CHAPTER II

Search Algorithms and Problem-Solving

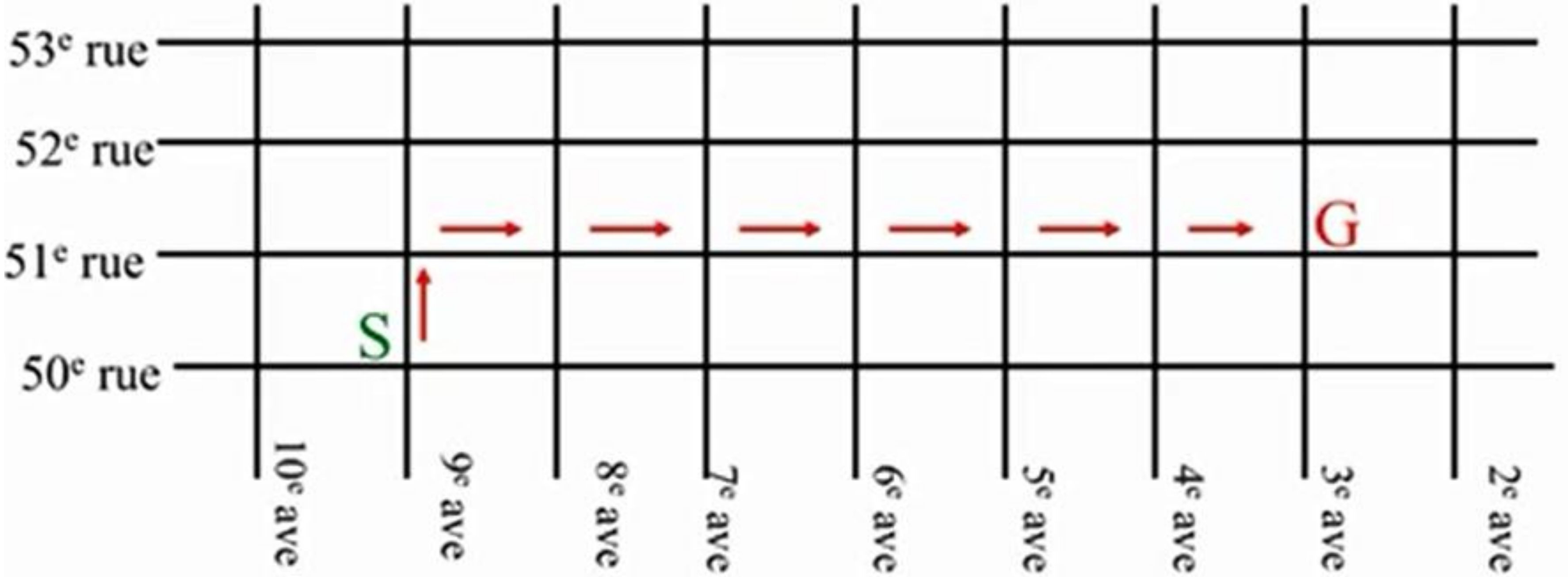
Solving a problem

- **Intuitive steps by a human**
  - Model the current situation
  - List possible solutions
  - Evaluate the value of each solution
  - Select the best option satisfying the goal
- **How to efficiently browse the list of solutions ?**
- **Several problems con be solved by searching in a graph :**
  - Each node represents a state of the environment
  - Each path through a graph represents a sequence of actions
  - The solution: simply look for the path that best satisfies our performance measurement

Problem-Solving

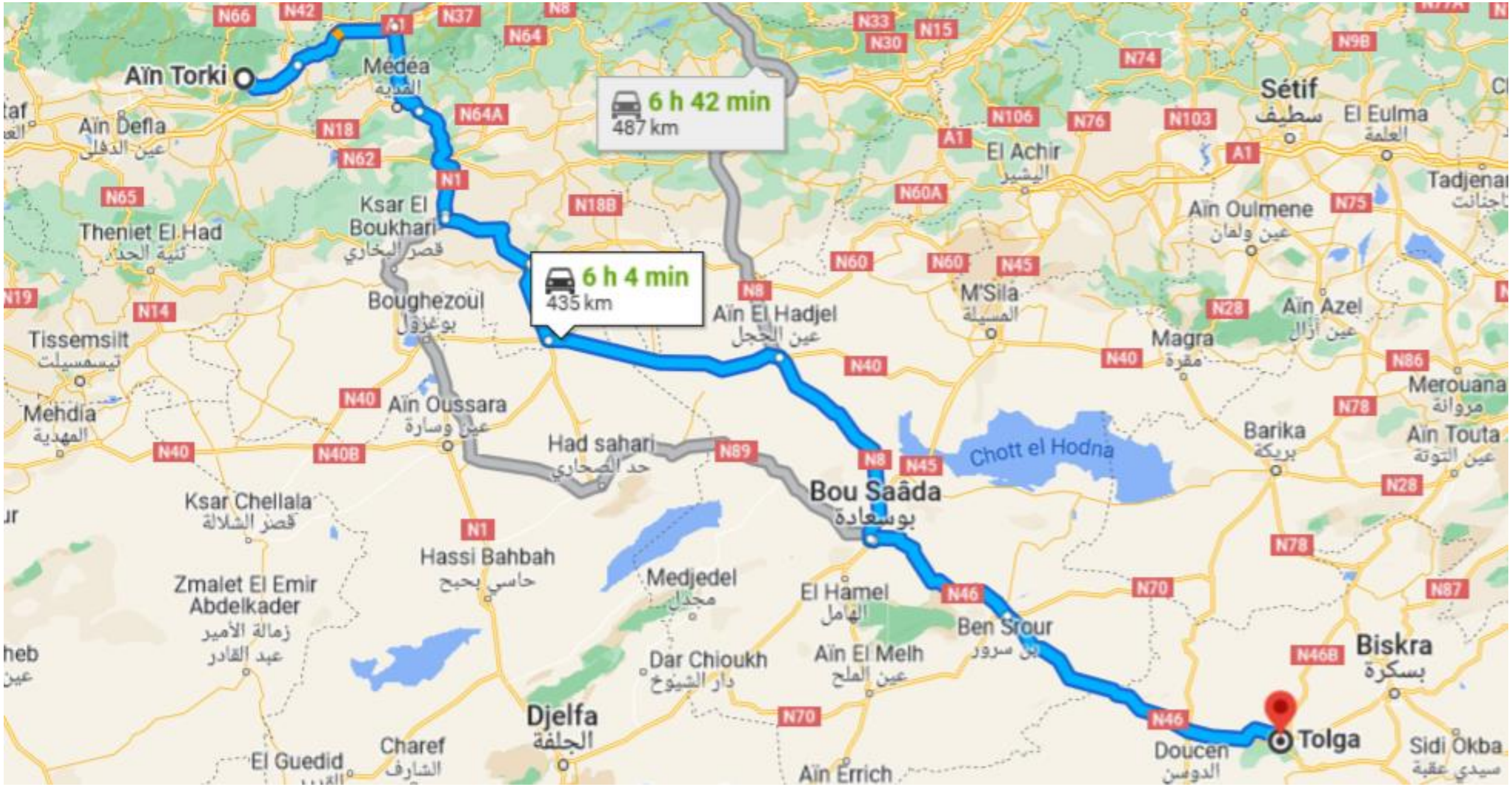
Example: Path-finding in a town

Find the best path between the 9th ave – 50th street to the 3rd ave -51st street



Problem-Solving

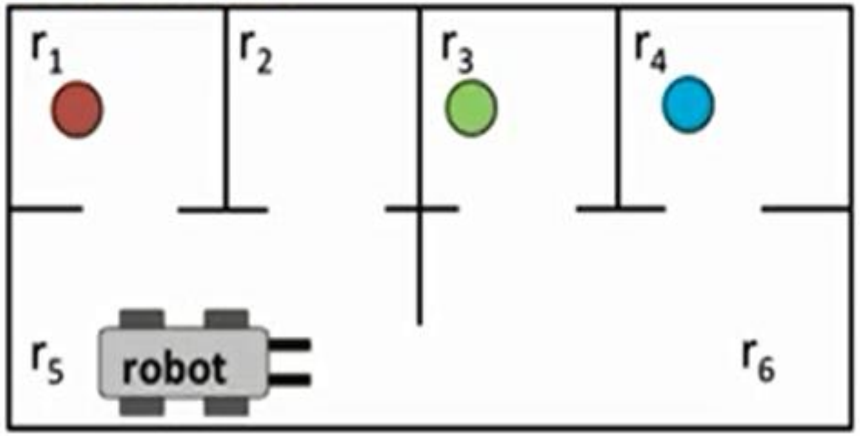
Example: Google Maps



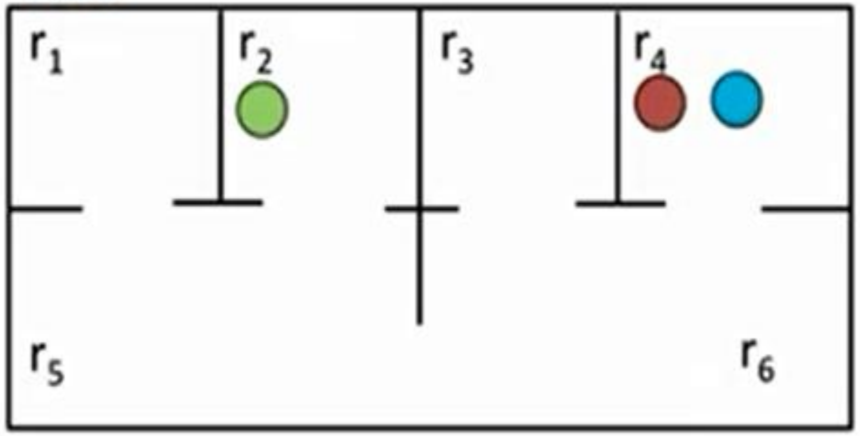
Problem-Solving

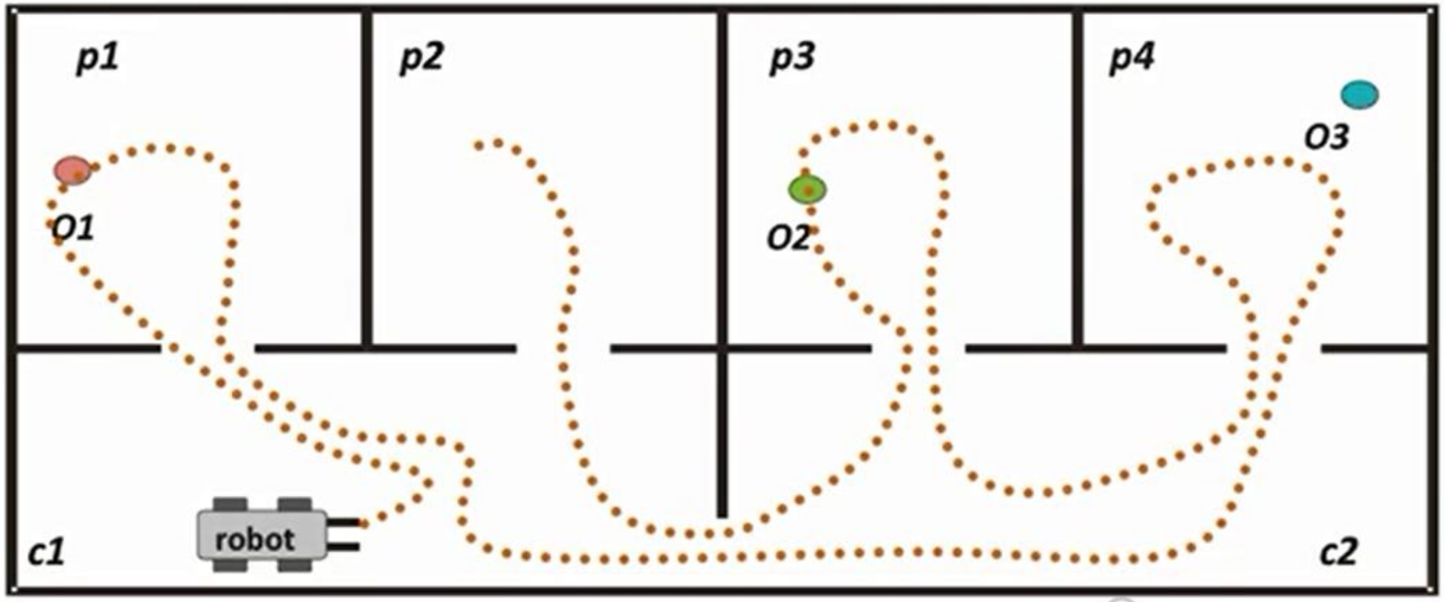
Example: Package delivery

Initial state



Goal



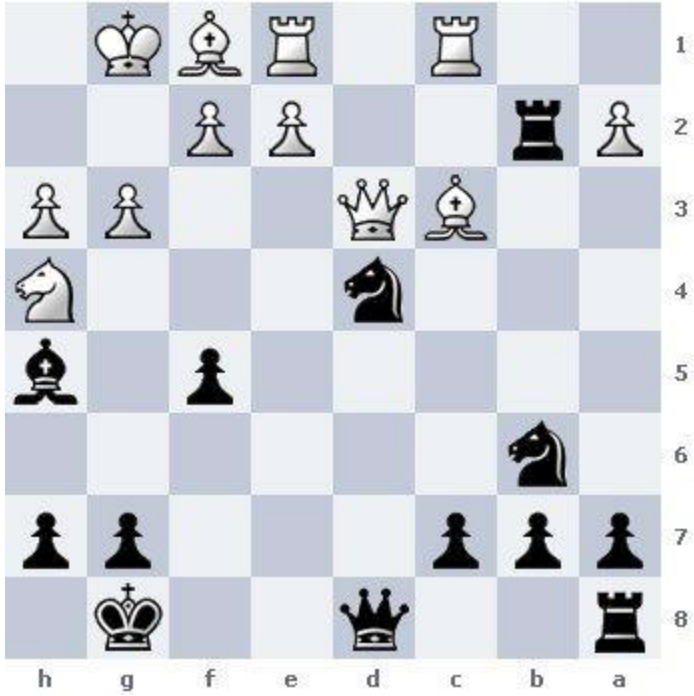


Problem-Solving

Example: Chess game

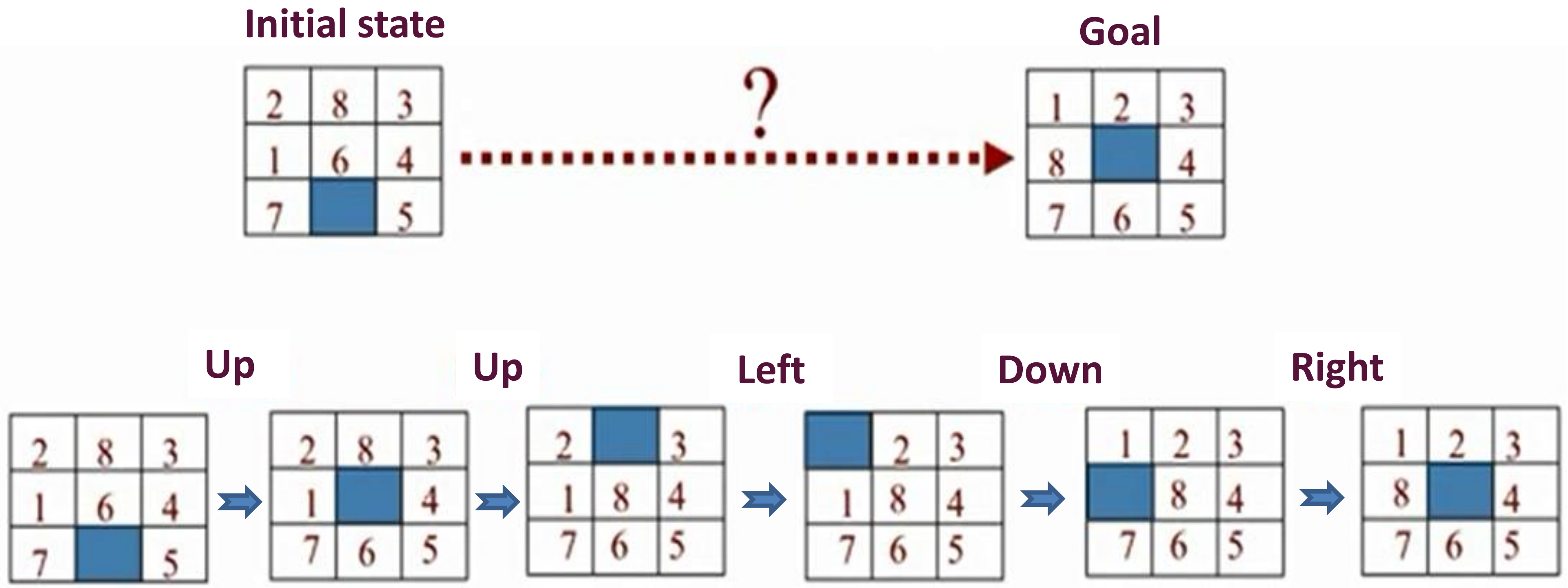
Initial state

Goal



Problem-Solving

Example: N-Puzzle



Initial state

2	8	3
1	6	4
7		5

Goal

1	2	3
8		4
7	6	5

Up

2	8	3
1		4
7	6	5

Up

2		3
1	8	4
7	6	5

Left

	2	3
1	8	4
7	6	5

Down

1	2	3
	8	4
7	6	5

Right

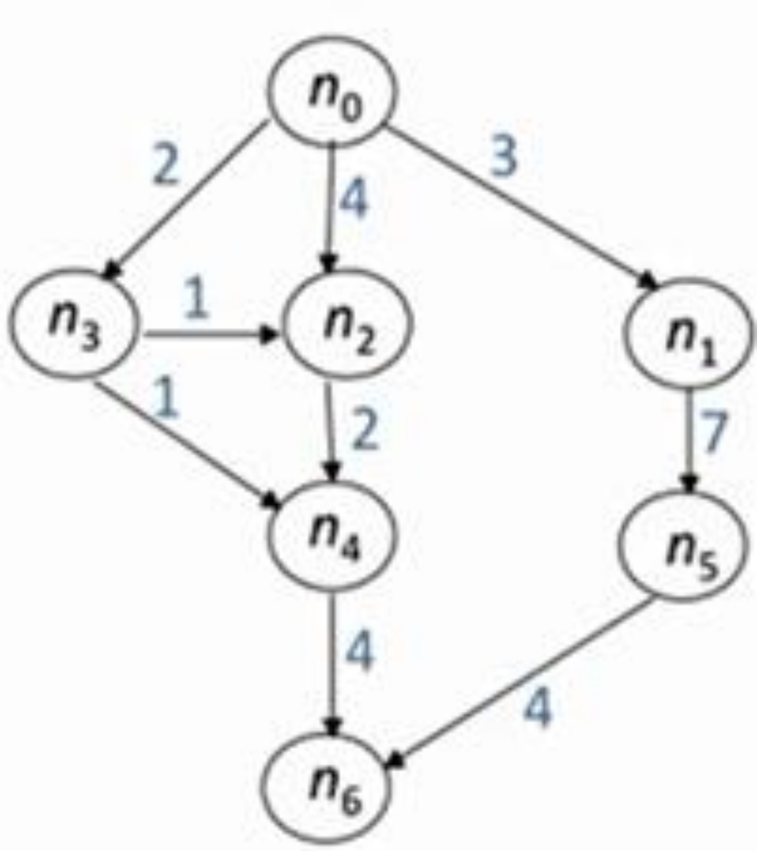
1	2	3
8		4
7	6	5

Graph search problem

- **Input:**
  - Initial node
  - Goal function **Goal(n)** which returns **True** if the goal is achieved
  - Transition function **Transition(n)** which returns the successor nodes of n
  - Cost function c(n,n’) strictly positive, which returns the cost of going from n to n’
- **Output:**
  - A path in the graph (nodes and edges)
    - The path cost is the sum of all the edges cost in the graph
    - There can be several goal nodes
- **Challenges:**
  - Find a solution path
  - Find an optimal path
  - Quickly find a path (in this case the optimality is not important)

Problem-Solving

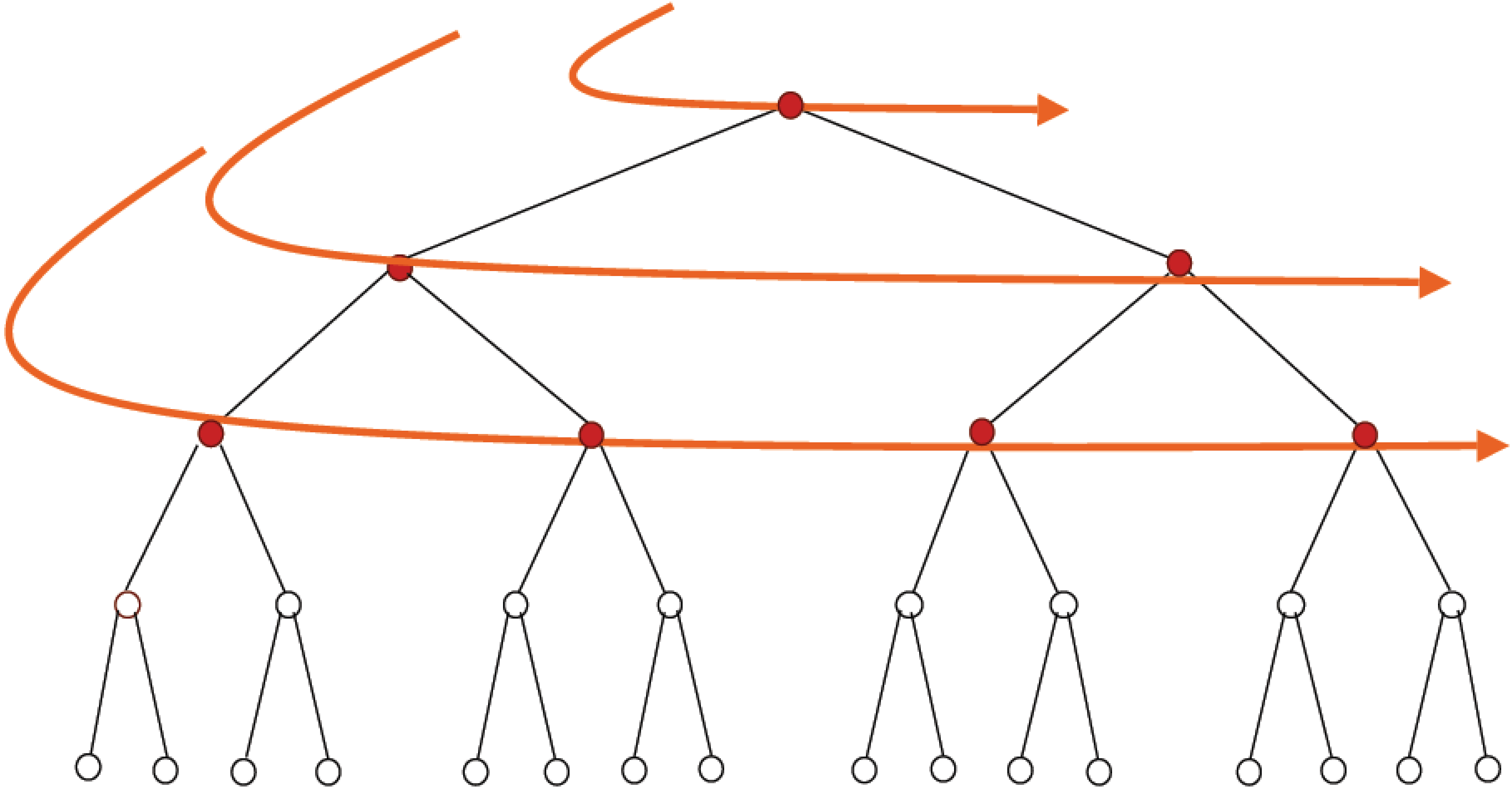
A real world example: Find a path between tow cities



- Cities: **Nodes**
- Paths between two cities: **Edges**
- Starting city: Initial node $\mathbf{n_0}$
- Roads between cities: **Transition(n_0) = (n_3, n_2, n_1)**
- Distance between cities: **c(n_0,n_2) = 4**
- Destination city: **Goal(n) = True** if n = n_6 (n_6 is the destination city)

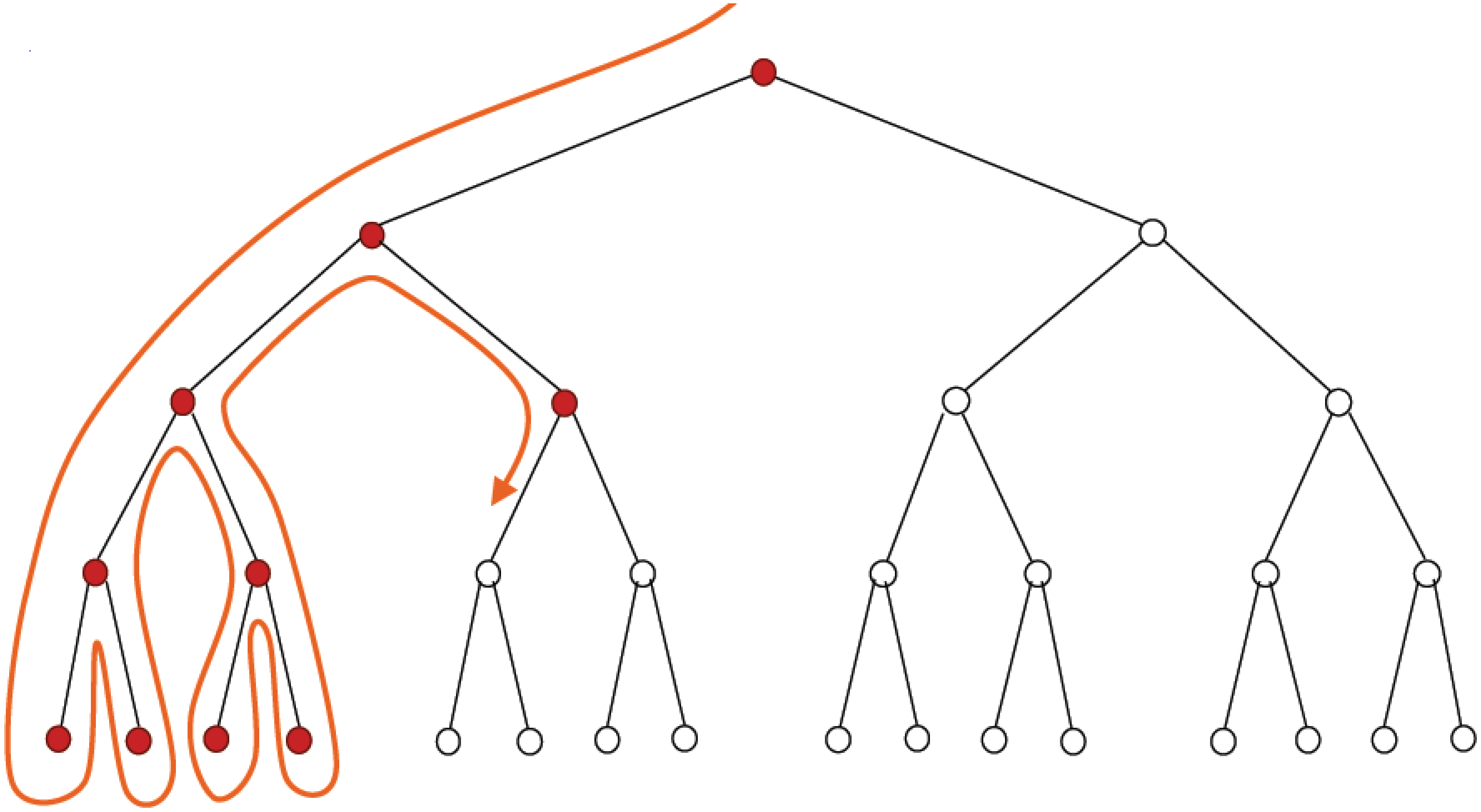
Search algorithm : Breadth-First Search

- For a given node, explore the sibling nodes before exploring their children.



Search algorithm : Depth-First Search

- For a given node, explore the first child node before exploring the sibling nodes.



Heuristic-Based Search Algorithms

Best-First Search

1. Start the search by a **List** containing the starting state (**initial node**) of the problem

2. If **List** not empty:

   - Select a state **n** with **minimal** measure to expand
   - If **n** is a final state (**Goal node**) then return Success
   - Else, add all **n successor nodes** to the List with respect of ascending order according to the utility measure.
   - Restart at point 2.

3. Else return Failure.

Heuristic-Based Search Algorithms

Greedy Best-First Search

- The utility measure is given by an estimation function **h**.

- For each state **n, h(n)** represents the **estimated cost** from **n** to a **final state**.

For example, in the problem of the shortest path between two cities,

we can take **h(n) = direct distance** between **n** and the **destination city**.

- Greedy search will choose the state that seems **closest** to a final state

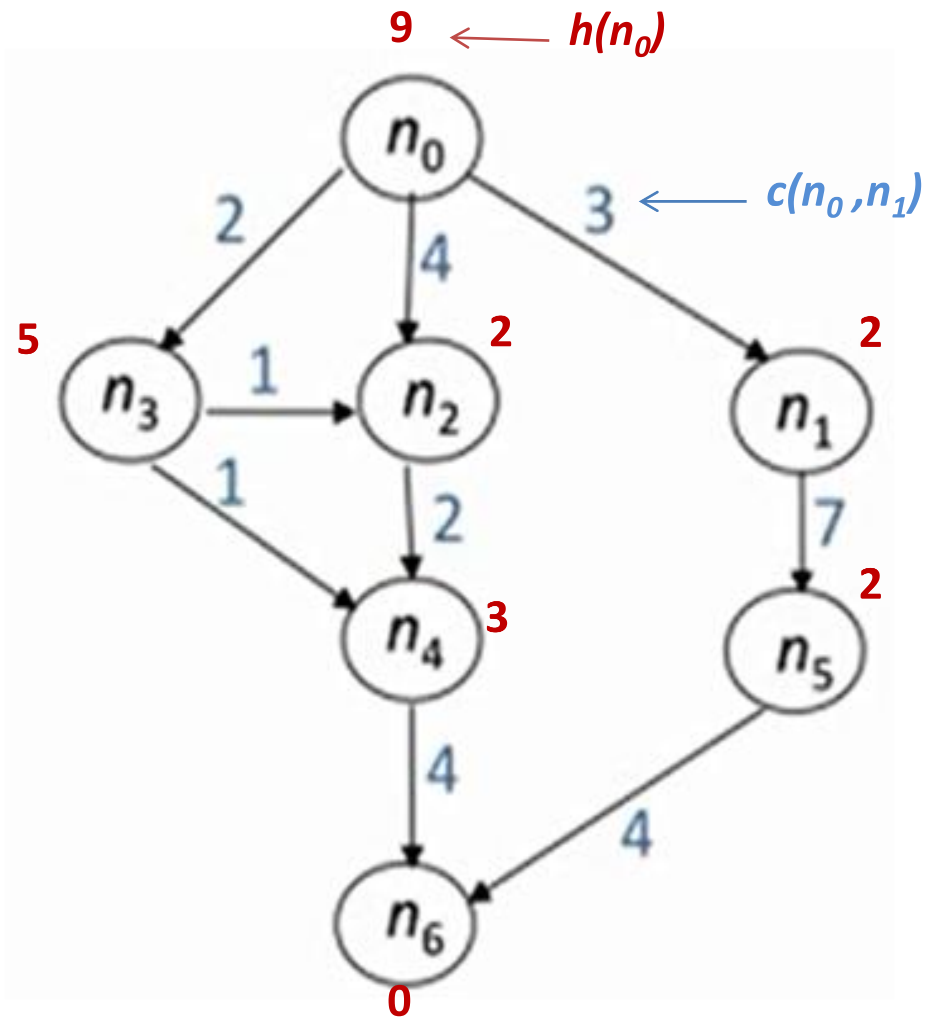
according to the **estimation function h**.

Greedy Best-First Search

Open List :

- $(n_0,9,void)$
- $(n_2,2,n_0)$, $(n_1,2,n_0)$, $(n_3,5,n_0)$
- $(n_1,2,n_0)$, $(n_4,3,n_2)$, $(n_3,5,n_0)$
- $(n_5,2,n_1)$, $(n_4,3,n_2)$, $(n_3,5,n_0)$
- $(n_6,0,n_5)$, $(n_4,3,n_2)$, $(n_3,5,n_0)$

Path : $n_0 \rightarrow n_1 \rightarrow n_5 \rightarrow n_6$



Heuristic-Based Search Algorithms

A* Search

- The utility measure is given by an evaluation function f

- For each node **n**: ***f*(*n*) = *g*(*n*) + *h*(*n*)**

  ○ ***g*(*n*)** *Is the cost till present to get* ***n***

  ○ ***h*(*n*)** Is the estimated cost to go from **n** to the **goal node**.

  ○ ***f*(*n*)** Is the total estimated cost to go from the **initial node** to the **goal node** going through **n**

h is said to be **admissible** if for all **n**: **h(n) ≤ c(n)**

c(n) being the real cost leading from **n** to the **final state**

A* Search Algorithm

```
1. Declare two nodes n, ns
2. Declare two lists Open and Closed (initially empty)
3. Add initial node to Open
4. If Open is empty Then Exit the loop with a failure
5. Current node n = node at the head of Open
6. Remove n from Open and add it to Closed.
7. If n= goal Then Exit the loop and return the path
   Else : For each successor ns of n:
       - Initilize the value $g(ns) = g(n) + c(n,ns)$
       - Set parent of ns to n
       - If Open or Closed contains a node ns'=ns with $f(ns) \leq f(ns')$
         Then remove ns' from Open or Closed and insert ns into Open (with respect to the ascending order of f)
         Else : Insert ns into Open (with respect to the ascending order of f)
       - Go to 4.
```

A* Search Algorithm

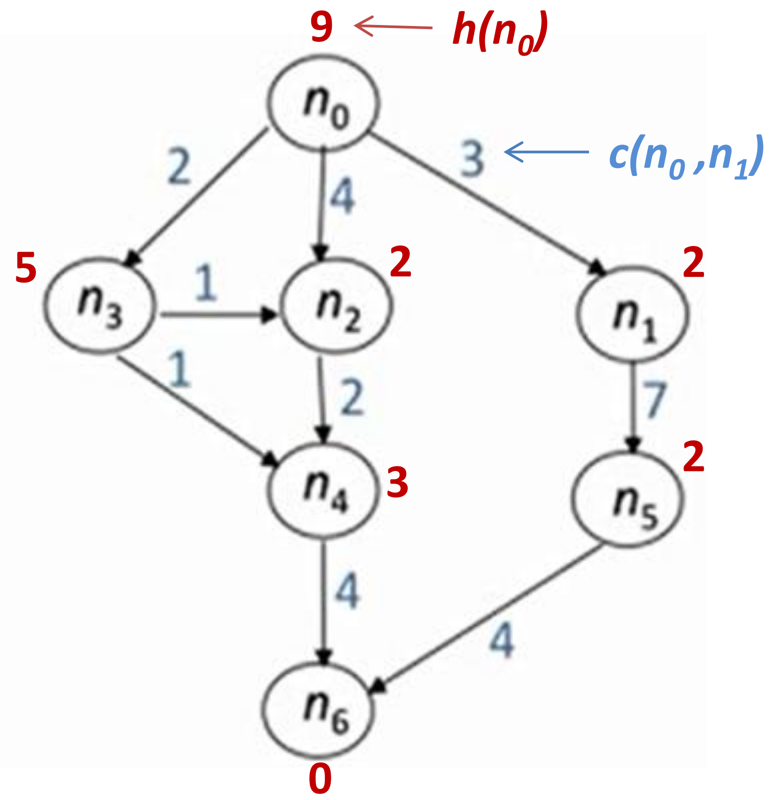
Illustrative example: Path-Finding between two cities

n_0 : Departure city (initial node)

n_6 : Destination city(goal node)

h : Direct distance between a city and the destination city (heuristic)

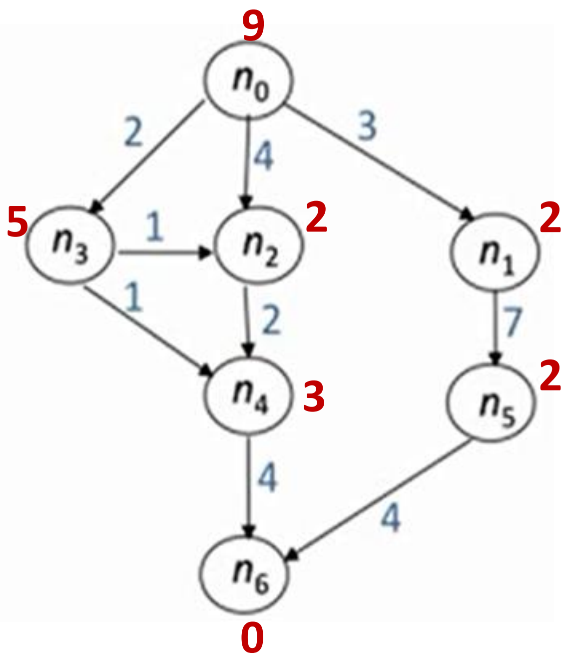
c : Real distance between two cities



A* Search Algorithm

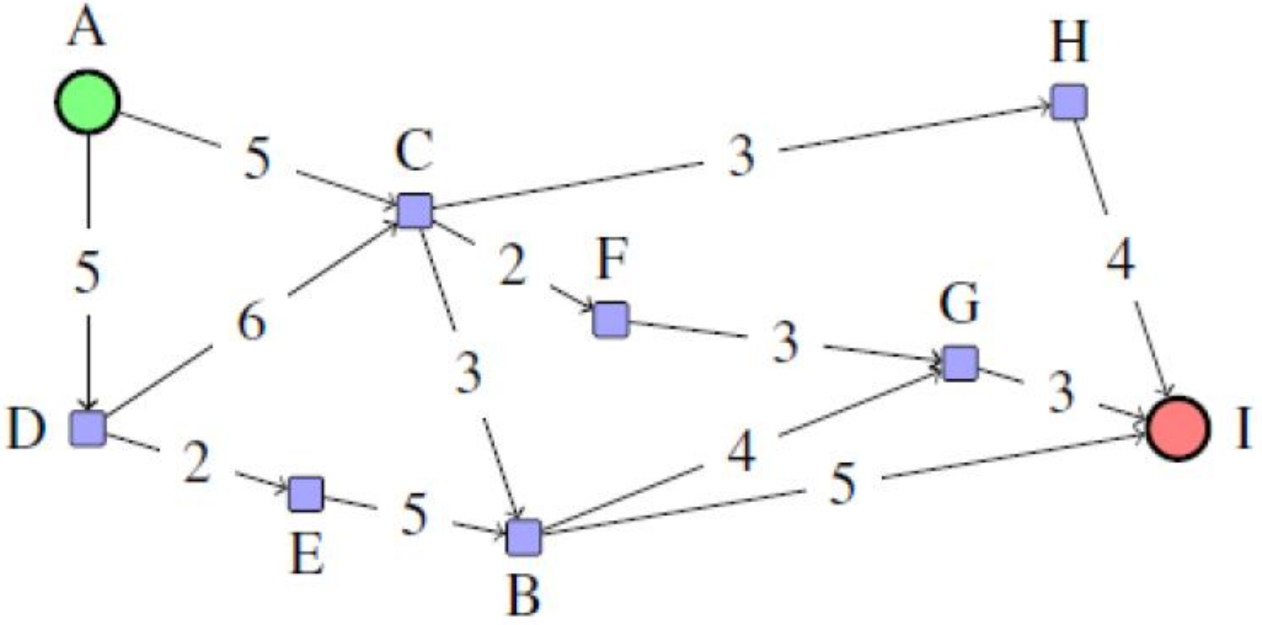
Illustrative example: Path-Finding between two cities

State of Open in each iteration	**State of Closed in each iteration**
(State, f, Parent)	**(State, f, Parent)**
1. (n_0, 9, void)	1. Vide
2. (n_1,5,n_0), (n_2,6,n_0), (n_3,7,n_0)	2. (n_0, 9, void)
3. (n_2,6,n_0), (n_3,7,n_0), (n_5,12,n_1)	3. (n_0, 9, void), (n_1,5,n_0)
4. (n_3,7,n_0), (n_4,9,n_2), (n_5,12,n_1)	4. (n_0, 9, void), (n_1,5,n_0), (n_2,6,n_0)
5. (n_2,5,n_3), (n_4,6,n_3), (n_5,12,n_1)	5. (n_0, 9, void), (n_1,5,n_0), (n_3,7,n_0)
6. (n_4,6,n_3), (n_5,12,n_1)	6. (n_0, 9, void), (n_1,5,n_0), (n_3,7,n_0), (n_2,5,n_3)
7. (n_6,7,n_4), (n_5,12,n_1)	7. (n_0, 9, void), (n_1,5,n_0), (n_3,7,n_0), (n_2,5,n_3), (n_4,6,n_3)
8. Solution : $\boldsymbol{n_0, n_3, n_4, n_6}$	8. (n_0, 9, void), (n_1,5,n_0), (n_3,7,n_0), (n_2,5,n_3), (n_4,6,n_3), (n_6,7,n_4)



Exercise

We consider the following map. The objective is to find the optimal path between A and I. We also give two heuristics h1 and h2:



Node	A	B	C	D	E	F	G	H	I
h1	10	5	5	10	10	3	3	3	0
h2	10	2	8	11	6	2	1	5	0

Find the optimal (we minimize) path using the following algorithms:

1. Greedy Best-First Search using h2 as heuristic function
2. A* Search using h1

Motivations

Reminder of the advantages of A*:

- As input, we have a function (goal(n)) identifying the goal node
- The solution is an **optimal** path and not only a final state
- Visited nodes are all stored to avoid revisiting them

Disadvantages: Memory space is too large (in order to save all the visited nodes)

Characteristics of a local search:

- Definition of an objective function to be optimized (e.g. a goal function which identifies a final node)
- The solution sought is just an optimal node (or close to it) and not the path that leads to the goal
- No need to save all visited states

Example: N-Queens

- **Problem:**
  - Place N queens on a chessboard of size N×N so that two queens do not attack each other (Never position two queens on the same diagonal, row or column)

Objective function
Minimize the number of queens that attack each other

Principle

- **Local search keeps only some visited nodes in the memory:**

- **Hill-Climbing:** a simple case which just keeps a (current) node in memory and iteratively improves it until it converges to a solution.
- **Genetic algorithm:** a more elaborate case which keeps a set of nodes (population) and evolves it until finding a solution

Local search objective

In general, there is an objective function to optimize (minimize or maximize)

- **Hill-Climbing:** the objective function allows to find the next visited node.
- **Genetic algorithm:** the objective function or fitness function is involved in the calculation of all the successor nodes of the current set.

➢ Local search does not guarantee an optimal solution but it has the capacity to find an acceptable solution quickly.

Hill-Climbing

Input:

- Initial node
- Objective function **F(n)** to optimize
- A function that generates successor nodes (neighbors)

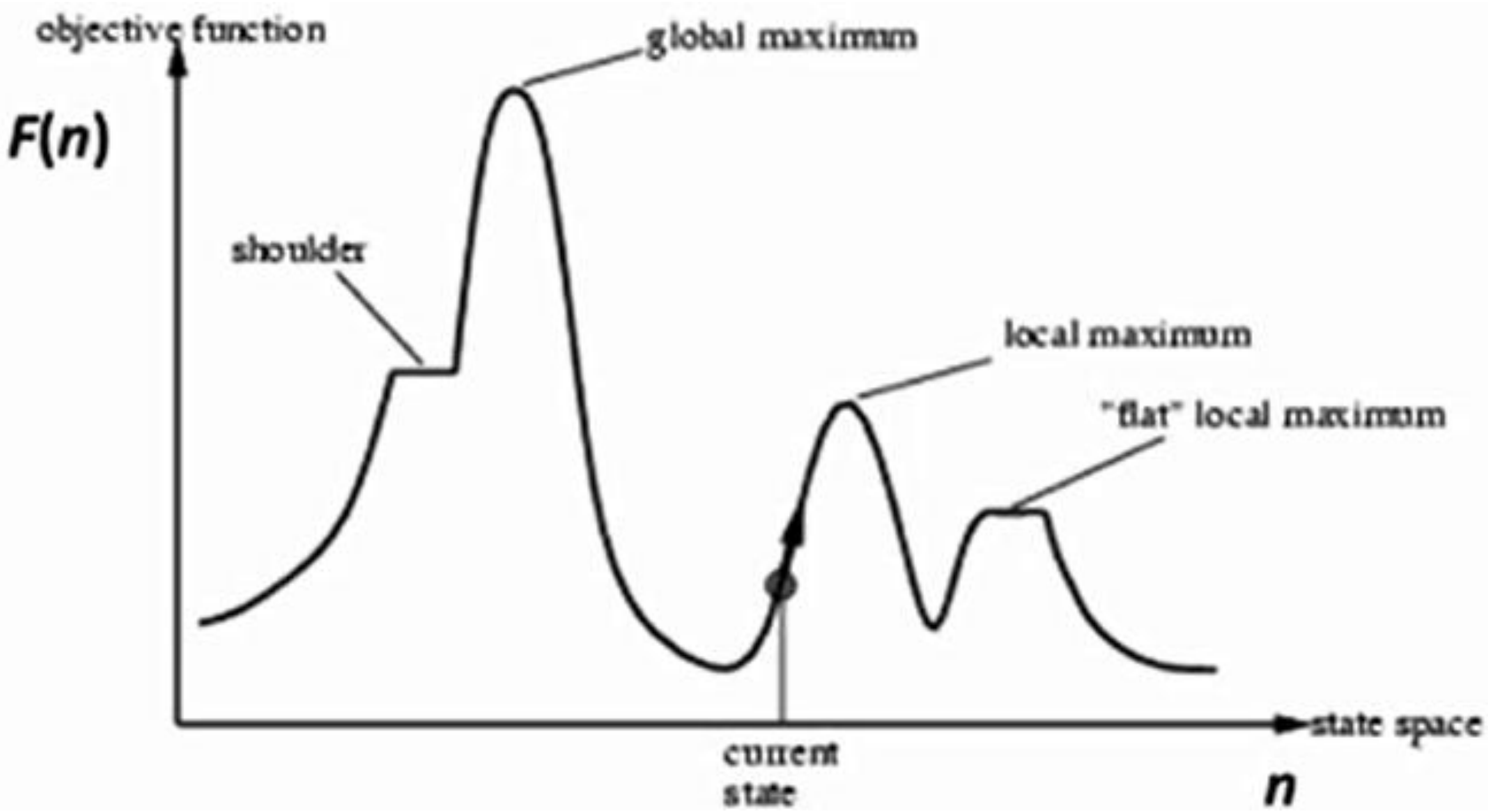
Procedure:

- The current node is initialized to the initial node
- Iterativelly, the current node is compared to its immediates successors (Neighbors):
    - The best neighbor **n'** having the highest value of **F(n')** such as **F(n')>F(n)** will be the current node.
    - If such a neighbor does not exist, we stop and return the current node as a solution.

Hill-Climbing

```
Algorithm HILL-CLIMBING(InitialNode)    //This version maximizes
1. Declare two nodes: n, n’
2. n = initial node
3. While(1):      //The exit criteria will be detrmined in the loop
    1. n’=Successor node of n having the highest value F(n’)
    2. If F(n’)≤F(n)  //If we minimize, the test will be F(n’)≥F(n)
        1. Return n //We couldn’t improve F(n)
    3. Else n = n’ (Go to 3)
```

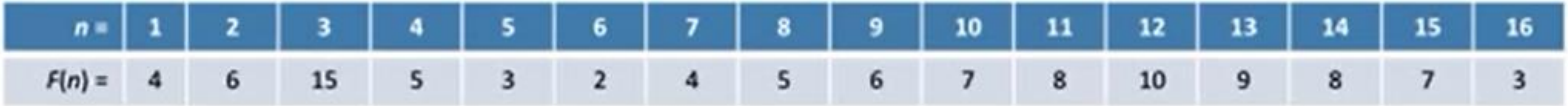
Hill-Climbing: Illustration



Objective: Trying to get to the top of a hill in a foggy environment

Hill-Climbing: Illustration

Consider the following objective function, defined for integers from 1 to 16

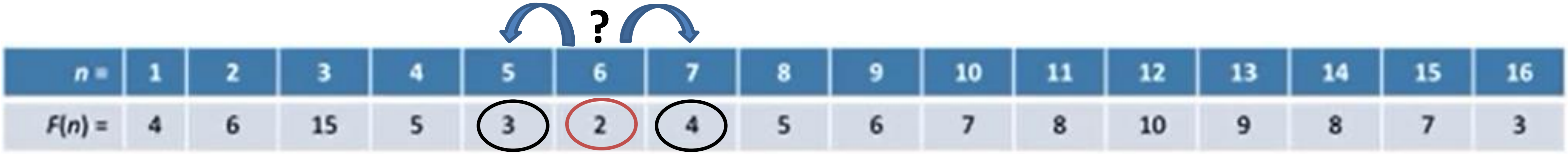


n =	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
F(n) =	4	6	15	5	3	2	4	5	6	7	8	10	9	8	7	3

- What value of **n** Hill-Climbing will find if the initial value of **n** is **6** and the used successors are n-1 (only if n>1) and n+1 (only if n<16)

Hill-Climbing: Illustration

Execution :
Initial node: n = 6



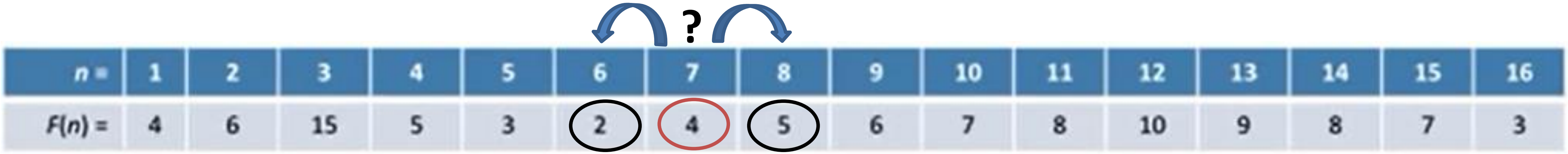
n =	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
F(*n*) =	4	6	15	5	3	2	4	5	6	7	8	10	9	8	7	3

Solution: Browsed values

6 → 7

Hill-Climbing: Illustration

Execution :
Initial node: n = 6



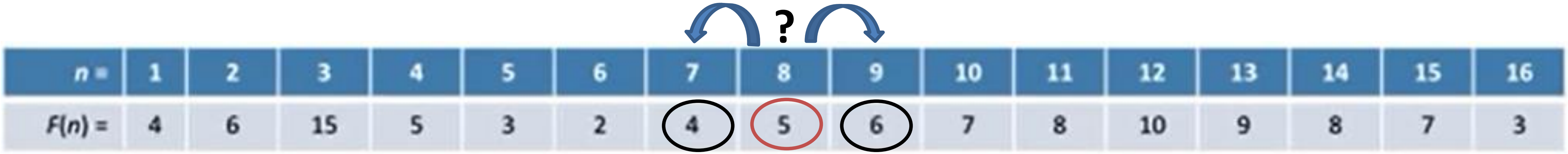
n =	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
F(n) =	4	6	15	5	3	2	4	5	6	7	8	10	9	8	7	3

Solution: Browsed values

6 → 7 → 8

Hill-Climbing: Illustration

Execution :
Initial node: n = 6



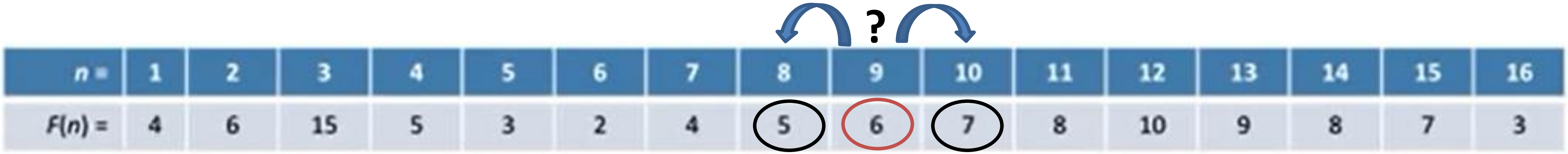
n =	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
F(*n*) =	4	6	15	5	3	2	4	5	6	7	8	10	9	8	7	3

Solution: Browsed values

6 → 7 → 8 → 9

Hill-Climbing: Illustration

Execution :
Initial node: n = 6



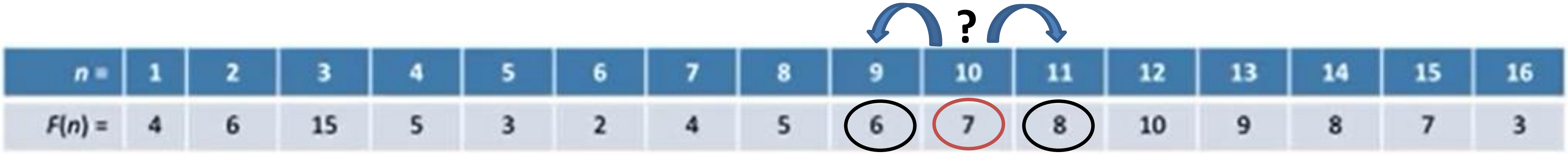
n =	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
F(n) =	4	6	15	5	3	2	4	5	6	7	8	10	9	8	7	3

Solution: Browsed values

6 → 7 → 8 → 9 → 10

Hill-Climbing: Illustration

Execution :
Initial node: n = 6



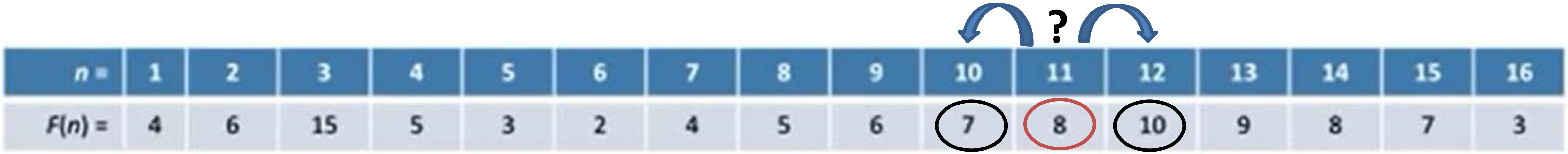
n =	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
F(n) =	4	6	15	5	3	2	4	5	6	7	8	10	9	8	7	3

Solution: Browsed values

6 → 7 → 8 → 9 → 10 → 11

Hill-Climbing: Illustration

Execution :
Initial node: n = 6



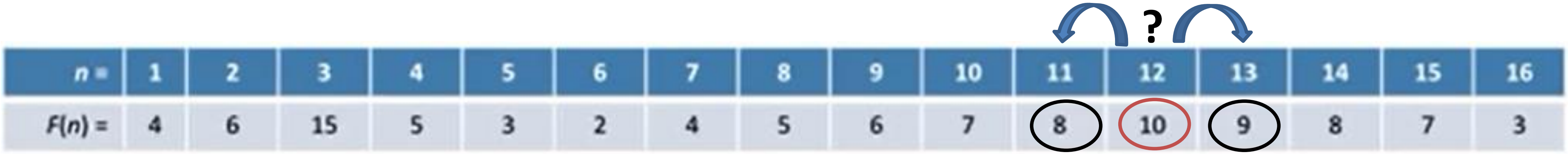
n =	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
F(n) =	4	6	15	5	3	2	4	5	6	7	8	10	9	8	7	3

Solution: Browsed values

6 → 7 → 8 → 9 → 10 → 11 → 12

Hill-Climbing: Illustration

Execution :
Initial node: n = 6



n =	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
F(n) =	4	6	15	5	3	2	4	5	6	7	8	10	9	8	7	3

Solution: Browsed values

6 → 7 → 8 → 9 → 10 → 11 → 12
Hill-Climbing stops and returns **n = 12**

Hill-Climbing: N-Queens

- **n:** Configuration of the chessboard with N queens
- **F(n):** Number of pairs of queens that attack each other directly or indirectly in the current configuration **n**
- We want to **minimize**

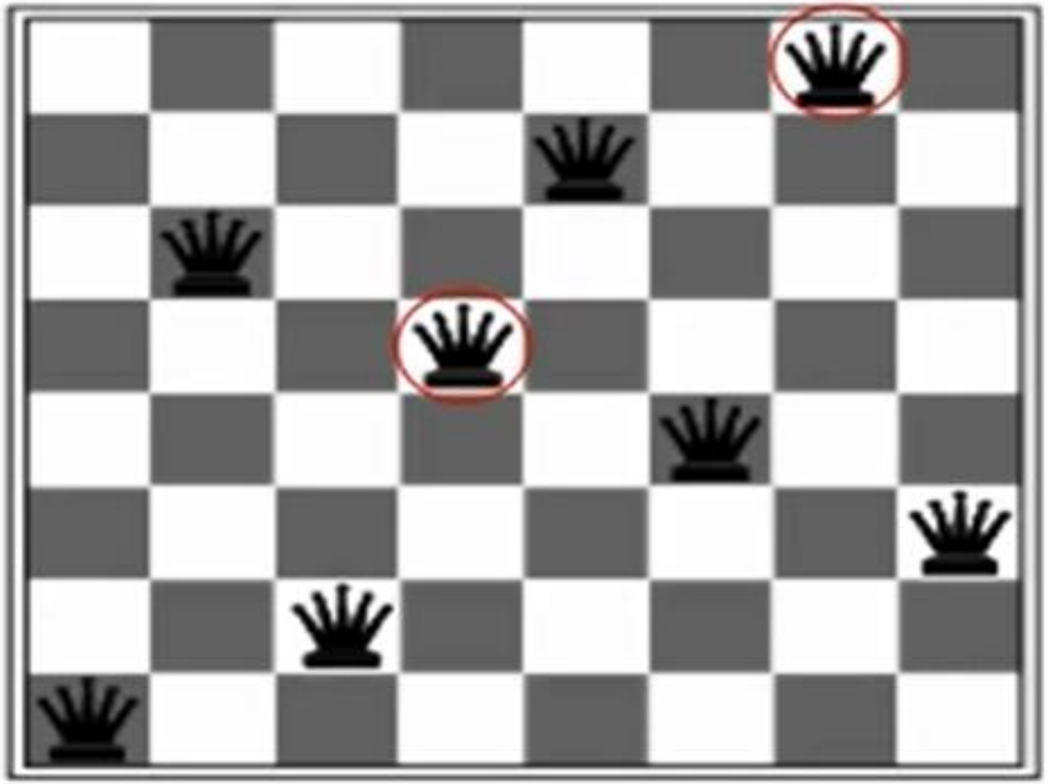
18	12	14	13	13	12	14	14
14	16	13	15	12	14	12	16
14	12	18	13	15	12	14	14
15	14	14	♛	13	16	13	16
♛	14	17	15	♛	14	16	16
17	♛	16	18	15	♛	15	♛
18	14	♛	15	15	14	♛	16
14	14	13	17	12	14	12	18

n=[5 6 7 4 5 6 7 6]

- **F(n)** for the displayed state is: **17**
- **Framed cells** are the best successors, if we move a queen in its column

Hill-Climbing: 8-Queens

- An example of a **local minimum** with **F(n)=1**



Hill-Climbing: 3-Puzzle

Exercise 1:

F(n): Number of misplaced digits

3	1	2
4	5	8
6		7

Initial state

→

	1	2
3	4	5
6	7	8

Goal state

Hill-Climbing: 3-Puzzle

Exercise 2:

F(n): Number of misplaced digits

4	1	2
3		5
6	7	8

Initial state

	1	2
3	4	5
6	7	8

Goal state

Simulated Annealing

- Improved version of Hill-Climbing (minimize the risk of being stuck in a local optimal)
    - Look for a **less good immediate** neighbor of the current node (with certain probability) instead of looking for a better immediate neighbor,
    - The probability of taking a less good neighbor is higher at the beginning then it gradually decreases,
- The number of iterations and the decrease in probabilities are defined using a temperature schedule in descending order.
    - Example: Schedule of 100 iterations $[2^{-0}, 2^{-1}, 2^{-2},.., 2^{-99}]$

Simulated Annealing

```
Algorithm SIMULATED-ANNEALING(InitialNode, Schedule)    //This version maximizes
1. Declare two nodes: n, n’
2. n = initial node
3. For t = 1 .. Size(Schedule):
    1. T = Schedule[t] //Temprature at the instant t
    2. n’ = a successor of n (selected randomly)
    3. $\Delta E = F(n') - F(n)$  //if we minimize, $\Delta E = F(n) - F(n')$
    4. If $\Delta E > 0$ Then assign n = n’   //improvement compared with n
    5. Else assign n = n’ only with a probability of $e^{\Delta E/T}$
4. Return n
```

The smaller **T** is, the smaller probability $e^{\Delta E/T}$ will be.

Local search

Improvement of Simulated Annealing

- Simulated annealing minimizes the risk of being trapped in local optima but it does not eliminate the risk of oscillating indefinitely by returning to a previously visited node.
- Solution1: **Tabu search** Algorithm
  - Save the ***k*** last visited nodes (Tabu set)

- Solution 2: **Beam search** Algorithm
  - Instead of maintaining a single solution node, we could maintain **k** different nodes (Beam):
    - Start with **k** nodes chosen randomly
    - At each iteration, generate all the successors of the **k** chosen nodes
    - Choose the **k** best nodes from the generated nodes and start again

Genetic Algorithms

Origin

- Inspired by the process of natural evolution of species:
- Human intelligence is the result of a process of evolution over millions of years:
  - *Theory of evolution (Darwin)*
  - *Theory of natural selection (Weismann)*
  - *Genetic concepts (Mendel)*
- Simulating evolution doesn't need to last millions of years on a computer

Genetic Algorithms

Principle

- We start with a set of **k** nodes chosen randomly: this set is called **population**.
- A successor is generated by combining two parents.
- A node is represented by a string (**word**) on an alphabet: it is the **genetic code** of a **node**.
- The objective function is called ***fitness function.***
- The next generation is produced by:

(1)Selection → ***(2)Cross-Over*** → ***(3)Mutation***

Genetic Algorithms

Representation

- We represent the solution space of a problem by a population (set of **chromosomes**).
  - A chromosome is a string of characters (**genes**) of fixed size. For example: **101101001**
  - A population generates children by a set of simple procedures that manipulate chromosomes:
    - *Parents Cross-Over*
    - *Mutation of a generated child*
- The children are kept according to their adaptation (**fitness**) determined by an adaptation function ***F(n)***

Genetic Algorithms

```
Algorithm GENETIC-ALGORITHM(k, nb_iterations)    //This version maximizes
1.  Population = set {$n_1$, $n_2$, $n_3$, …, $n_k$} of k chromosomes generated randomly
2.  For t = 1…nb_iterations:
    1.  New_population = {}
    2.  For i = 1… k:
        1.  n = chromosome selected form Population with a higher probability relatively to F(n)
        2.  n' = a different chromosome selected from Population – {n} with the same way as n
        3.  n* = result of Cross-Over between n and n'
        4.  With a small probability, apply a mutatation to n*
        5.  Add n* to New_population
    3.  Population = New_population
3.  Return n in Population with the highest value of F(n)
```

Genetic Algorithms

Example of Cross-Over : 8-Queens

Genetic Algorithms

8-Queens

- Adaptation function : Number of queens that do not attack each other (min=0, max=28).
- Probability of selection of the first chromosome (proportional to the adaptation) :
  - 24/(24+23+20+11) = 31%
  - 23/(24+23+20+11) = 29%
  - 20/(24+23+20+11) = 26%
  - 11/(24+23+20+11) = 14%

Towards adversary search

- Is it possible to use A* in two-player games ?
  - ✓ We could define a state for the game (Chess: position of all pieces in the chessboard)
  - ✓ The goal state is the configuration of the board such that a player wins the game
  - × What would be the transition function ?

- Yes, but not directly (go through intermediate goal states)
  - Multi-agent environment (the opposing player can modify the state of the environment)

Adversary games

Types of games

- Cooperative game
  - All the players want to achieve the same goal
- Adversary game
  - The players are competing
  - A win for some is a defeat for the others (or a draw)
  - Special case : Zero-sum game
    - Examples: Chess, Tic-Tac-Toe,..

We assume : - Games with two opponents who take turns
- Zero-sum-game
- Detrministic and fully observable environment

MiniMax Algorithm

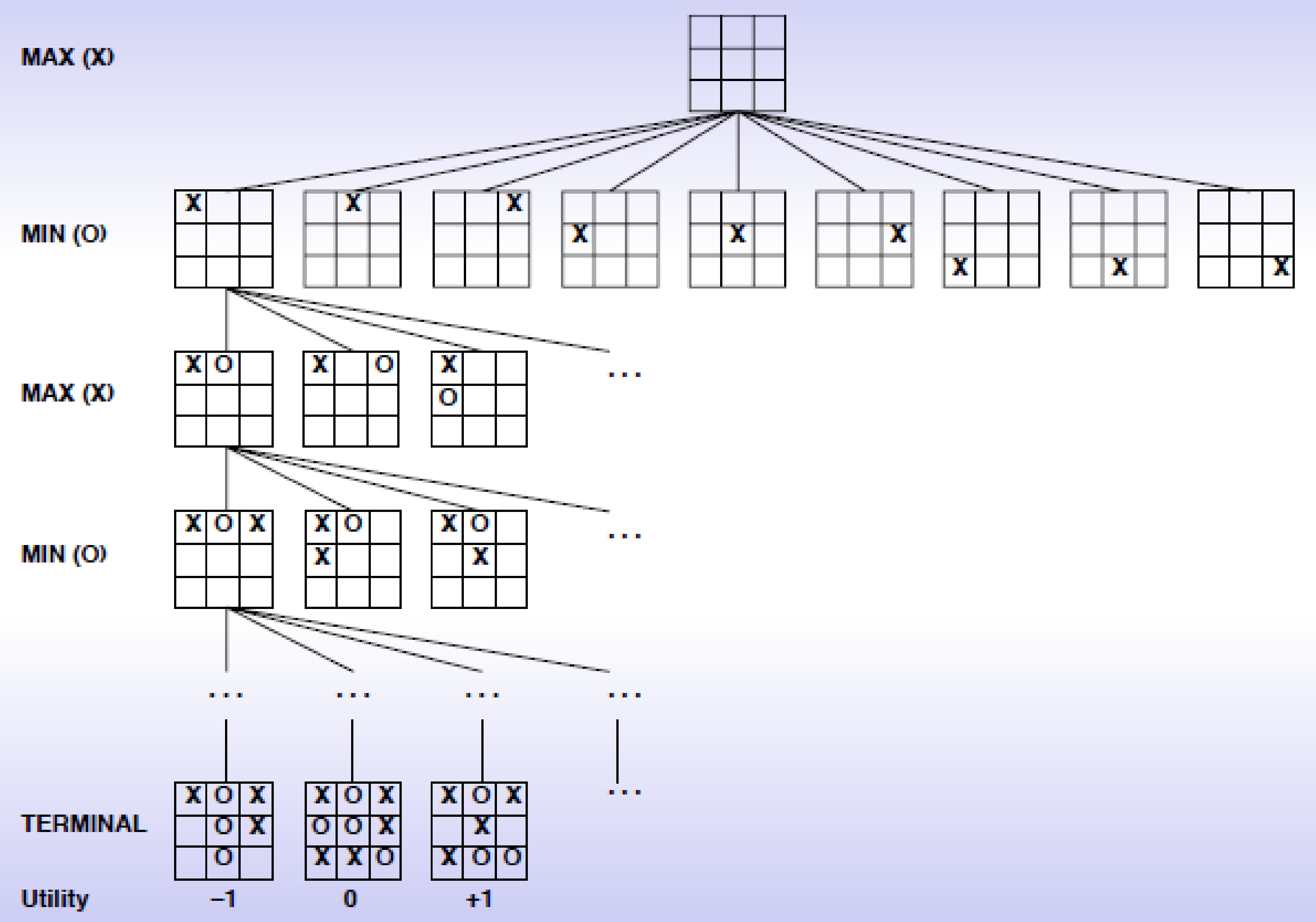
- Two players : Max vs Min
- Max is the first to play
- We consider the result of a game as a reward distributed to the player Max
    - Max tries to maximize the reward
    - Min tries to minimize the Max’s reward

MiniMax Algorithm

- The problem to solve is seen as a **tree-search** problem
    - An **Initial node** (initial configuration of the game)
    - A **transition function** that retourns pairs **(action, successor nodes)**
    - A **termination test** (indicates if the game is over)
    - **Utility function** for the final states (Reward received by Max)

Two-player games

Tic-Tac-Toe Search tree



MiniMax Algorithm

- We assume that the most profitable action for Max or Min is taken (obtain the **greatest MiniMax value**)

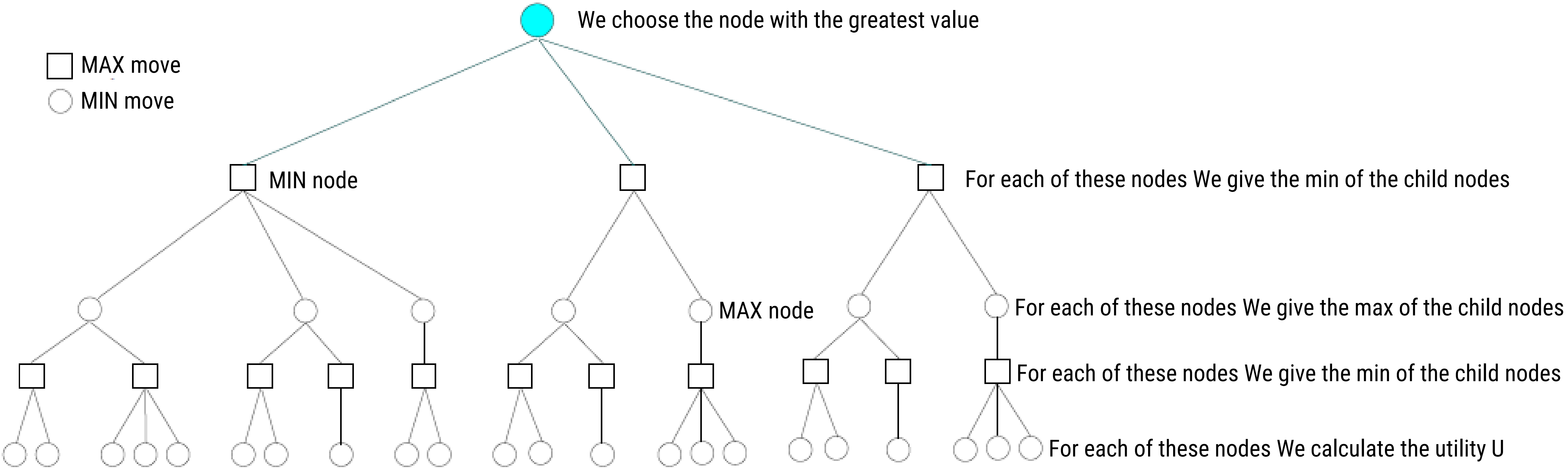
$$\boldsymbol{MINIMAX\_VALUE(n)} = \begin{cases} \boldsymbol{UTILITY(n)} & if\ n\ is\ a\ Terminal\ node \\ \boldsymbol{max}_{n'\,successor\ of\ n}\ \boldsymbol{MINIMAX\_VALUE(n')} & if\ n\ is\ a\ Max\ node \\ \boldsymbol{min}_{n'\,successor\ of\ n}\ \boldsymbol{MINIMAX\_VALUE(n')} & if\ n\ is\ a\ Min\ node \end{cases}$$

- The calculation of the minimax values for all the nodes of the search tree is done using a recursive program

MiniMax Algorithm

```
Algorithm MiniMax(initial node)
-   Return the action chosen by TURN-MAX(initial node)
TURN-MAX(n){
1.   If n corresponds to an end of game Then return the utility value UTILITY(n)
2.   U=-∞ ,  a=void
3.   For Each pair(a', n') given by TRANSITION(n):
       -  If the utilility of TURN-MIN(n') > u Then assign a=a' , u=utility of TURN-MIN(n')
          Else Return the utility u and the action a }
TURN-MIN(n){
1.   If n corresponds to an end of game Then return the utility value UTILITY(n)
2.   U=+∞ ,  a=void
3.   For Each pair(a', n') given by TRANSITION(n):
       -  If the utility of TURN-MAX(n') < u Then assign a=a' , u=utility of TURN-MAX(n')
          Else Return the utility u and the action a}
```

MiniMax Algorithm / Game Tree



We choose the node with the greatest value

☐ MAX move

○ MIN move

MIN node

For each of these nodes We give the min of the child nodes

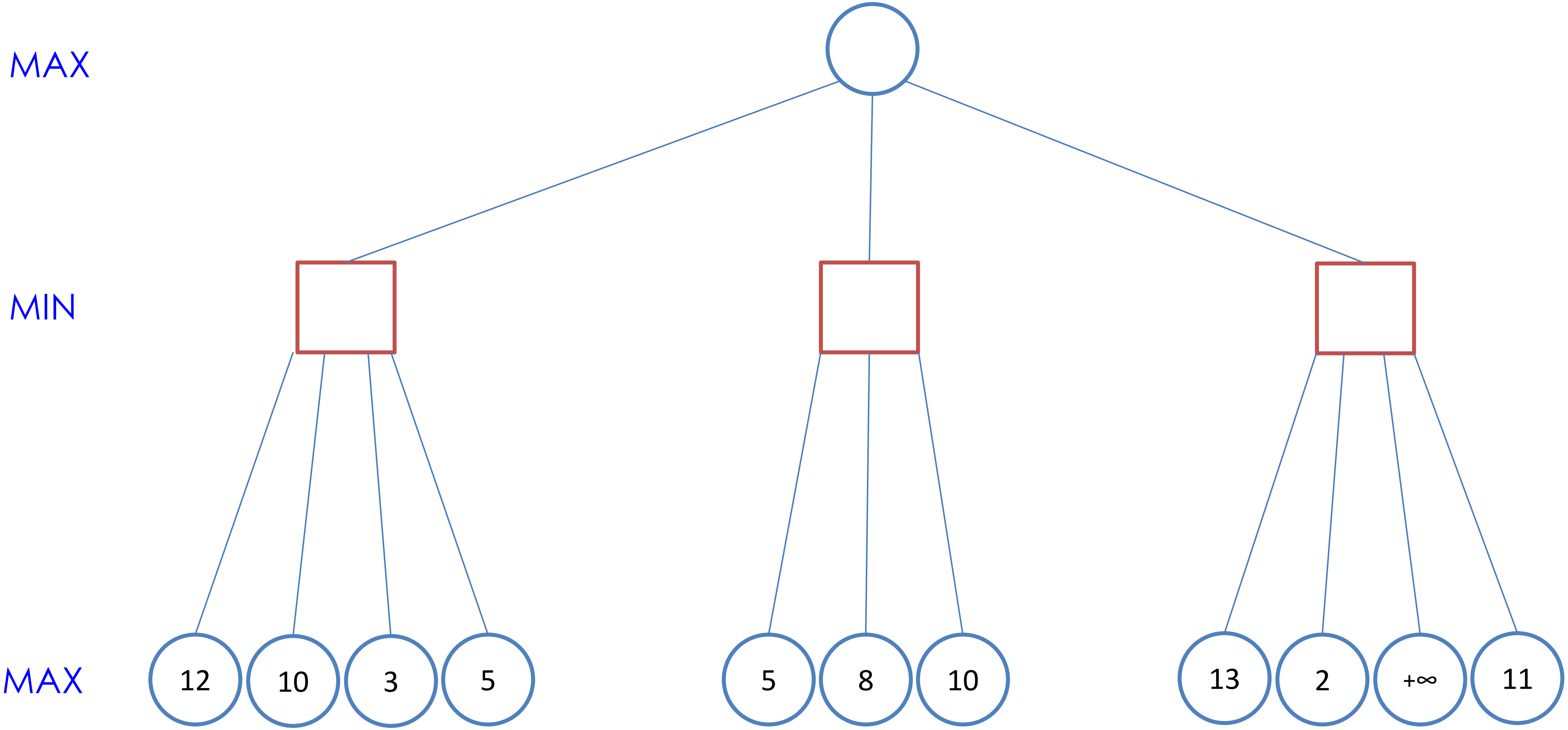
MAX node

For each of these nodes We give the max of the child nodes

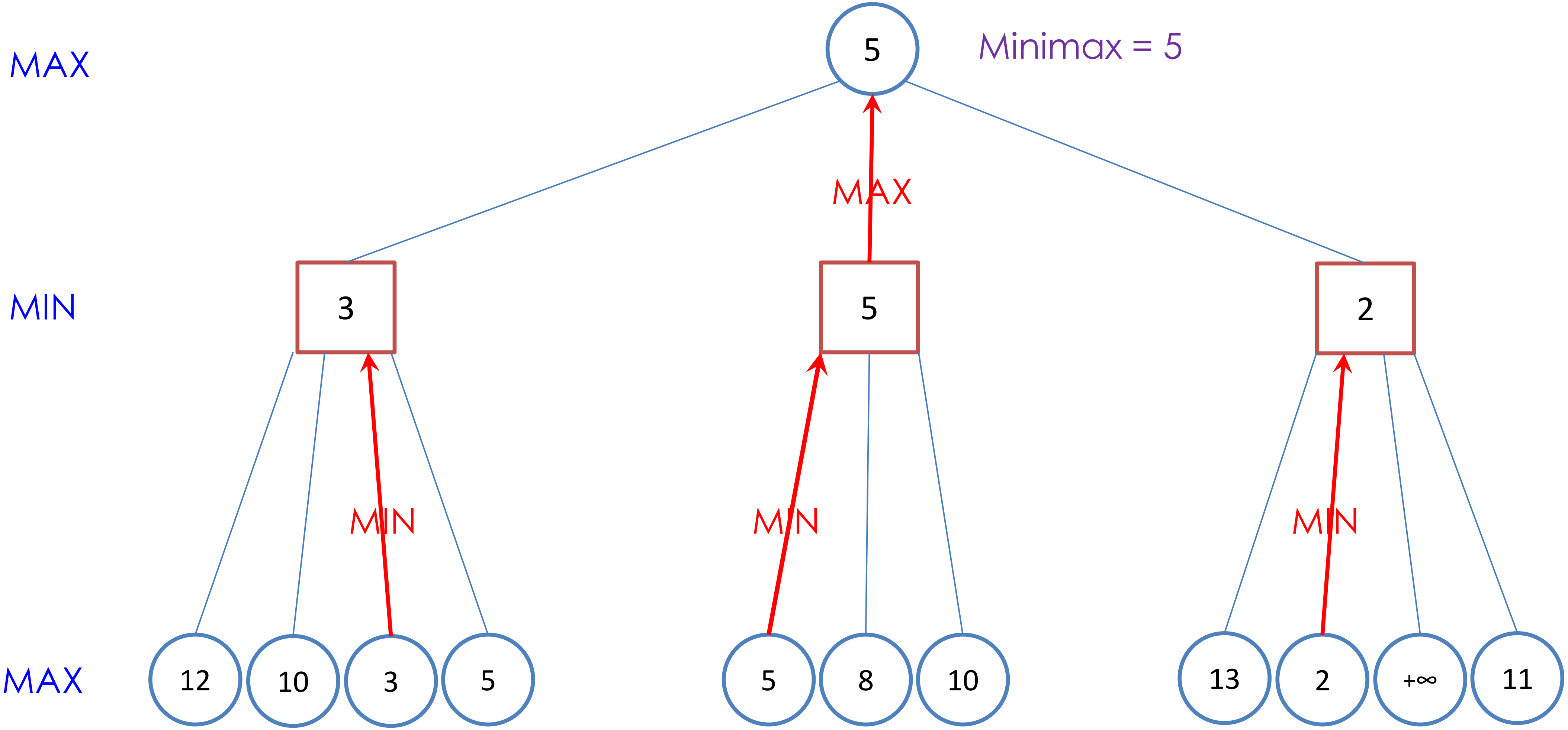
For each of these nodes We give the min of the child nodes

For each of these nodes We calculate the utility U

MiniMax Algorithm / Illustration



MiniMax Algorithm / Illustration



MiniMax Algorithm / Complexity

- Time complexity = O(b^m)

- b : maximum number of choices by move (choices or actions) in each step (branching factor)
- m : maximum number of moves in a game (number of levels in a DFS search)

- Space complexity = O(bm)

Chess game:

Number of choices per move: 35 ($b \approx 35$)

Average number of moves for each player: 50 ($m \approx 100$)

Alpha-Beta Pruning

- MINIMAX : Evaluation of positions after generation of the tree (Expand all leaf nodes to a limiting depth)

- Idea : Evaluation to the leaf nodes and propagation to the ancestors when the tree is generated (Alpha-Beta Pruning)):
  - A leaf node is evaluated once produced.
  - Identify the paths (in the tree) which are explored unnecessarily (know if a leaf node is uninteresting)

Two-player games

Alpha-Beta Cut-Off

```
Algorithm Alpha-Beta-Pruning(initial node)
-   Return the action chosen by TURN-MAX(initial node, $-\infty$ , $+\infty$ )
TURN-MAX(n,$\alpha, \beta$) {
1.  If n corresponds to an end of game Then return the utility value UTILITY(n)
2.  U=$-\infty$ ,  a=void
3.  For Each pair(a', n') given by TRANSITION(n)
      -  If the utilility of TURN-MIN(n', $\alpha, \beta$) > U Then assign a=a' , U=utility of TURN-MIN(n' , $\alpha, \beta$)
         If U $\geq \beta$ Return the utility U and the action a
         Else  $\alpha$ = Max ($\alpha$ , U)
4.  Return the utility U and the action a,
    }
TURN-MIN(n,$\alpha, \beta$) {
1.  If n corresponds to an end of game Then return the utility value UTILITY(n)
2.  U=$+\infty$ ,  a=void
3.  For each pair(a', n') given by TRANSITION(n)
      -  If the utility of TURN-MAX(n', $\alpha, \beta$) < U Then assign a=a' , U=utility of TURN-MAX(n' , $\alpha, \beta$)
         If U $\leq \alpha$ Return the utility U and the action a
         Else  $\beta$ = Min ($\beta$ , U)
4.  Return the utility U and the action a,
    }
```

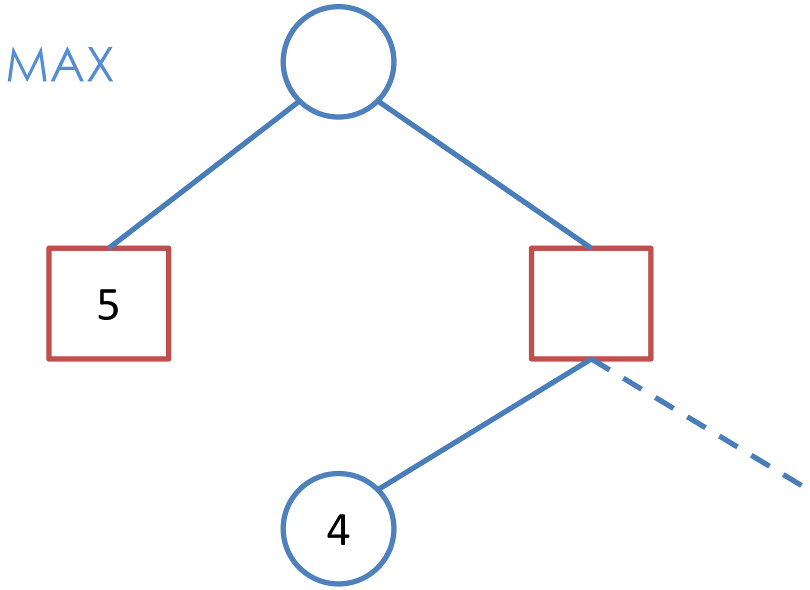
Two-player games

Alpha-Beta Cut-Off

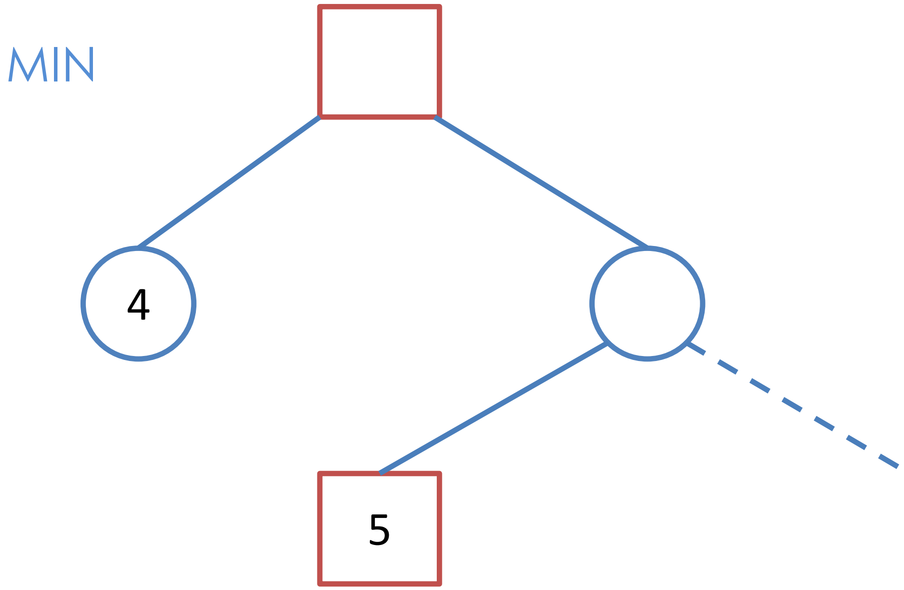
Cut-Off Principle:

- We add the parameters $\alpha \; and \; \beta$ (initially $-\infty \; and +\infty$)
- The cut nodes (pruned) are those such that $u(n) \in [\alpha, \beta] \; and \; \alpha \geq \beta$
- The uncut nodes are those such that:

$$[\boldsymbol{\alpha}, \boldsymbol{\beta}] = \begin{cases} [-\infty, +\infty] & \boldsymbol{or} \\ [-\infty, \boldsymbol{b}] \quad with \; \boldsymbol{b} \neq +\infty & \boldsymbol{or} \\ [\boldsymbol{a}, +\infty] \quad with \; \boldsymbol{a} \neq -\infty & \end{cases}$$



Coupure α



Coupure β

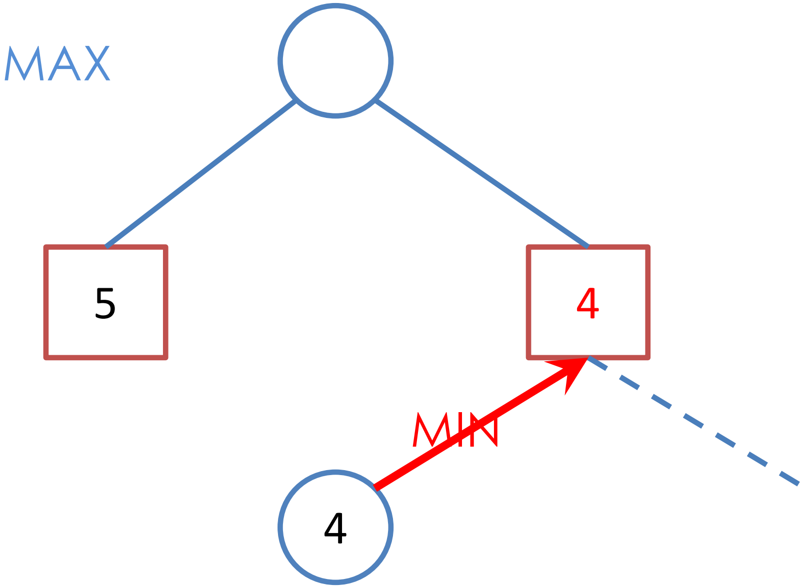
Two-player games

Alpha-Beta Cut-Off

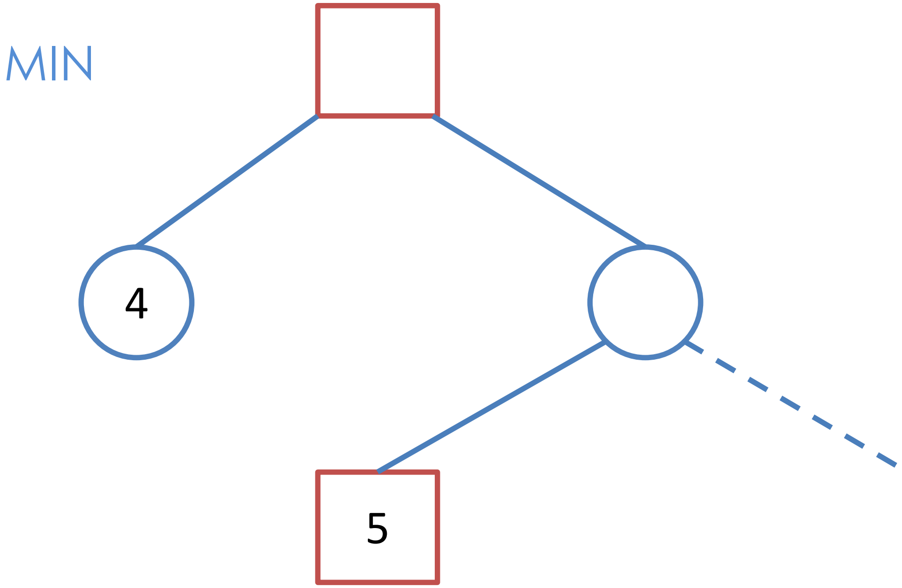
Cut-Off Principle:

- We add the parameters $\alpha\ and\ \beta$ (initially $-\infty\ and + \infty$)
- The cut nodes (pruned) are those such that $u(n) \in [\alpha, \beta]\ and\ \alpha \geq \beta$
- The uncut nodes are those such that:

$$[\boldsymbol{\alpha}, \boldsymbol{\beta}] = \begin{cases} [-\infty, +\infty] & \boldsymbol{or} \\ [-\infty, \boldsymbol{b}] \quad with\ \boldsymbol{b}\ \neq\ +\infty & \boldsymbol{or} \\ [\boldsymbol{a}, +\infty] \qquad with\ \boldsymbol{a}\ \neq -\infty & \end{cases}$$



Coupure α



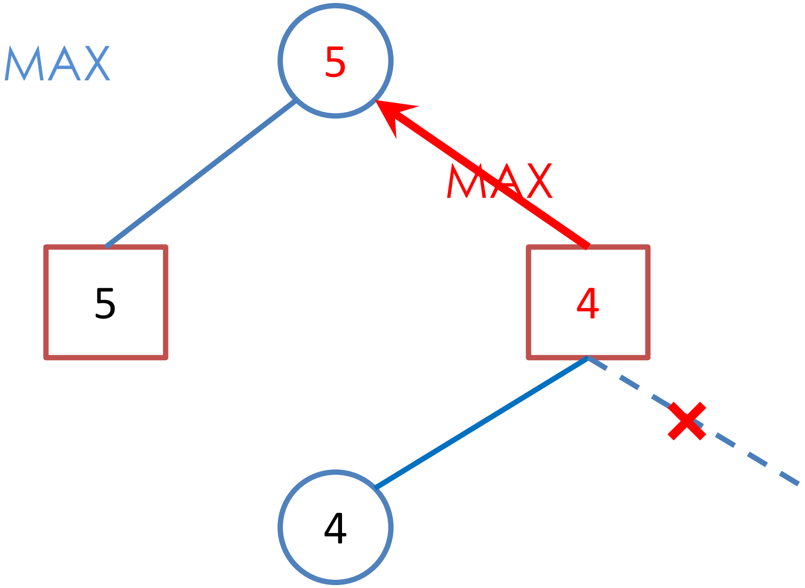
Coupure β

Alpha-Beta Cut-Off

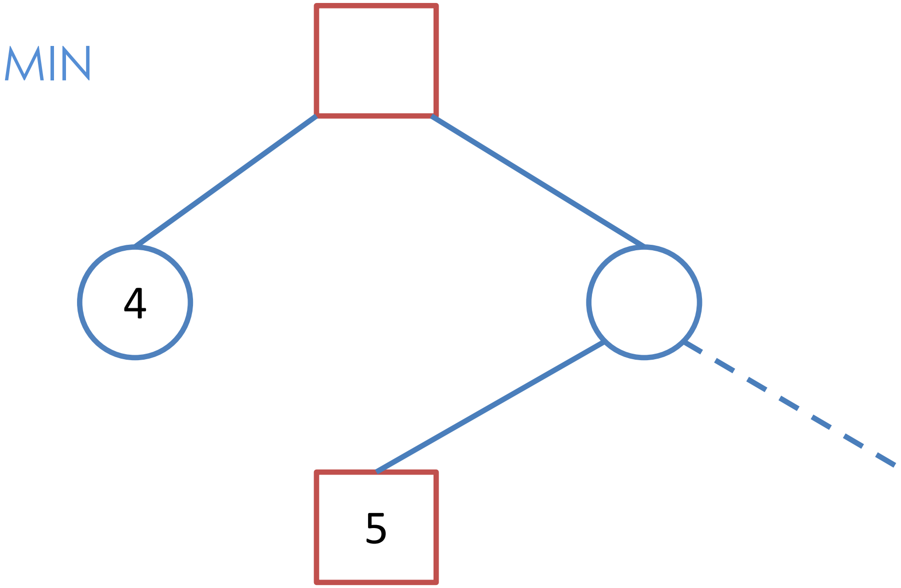
Cut-Off Principle:

- We add the parameters α and β (initially $-\infty$ and $+\infty$)
- The cut nodes (pruned) are those such that $u(n) \in [\alpha, \beta]$ and $\alpha \geq \beta$
- The uncut nodes are those such that:

$$[\boldsymbol{\alpha}, \boldsymbol{\beta}] = \begin{cases} [-\infty, +\infty] & \textbf{or} \\ [-\infty, \boldsymbol{b}] \quad with\ \boldsymbol{b} \neq +\infty & \textbf{or} \\ [\boldsymbol{a}, +\infty] \quad with\ \boldsymbol{a} \neq -\infty & \end{cases}$$



Coupure α



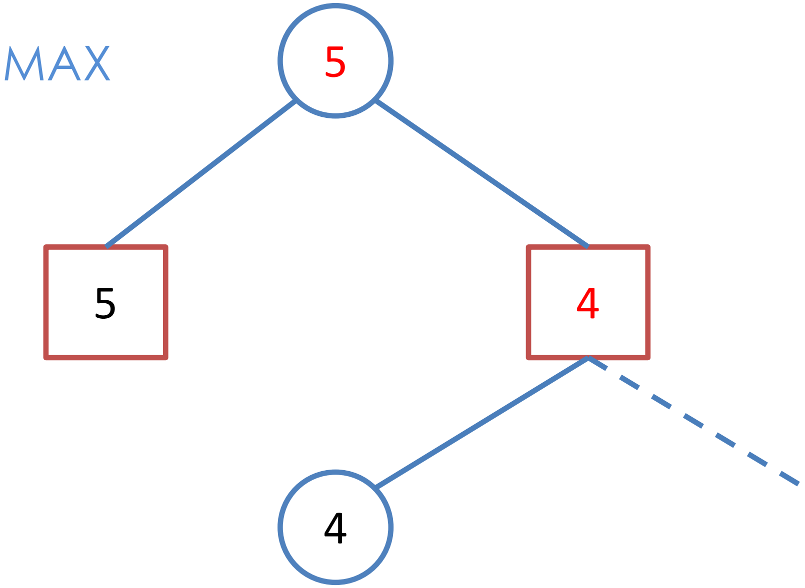
Coupure β

Alpha-Beta Cut-Off

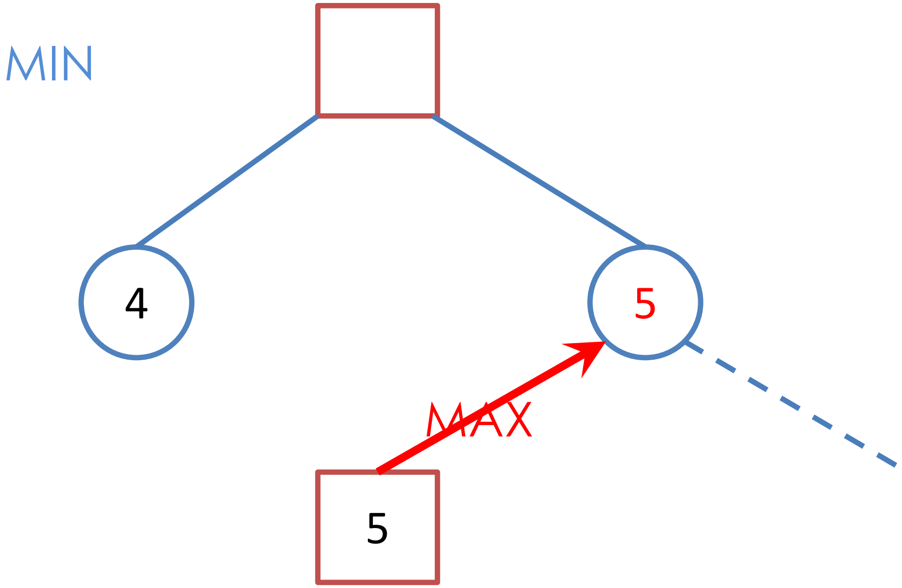
Cut-Off Principle:

- We add the parameters $\alpha \text{ and } \beta$ (initially $-\infty \text{ and } +\infty$)
- The cut nodes (pruned) are those such that $u(n) \in [\alpha, \beta] \text{ and } \alpha \geq \beta$
- The uncut nodes are those such that:

$$[\boldsymbol{\alpha}, \boldsymbol{\beta}] = \begin{cases} [-\infty, +\infty] & \boldsymbol{or} \\ [-\infty, \boldsymbol{b}] \quad with\ \boldsymbol{b} \neq +\infty & \boldsymbol{or} \\ [\boldsymbol{a}, +\infty] \quad with\ \boldsymbol{a} \neq -\infty & \end{cases}$$



Coupure α



Coupure β

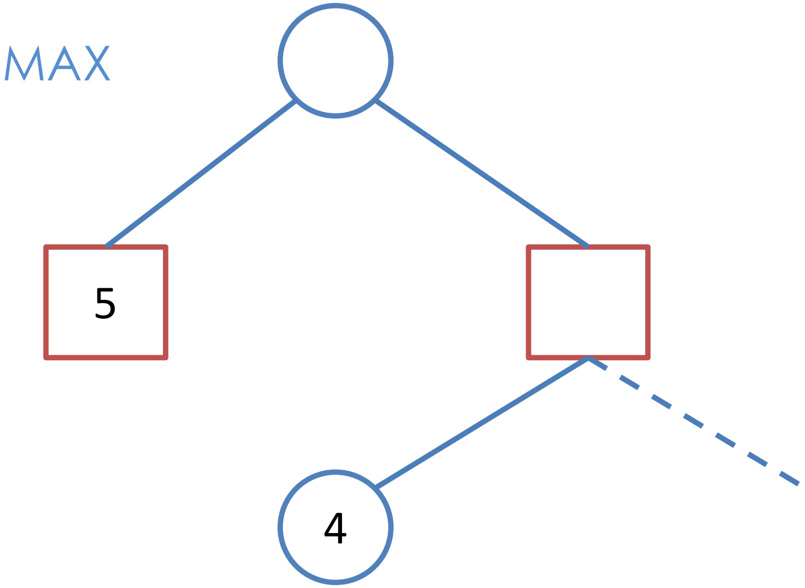
Two-player games

Alpha-Beta Cut-Off

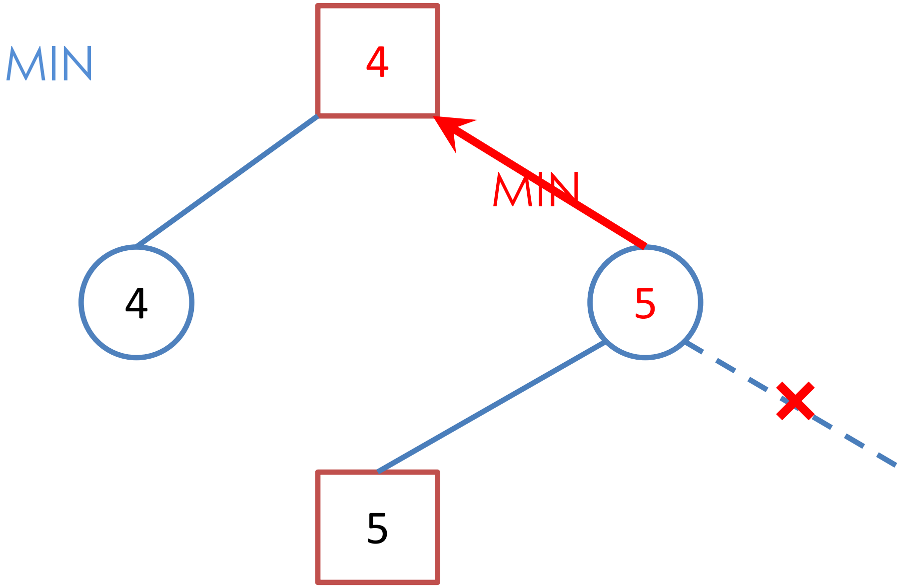
Cut-Off Principle:

- We add the parameters α *and* β (initially $-\infty$ *and* $+\infty$)
- The cut nodes (pruned) are those such that $u(n) \in [\alpha, \beta]$ *and* $\alpha \geq \beta$
- The uncut nodes are those such that:

$$[\boldsymbol{\alpha}, \boldsymbol{\beta}] = \begin{cases} [-\infty, +\infty] & \boldsymbol{or} \\ [-\infty, \boldsymbol{b}] \quad with\ \boldsymbol{b} \neq +\infty & \boldsymbol{or} \\ [\boldsymbol{a}, +\infty] \quad with\ \boldsymbol{a} \neq -\infty & \end{cases}$$



Coupure α

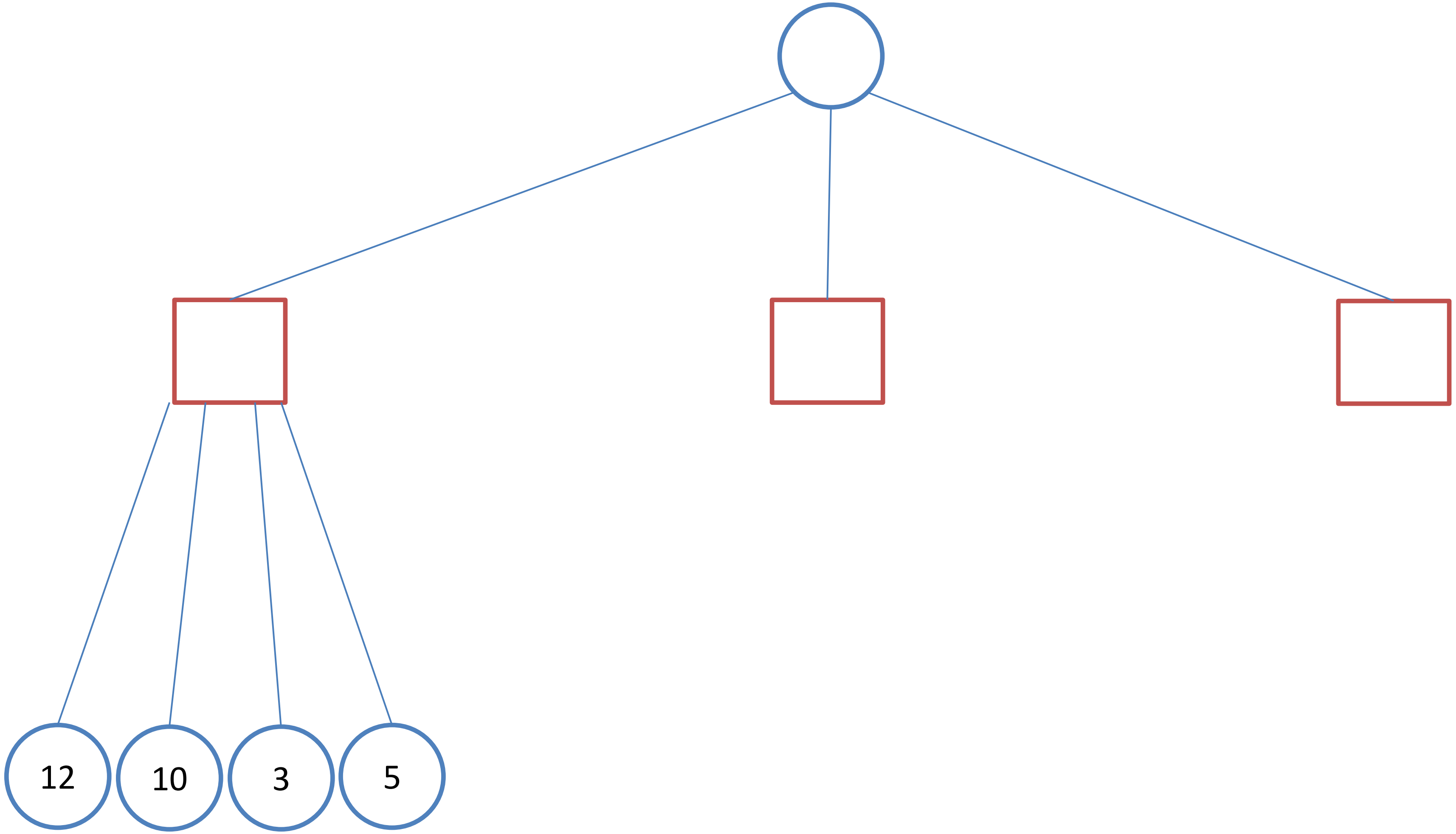


Coupure β

Two-player games

Alpha-Beta Cut-Off

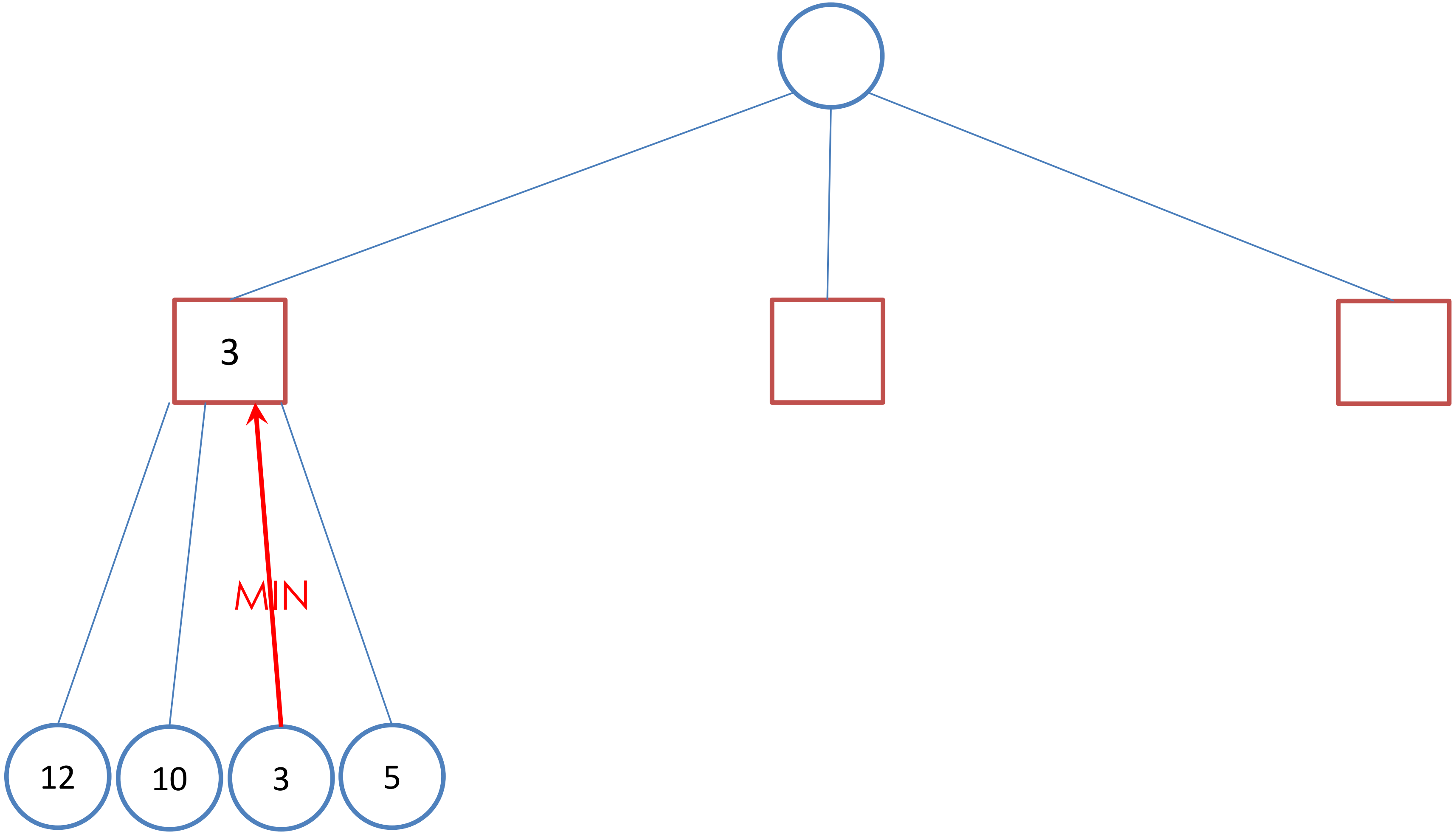
$[\alpha, \beta] = -\infty, +\infty$



Two-player games

Alpha-Beta Cut-Off

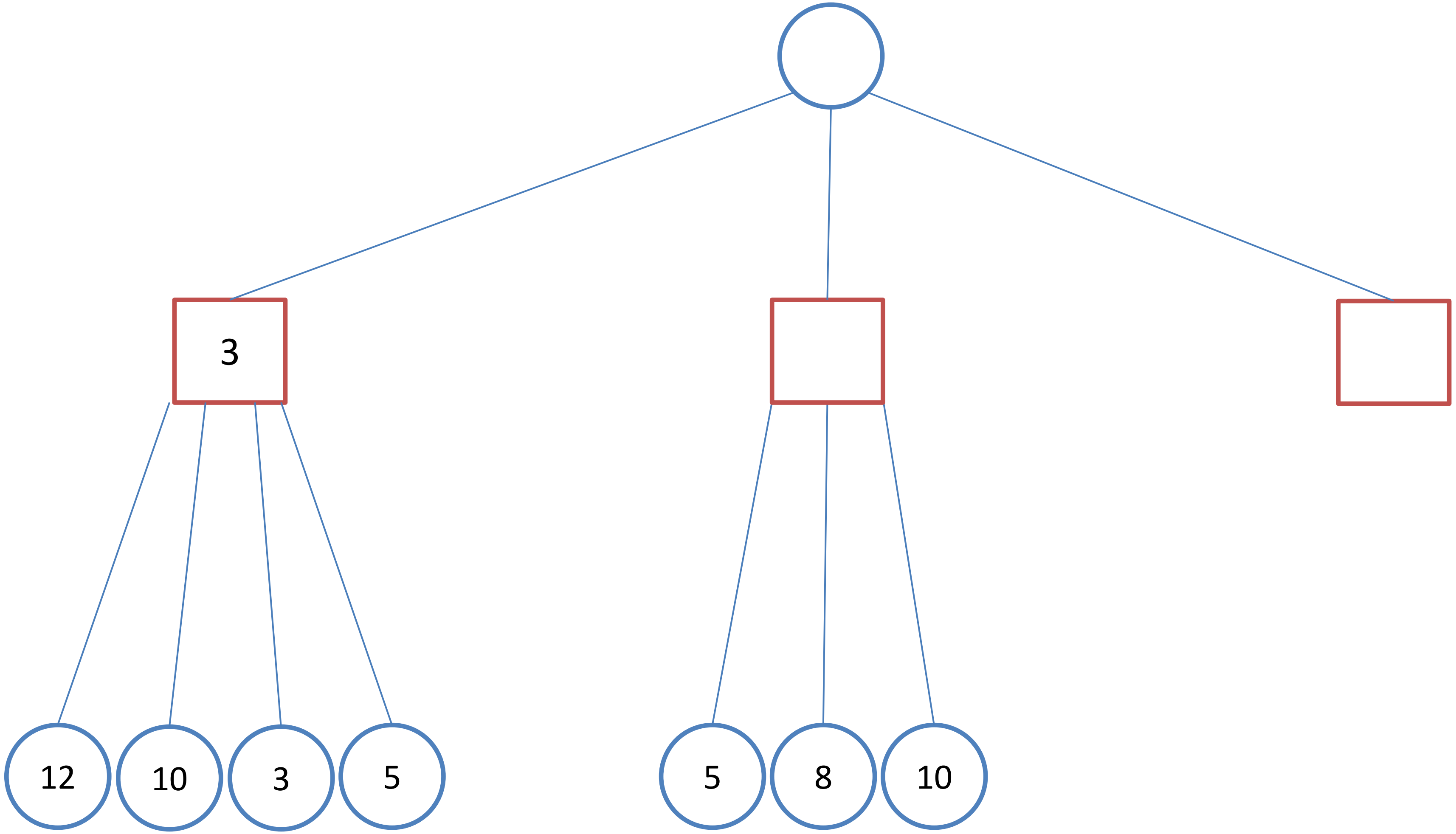
$[\alpha, \beta] = [3, +\infty]$



Two-player games

Alpha-Beta Cut-Off

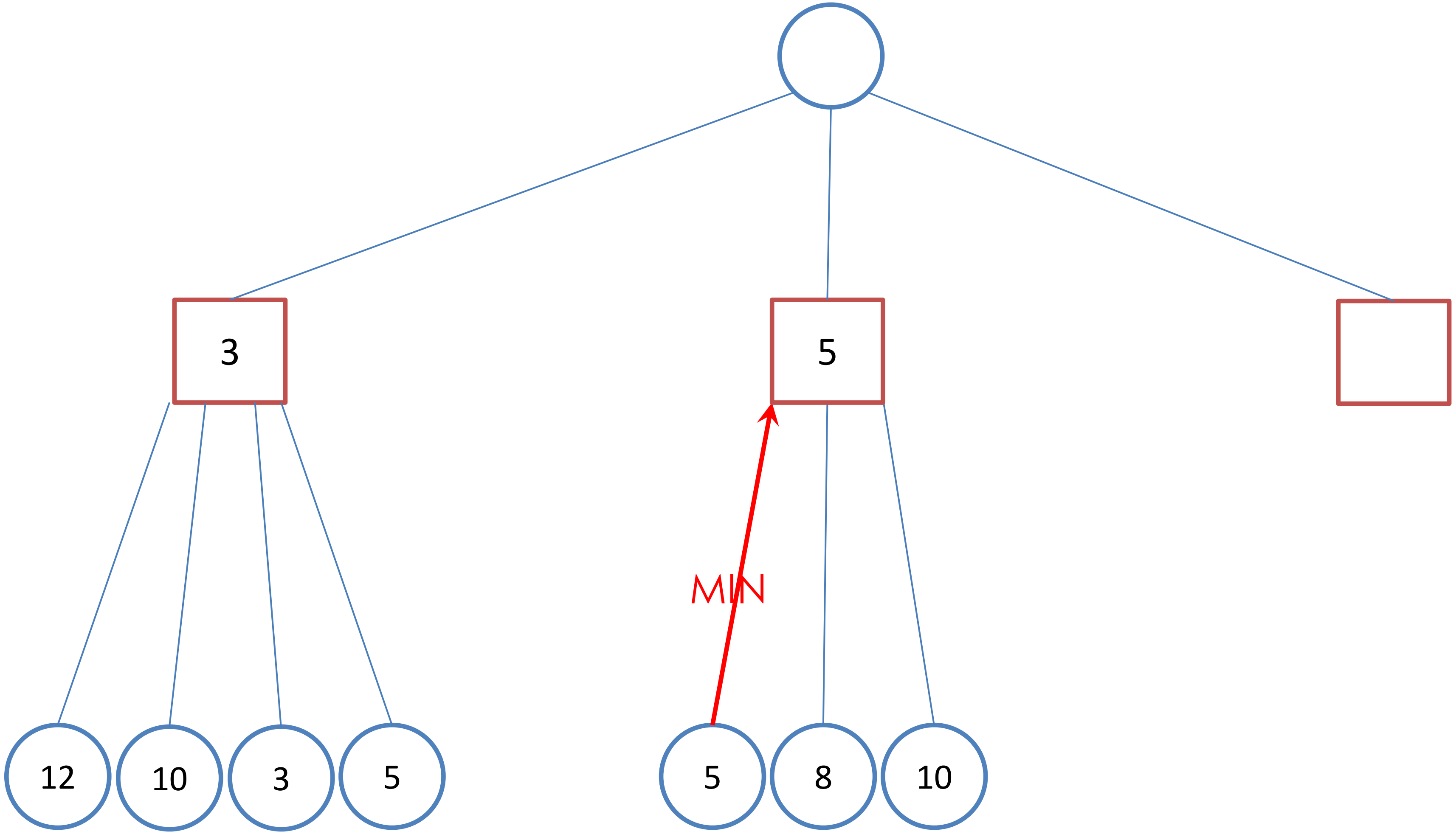
$[\alpha, \beta] = [3, +\infty]$



Two-player games

Alpha-Beta Cut-Off

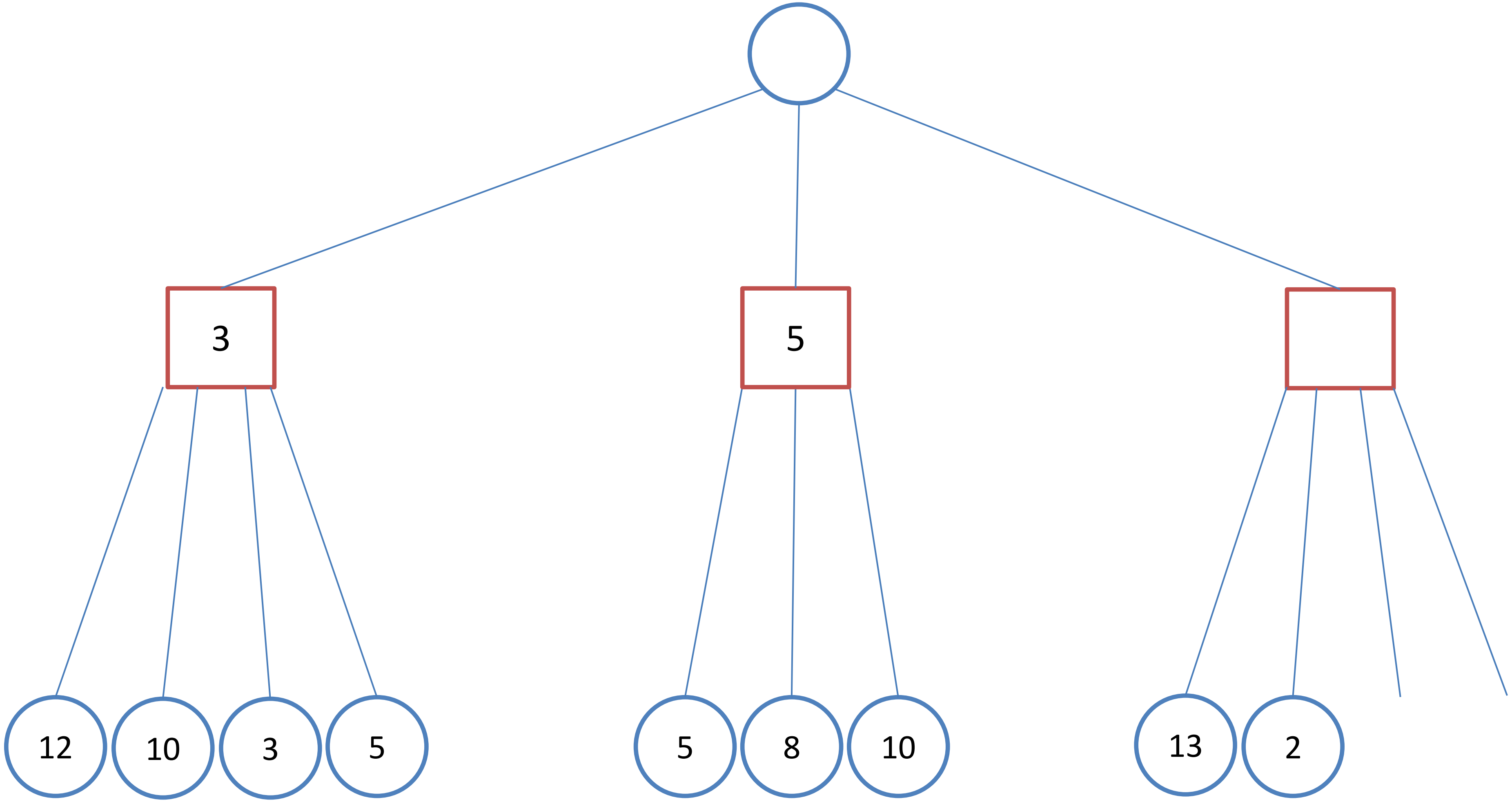
$[\alpha, \beta] = 5, +\infty$



Two-player games

Alpha-Beta Cut-Off

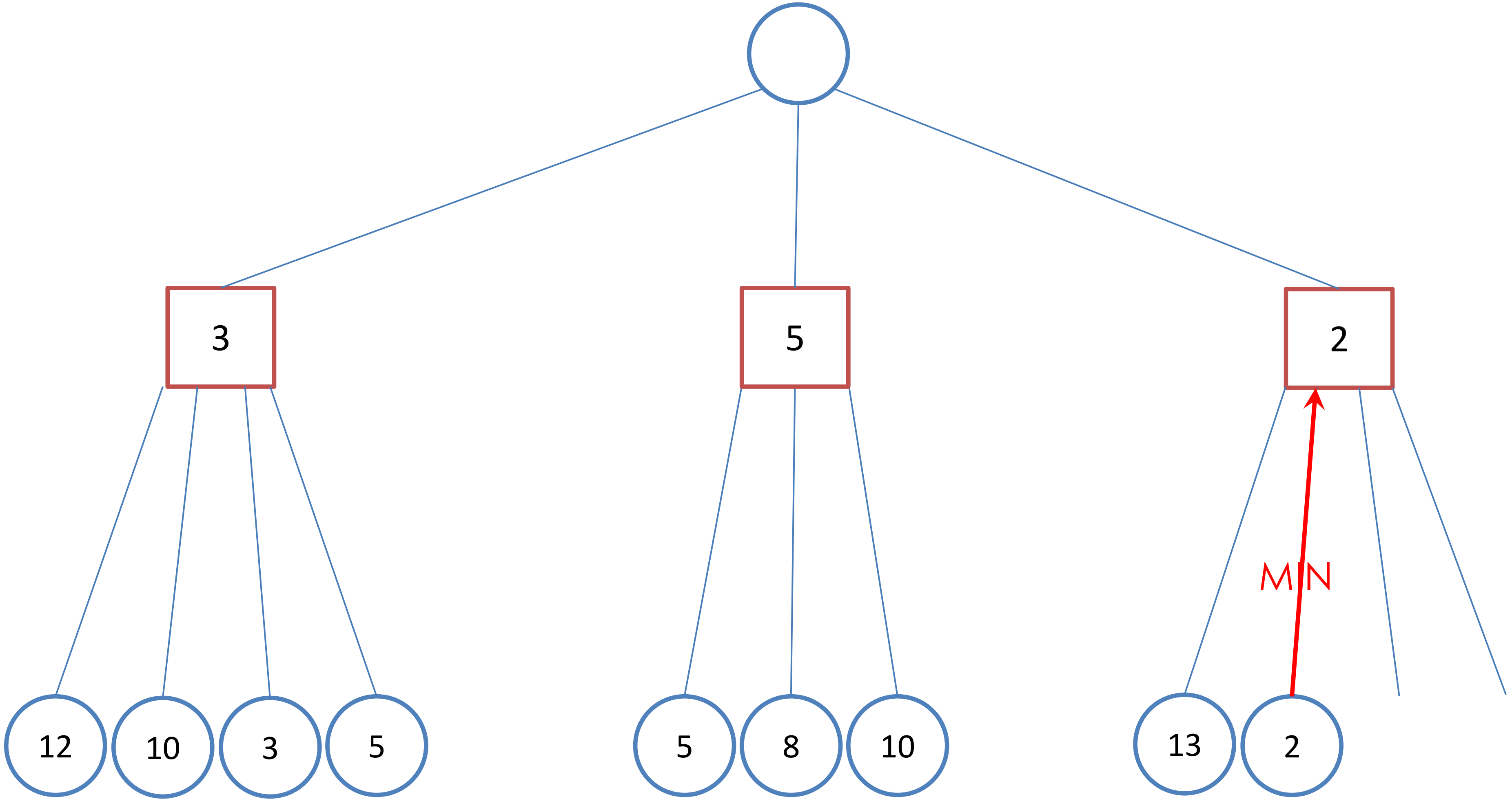
$[\alpha, \beta] = 5, +\infty$



Two-player games

Alpha-Beta Cut-Off

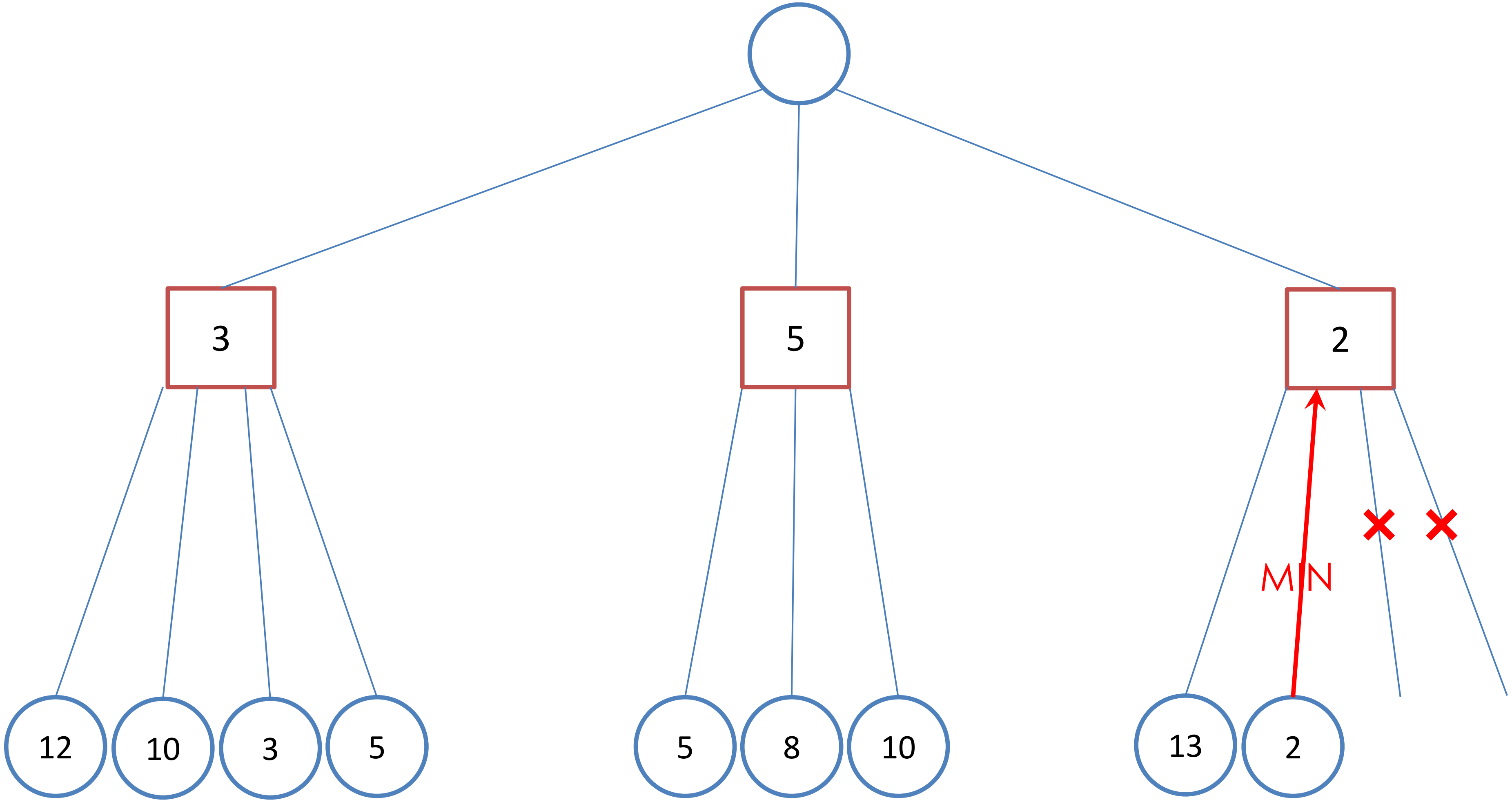
$[\alpha, \beta] = 5, +\infty$



Two-player games

Alpha-Beta Cut-Off

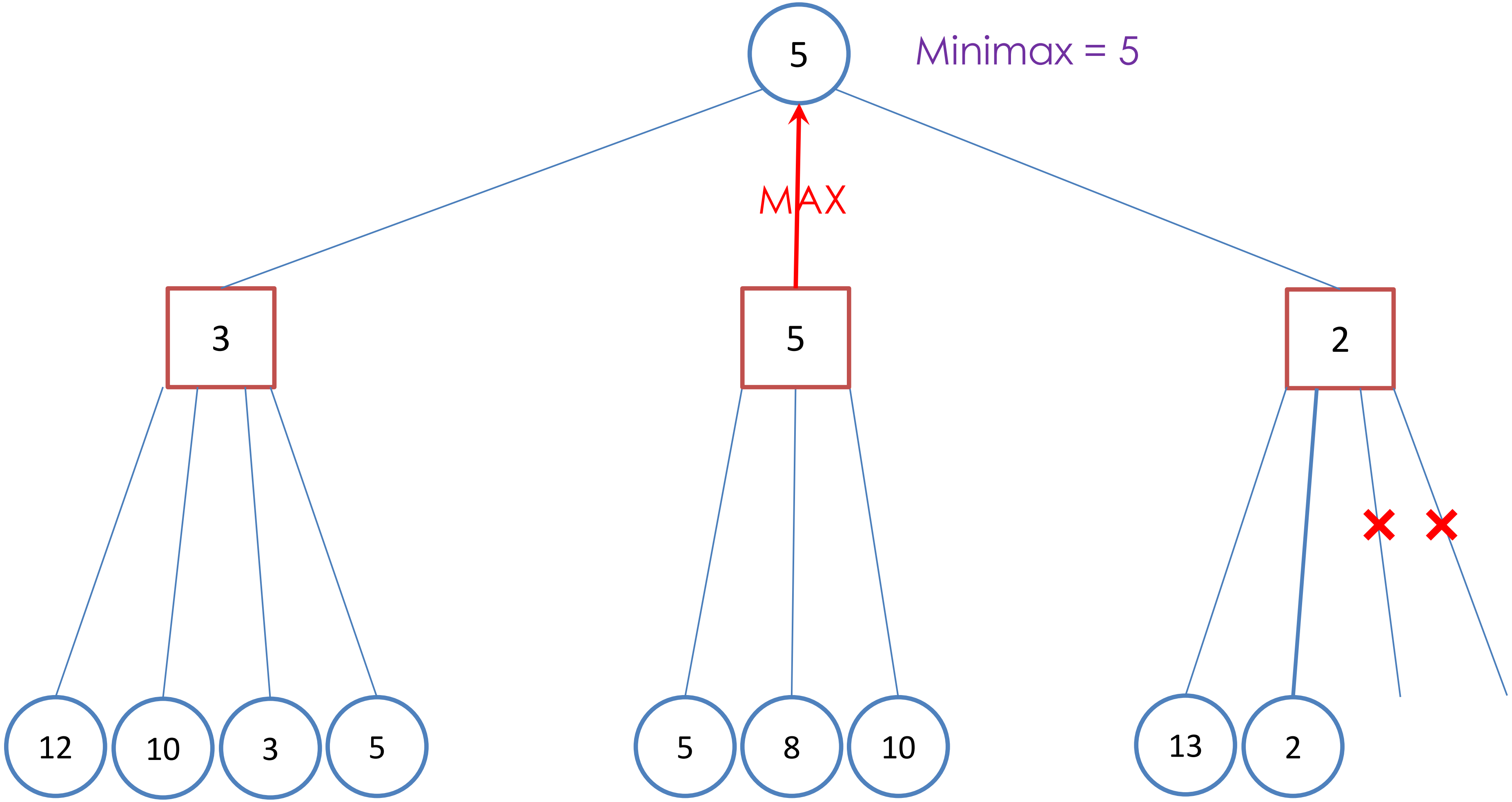
$[\alpha, \beta] = 5, +\infty$



Two-player games

Alpha-Beta Cut-Off

$[\alpha, \beta] = 5, +\infty$



Two-player games

MinMax

Alpha-Beta Cut-Off