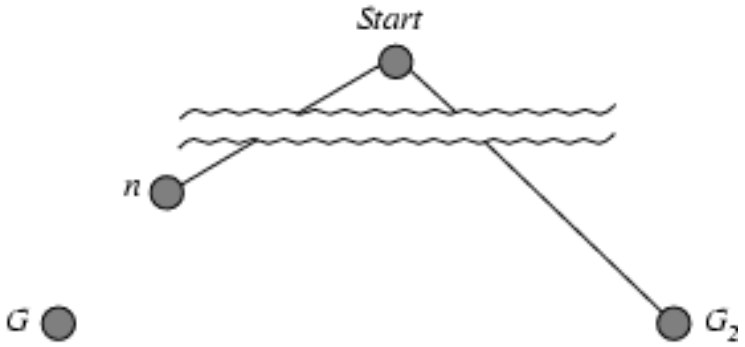


Admissible heuristics

- *A heuristic h(n) is admissible if* $0 \leq h(n) \leq h^*(n)$ *for every node n, where* $h^*(n)$ *is the true cost to reach to the goal state from n*
- An admissible heuristic *never overestimates* the cost to reach the goal
  - It is *optimistic*
- Example: The heuristic $h_{SLD}(n)$ *underestimates* the actual road distance
- **Theorem:** If *h(n)* is admissible, A* using `TREE-SEARCH` is optimal

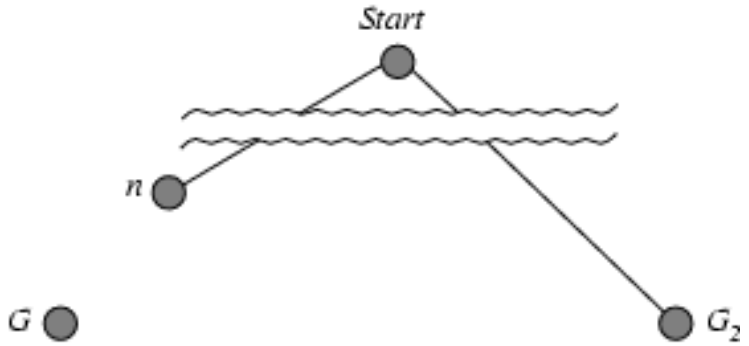
Optimality of A*: Proof (1)

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



- We have: 1) $f(G_2) = g(G_2)$ since $h(G_2) = 0$
- We have: 2) $g(G_2) > g(G)$ since G_2 is suboptimal
- We have: 3) $f(G) = g(G)$ since $h(G) = 0$
- From 1)+2)+3) we have: 4) $f(G_2) > f(G)$

Optimality of A*: Proof (2)



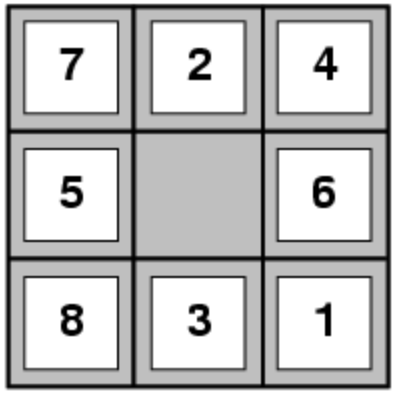
- We have: 5) $h(n) \leq h^*(n)$ since *h* is admissible
- From 5) we have: 6) $g(n) + h(n) \leq g(n) + h^*(n)$
- We have: 7) $g(n) + h^*(n) = f(G)$ since *n* is in the path to *G*
- From 6)+7) we have: 8) $f(n) \leq f(G)$
- From 4)+8) we have: $f(G_2) > f(n)$ A* never selects G_2 for expansion

Admissible heuristics (1)

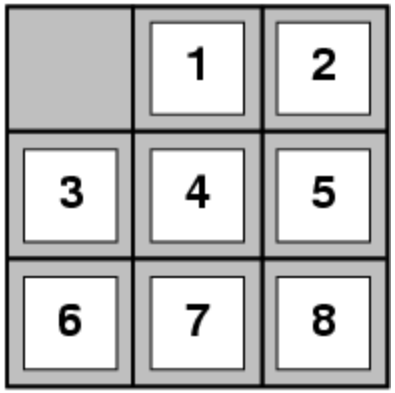
Example: For the 8-puzzle:

- $h_1(n)$ = number of misplaced tiles
- $h_2(n)$ = the least number of moves ($\leftarrow,\rightarrow,\uparrow,\downarrow$) to move the misplaced tiles to their correct position

- $h_1(S)$ = ?
- $h_2(S)$ = ?



Start State



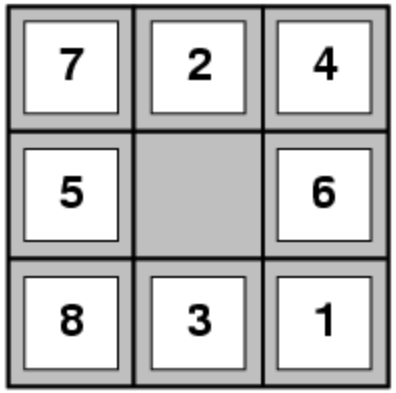
Goal State

Admissible heuristics (2)

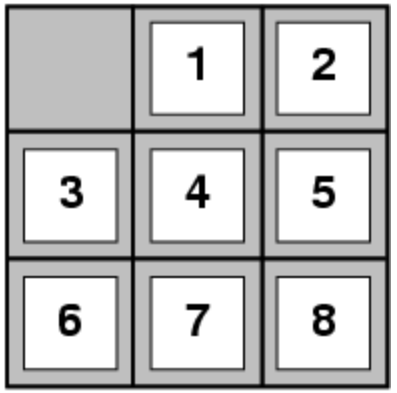
Example: For the 8-puzzle:

- $h_1(n)$ = number of misplaced tiles
- $h_2(n)$ = the least number of moves ($\leftarrow,\rightarrow,\uparrow,\downarrow$) to move the misplaced tiles to their correct position

- $h_1(S) = 8$
- $h_2(S) = 3+1+$
  $2+2+$
  $2+3+$
  $3+2 = 18$



Start State



Goal State

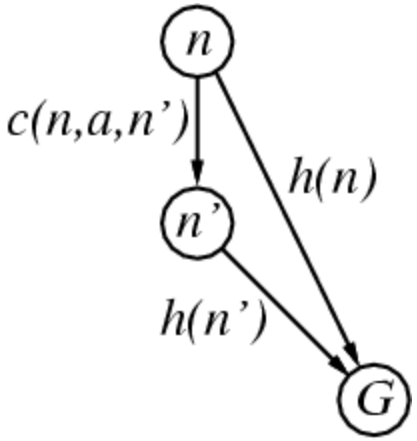
Dominant heuristic

- Heuristic h_2 **dominates** heuristic h_1 if:
  - Both $h_1(n)$ and $h_2(n)$ are admissible, and
  - $h_2(n) \geq h_1(n)$ for every node n
- If heuristic h_2 dominates heuristic h_1, then h_2 is better (to be used) for search
- The 8-puzzle: *Search cost* = *Average number of nodes expanded*
  - For the depth d =12
    - IDS (Iterative Deepening Search): 3,644,035 nodes
    - A*(using heuristic h_1): 227 nodes
    - A*(using heuristic h_2): 73 nodes
  - For the depth d =24
    - IDS (Iterative Deepening Search): Too many nodes
    - A*(using heuristic h_1): 39,135 nodes
    - A*(using heuristic h_2): 1,641 nodes

Consistent heuristics

- A heuristic *h* is **consistent** if for every node *n* and every successor *n'* of *n* (generated by action *a*):

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



- If heuristic *h* is consistent, we have:

  $f(n') = g(n') + h(n')$
  $= g(n) + c(n,a,n') + h(n')$
  $\geq g(n) + h(n) = f(n)$

That means: *f(n)* is non-decreasing along any path passing through *n*

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

Properties of A*

- Complete?
  - Yes (Unless there are infinitely many nodes with f ≤ *f(G)*)
- Time?
  - Exponential (The number of considered nodes is an exponential function of the solution's path length)
- Space?
  - Keeps all nodes in memory
- Optimal?
  - Yes, for some special conditions

Local search algorithms

- In many optimization problems, the path to the goal is irrelevant
  - The goal state itself = The solution
- The state space = A set of "complete" **configurations**
- ***Goal:*** *Find a configuration satisfying all the constraints*
  - Example: The *n*-queens problem (i.e., arrange *n* queens on a board of size $n \times n$ so that they do not attack each other)
- In such problems, we can use local search algorithms
- Save only a single "current" state (i.e., configuration) at a time
  Idea: Try to "improve" this current state using a (predefined) criterion

Example: *n*-queens problem

- Arrange *n* (=4) queens on a board of size $n \times n$ with no two queens on the same row, column or diagonal

Hill-climbing search: Algorithm

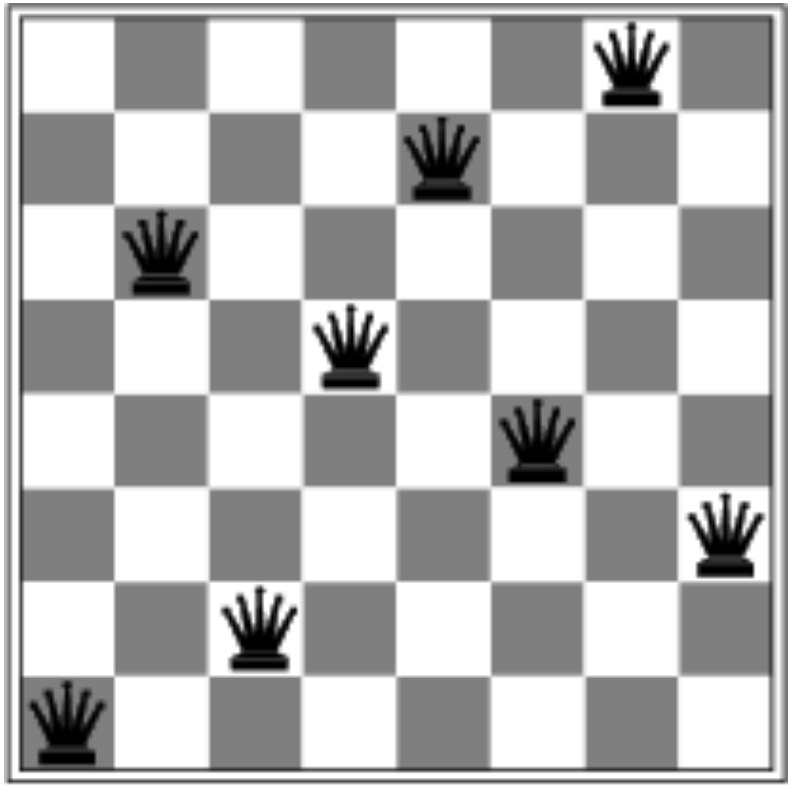
```
function HILL-CLIMBING(problem) returns a state that is a local maximum
    inputs: problem, a problem
    local variables: current, a node
                     neighbor, a node

    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
```

Hill-climbing search: *n*-queens problem (1)

- Heuristic *h* = The number of pairs of queens attacking each other, either directly or indirectly
- For the above state: *h* =17

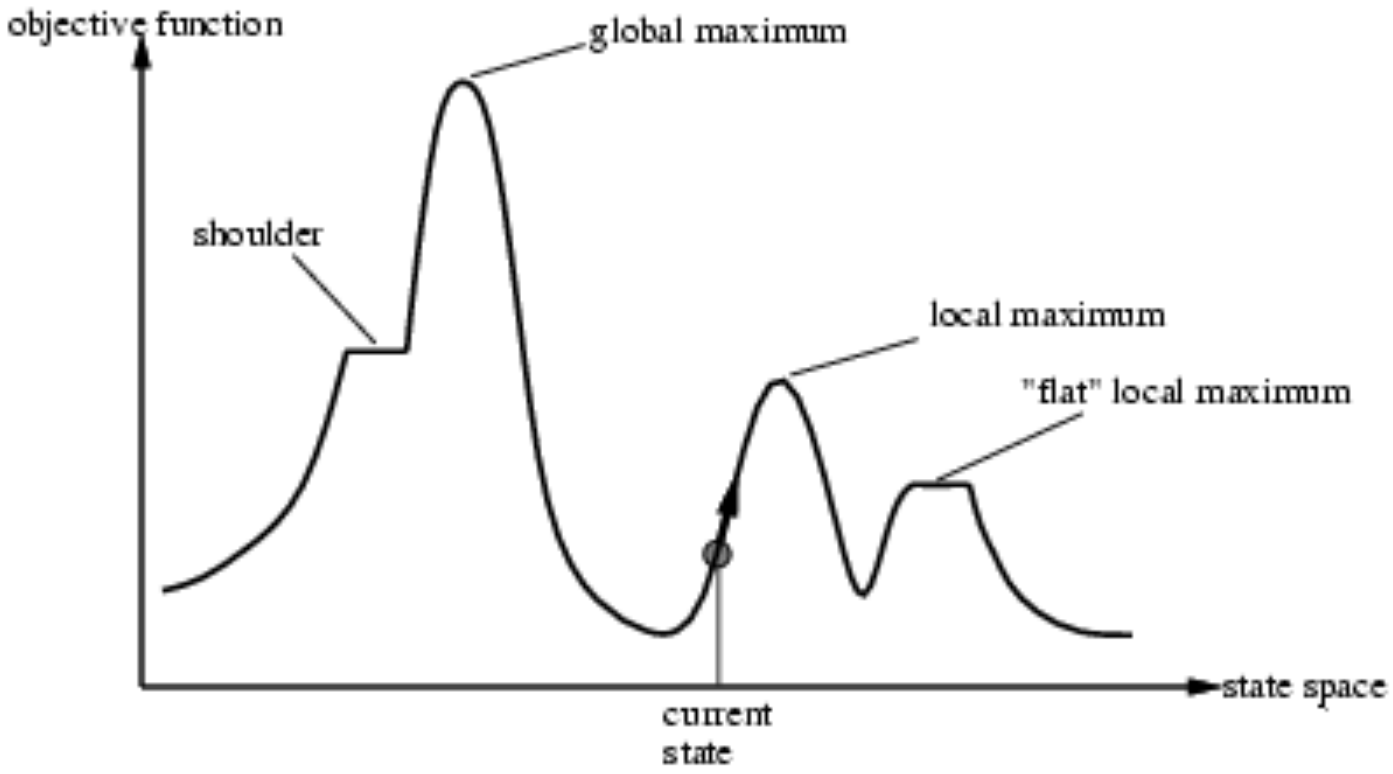
Hill-climbing search: *n*-queens problem (2)



- The above state is a local minimum solution
  - With *h* =1 (i.e., there is still a pair of queens attacking each other)

Hill-climbing search: issue

- Problem: Depending on initial state, can get stuck in local optimal
  - Cannot return the global optimal solution



Simulated annealing search

- *Annealing process:* The metal cools and freezes into a crystallized structure
- Simulated annealing search may avoid returning local optima
- Simulated annealing search uses random search strategy, which accepts changes that increase the value of the target function (need to maximize) and *also accepts (but limited) changes that decrease* the value of the target function
- Simulated annealing search uses a control parameter *T* (as in temperature systems)
  - *T* is a high value at the beginning of the search, and then decreases gradually to 0

Simulated annealing search: Algorithm

- Intuitive 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)
```

Simulated annealing search: Properties

- If *T* (i.e., that defines the degree of frequency reduction for “bad moves”) decreases slowly enough, then simulated annealing search will find a global optimum with probability approaching 1

Local beam search

- At a time in the search process, keep track of *k* – rather than just 1 – best states
- At the beginning of the search: Select *k* states randomly
- At each iteration, generate all the successors of all *k* states
- If any one is a goal state, stop (successfully); else select the *k* best successors from the complete list and repeat

Adversarial search

- IDS and A* only consider the search problems with one agent
- How about an environment with two agents which may have conflict of interest?
  - Adversarial search (Tìm kiếm có đối thủ)
- Adversarial search is often used in games

Issues of search in games

- Hard to predict the reaction of the opponent
  - Need to determine a suitable move for each reaction (or move) of the opponent
- Time limit (time-counting games)
  - Difficult (or unable) to find an optimal solution $\rightarrow$ approximation
- Adversarial search often requires effectiveness (quality of each move and time cost)
  $\rightarrow$ a hard requirement
- In a zero-sum adversarial game:
  - Winner >< Loser
  - Winning score of the winner = Losing score of the loser

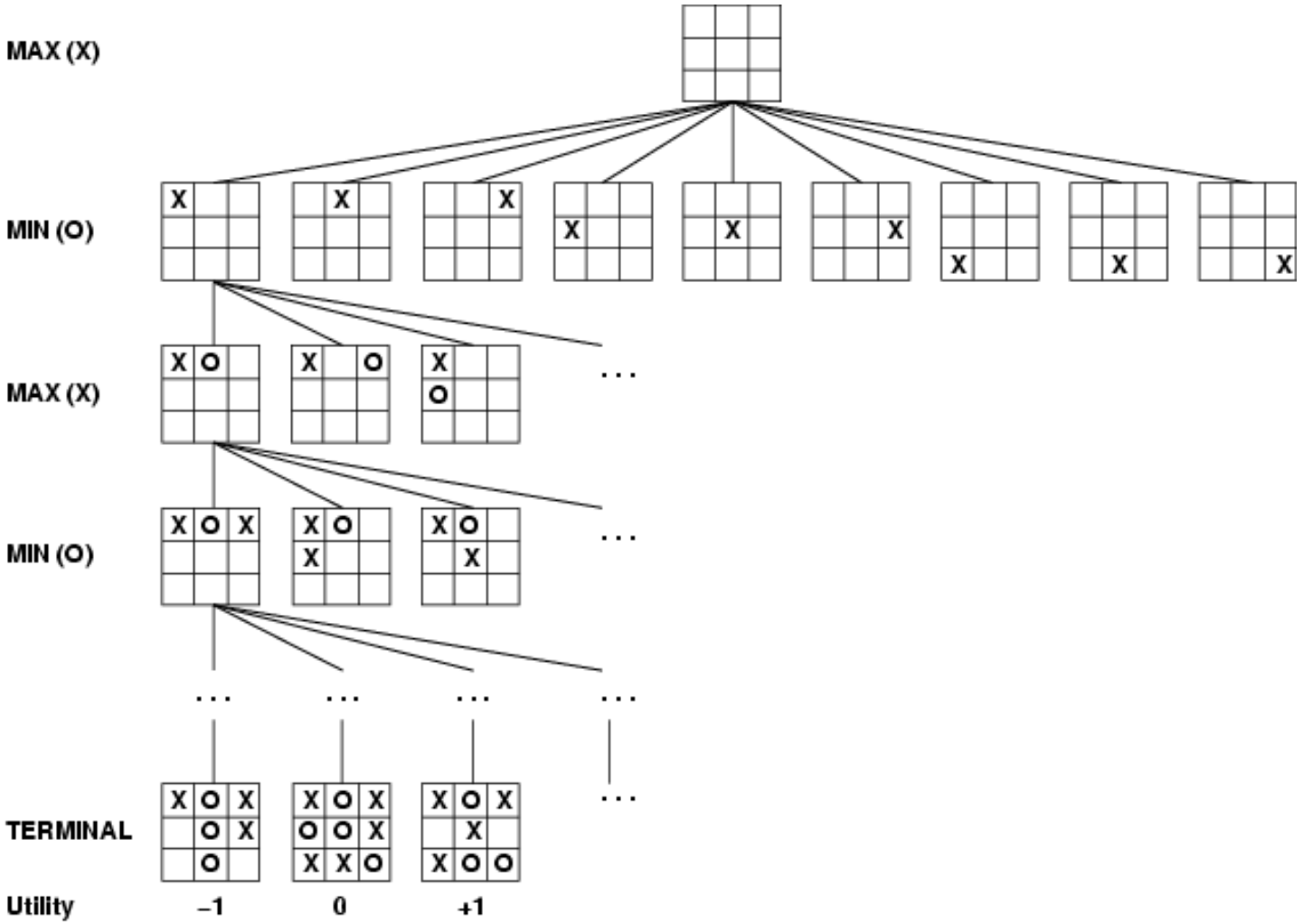
Tic Tac Toe (cờ ca-rô)

- This is an adversarial game
  - E.g.: http://www.ourvirtualmall.com/tictac.htm
- It consists of two players (e.g., MAX and MIN)
  - Each will move just after the other's move
  - Game termination: Winner will have bonus, while loser will be penalized

Representing an adversarial game

- Components for representing a game
  - *Initial state:* State of the game + Who will move first
  - *Successor function:* return some information (given a move, states)
    - All the admissible moves
    - New state (after the move)
  - *Terminal test*
  - *Utility function* to evaluate each state
- Initial state + admissible moves = Game tree

Game tree for Tic Tac Toe

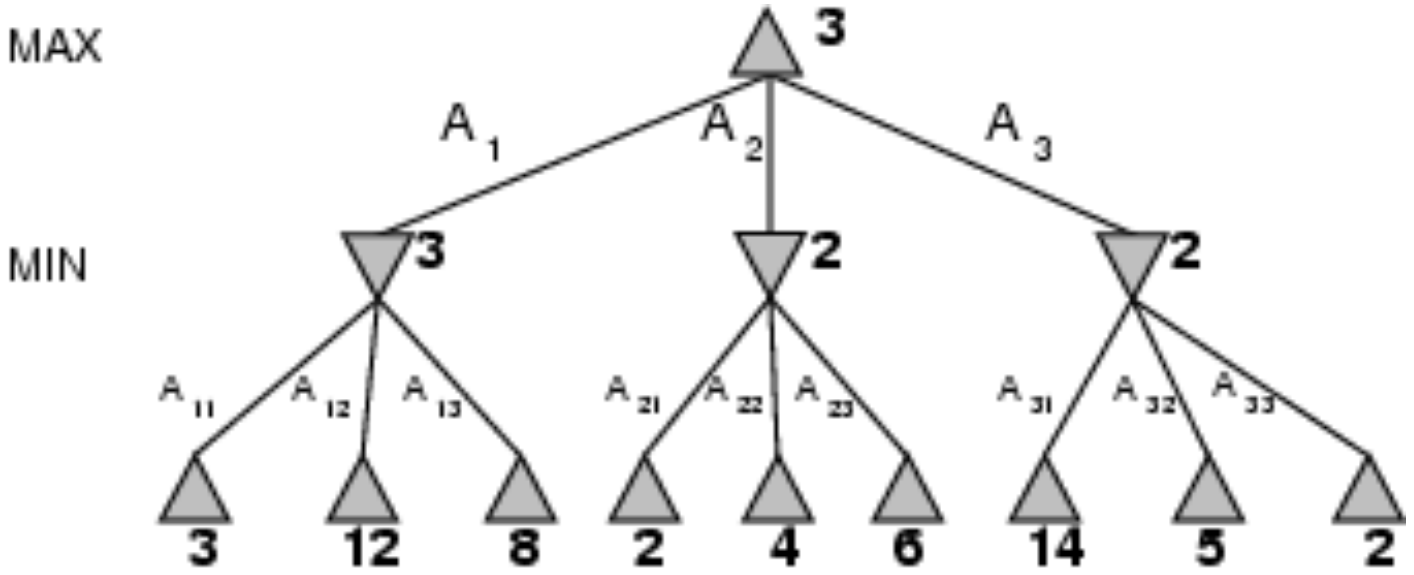


Optimal strategies

- An *optimal strategy* is a sequence of moves to achieve the goal (e.g., winner)
- The strategy of MAX can depend on the moves by MIN, and vice versa
- MAX needs choosing a strategy that *maximizes* its objective function, assuming that MIN uses optimal moves
  - MIN needs choosing a strategy that *minimizes* its objective function
- This strategy can be determined by considering the MINIMAX value at each node in a game tree

MINIMAX value

- MAX chooses a move with *maximal* MINIMAX value (to maximize its objective function)
- In contrast, MIN chooses a move with *minimal* MINIMAX value (to minimize its objective function)



MINIMAX algorithm

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

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

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

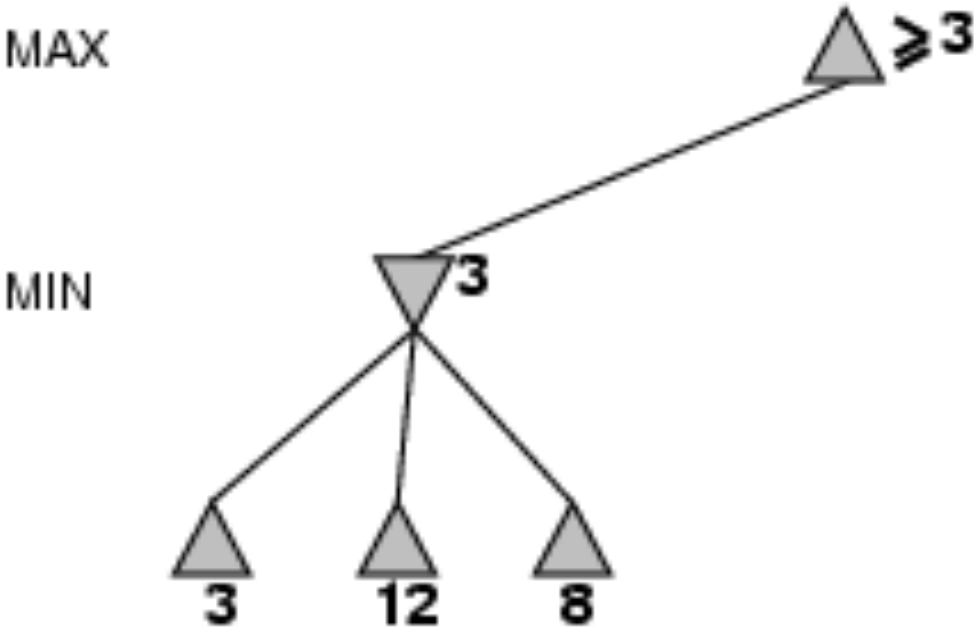
MINIMAX algorithm: properties

- Completeness
  - Yes (if the game tree is finite)
- Optimality
  - Yes (if the players always choose the optimal move at each step)
- Time complexity
  - $O(b^m)$
- Memory complexity
  - O(bm) (based on DFS)
- For Chess, branching factor $b \approx 35$ and tree depth $m \approx 100$
  - Too expensive – cannot find an optimal strategy

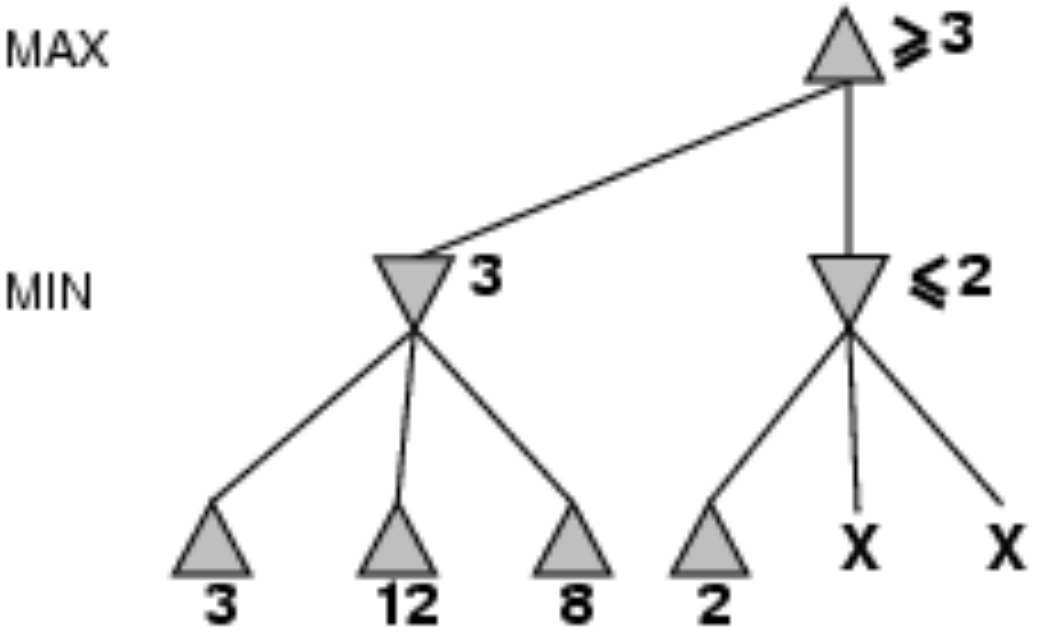
Pruning (cắt tỉa)

- Issue: MINIMAX algorithm may have an exponential number of moves to be considered $\rightarrow$ may not be practical
- We can prune some branches in the tree
- α-β pruning (Alpha-beta pruning):
  - *Idea:* if a branch cannot improve the objective function, we can ignore it!
  - Pruning a bad branch may not affect the solution.

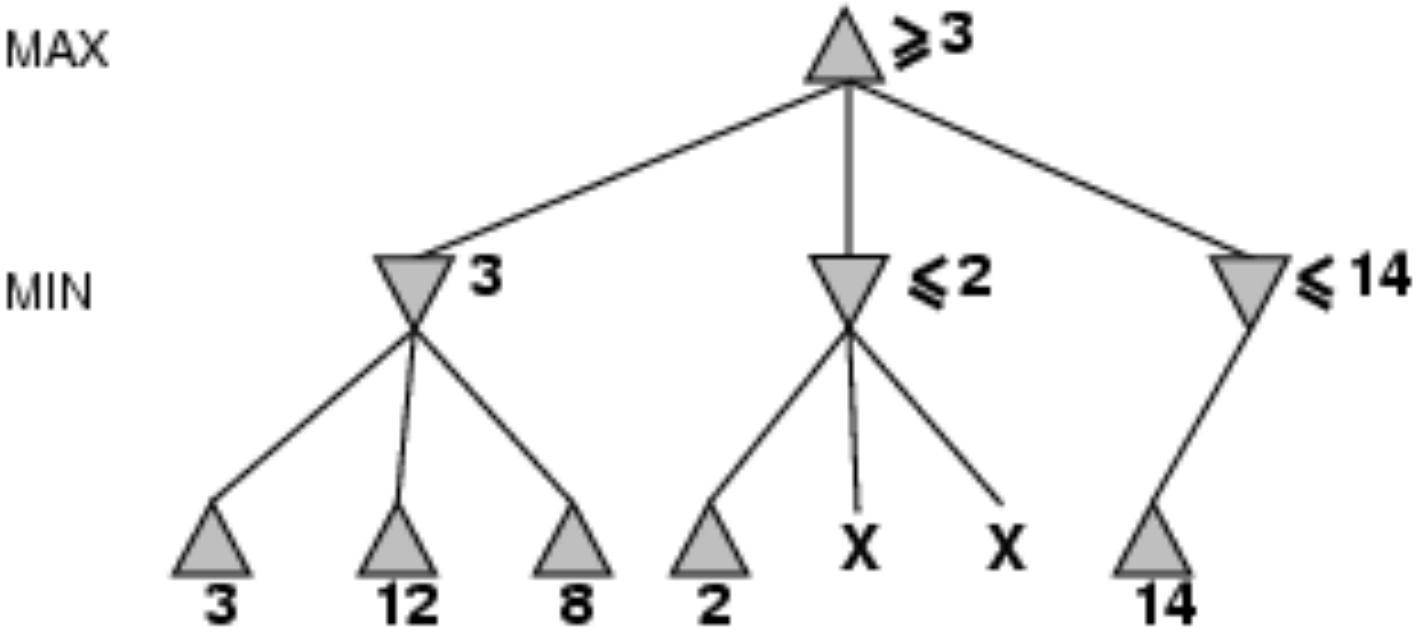
α-β pruning: example (1)



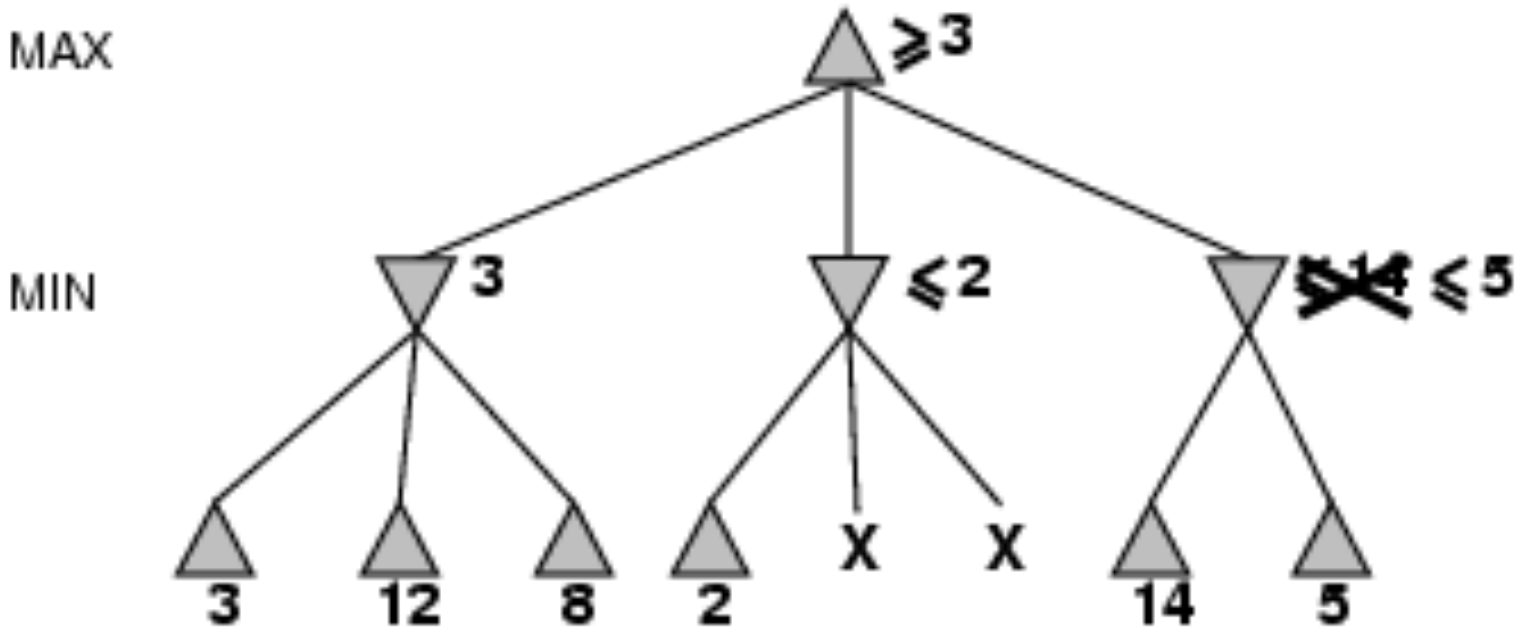
α-β pruning: example (2)



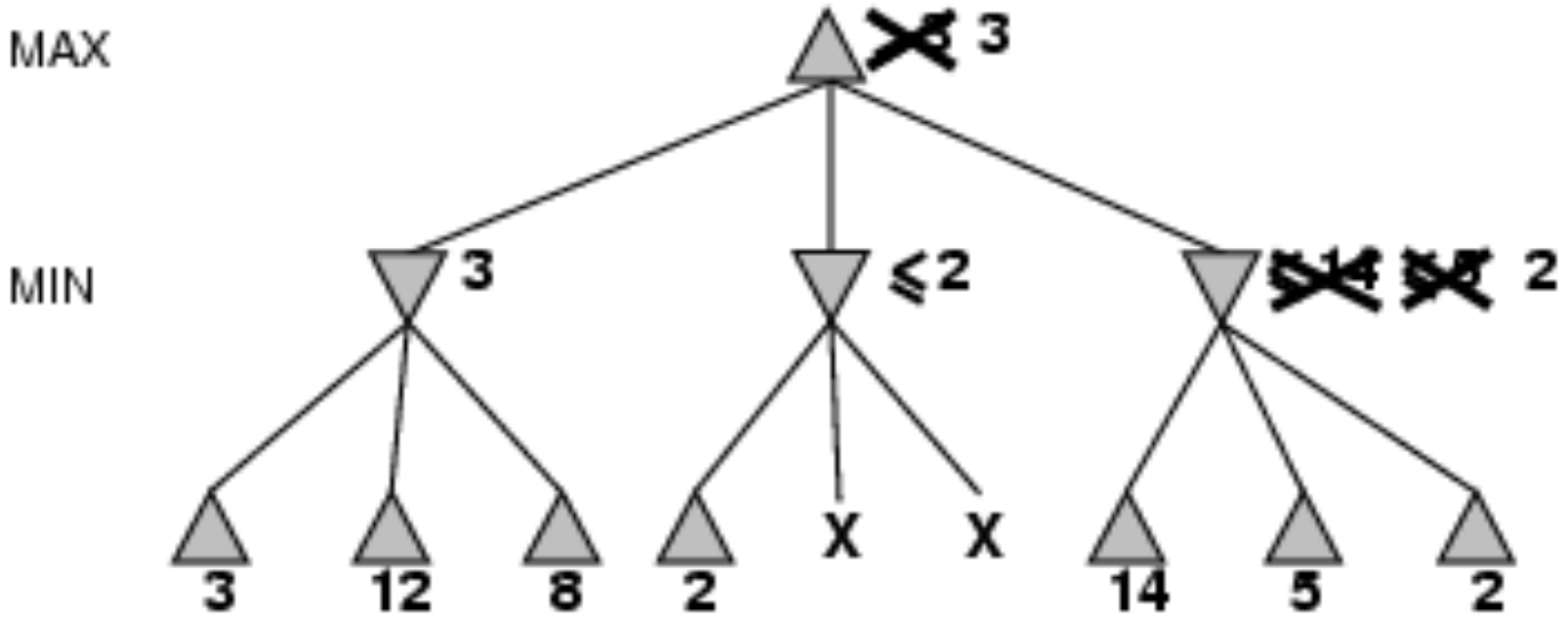
α-β pruning: example (3)



α-β pruning: example (4)

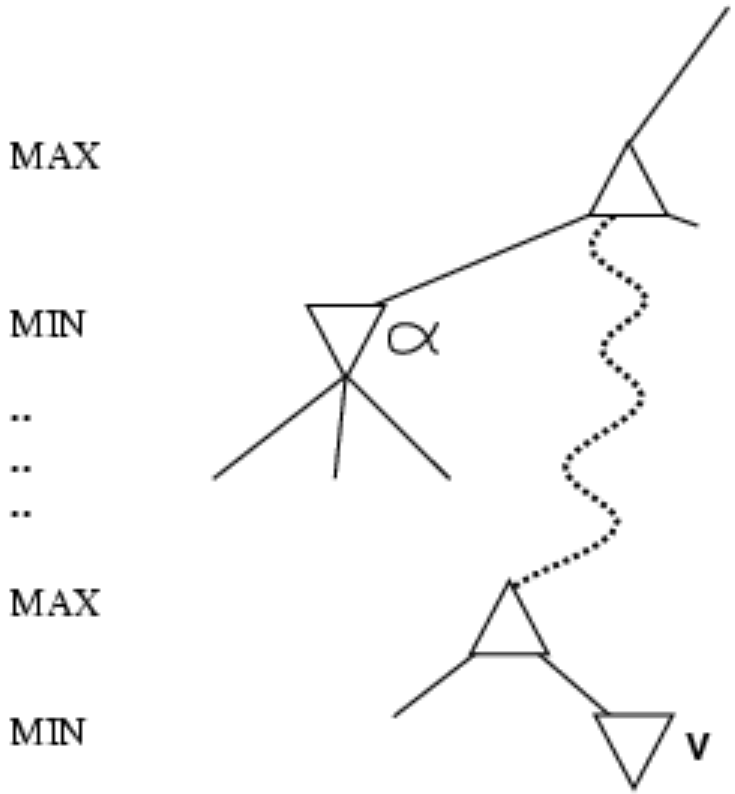


α-β pruning: example (5)



Why calling α-β pruning?

- α is the best move's value of MAX until now at the current branch
- If v is worse than α, MAX will ignore the moves with value v
  - Prune the branches with value v
- β has the same meaning for MIN



α-β pruning algorithm (1)

```
function ALPHA-BETA-SEARCH(state) returns an action
    inputs: state, current state in game

    v ← MAX-VALUE(state, −∞, +∞)
    return the action in SUCCESSORS(state) with value v

---

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

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

α-β pruning algorithm (2)

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

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

α-β pruning

- For games with a large state space, α-β pruning is still not good
  - The pruned space is still large
- Domain knowledge about the game can be used to reduce the search space
  - Such a knowledge can enable us to evaluate each state
  - Such an additional knowledge plays a similar role with heuristic function *h(n)* in the A* algorithm