CSC 480: Artificial Intelligence

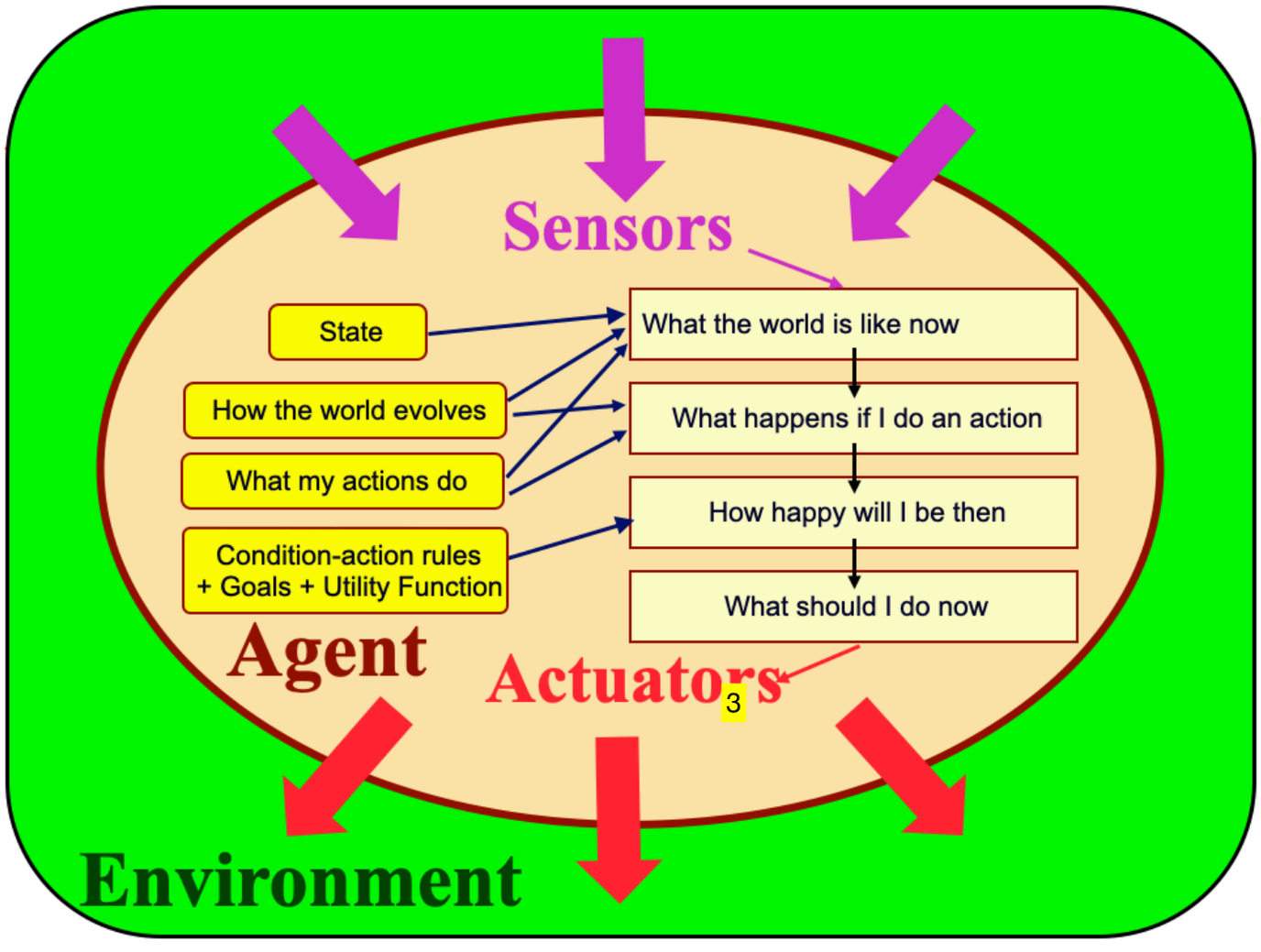
3- Tree Search

Rodrigo Canaan
Assistant Professor
Computer Science Department
Cal Poly, San Luis Obispo
rcanaan@calpoly.edu

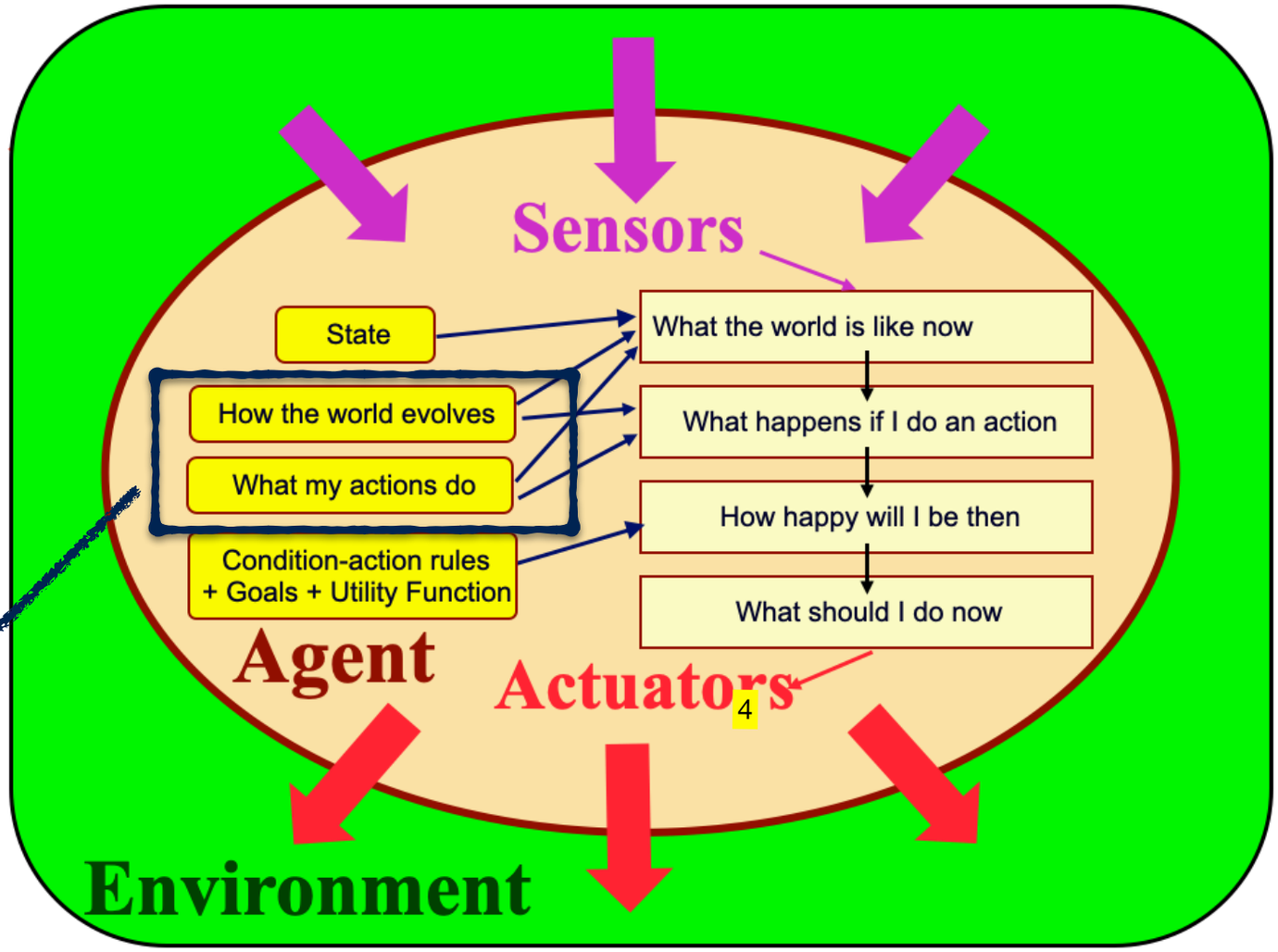
Aug 5, 2024

Tree Search

Recap: Utility-based agents



Recap: Utility-based agents



Forward model

Search

Search

- **Key Insight:** if we have (or can build) a forward model of the environment, we can search for sequences of actions that lead to a goal state (or a state with high utility)

Search

- **Key Insight:** if we have (or can build) a forward model of the environment, we can search for sequences of actions that lead to a goal state (or a state with high utility)
- Forward model: a function that takes a state and an action and (assuming environment is deterministic) outputs the next state

Search

- **Key Insight:** if we have (or can build) a forward model of the environment, we can search for sequences of actions that lead to a goal state (or a state with high utility)

- Forward model: a function that takes a state and an action and (assuming environment is deterministic) outputs the next state

- For this to work, you must be able to simulate states with the forward model **much** faster than your observation-action loop

States

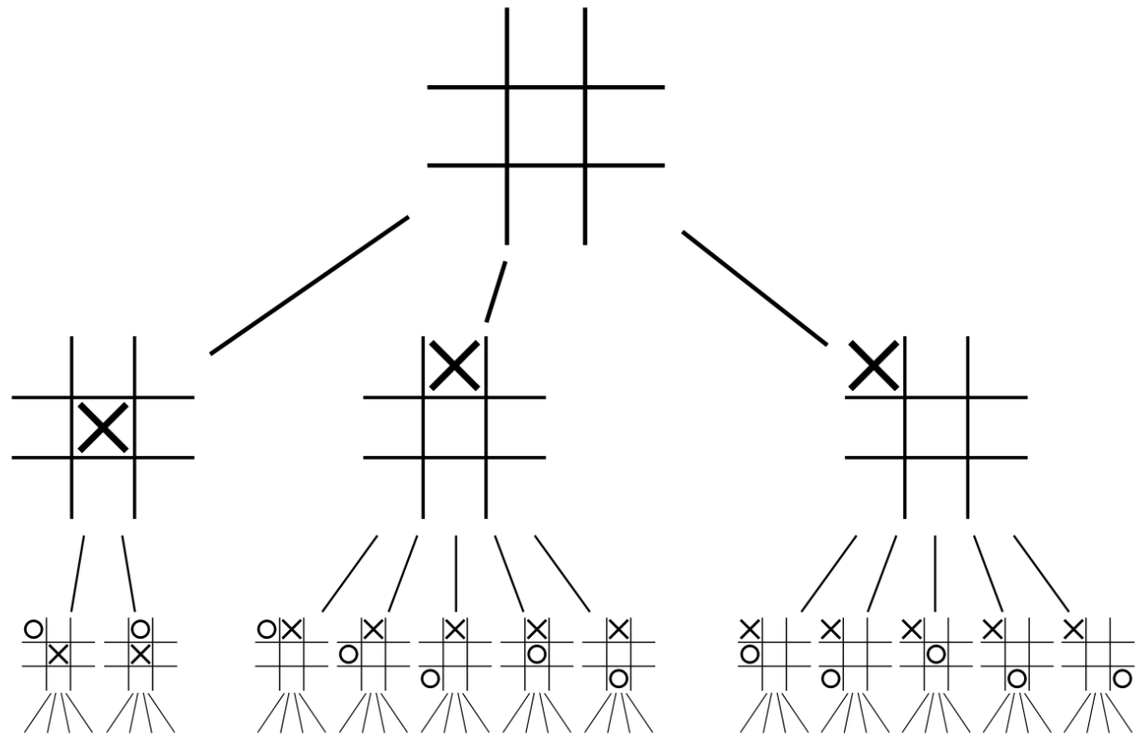
States

- **State:** representation of the current configuration of the environment

States

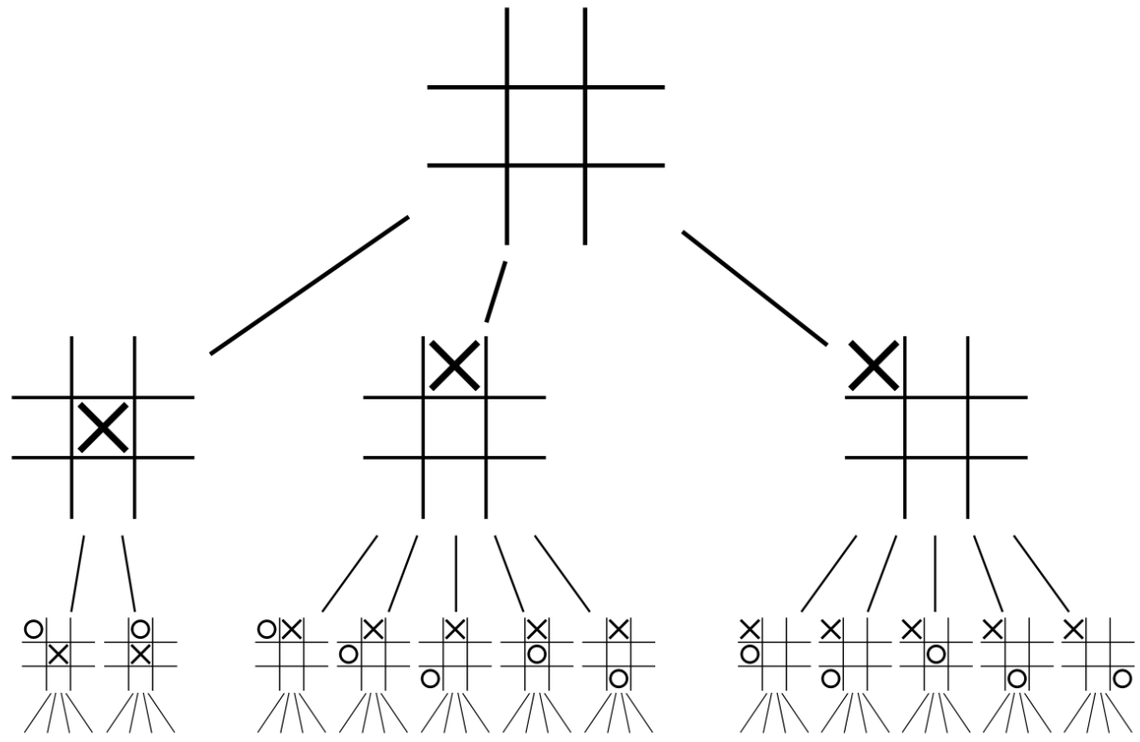
- **State:** representation of the current configuration of the environment
- Can be quite complex!

Simplifying assumptions



Simplifying assumptions

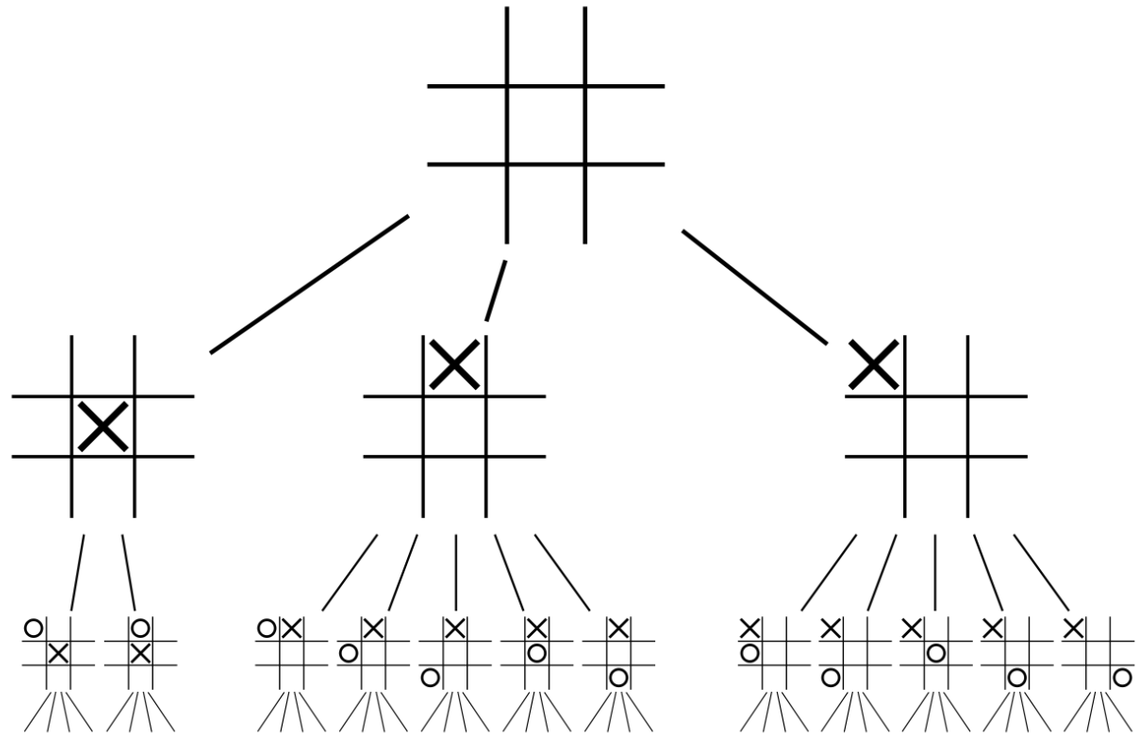
If we can…



Simplifying assumptions

If we can…

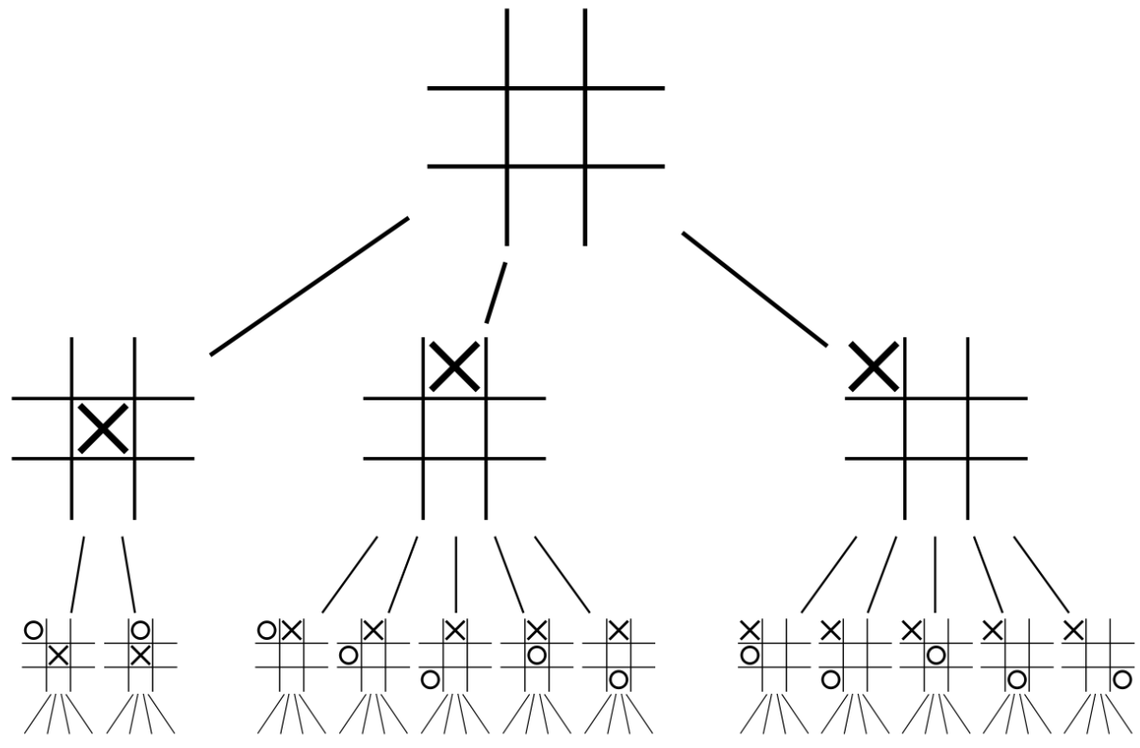
- Represent states as discrete “black box” entities (**atomic representation**) and..



Simplifying assumptions

If we can…

- Represent states as discrete "black box" entities (**atomic representation**) and..
- quickly compute, given a state, which actions lead to which states (**forward model**)



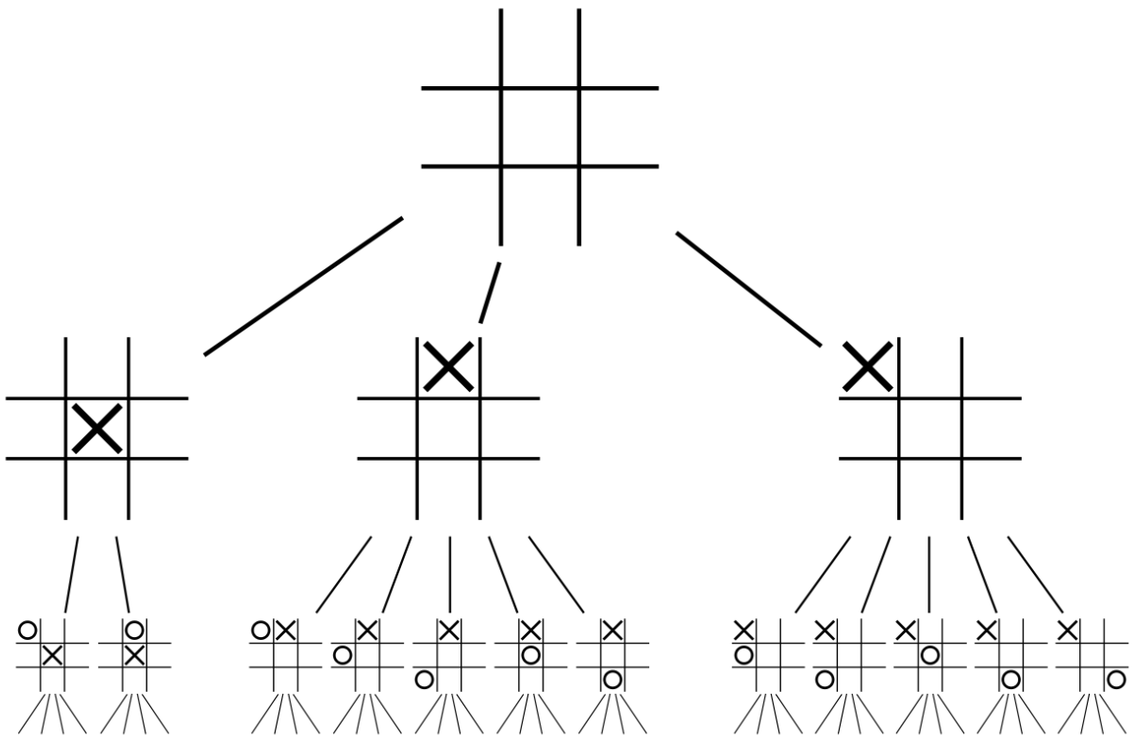
Simplifying assumptions

If we can…

- Represent states as discrete "black box" entities (**atomic representation**) and..

- quickly compute, given a state, which actions lead to which states (**forward model**)

Then searching for a solution becomes similar to finding a path on a map (or graph)



Problem Formulation

Problem Formulation

- What's the starting state?

Problem Formulation

- What's the starting state?
- How are states represented?

Problem Formulation

- What's the starting state?
- How are states represented?
- What actions are available at each state?

Problem Formulation

- What's the starting state?
- How are states represented?
- What actions are available at each state?
- Which state does each action at a given state take to? (forward model)

Problem Formulation

- What's the starting state?
- How are states represented?
- What actions are available at each state?
- Which state does each action at a given state take to? (forward model)
- Which states are terminal?

Problem Formulation

- What's the starting state?
- How are states represented?
- What actions are available at each state?
- Which state does each action at a given state take to? (forward model)
- Which states are terminal?
- What's the reward at each terminal state?

Representation of states

Representation of states

Representation	Description	What can we do with a state?	Example (Route Finding)

Representation of states

Representation	Description	What can we do with a state?	Example (Route Finding)
Atomic	Indivisible (black box) states	Check for terminal states, parent/children of a state, equality between states (maybe)	Name of the city/road we are in

Representation of states

Representation	Description	What can we do with a state?	Example (Route Finding)
Atomic	Indivisible (black box) states	Check for terminal states, parent/children of a state, equality between states (maybe)	Name of the city/road we are in
Factored	States represented by **arrays** of properties	Check for value of each property, do math/logic with it	GPS coordinates, fuel levels

Representation of states

Representation	Description	What can we do with a state?	Example (Route Finding)
Atomic	Indivisible (black box) states	Check for terminal states, parent/children of a state, equality between states (maybe)	Name of the city/road we are in
Factored	States represented by **arrays** of properties	Check for value of each property, do math/logic with it	GPS coordinates, fuel levels
Structured	States represented by complex hierarchy of objects with their own properties	Check for objects within a state, their relationships and properties	Passengers, each with their names, luggage weight, hunger level, favorite food…

Representation of states

Representation	Description	What can we do with a state?	Example (Route Finding)
Atomic	Indivisible states	Check for terminal states, parent/children of a state, equality between states (maybe)	Name of the city/road we are in
Factored	States represented by arrays of properties	Check for value of each property, do math/logic with it	GPS coordinates, fuel levels
Structured	States represented by complex hierarchy of objects with their own properties	Check for objects within a state, their relationships and properties	Passengers, each with their names, luggage weight, hunger level, favorite food…

Recap: Atomic State

Recap: Atomic State

- Simplifying assumption: all you can ask of a node is

Recap: Atomic State

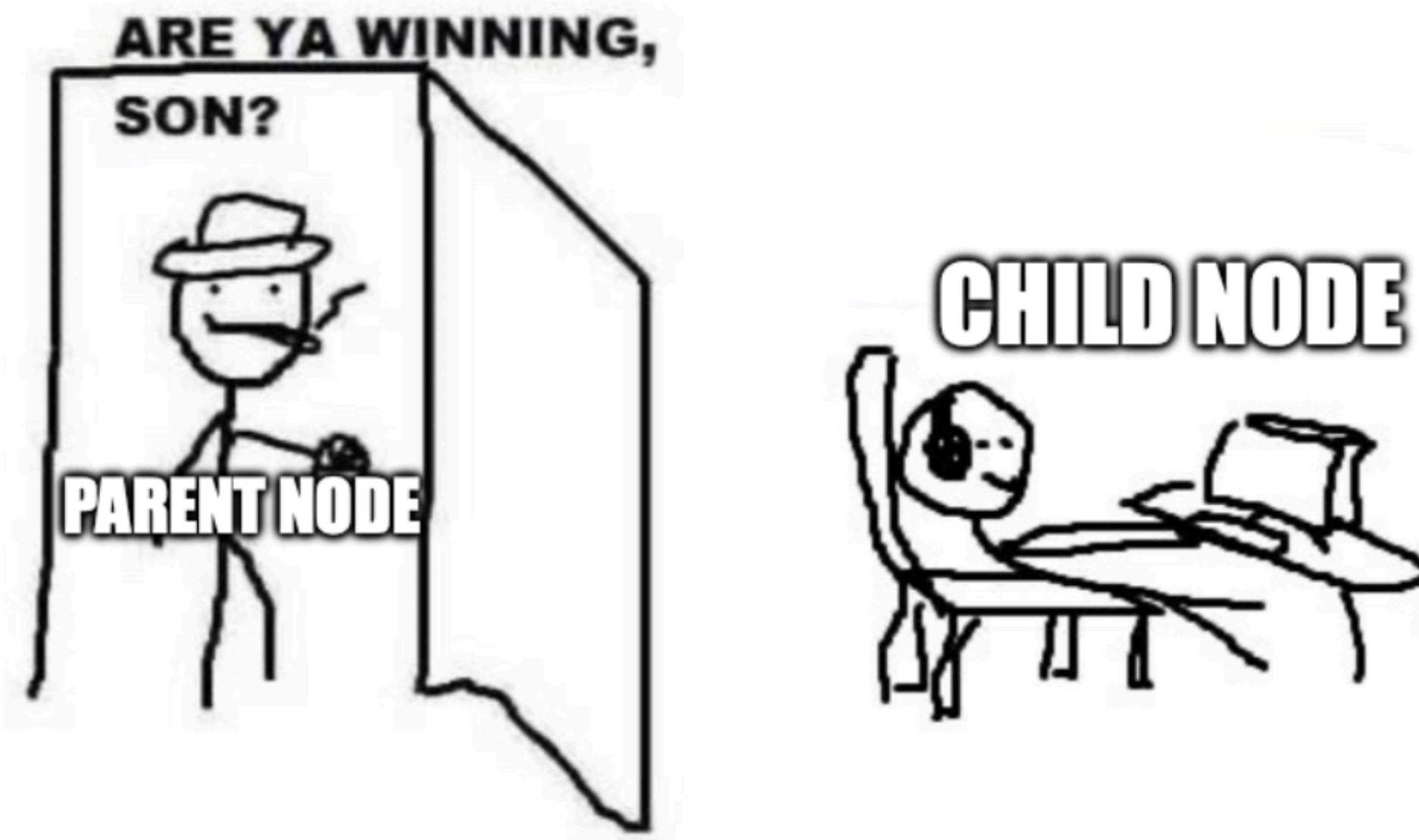
- Simplifying assumption: all you can ask of a node is
- Who is its parent?

Recap: Atomic State

- Simplifying assumption: all you can ask of a node is
- Who is its parent?
- Who are its children?

Recap: Atomic State

- Simplifying assumption: all you can ask of a node is
- Who is its parent?
- Who are its children?
- Is the state a win?

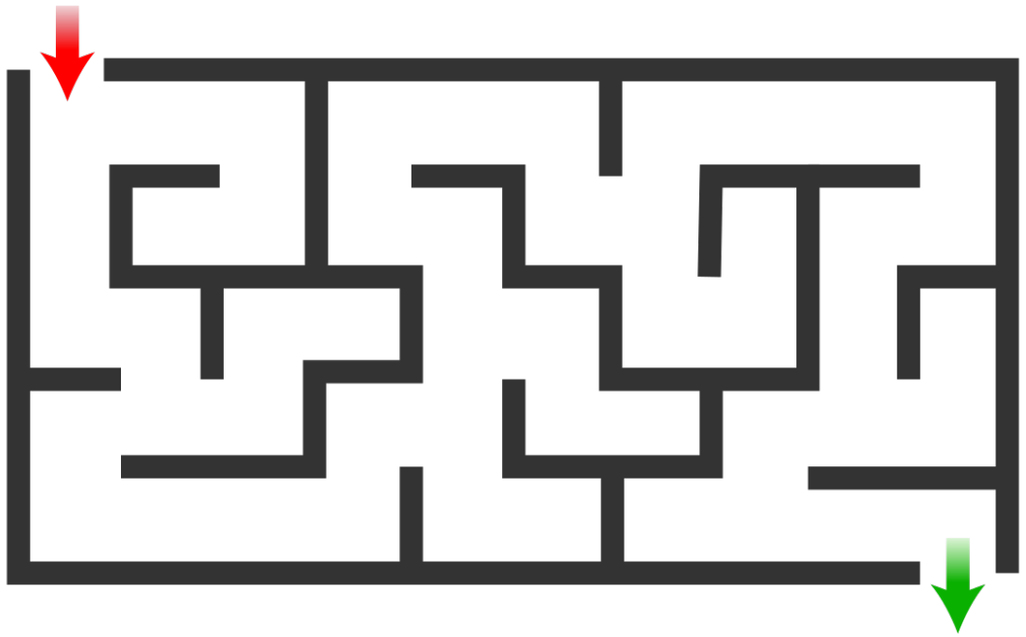

ARE YA WINNING,
SON?
PARENT NODE
CHILD NODE

Search in “physical” space versus game-state space

Question: If you move one step right, then left, are you in the same state?

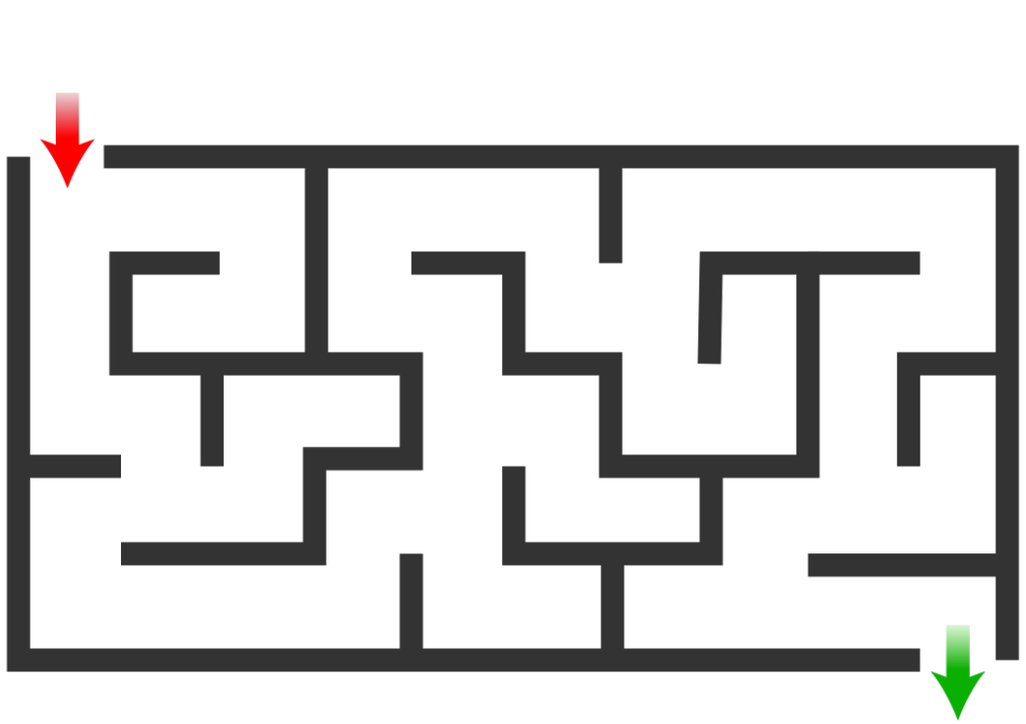
Search in “physical” space versus game-state space

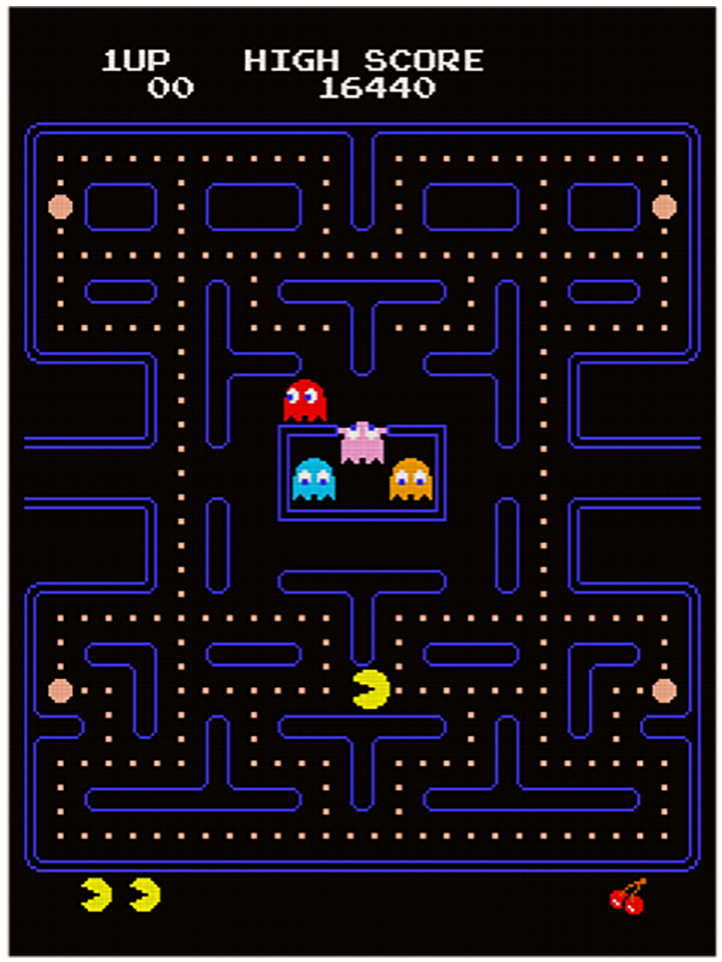
Question: If you move one step right, then left, are you in the same state?



Search in “physical” space versus game-state space

Question: If you move one step right, then left, are you in the same state?





Classical tree search

Classical tree search

- Atomic representation of states

Classical tree search

- Atomic representation of states
- Typical assumptions:

Classical tree search

- Atomic representation of states
- Typical assumptions:

➡ No randomness

Classical tree search

- Atomic representation of states
- Typical assumptions:

➡ No randomness

➡ No hidden information

Classical tree search

- Atomic representation of states
- Typical assumptions:
  - ➡ No randomness
  - ➡ No hidden information
  - ➡ Single-agent

Informed vs Uninformed Search

Informed vs Uninformed Search

- Uninformed: no domain knowledge

Informed vs Uninformed Search

- Uninformed: no domain knowledge

  - All we can do is ask whether a state is terminal and what its reward is.

Informed vs Uninformed Search

- Uninformed: no domain knowledge
    - All we can do is ask whether a state is terminal and what its reward is.
- Informed: some domain knowledge

Informed vs Uninformed Search

- Uninformed: no domain knowledge
  - All we can do is ask whether a state is terminal and what its reward is.
- Informed: some domain knowledge
  - A domain-specific *heuristic function* estimates the desirability of each state (e.g. expected score at the end of the game, expected cost to nearest win state).

States vs Nodes

States vs Nodes

- **State**: a representation of the environment at a certain point in time

States vs Nodes

- **State**: a representation of the environment at a certain point in time
    - Changes only when we choose an action

States vs Nodes

- **State**: a representation of the environment at a certain point in time
    - Changes only when we choose an action
- **Node**: a data structure used to build the tree that represents our search for a solution

States vs Nodes

- **State**: a representation of the environment at a certain point in time
    - Changes only when we choose an action
- **Node**: a data structure used to build the tree that represents our search for a solution
    - Usually contains a state and other metadata (pointer to parent/children, number of visits, heuristics).

States vs Nodes

- **State**: a representation of the environment at a certain point in time
    - Changes only when we choose an action
- **Node**: a data structure used to build the tree that represents our search for a solution
    - Usually contains a state and other metadata (pointer to parent/children, number of visits, heuristics).
    - While searching for an action, we may visit many nodes, but we don't move to a different state until we commit to an action.

States vs Nodes

- **State**: a representation of the environment at a certain point in time
    - Changes only when we choose an action
- **Node**: a data structure used to build the tree that represents our search for a solution
    - Usually contains a state and other metadata (pointer to parent/children, number of visits, heuristics).
    - While searching for an action, we may visit many nodes, but we don't move to a different state until we commit to an action.
    - For this to be possible, you must be able to simulate sequences of states **without** actually changing the environment

States vs Nodes

- **State**: a representation of the environment at a certain point in time
    - Changes only when we choose an action
- **Node**: a data structure used to build the tree that represents our search for a solution
    - Usually contains a state and other metadata (pointer to parent/children, number of visits, heuristics).
    - While searching for an action, we may visit many nodes, but we don't move to a different state until we commit to an action.
    - For this to be possible, you must be able to simulate sequences of states **without** actually changing the environment
    - This allows you to backtrack!

Cost vs Complexity

Cost vs Complexity

- Do not confuse:

Cost vs Complexity

- Do not confuse:

  ➡ **Cost of a solution (path cost):** usually, # of nodes in the path (but edges may have different weights)

Cost vs Complexity

- Do not confuse:

  ➡ **Cost of a solution (path cost):** usually, # of nodes in the path (but edges may have different weights)

  ➡ **Computational complexity of the search (search cost):** usually, proportional to total # of nodes visited (even ones that were not in the path to the solution)

Completeness and Optimality

- **Complete**: if a solution exists (with finite depth), the algorithm will find it

- **Optimal:** if multiple solutions exist, the algorithm will return one with minimal depth (or path cost)

Example Environments

Route Finding

- **states**
  - locations
- **initial state**
  - starting point
- **successor function (actions)**
  - move from one location to another
  - it is common to simplify the space to a discrete sets of points, which leads to search in a graph
    - Each location is a vertex
    - If you can move from locations A to B, there is an edge between the corresponding vertices
- **goal test**
  - arrive at a certain location
- **path cost**
  - may be quite complex
    - money, time, travel comfort, scenery, …
- **Important: All other problems in this section can be thought of as finding a route in a suitably-defined graph!**

n-queens

- **Put n queens on a n × n board with no two queens on the same row, column, or diagonal**
  - see also Wikipedia Eight Queens Puzzle

8-Queens
Incremental Approach

- **start with an empty board**
- **add queens one by one**
  - no violation of constraints
    - different row, column, diagonal from sitting queens
- **incremental formulation**
  - states
    - arrangement of up to 8 queens on the board
  - initial state
    - empty board
  - successor function (actions)
    - add a queen to any square
  - goal test
    - all queens on board
    - no queen attacked
  - path cost
    - irrelevant (all solutions equally valid)
- **Properties: 3*1014 possible sequences; can be reduced to 2,057**

8-Queens
Complete-State Approach

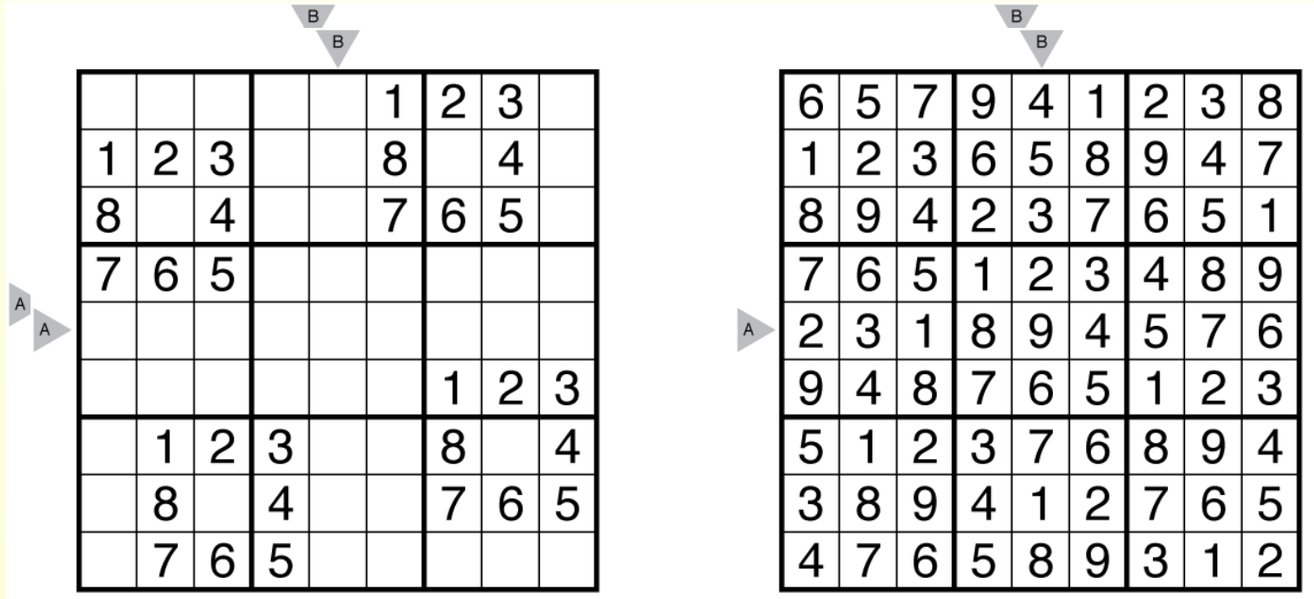
- **start with a full board**
  - all n queens placed on the board
  - conflicts are to be expected
- **try to find a better configuration**
  - reduced number of conflicts
- **complete-state formulation**
  - states
    - arrangement of 8 queens on the board
  - initial state
    - all 8 queens on board
  - successor function (actions)
    - move a queen to a different square
  - goal test
    - no queen attacked
  - path cost
    - irrelevant (all solutions equally valid)
- **Properties: good strategies can reduce the number of possible sequences considerably**

8-Queens Refined

- **simple solutions may lead to very high search costs**
  - 64 fields, 8 queens ==> 648 possible sequences
- **more refined solutions trim the search space, but may introduce other constraints**
  - place queens on “unattacked” places
    - much more efficient
    - may not lead to a solution depending on the initial moves
  - move an attacked queen to another square in the same column, if possible to an “unattacked” square
    - much more efficient

Sudoku

- **states**
  - Grid puzzle with cells filed with numbers 0-9 or a blank
- **initial state**
  - A few cells initially filled
- **successor function (actions)**
  - Add a number to a blank cell
- **goal test**
  - All cell filled
  - No cell sharing a row, column or 3x3 square have the same number
- **path cost**
  - all solutions are equally valid



					1	2	3	
1	2	3			8		4	
8		4			7	6	5	
7	6	5						
						1	2	3
	1	2	3			8		4
	8		4			7	6	5
	7	6	5					

6	5	7	9	4	1	2	3	8
1	2	3	6	5	8	9	4	7
8	9	4	2	3	7	6	5	1
7	6	5	1	2	3	4	8	9
2	3	1	8	9	4	5	7	6
9	4	8	7	6	5	1	2	3
5	1	2	3	7	6	8	9	4
3	8	9	4	1	2	7	6	5
4	7	6	5	8	9	3	1	2

Bridge and Torch problem

- states
  - Location of four adventurers (A,B,C,D) and a torch (*) on either bank of a river crossed by a bridge
  - Example (A,B,* I C,D)
- initial state
  - All adventurers and the torch on the left bank of the river (A,B,C,D,* I)
- successor function (actions)
  - Move one or to two adventurers and the torch from the bank with the torch to the opposite bank
- Cost of an action
  - Each adventure takes a number of seconds to cross (1,2,5,8)
  - If two adventurers cross simultaneously, use the cost of the slower adventurer
- goal test
  - All adventurers and the torch on the right bank of the river (I A,B,C,D,*)
- path cost
  - Total cost of all crossing
- Variations exist, such as as “goats and cabbage”, “wolves and sheep” etc.
- In general, can be called “river-crossing problems”

Robot Navigation

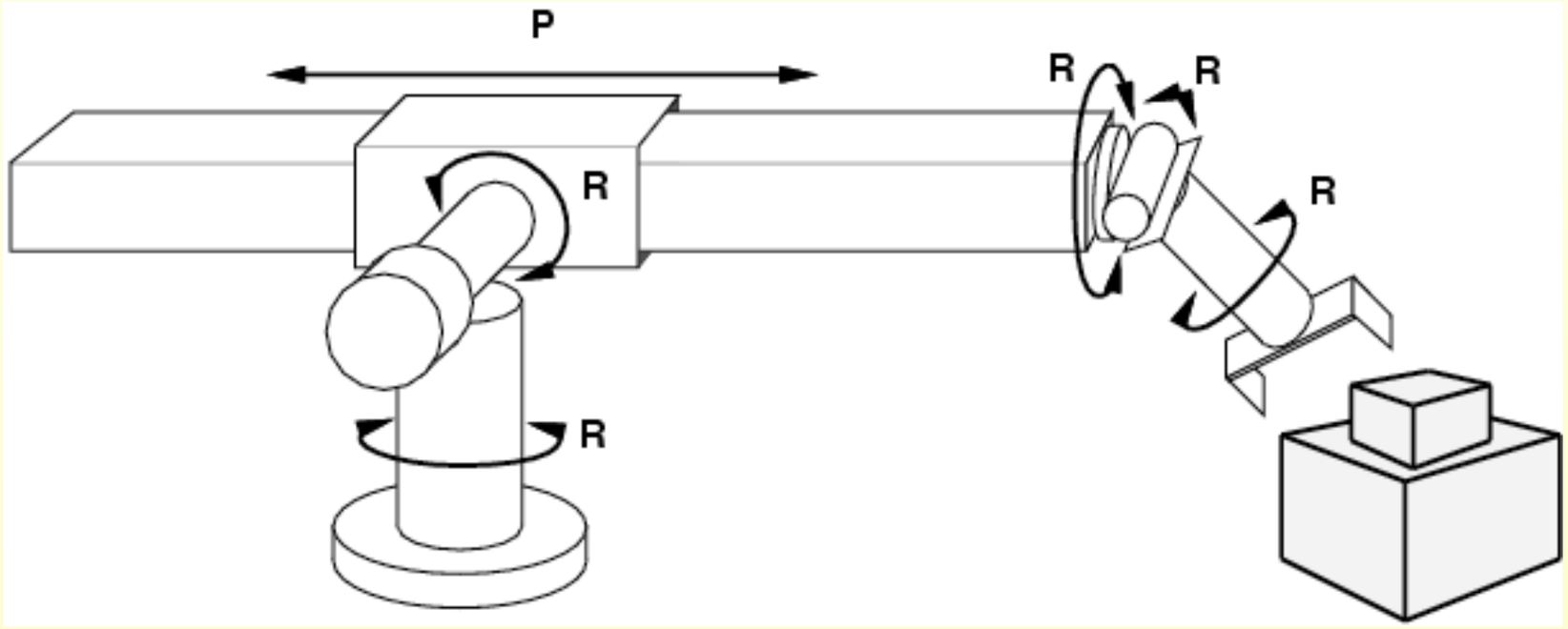
- **states**
  - locations
  - position of actuators
- **initial state**
  - start position (dependent on the task)
- **successor function (actions)**
  - movement, actions of actuators
- **goal test**
  - task-dependent
- **path cost**
  - may be very complex
    - distance, energy consumption

Assembly Sequencing

- **states**
  - location of components
- **initial state**
  - no components assembled
- **successor function (actions)**
  - place component
- **goal test**
  - system fully assembled
- **path cost**
  - number of moves

Example: robotic assembly

- **states:**
  - real-valued coordinates of robot joint angles parts of the object to be assembled
- **actions:**
  - continuous motions of robot joints
- **goal test:**
  - complete assembly
- **path cost:**
  - time to execute



Uninformed Search

General Tree Search Algorithm

General Tree Search Algorithm

1. **Initialize fringe (or frontier) with node containing the initial state**
   - ➡ **Fringe:** data structure with nodes that we want to explore next
   - ➡ Initially contains a node corresponding to the start state
2. **Repeat while you have computational budget:**
   A. Check if fringe is empty
      - If so, return failure
   B. (If not) Select a node n in the fringe and remove it from the fringe
   C. Check if n contains a goal state
      - If so, go to 3 (return solution)
   D. (If not) Expand n, adding nodes containing all its successor states to the fringe
3. **Return the action that leads from the initial node to the second node on the path to the goal (online problems) or the entire path (offline)**

General Tree Search Algorithm

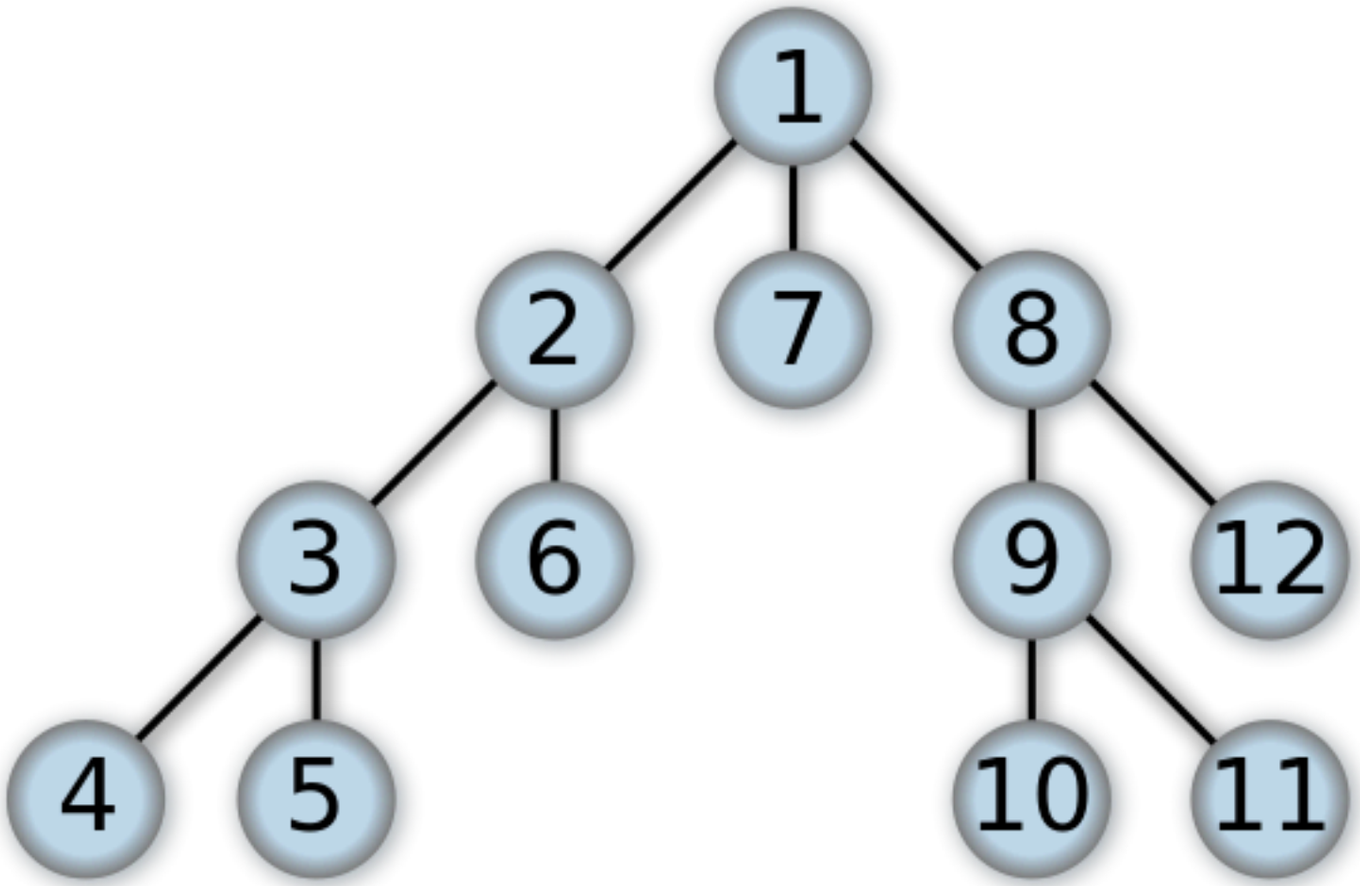
1. **Initialize fringe (or frontier) with node containing the initial state**
   - ➡ **Fringe:** data structure with nodes that we want to explore next
   - ➡ Initially contains a node corresponding to the start state

2. **Repeat while you have computational budget:**
   A. Check if fringe is empty
      - If so, return failure
   B. (If not) Select a node n in the fringe and remove it from the fringe
   C. Check if n contains a goal state
      - If so, go to 3 (return solution)
   D. (If not) Expand n, adding nodes containing all its successor states to the fringe

3. **Return the action that leads from the initial node to the second node on the path to the goal (online problems) or the entire path (offline)**

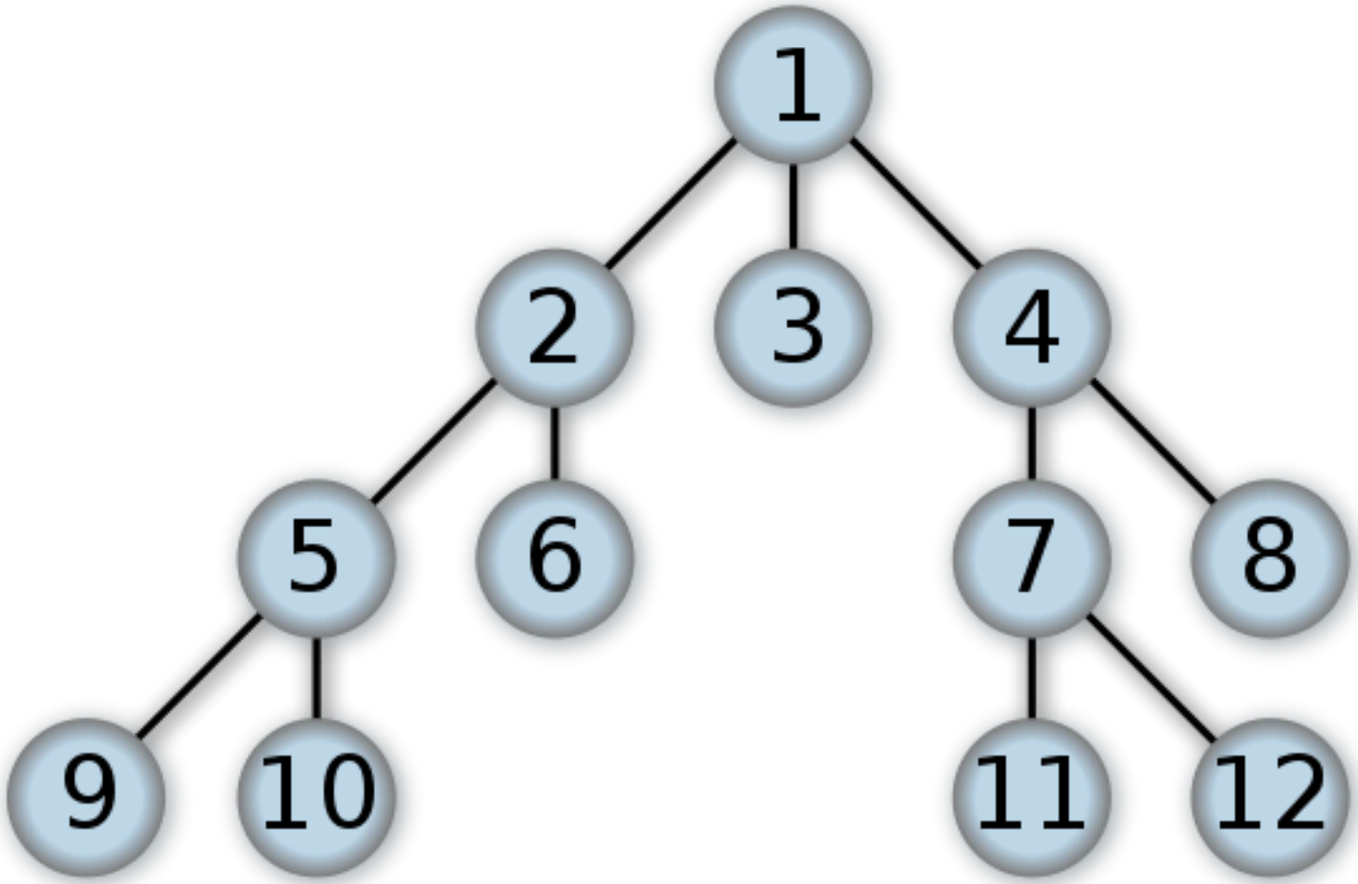
The order in which elements are added (or removed) to the fringe defines the search algorithm!

Depth-first search



- Search deeply first
- Fringe is a LIFO stack

Breadth-first search



- Search widely first
- Fringe is a FIFO queue

DFS versus BFS

- **Depth-First Search (DFS)** goes off into one branch until it reaches a leaf node, then backtracks

    - Low memory usage: Only needs to keep nodes in current path in memory

    - Search is incomplete**:** may fail to return a solution, even if one exists (loops or infinite branches).

    - Not optimal: if a solution is found, it may not be the cheapest one.

DFS versus BFS

- **Breadth-First Search (DFS)** visits all nodes at a given depth before searching the next level
    - High memory usage: Keeps all nodes at a given level in-memory
    - Search is complete: always finds a solution, if one exists at finite depth
    - Optimal: if a solution is found, it will be the cheapest one (assuming equal edge weights).

DFS versus BFS

- **Breadth-First Search (DFS)** visits all nodes at a given depth before searching the next level
    - High memory usage: Keeps all nodes at a given level in-memory
    - Search is complete: always finds a solution, if one exists at finite depth
    - Optimal: if a solution is found, it will be the cheapest one (assuming equal edge weights).

DFS versus BFS

- **Breadth-First Search (DFS)** visits all nodes at a given depth before searching the next level
    - High memory usage: Keeps all nodes at a given level in-memory
    - Search is complete: always finds a solution, if one exists at finite depth
    - Optimal: if a solution is found, it will be the cheapest one (assuming equal edge weights)

How can we get the advantages of DFS and BFS at the same time?

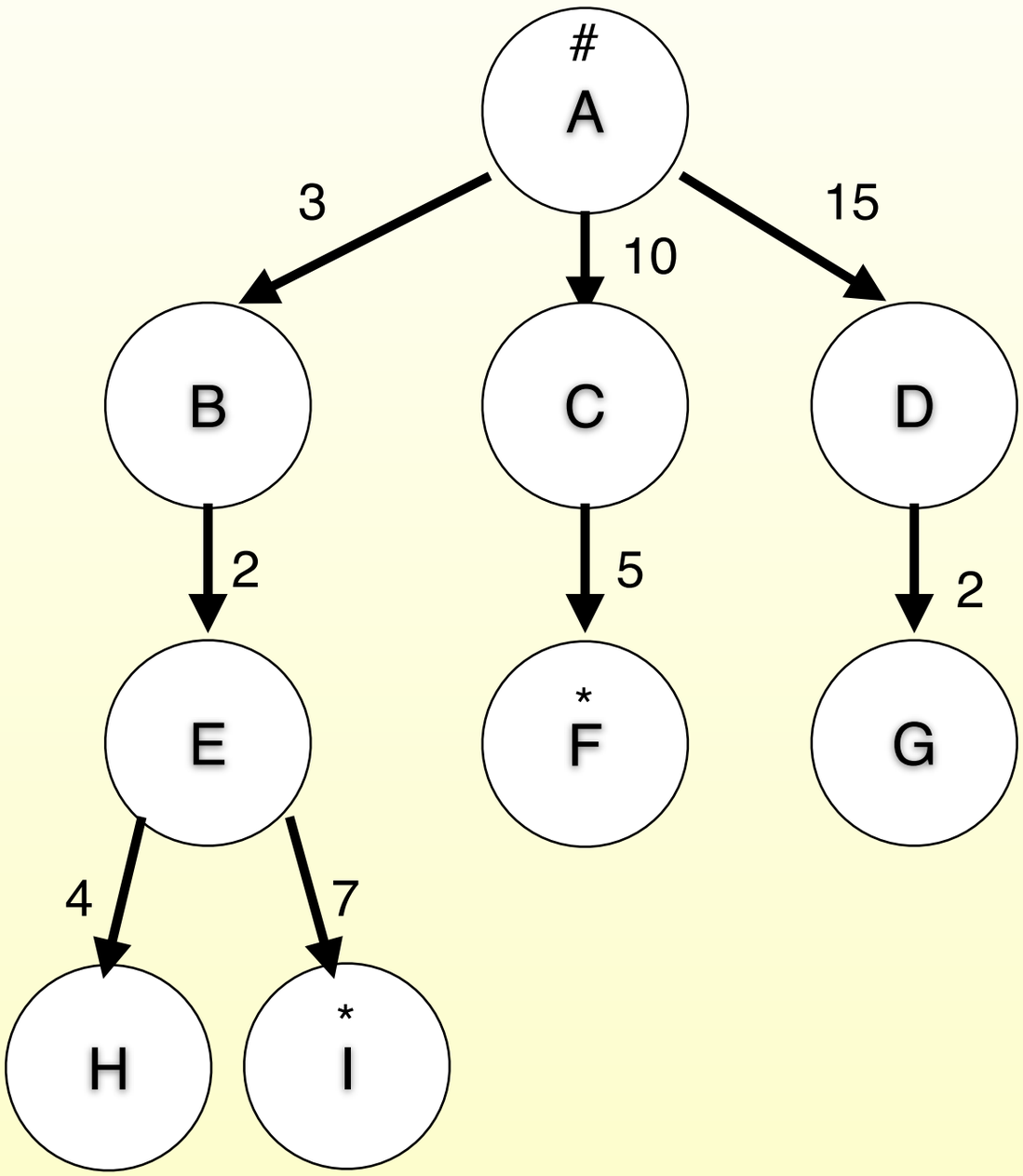
Depth-Limited search and iterative Deepening

- Depth-limited: cuts off DFS at a specified depth
- Iterative deepening: iteratively applies depth-limited with higher and higher depths
  - Shallow nodes may be re-visited many times
  - Seems like a lot of overhead at a glance, but asymptotical analysis shows this is not the case (almost all nodes are deep)
  - Combines **low memory usage** of DFS with **completeness and optimality** of BFS. Very good performance in practice!

Uniform Cost search

- A variation of Breadth-First Search that handles edges with variable cost
- Strategy: associate a cost **g(n)** to each node n added to the fringe
- Always take the node with smallest g to expand next
- Done naively, this may require sorting the fringe each time, which is expensive.
- **Priority Queues** are an abstract data type that allow for quickly finding the smallest element.
  - Heaps are the most common implementation of priority queues

Example State Space



Legend:

- Starting state

* - Goal state

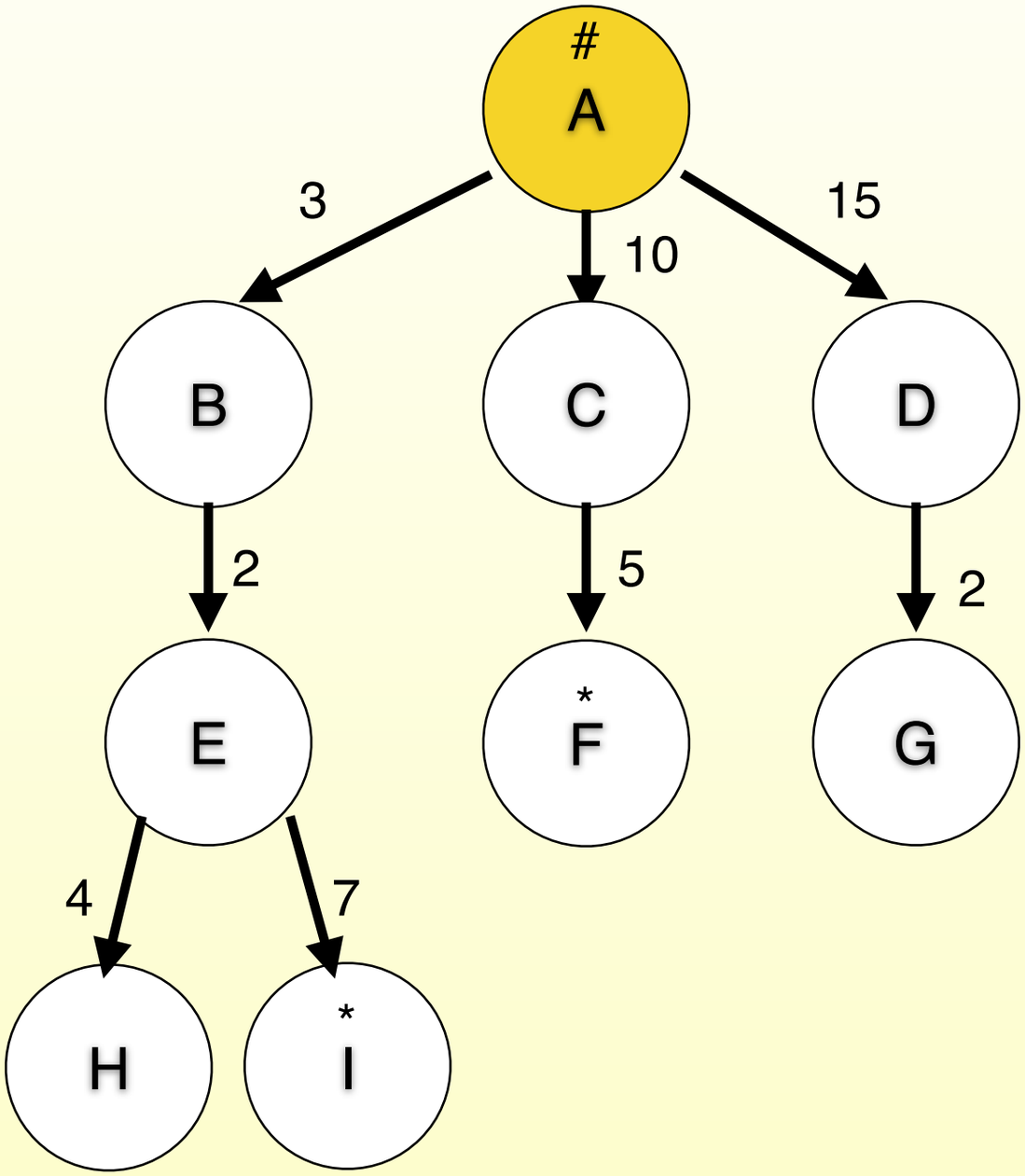
Visited (expanded)

Current state

Fringe

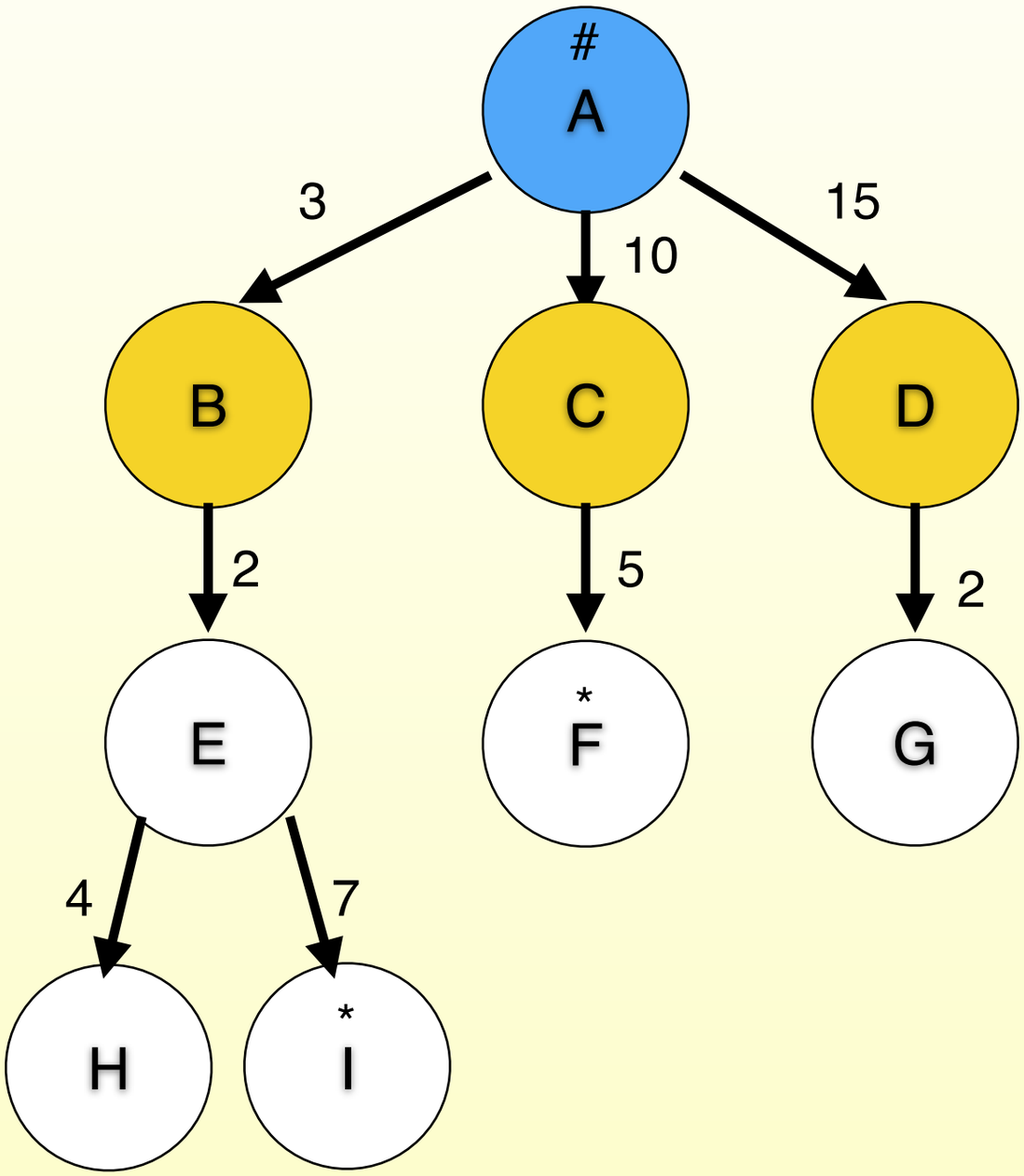
c → Transition cost = c

Uniform Cost Search Example



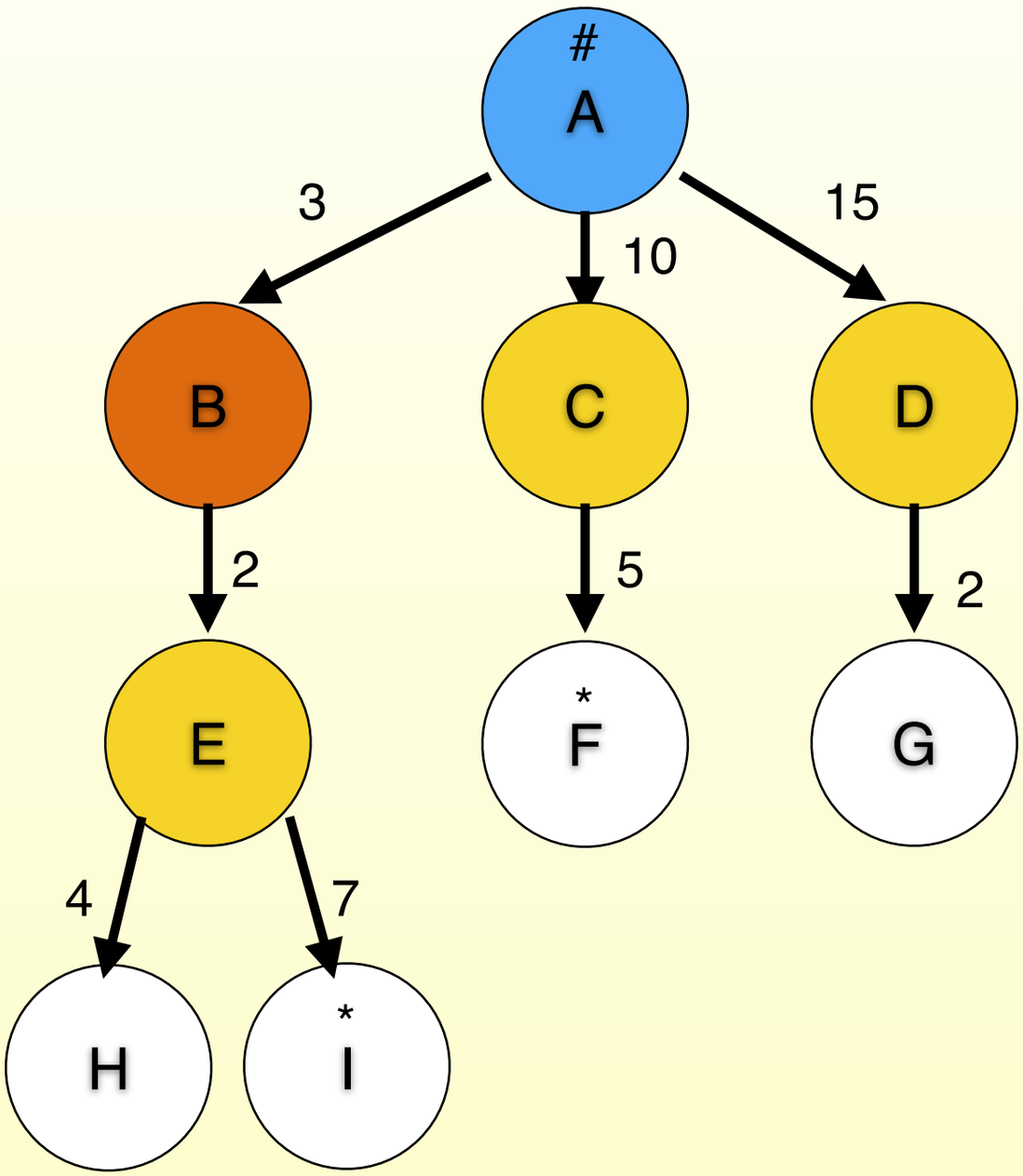
Node	g
A	0

Uniform Cost Search Example



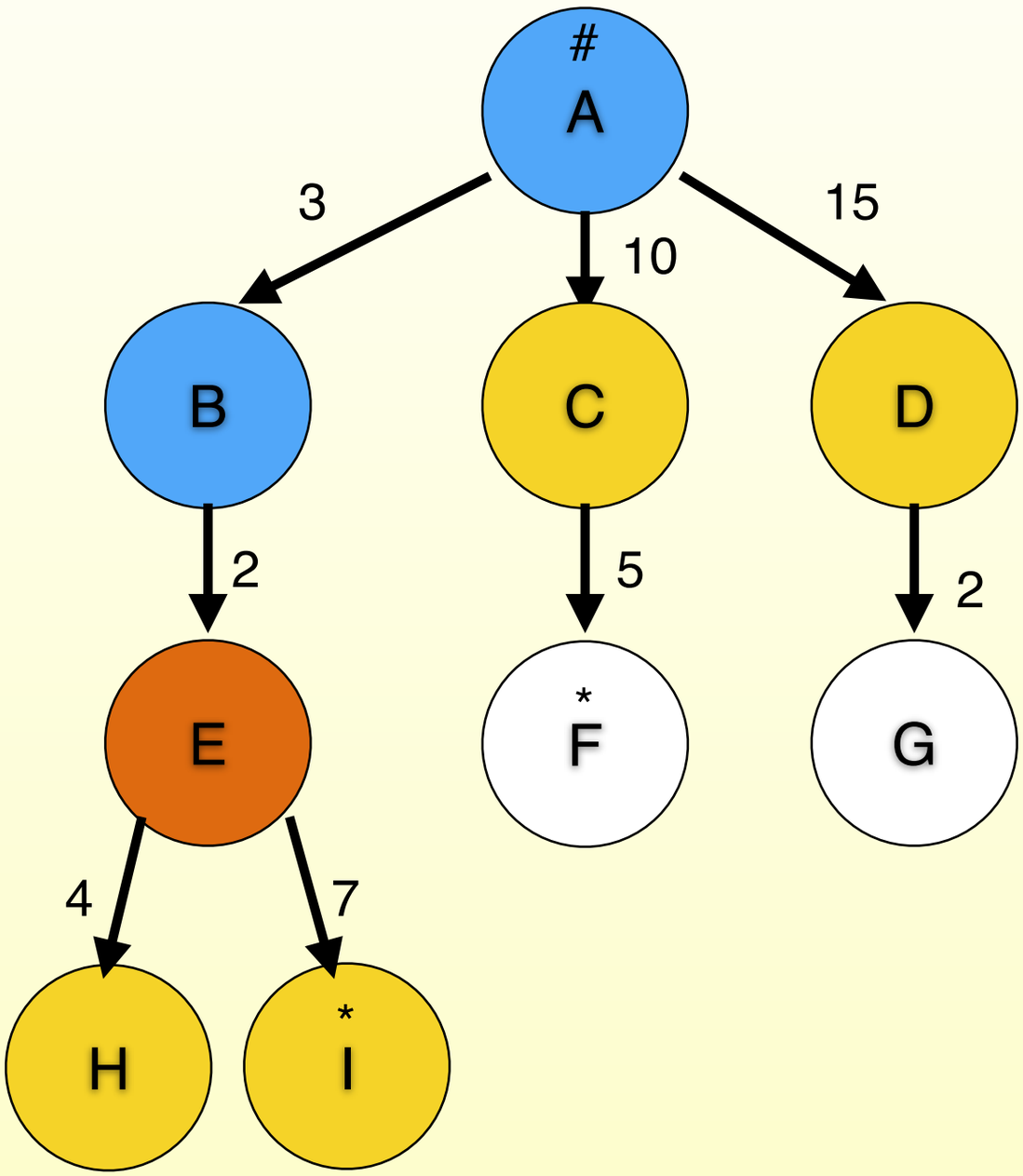
Node	g
B	3
C	10
D	15

Uniform Cost Search Example



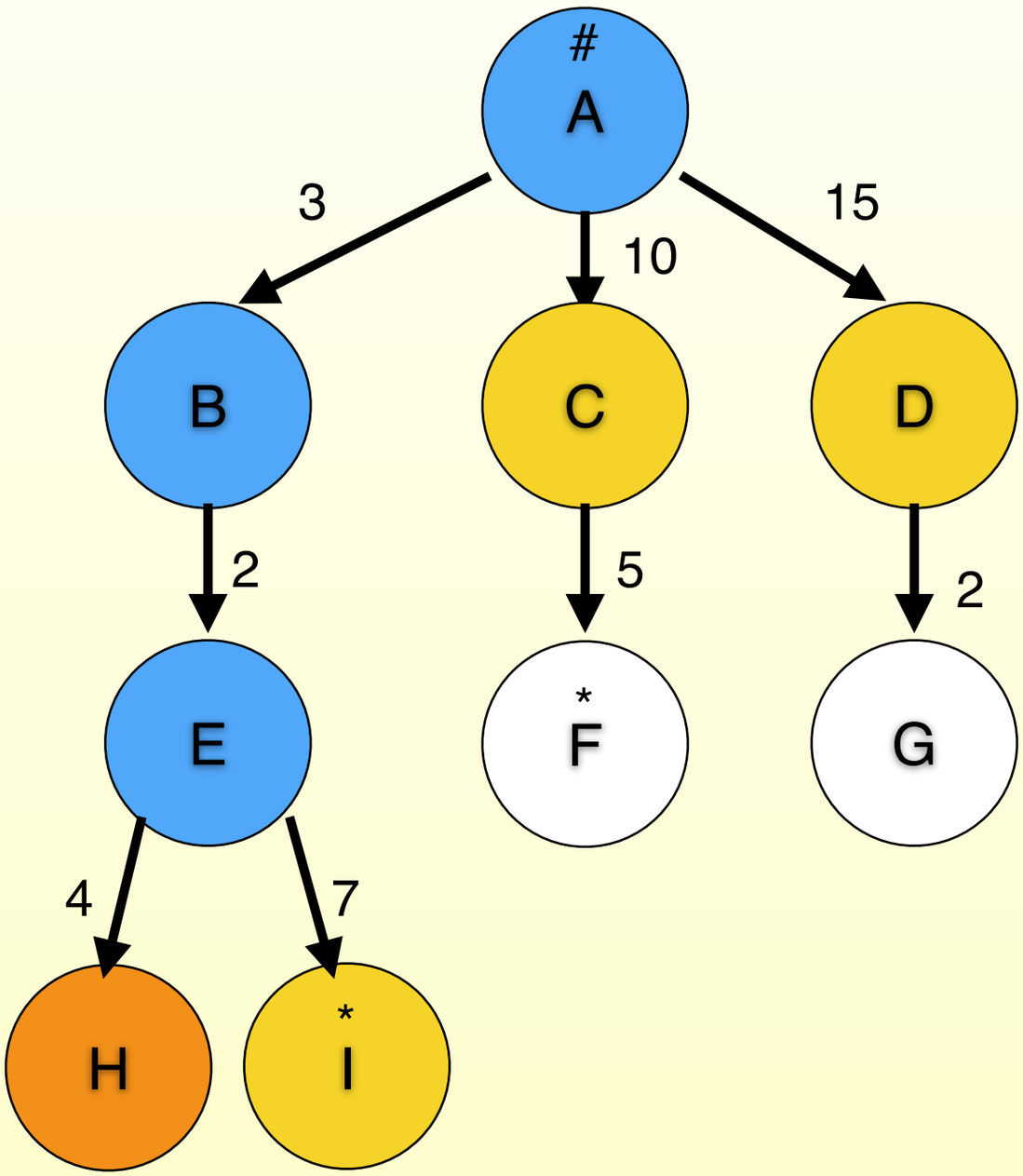
Node	g
E	3+2 = 5
C	10
D	15

Uniform Cost Search Example



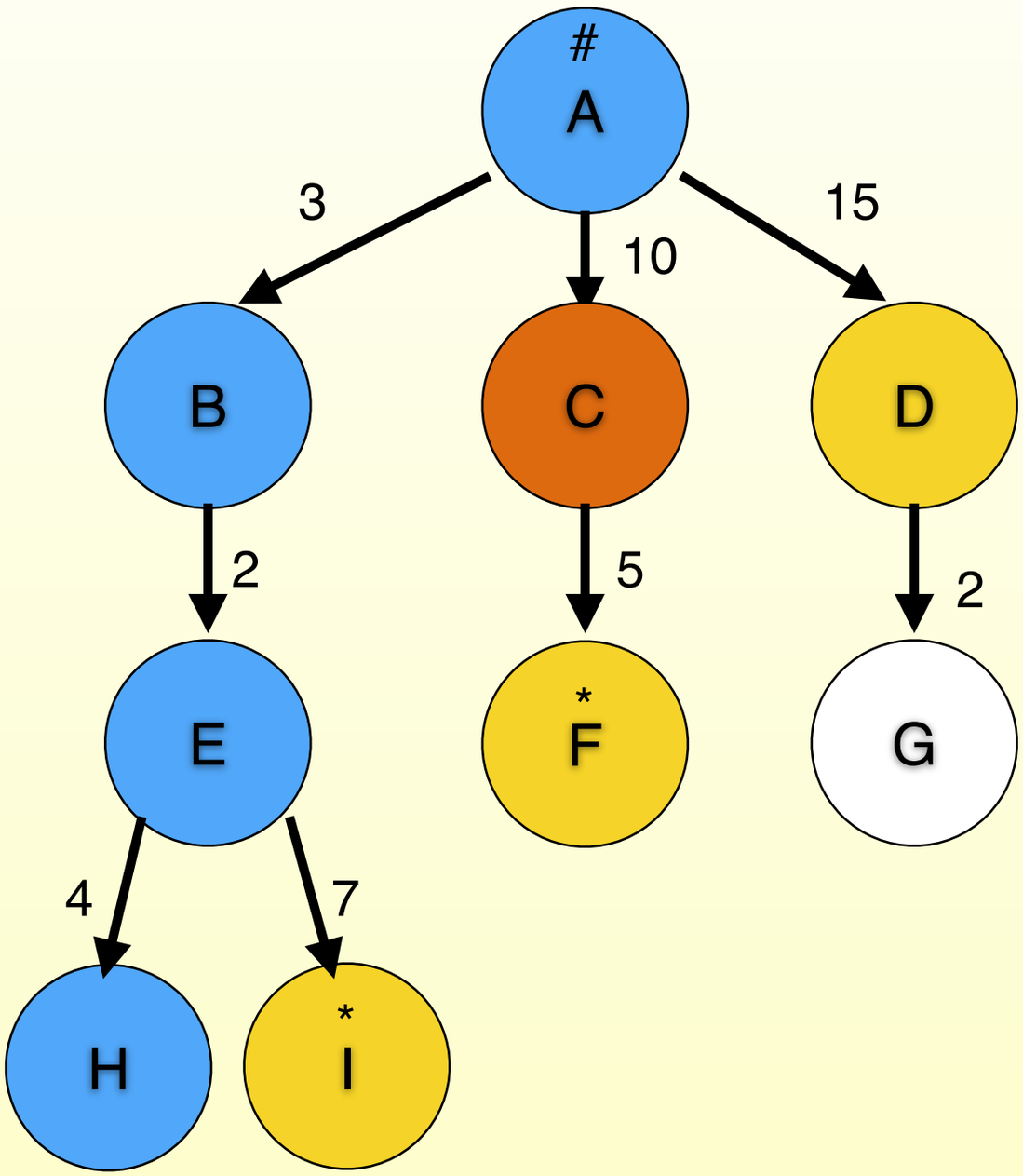
Node	g
H	5+4 = 9
C	10
I	5+7=12
D	15

Uniform Cost Search Example



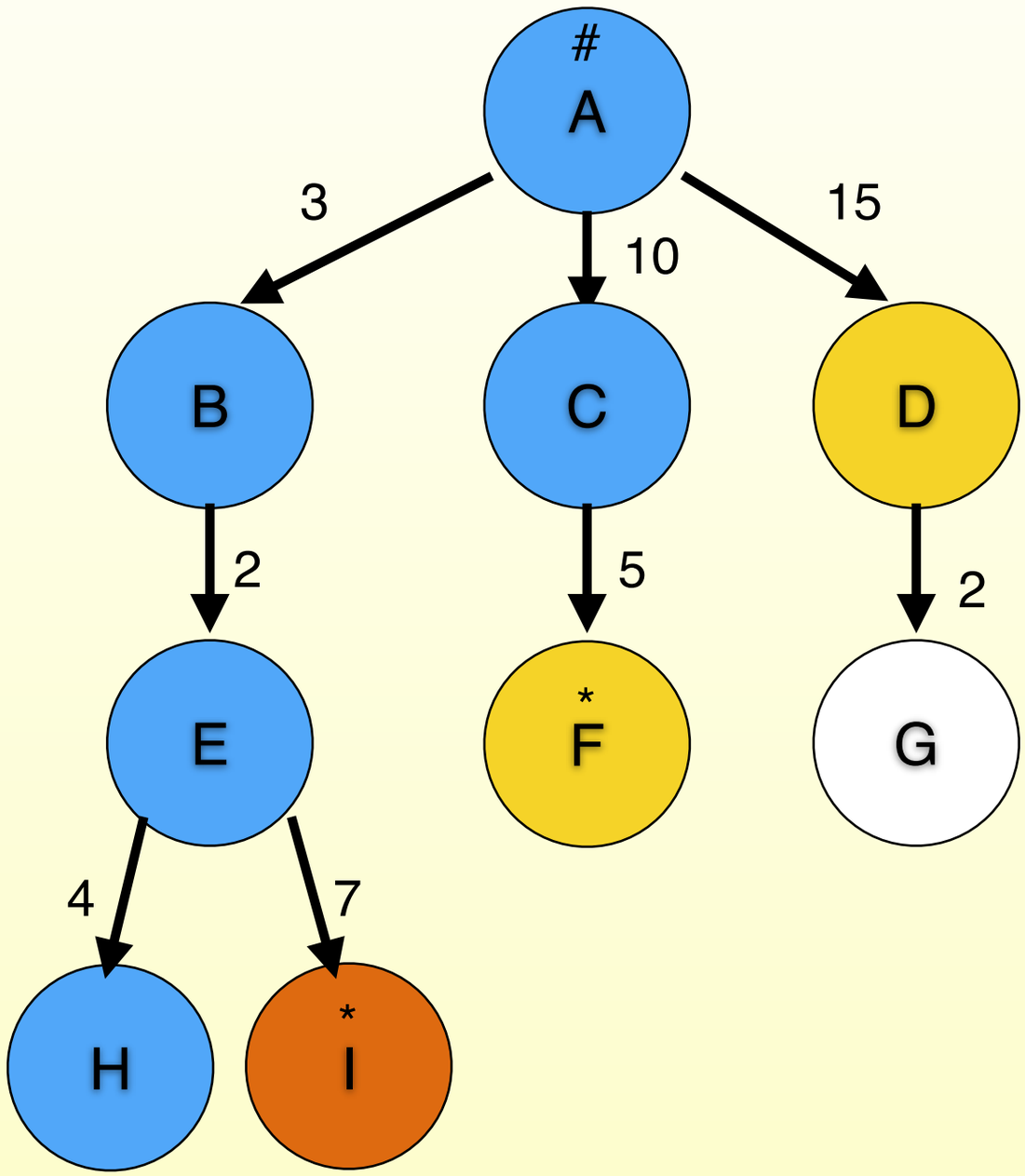
Node	g
H	5+4 = 9
C	10
I	5+7=12
D	15

Uniform Cost Search Example



Node	g
I	12
F	10+5=15
D	15

Uniform Cost Search Example



Node	g
I	12
F	10+5=15
D	15

Goal found with cost 12!

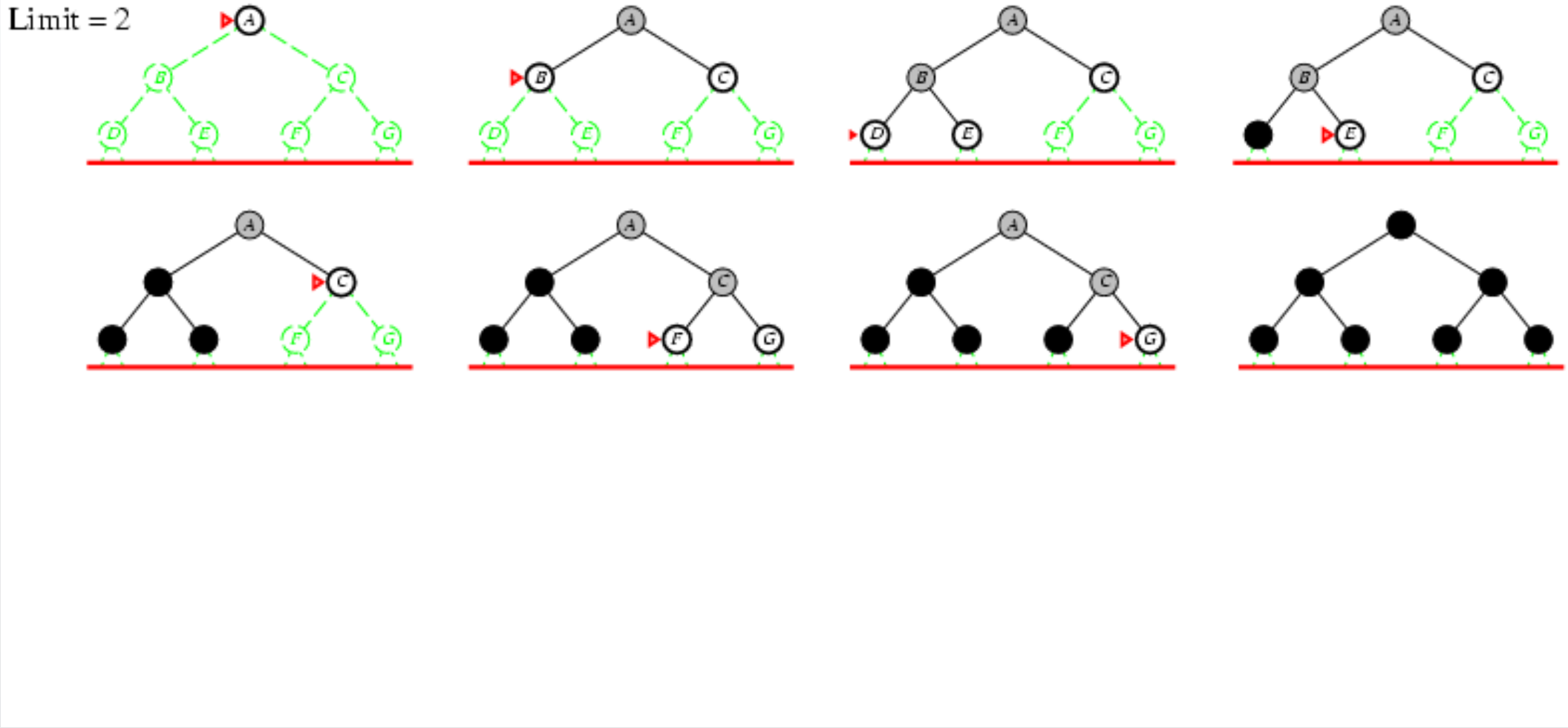
Uniform Cost search vs other algorithms

- Uniform cost search can be seen as a variation of Djikstra's Algorithm
- The typical formulation of Djisktra is to find the shortest path from the root to **all nodes**
- In uniform cost, we typically stop once the first **goal node** is found
- DFS can be seen as a special case of uniform cost (where all edges have weight 1)
- Uniform cost can be seen as a special case of A* (where h=0 for all nodes)
- See more at https://en.wikipedia.org/wiki/Dijkstra's_algorithm#Related_problems_and_algorithms

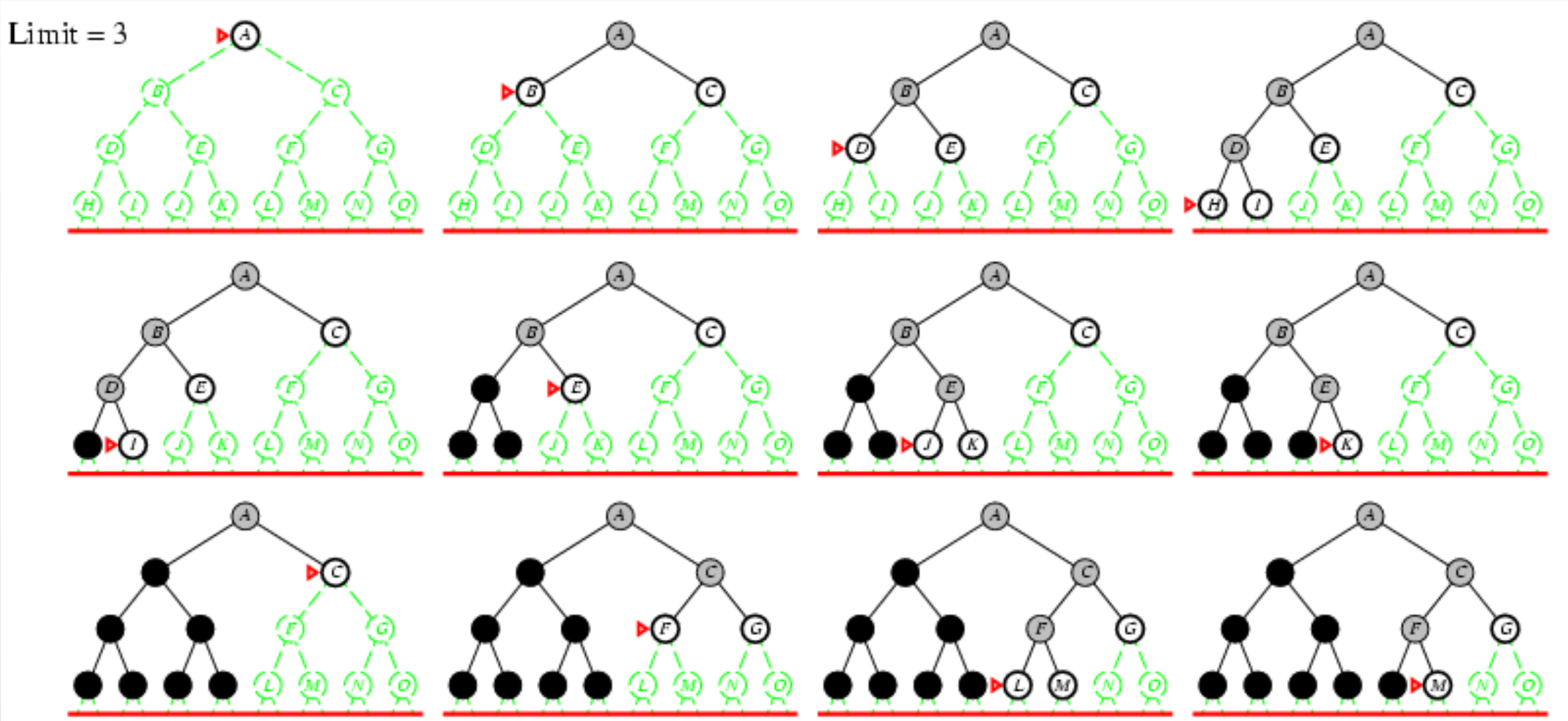
Iterative deepening search l =0

Iterative deepening search l =1

Iterative deepening search l =2



Iterative deepening search l =3



Iterative deepening search

- **Number of nodes generated in a depth-limited search to depth** d **with branching factor** b:

$$N_{DLS} = b^0 + b^1 + b^2 + \ldots + b^{d-2} + b^{d-1} + b^d$$

- **Number of nodes generated in an iterative deepening search to depth** d **with branching factor** b:

$N_{IDS} = (d+1)b^0 + d\,b\hat{}1 + (d-1)b\hat{}2 + \ldots + 3b^{d-2} + 2b^{d-1} + 1b^d$

- **For** b = 10, d = 5,
  - $N_{DLS} = 1 + 10 + 100 + 1{,}000 + 10{,}000 + 100{,}000 = 111{,}111$
  - $N_{IDS} = 6 + 50 + 400 + 3{,}000 + 20{,}000 + 100{,}000 = 123{,}456$

- **Overhead = (123,456 - 111,111)/111,111 = 11%**

Summary - Uninformed Search

Method	Complete?	Optimal?	Space complexity	Time complexity	Data structure for fringe	Notes
BFS	Yes (finite b)	Yes (if costs are constant)	exponential	exponential	FIFO (queue)	Complete, Optimal, bad memory complexity
Uniform-Cost	Yes	Yes (even if variable costs)	exponential	exponential	Priority Queue	Handles different costs
DFS	No	No	linear ~ b*m	exponential	LIFO (stack)	Low memory requirement, but fewer guarantees than BFS
Backtracking DFS	No	No	linear ~ m	exponential	Only 1 node in fringe (but may use stack as part of backtracking implementation)	Even lower memory requirement, but a bit more complicated than DFS
Depth-Limited	No	No	Linear ~ bl	exponential	Stack (variation of BFS)	Alleviates DFS problems with infinite spaces and loops
Iterative Deepening	Yes (if reachable with available memory)	Yes (if reachable with available memory)	Linear ~ bd	exponential	Stack (variation of BFS)	Combines low memory of DFS with completeness and optimality of BFS. Re-explores shallow nodes many times (not too bad)
Bi-directional	Yes (if using BFS)	Yes (if using BFS)	Exponential (square root of complexity with BFS)	exponential (square root of complexity with most other strategies)	Depends on which strategy each side is using	Needs to generate predecessors, needs to enumerate goals, at least one side needs to store all frontier nodes

Note: b = branching factor, d = depth of shallowest goal, m = largest depth of any path, l = threshold for depth-limited

Informed Search

Informed Search

- **relies on additional knowledge about the problem or domain**
  - frequently expressed through heuristics (“rules of thumb”)
- **used to distinguish more promising paths towards a goal**
  - may be mislead, depending on the quality of the heuristic
- **in general, performs much better than uninformed search**
  - but frequently still exponential in time and space for realistic problems

Informed Search

In this lecture, we will discuss:

- A general approach for informed search called best-first search
- A few instantiations of this approach:
  - Greedy best-first search
  - A*
  - Some memory-bounded variations of Greedy and A*
  - Also MCTS
- Ways to generate heuristics for informed search

Best-First Search

- **relies on an evaluation function f(n) that gives an indication of how useful it would be to expand a node**
  - family of search methods with various evaluation functions
  - often based on heuristics, but can use other information as well (see: A*)
  - the evaluation function is different from the cost function g(n) in Uniform-Cost Search
    - cost function: path cost calculated from known costs for steps along the path to the current node
    - evaluation function: estimate of the cost to the goal
- **the node with the lowest value is expanded first**
  - the "best" in the name is a little misleading: the node with the lowest value for the evaluation function is not necessarily one that is on an optimal path to a goal
  - if we really know which one is the best, there's no need to do a search

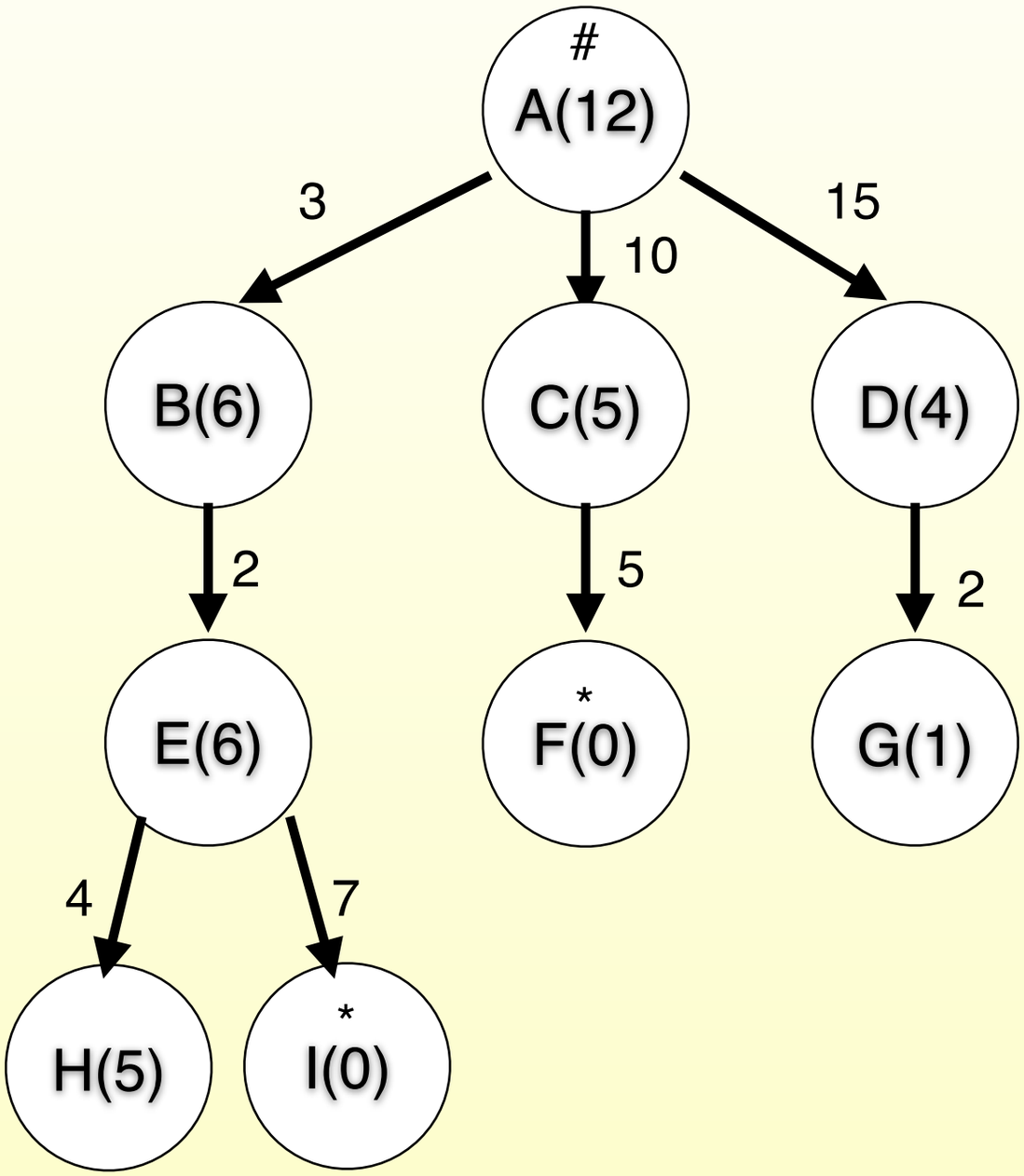
```
function BEST-FIRST-SEARCH(problem, EVAL-FN) returns solution
        fringe := queue with nodes ordered by EVAL-FN
        return TREE-SEARCH(problem, fringe)
```

Greedy Best-First Search

- **minimizes the estimated cost to a goal**
  - expand the node that seems to be closest to a goal
  - utilizes a heuristic function as evaluation function
    - *f(n) = h(n) =* estimated cost from the current node to a goal
    - heuristic functions are problem-specific
    - often straight-line distance for route-finding and similar problems
  - often better than depth-first, although worst-cases complexities are equal or worse (space)

```
function GREEDY-SEARCH(problem) returns solution
    return BEST-FIRST-SEARCH(problem, h)
```

Example State Space



Legend:

- Starting state

* - Goal state

Visited (expanded)

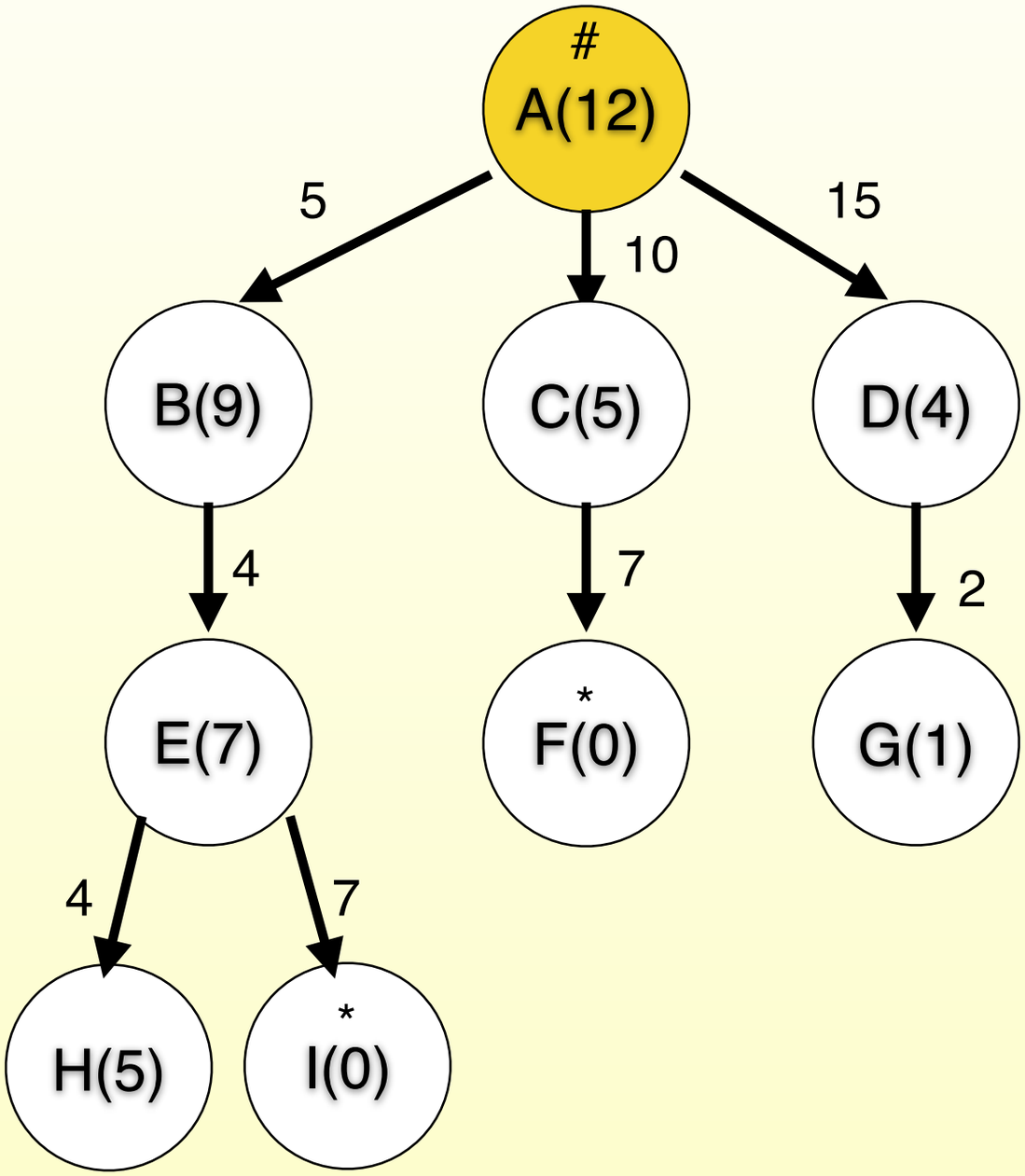
Current state

Fringe

c → Transition cost = c

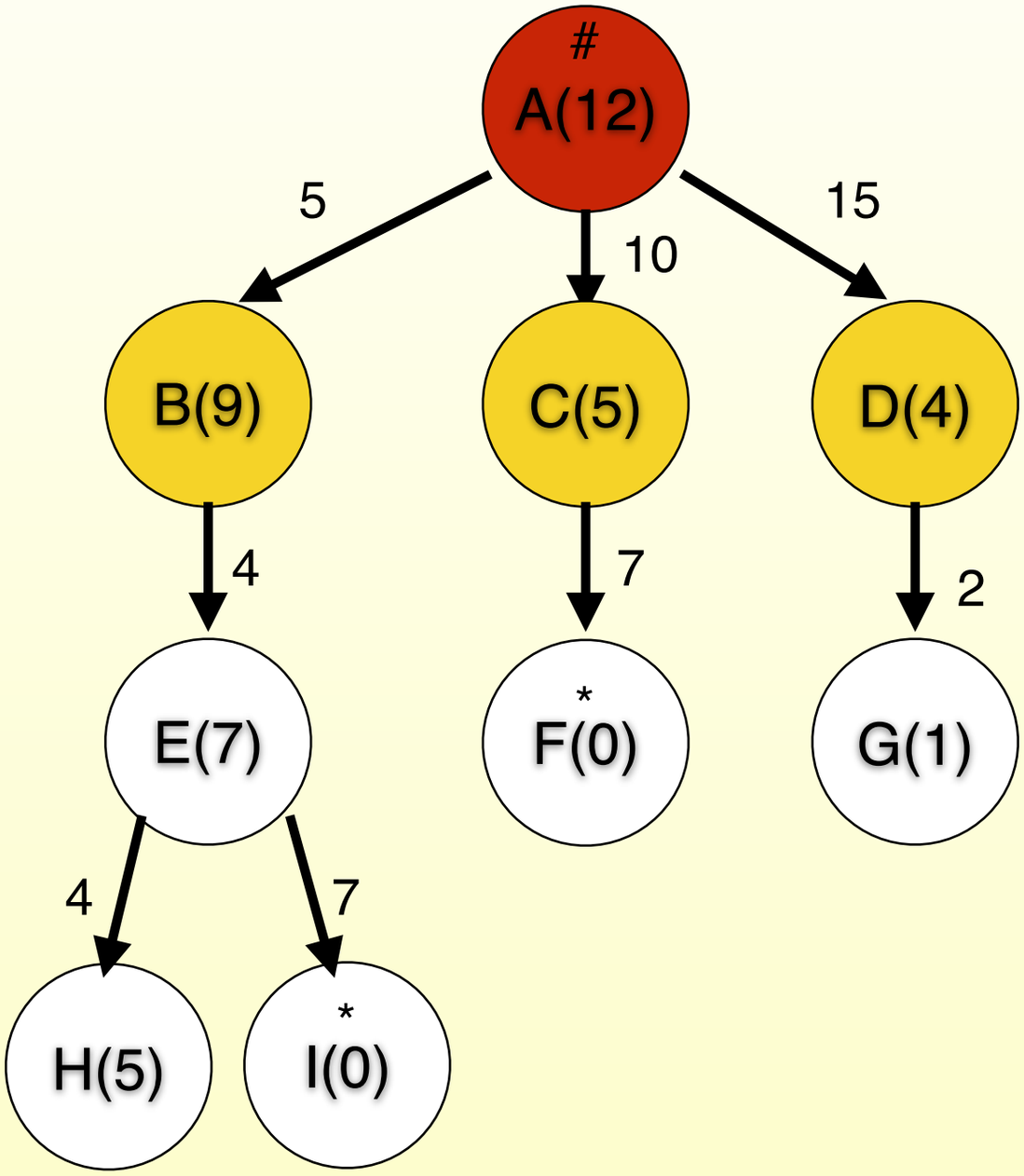
(h) Heuristic = h

Greedy Best-First Example



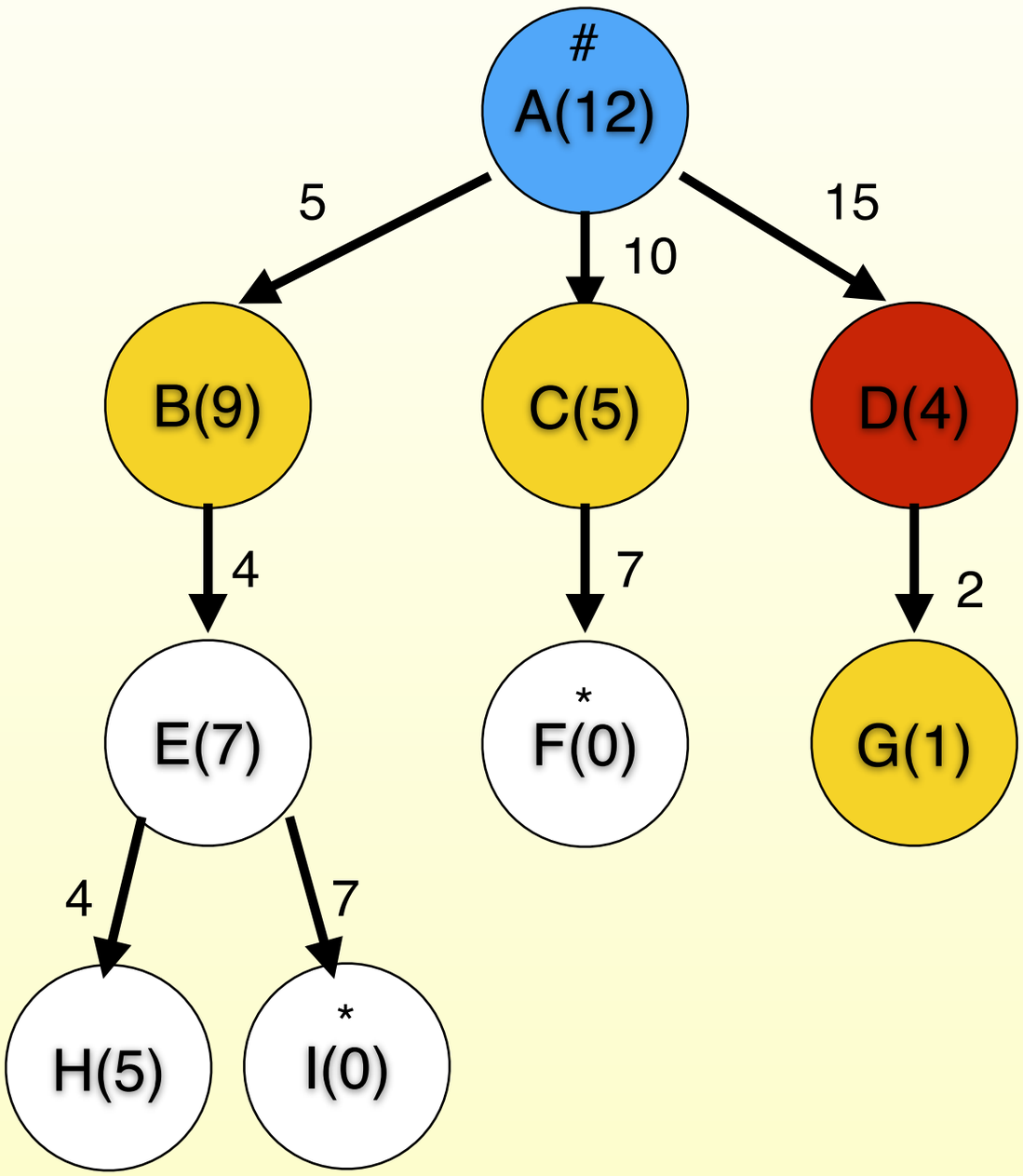
Node	f = h
A	12

Greedy Best-First Example



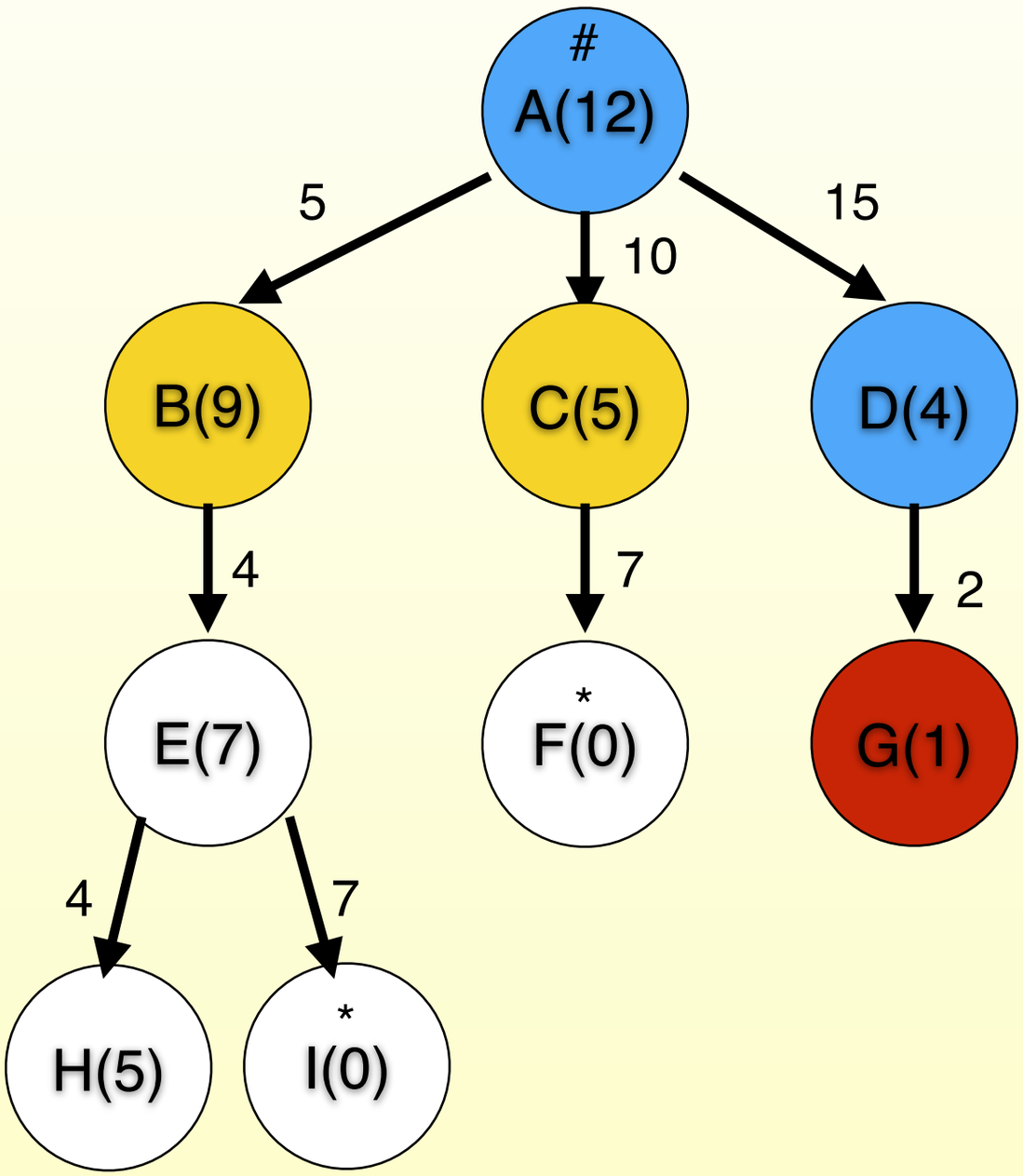
Node	f = h
D	4
C	5
B	9

Greedy Best-First Example



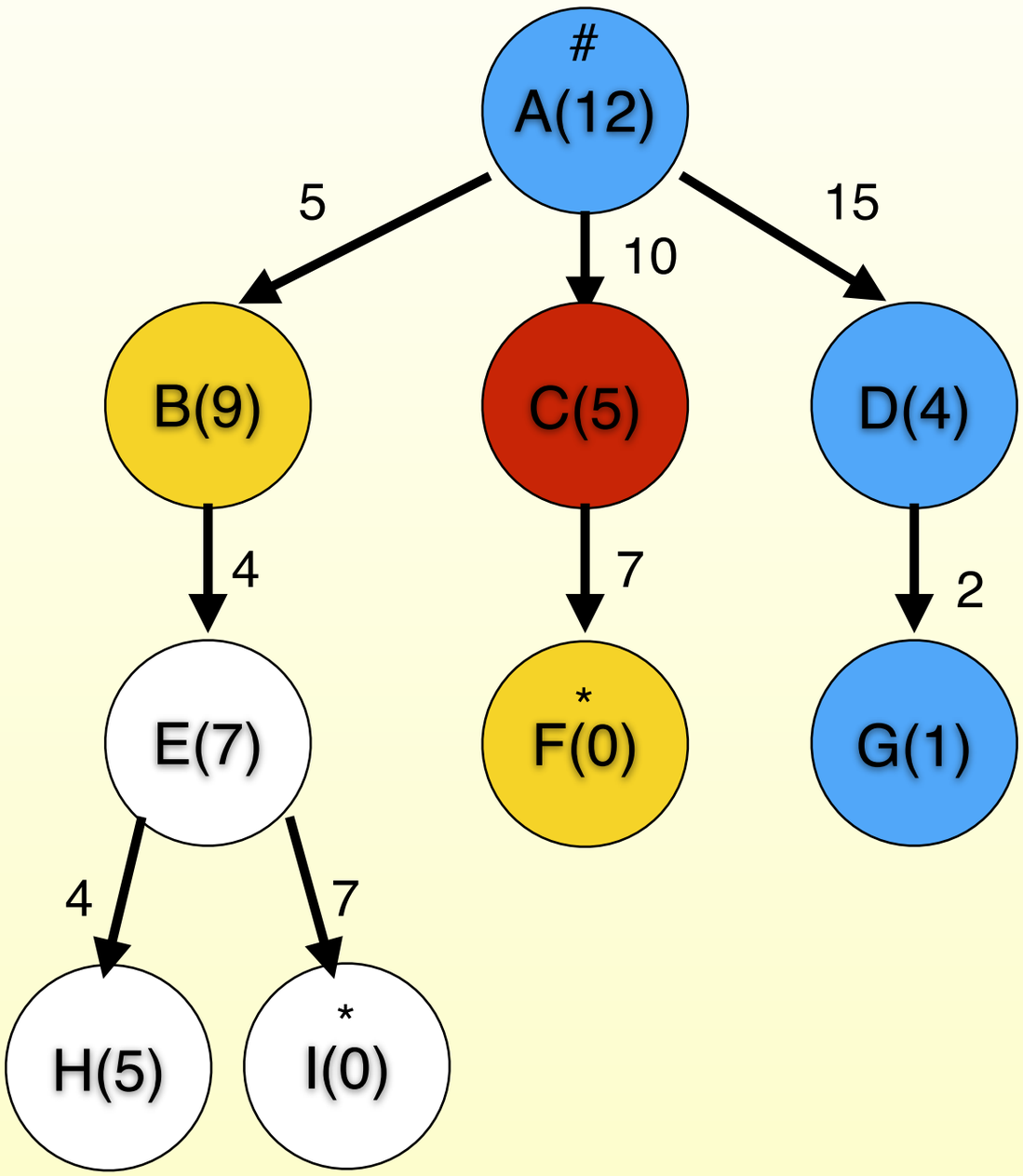
Node	f = h
G	1
C	5
B	9

Greedy Best-First Example



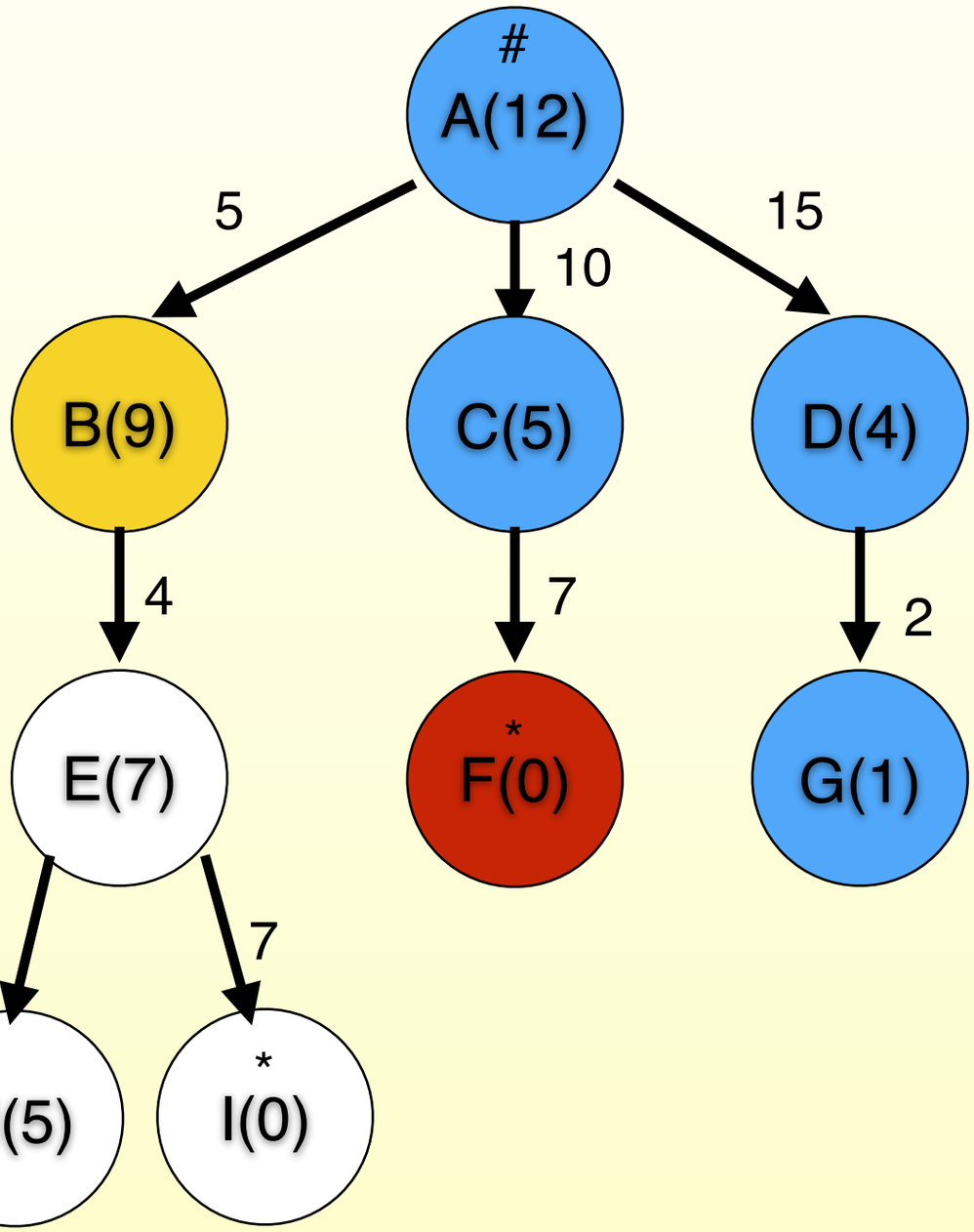
Node	f = h
C	5
B	9

Greedy Best-First Example



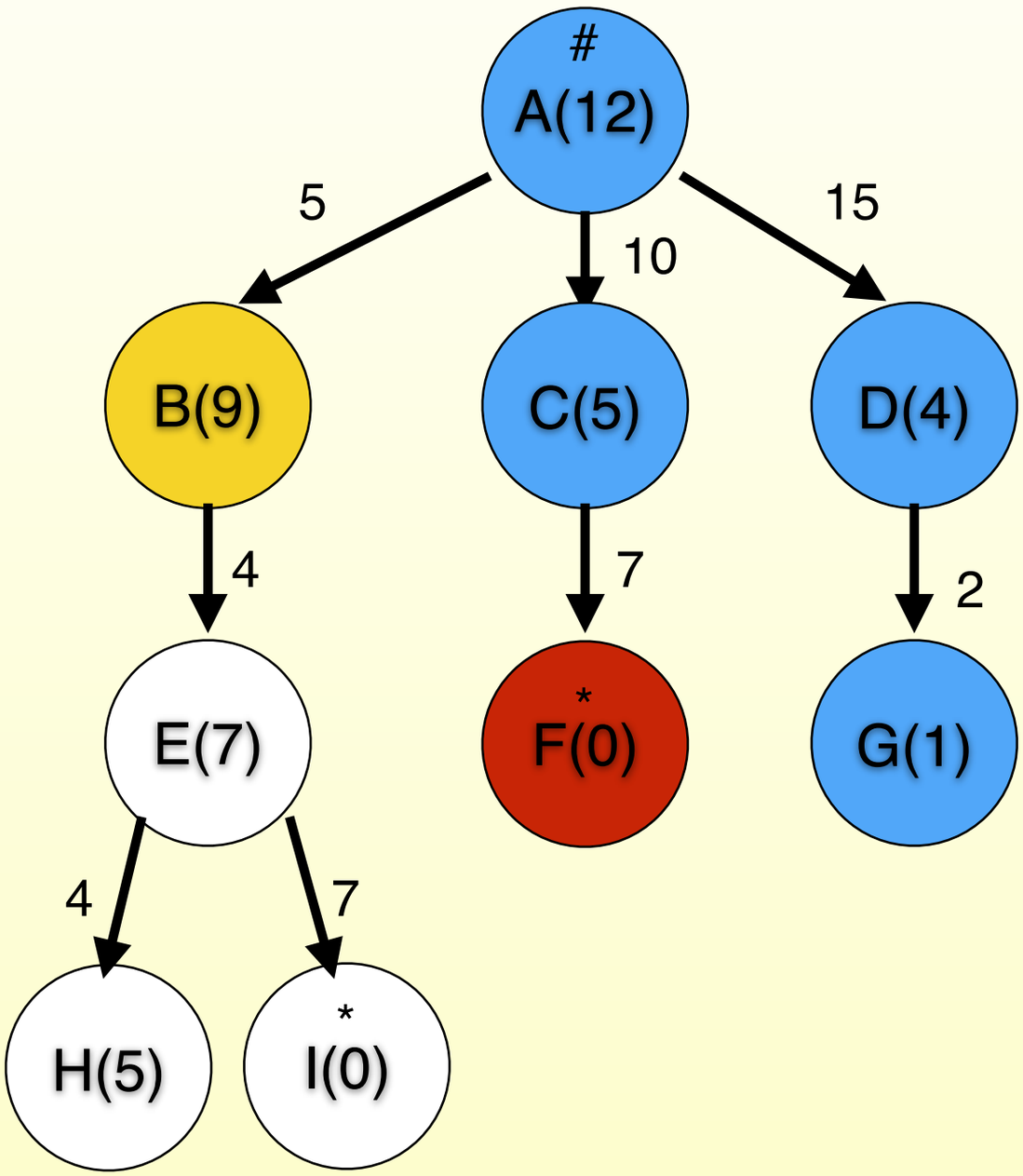
Node	f = h
F	0
B	9

Greedy Best-First Example



Node	f = h
B	9

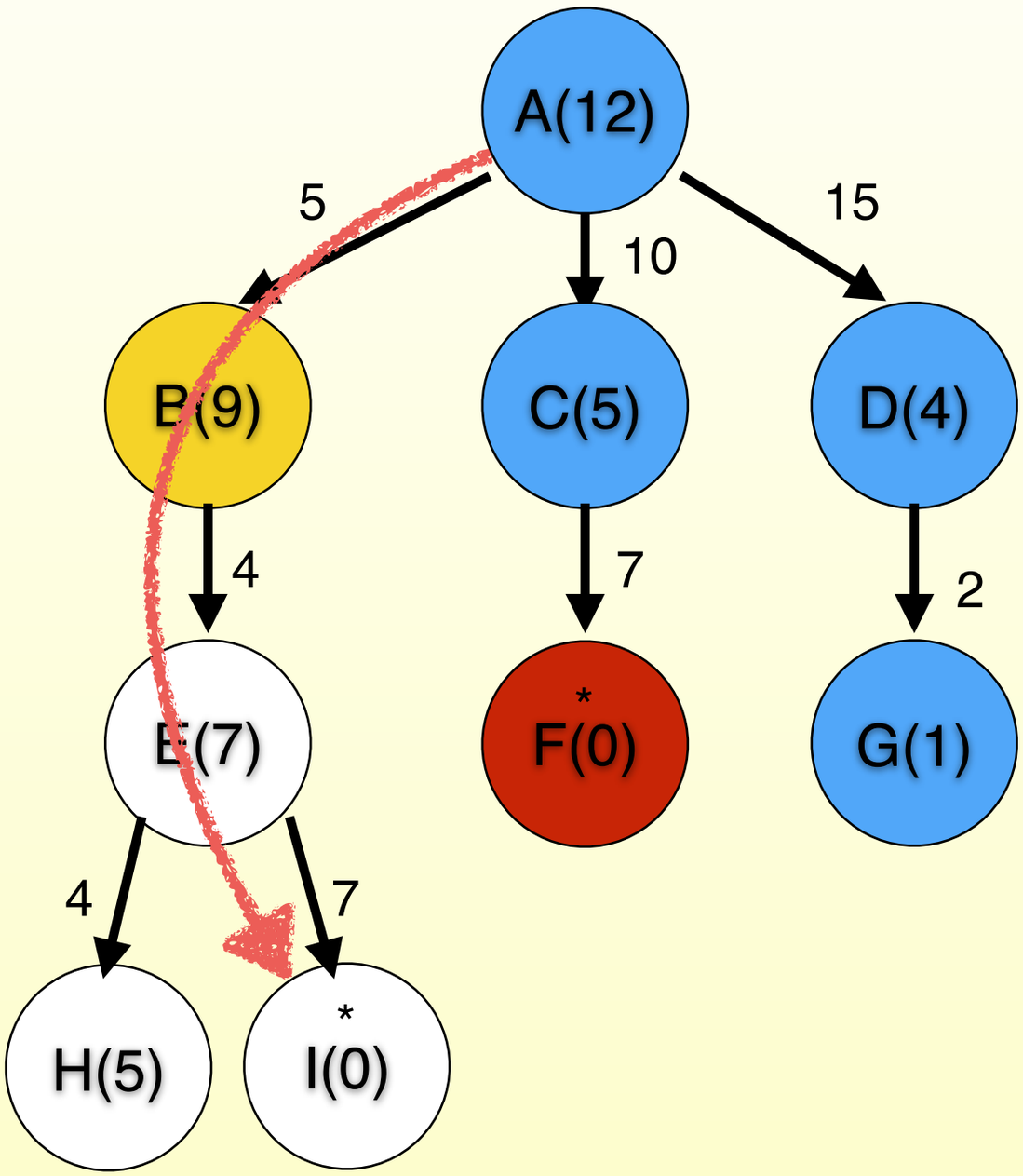
Greedy Best-First Example



Node	f = h
B	9

Goal is found agent returns path ACF with cost 17

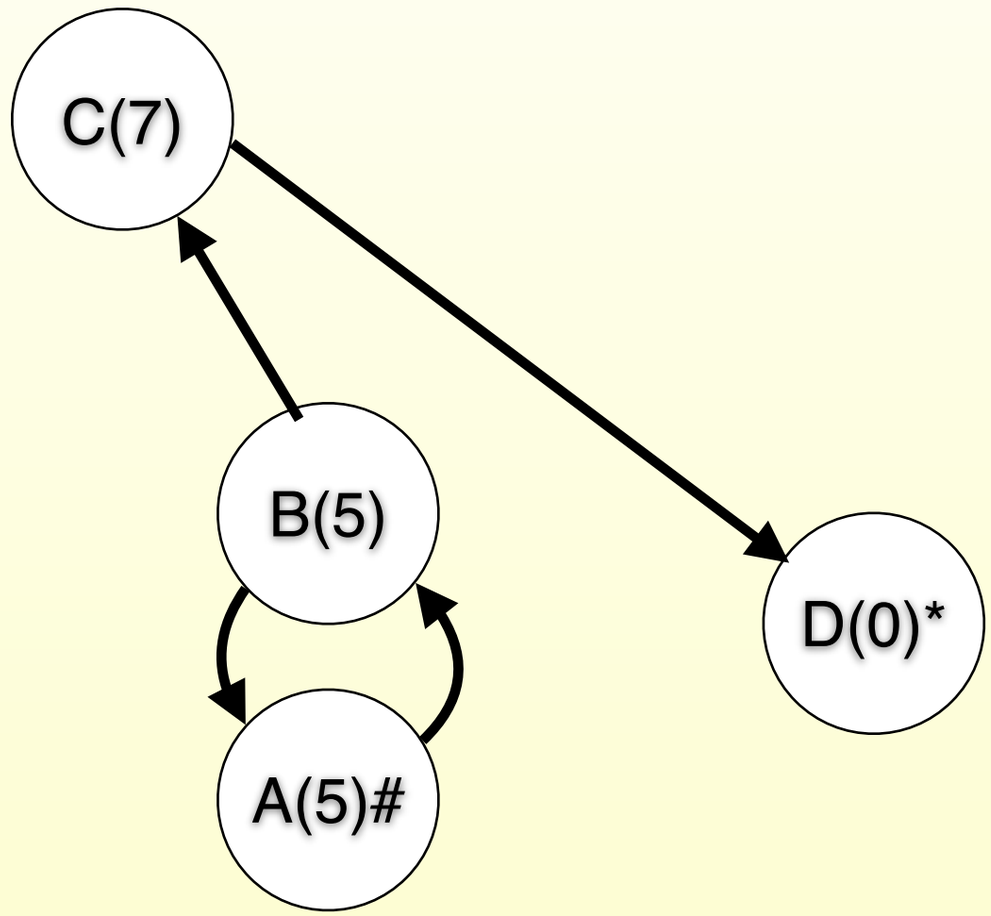
Greedy is not Optimal!



Node	f = h
B	9

A cheaper path, ABEI exists with cost 16!

Greedy is also incomplete!



Has the same problems with loops and infinite paths as Depth-First Search

Greedy Best-First Search

- **minimizes the estimated cost to a goal**
  - expand the node that seems to be closest to a goal
  - utilizes a heuristic function as evaluation function
    - *f(n) = h(n) =* estimated cost from the current node to a goal
    - heuristic functions are problem-specific
    - often straight-line distance for route-finding and similar problems
  - often better than depth-first, although worst-cases complexities are equal or worse (space)

```
function GREEDY-SEARCH(problem) returns solution
    return BEST-FIRST-SEARCH(problem, h)
```

Completeness	Time Complexity	Space Complexity	Optimality
no	b^m	b^m	no

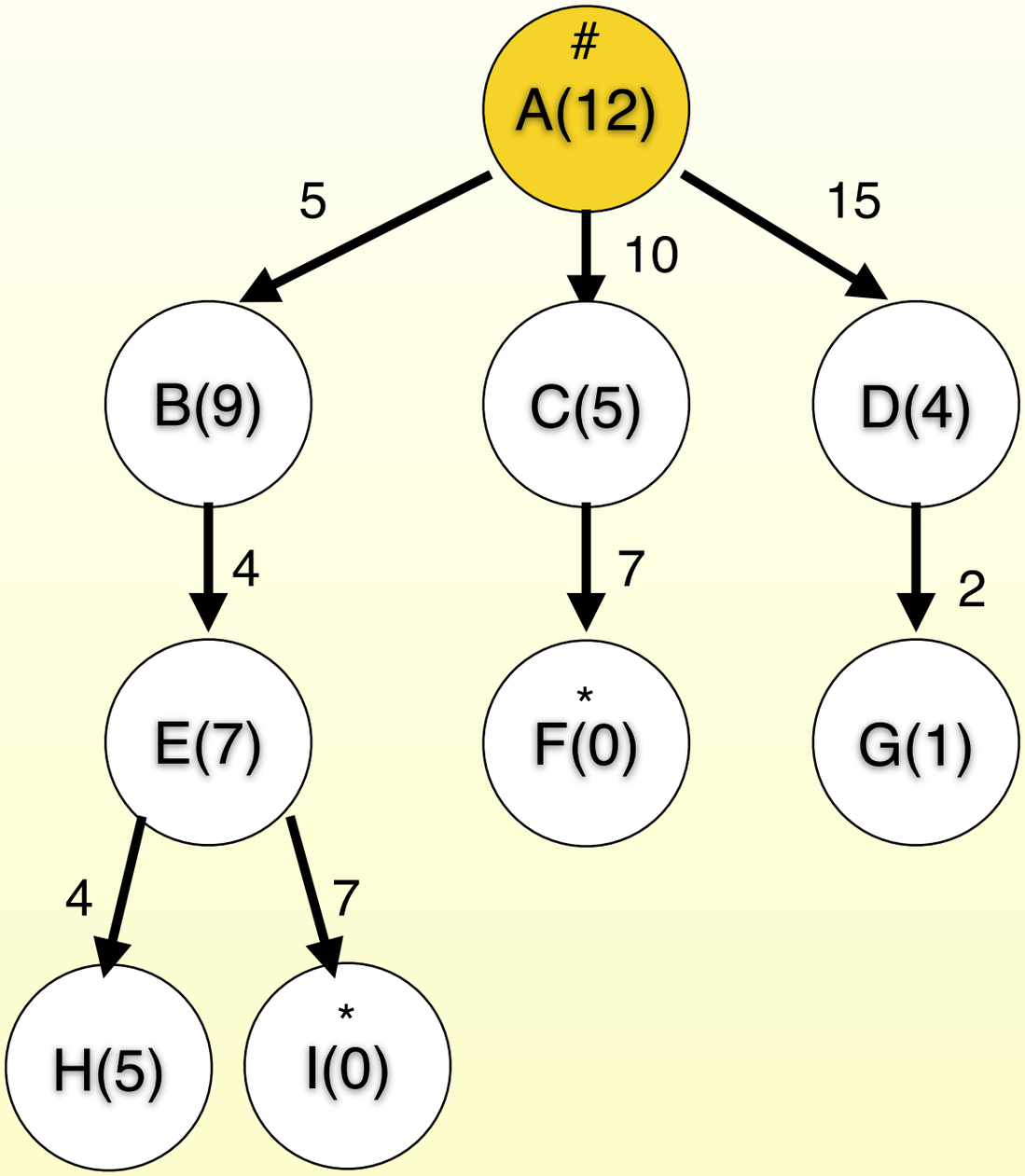
b: branching factor, d: depth of the solution, m: maximum depth of the search tree, l: depth limit

A* Search

- **combines greedy and uniform-cost search to find the (estimated) cheapest path through a prospective node to expand**
  - *f(n) = g(n) + h(n)*
    = path cost + estimated cost to the goal
  - very good search method, but still with (worst-case) complexity problems

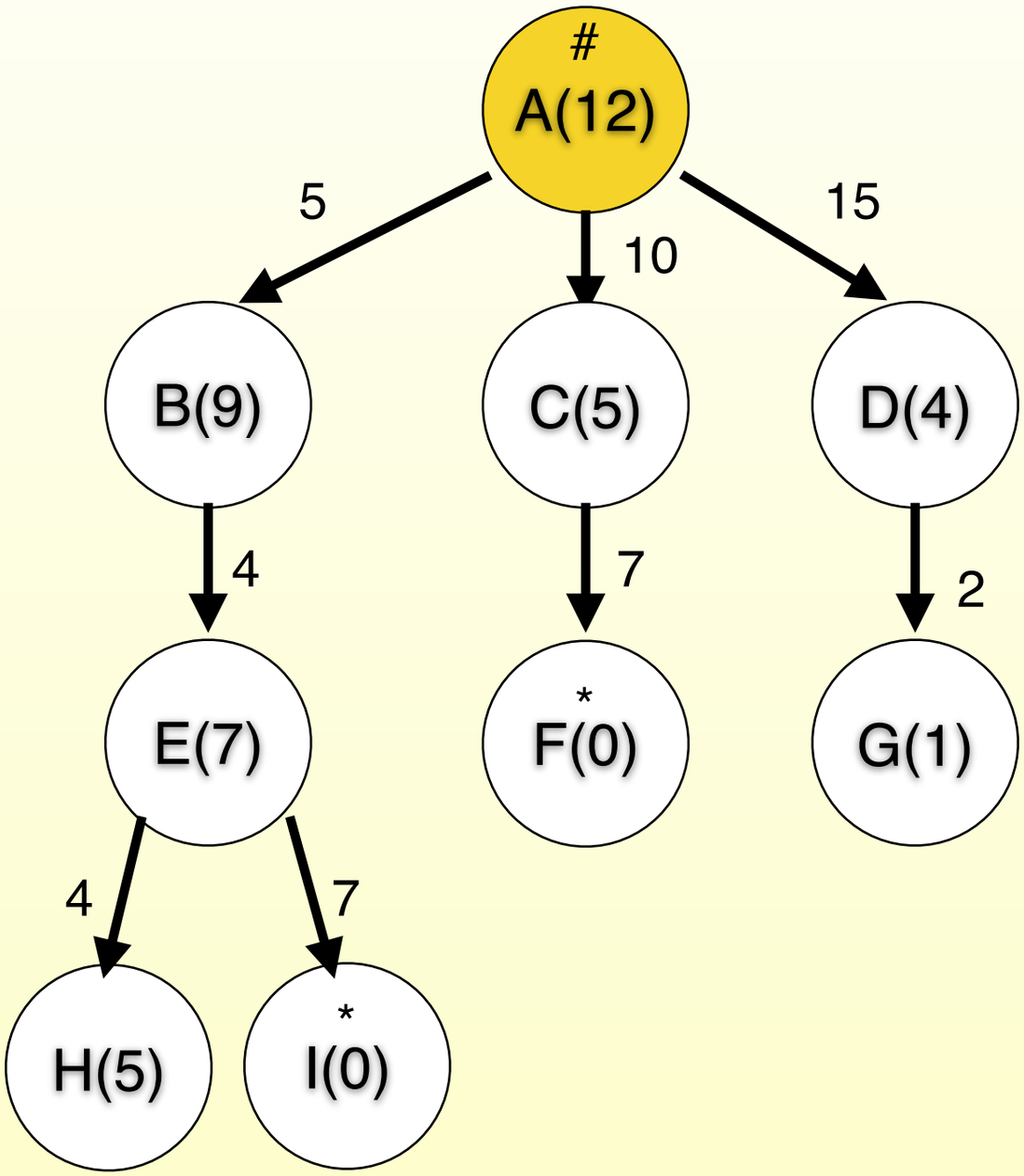
```
function A*-SEARCH(problem) returns solution

    return BEST-FIRST-SEARCH(problem, g+h)
```

A* search example



Node	f	g	h
A	12	0	12

A* search example



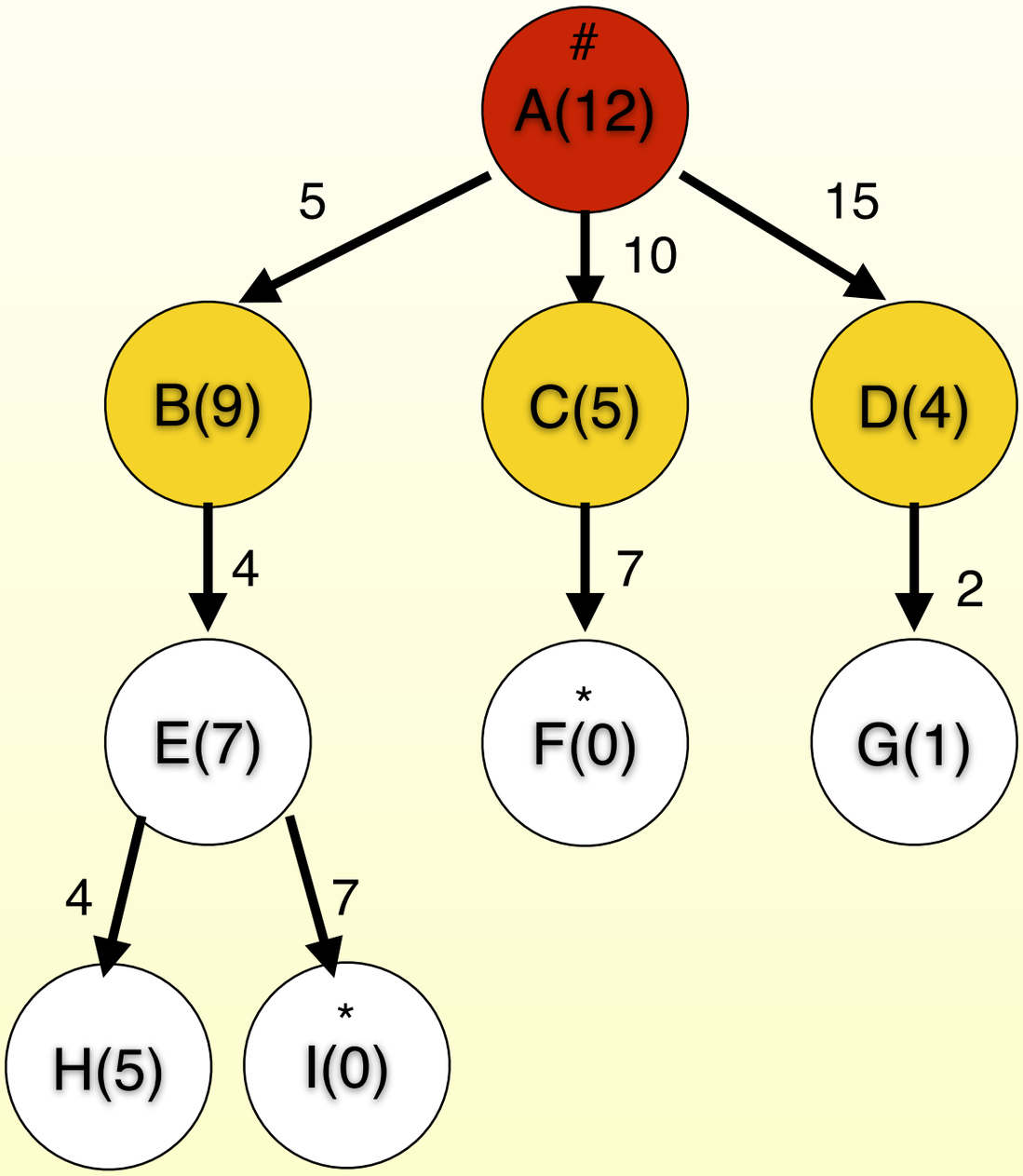
g = How much did it actually cost me to get here?

f = How much do I think it will cost me to reach the goal via this node?

h = How far do I think I am from the goal?

Node	f	g	h
A	12	0	12

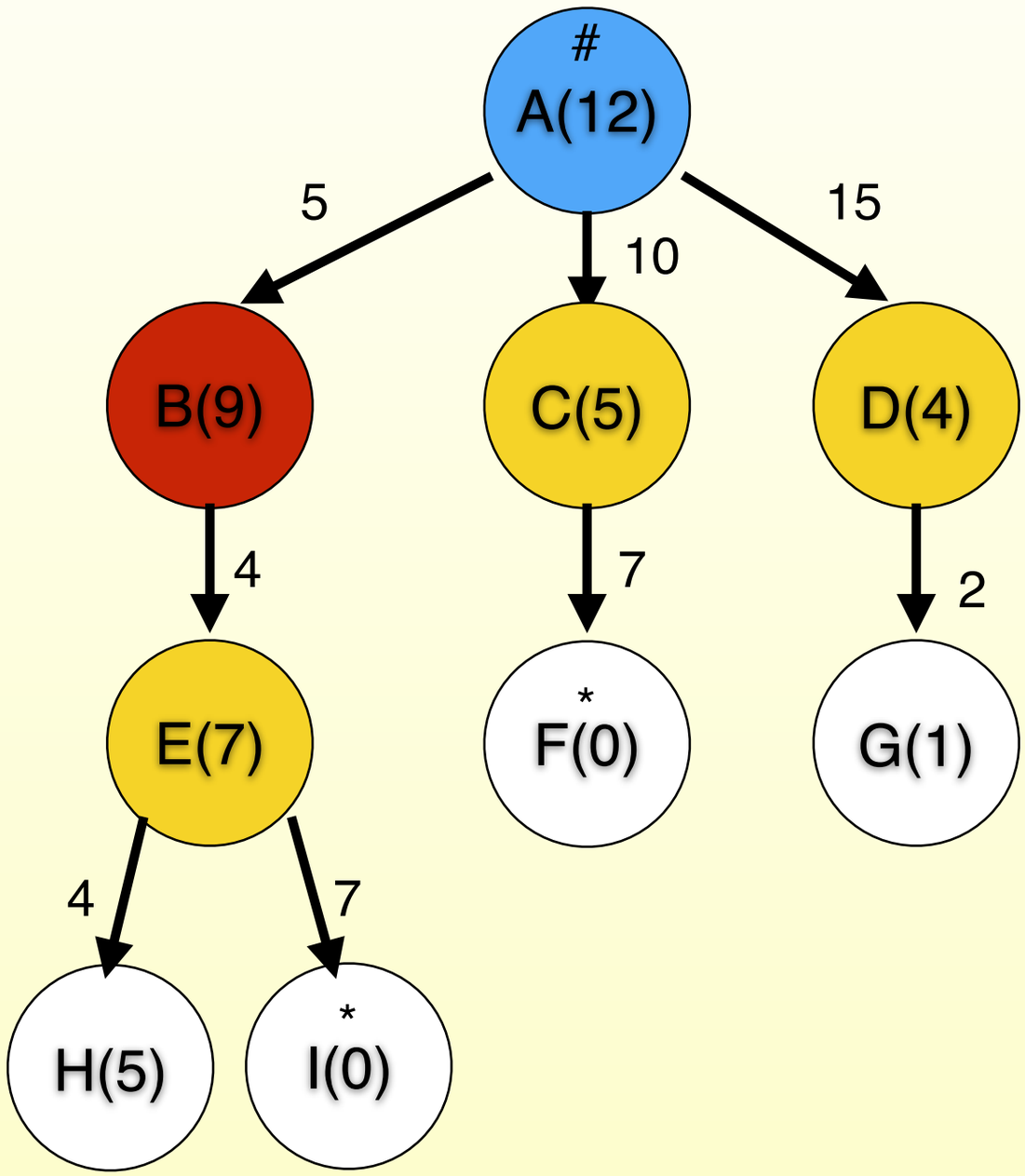
A* search example



Current node: A (g=0)

Node	f	g	h
B	14	5	9
C	15	10	5
D	19	15	4

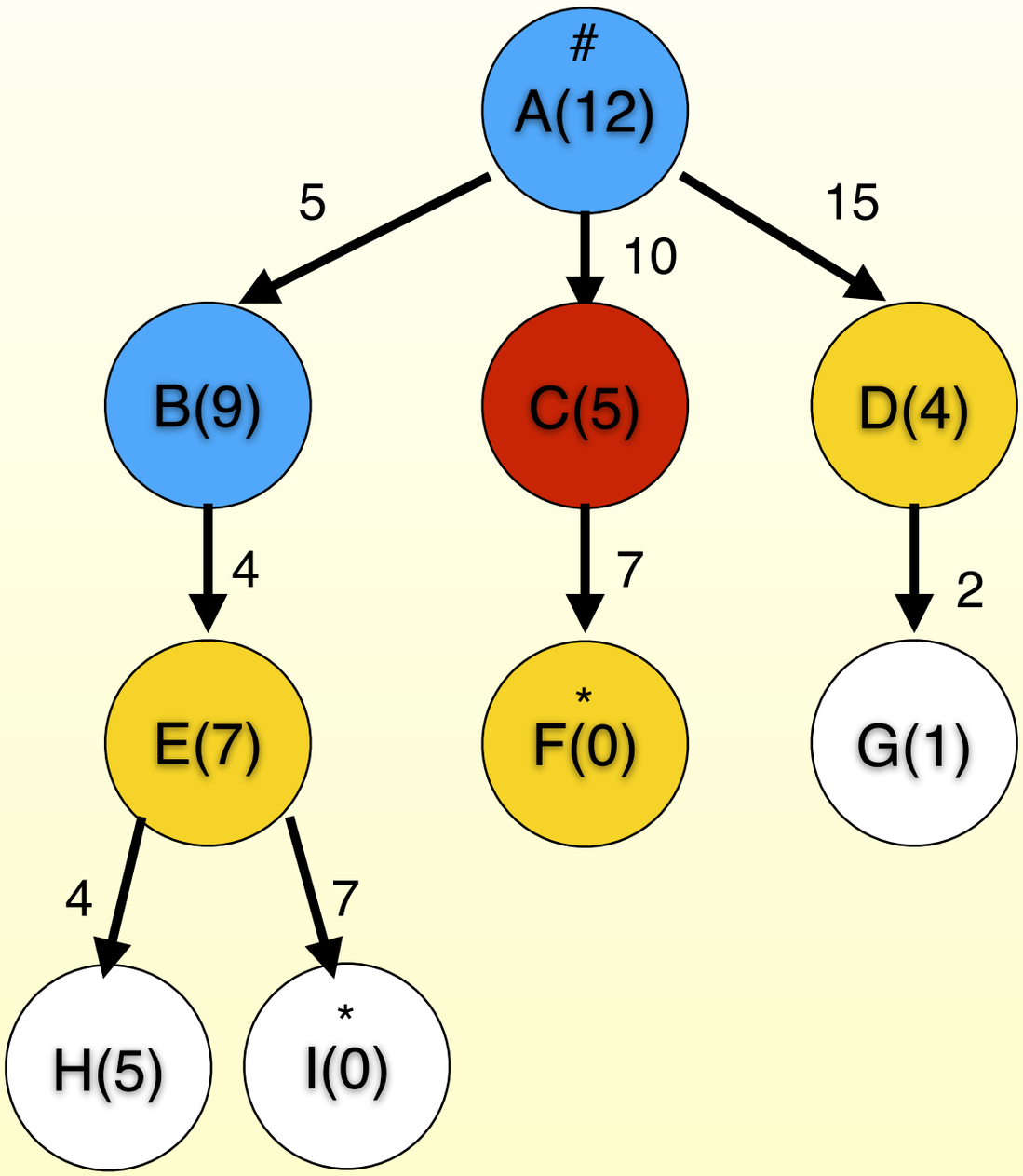
A* search example



Current node: B (g=5)

Node	f	g	h
C	15	10	5
E	16	5+4=9	7
D	19	15	4

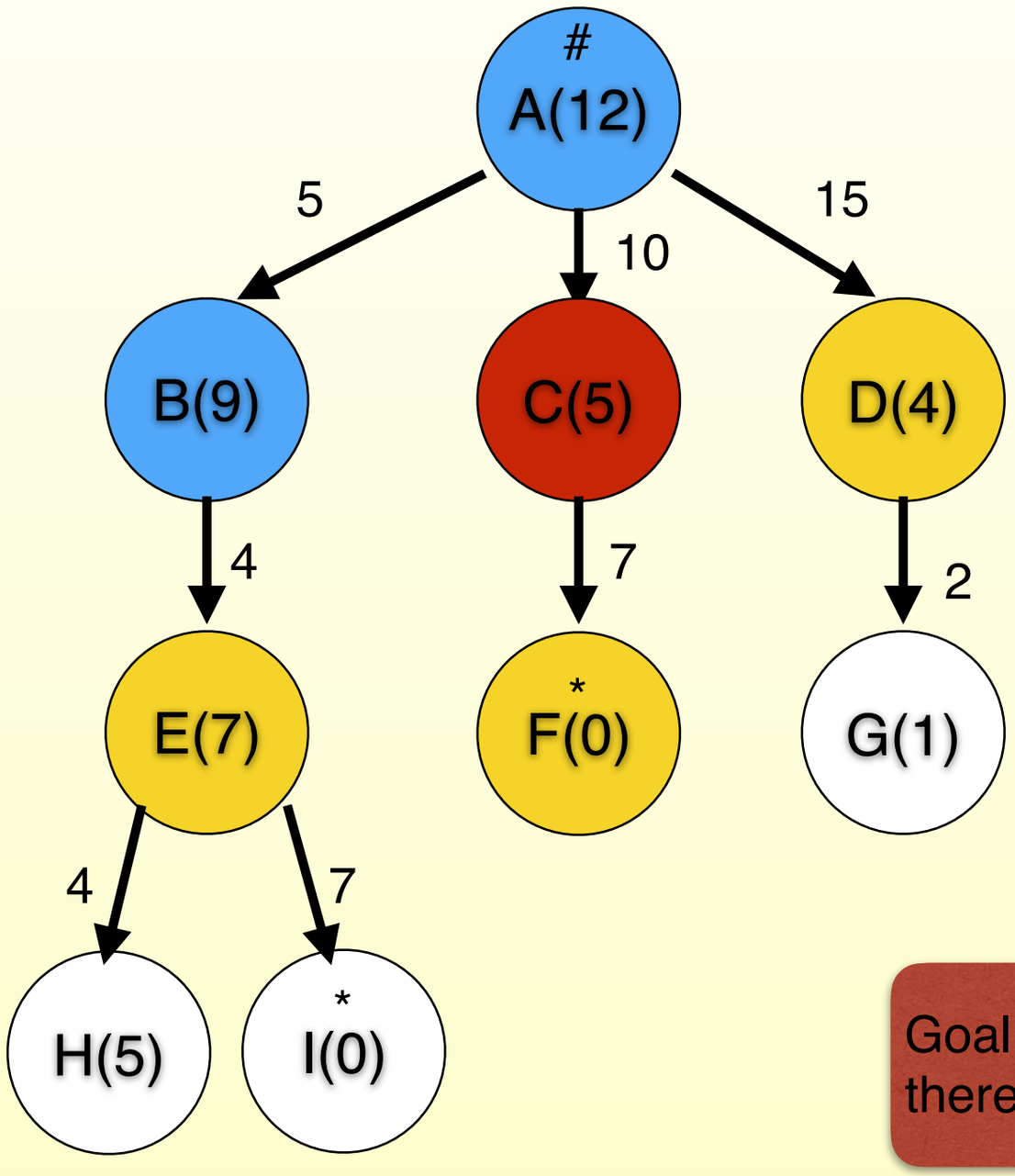
A* search example



Current node: C (g=10)

Node	f	g	h
E	16	9	7
F	17	10+7=17	0
D	19	15	4

A* search example

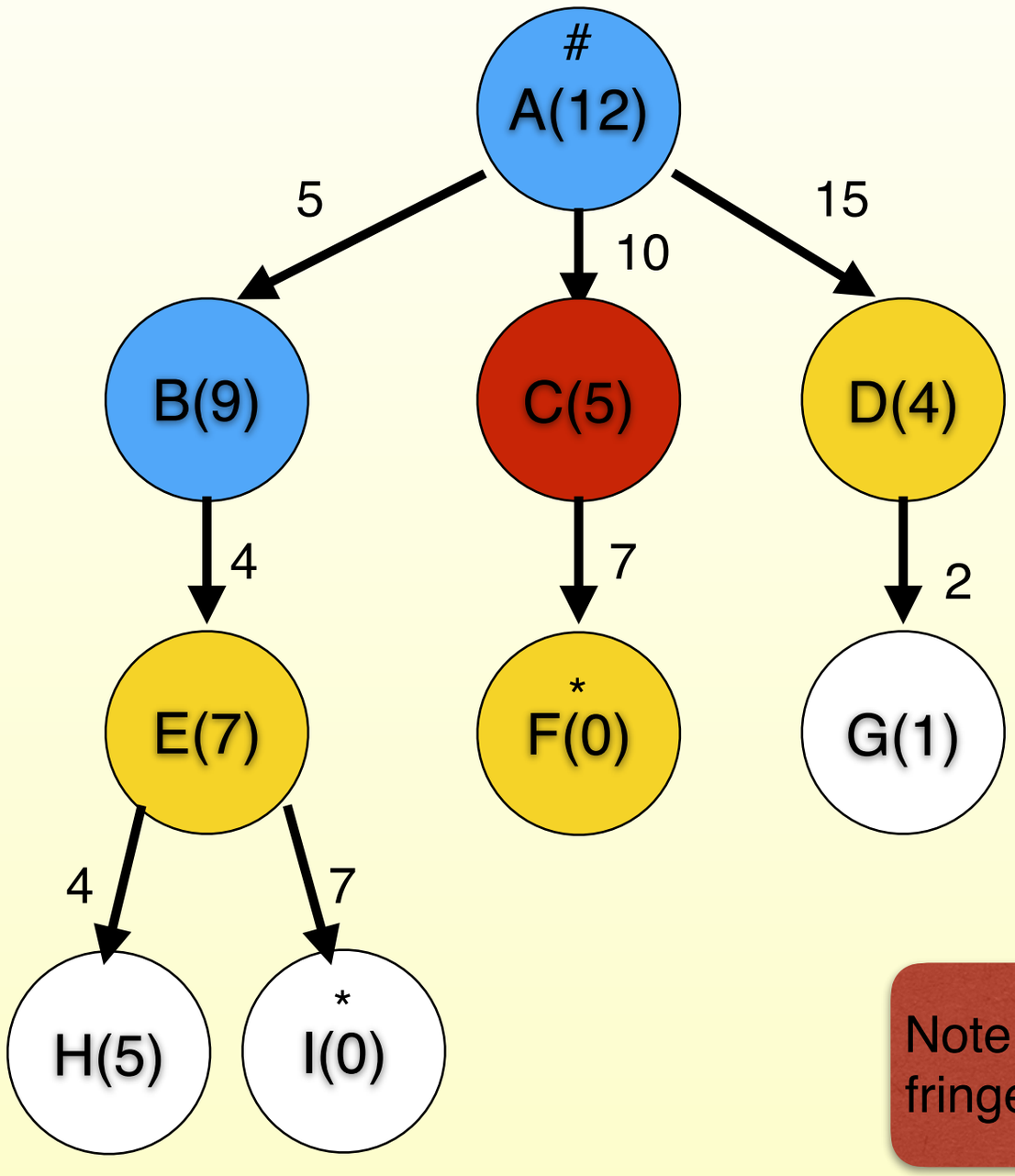


Current node: C (g=10)

Node	f	g	h
E	16	9	7
F	17	10+7=17	0
D	19	15	4

Goal state F is visible but we cannot explore it yet because there might still be a better solution through E

A* search example

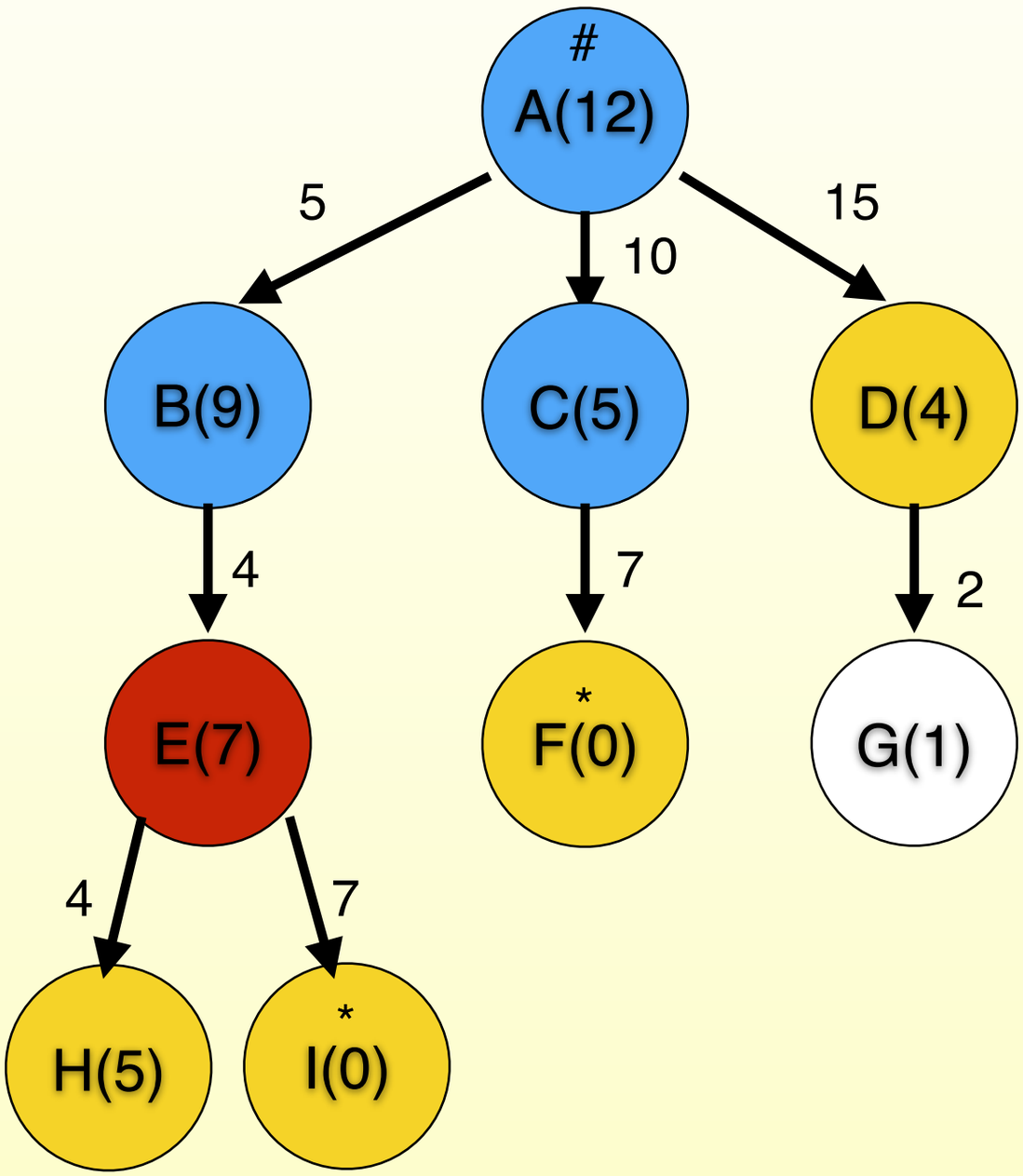


Current node: C (g=10)

Node	f	g	h
E	16	9	7
F	17	10+7=17	0
D	19	15	4

Note that at this point we could drop (prune) D from the fringe

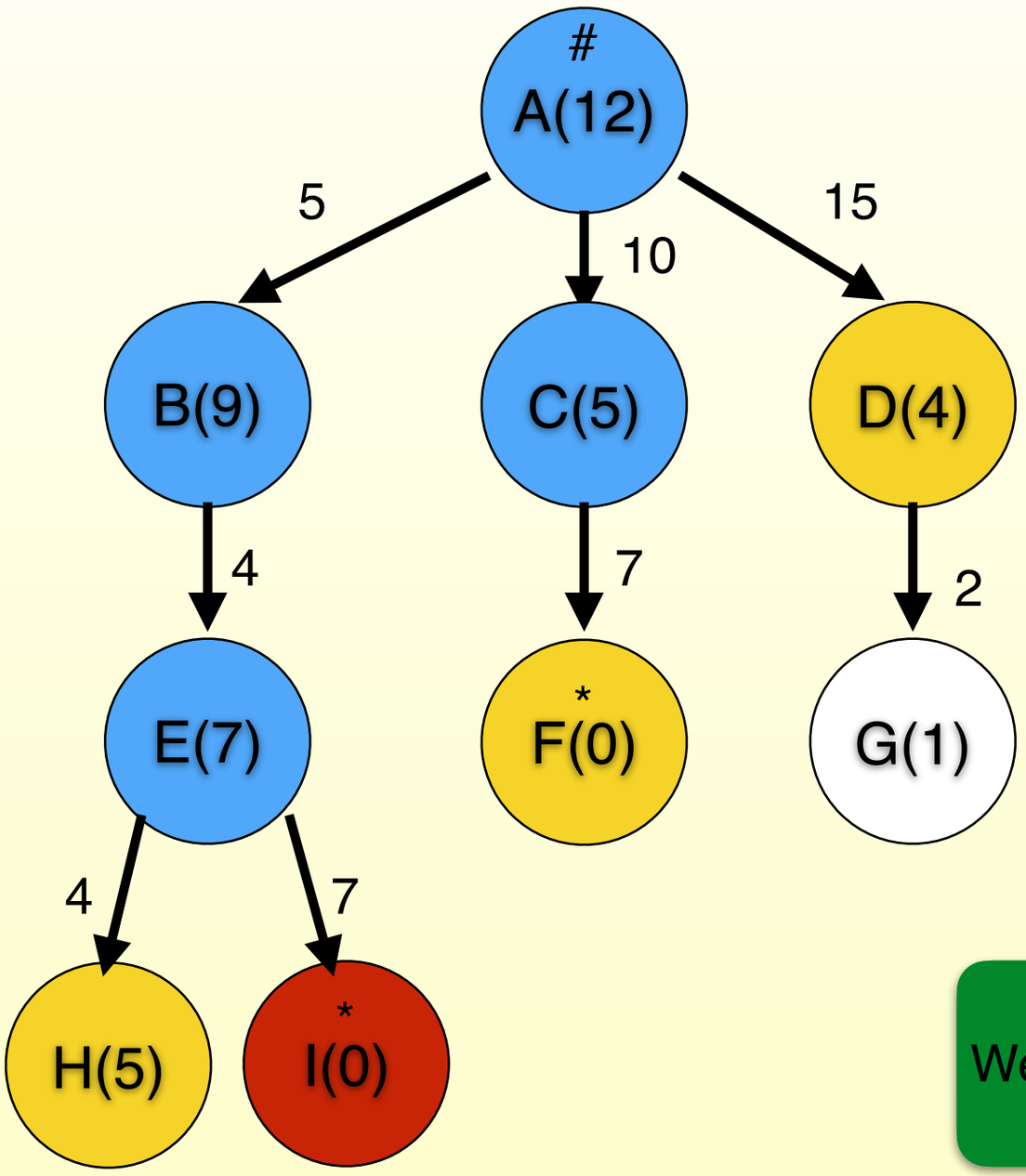
A* search example



Current node: E (g=9)

Node	f	g	h
I	16	9+7=16	0
F	17	10+7=17	0
H	18	9+4=13	5
D	19	15	4

A* search example



Current node: E (g=9)

Node	f	g	h
I	16	9+7=16	0
F	17	10+7=17	0
H	18	13	5
D	19	15	4

We find the goal state I, and return the optimal path ABEI

MARIO: 1624 COINS 11 DIFFICULTY 15 TIME 148

FPS: 24
Attempt: 1 of 1
AStarAgent
Selected Actions:

WorldPause False

LEFT

SPEED

MARIO: 1624 COINS 11 DIFFICULTY 15 TIME 148

FPS: 24
Attempt: 1 of 1
AStarAgent
Selected Actions:

WorldPause False

LEFT

SPEED

A* Search

- **combines greedy and uniform-cost search to find the (estimated) cheapest path through the current node**
  - *f(n) = g(n) + h(n)*
    = path cost + estimated cost to the goal
    - *g(n)* and *h(n)* may have to be balanced (normalized)
  - heuristics must be admissible to guarantee optimality (see next slides)
  - very good search method, but still with (worst-case) complexity problems

```
function A*-SEARCH(problem) returns solution

    return BEST-FIRST-SEARCH(problem, g+h)
```

Completeness	Time Complexity	Space Complexity	Optimality
yes	b^d	b^d	yes

b: branching factor, d: depth of the solution, m: maximum depth of the search tree, l: depth limit

Admissible Heuristic

- **A heuristic is admissible if it never overestimates the cost to the goal.**
  - In other words, it is “optimistic”

- **Examples of admissible heuristics**
  - Euclidean distance, in most maze navigation problems
  - Number of misplaced tiles in an 8-puzzle (and many puzzles where you move 1 piece at a time)
    - Also called Hamming Distance

Optimality of A*

With an admissible heuristic:

- **A* will find the optimal solution**
  - the first solution found is the optimal one
- **A* is optimally efficient**
  - no other optimal algorithm is guaranteed to expand fewer nodes than A*
- **A* is not always "the best" algorithm**
  - optimality refers to the expansion of nodes
    - other criteria might be more relevant
    - described in abstract Big-O terminology
  - it generates and keeps all nodes in memory
    - improved in variations of A*

Consistence

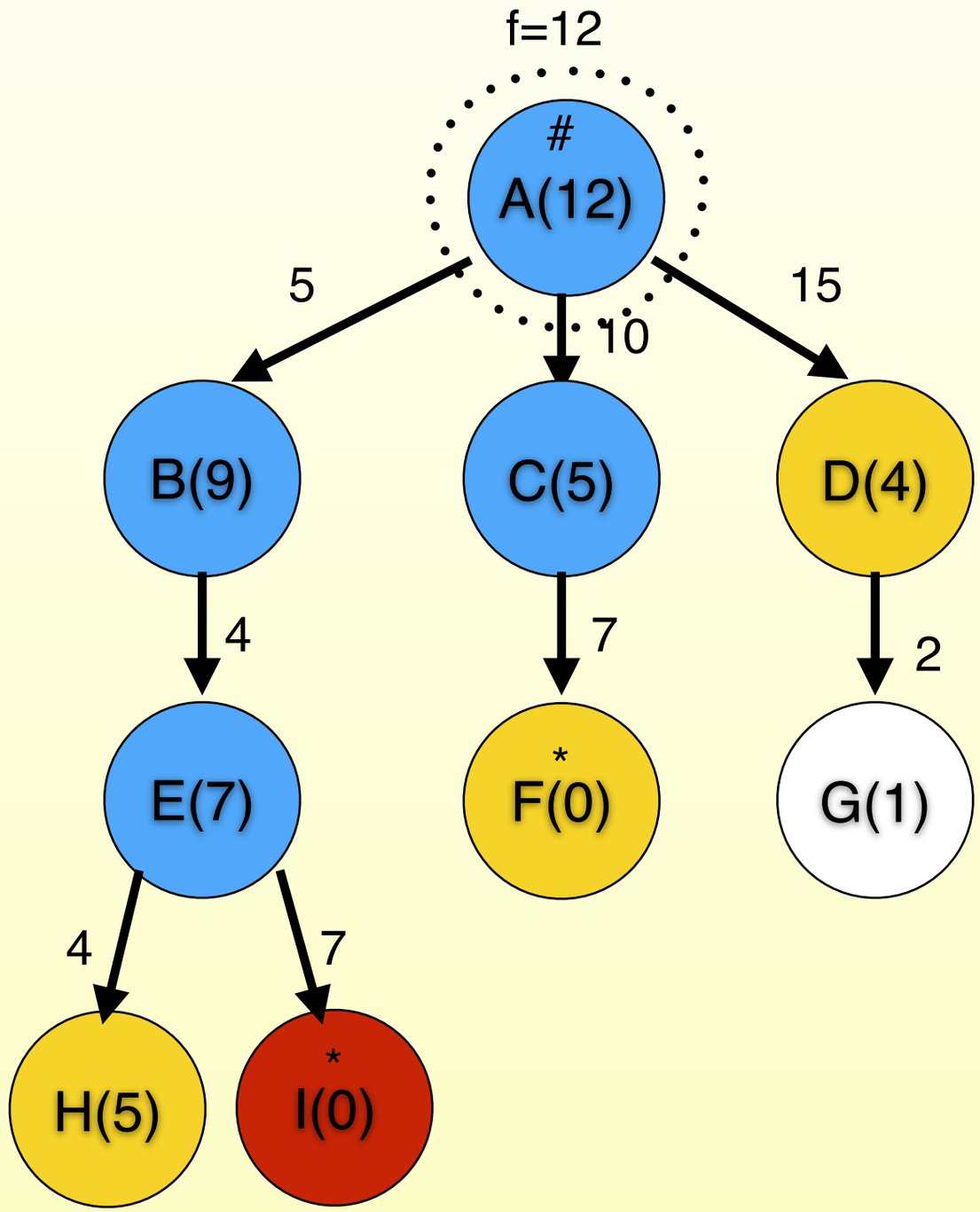
- **A related (stronger) property than admissibility is is consistency:**
  - For all children c of n, F(n) < = F(c) + edge(n,c)
  - A form of triangle inequality

- **With a consistent heuristic, we the first time we reach any node is always through the optimal path**
  - This lets us expand and store even fewer states (since we don't have to worry about duplicates)

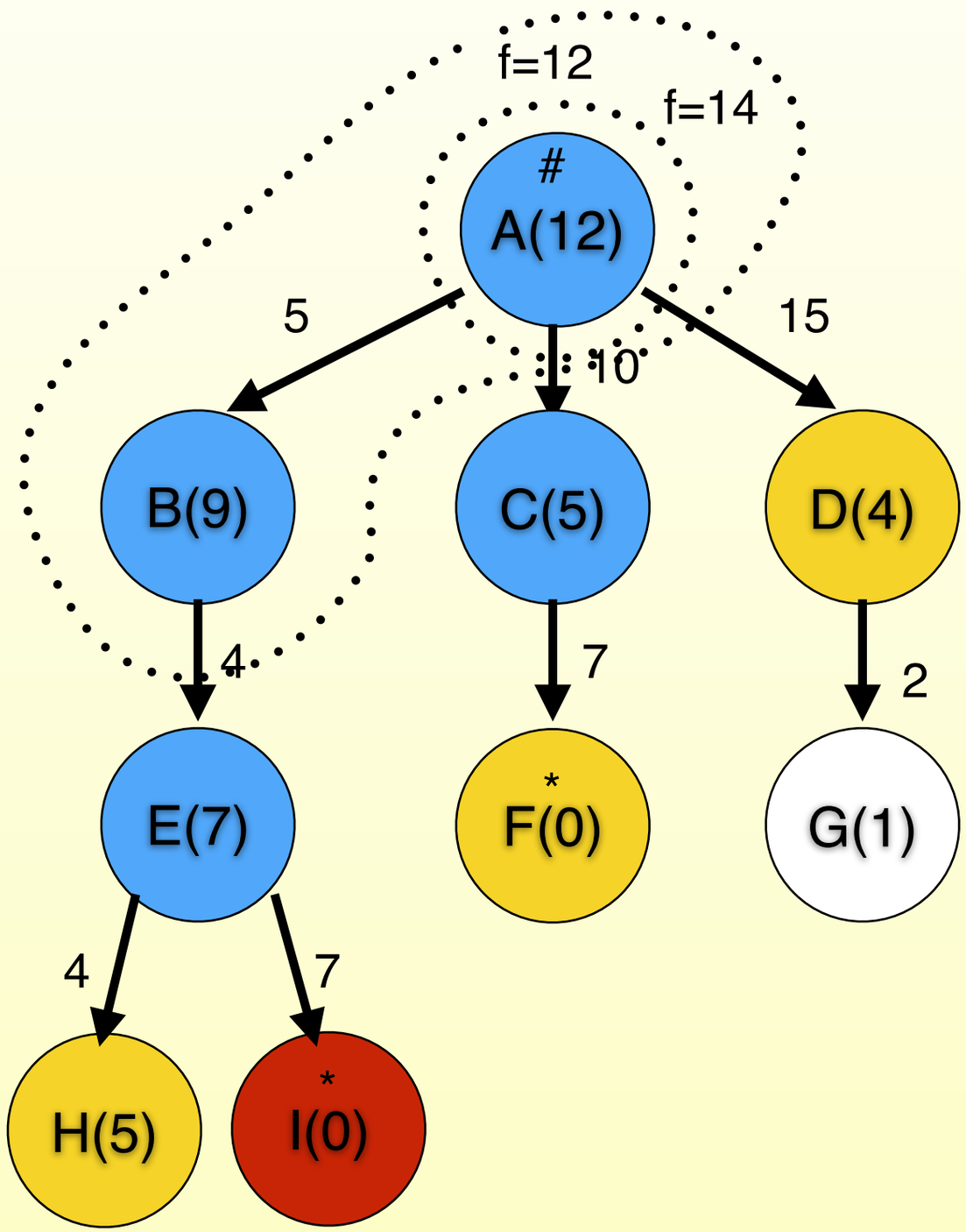
Contours

- **With a consistent heuristic, value of f never decreases along any path starting from the initial node**
  - also known as *monotonicity* of the function
  - almost all admissible heuristics show monotonicity
    - those that don't can be modified through minor changes
- **this property can be used to draw contours**
  - regions where the f-cost is below a certain threshold
  - with uniform cost search (h = 0), the contours are circular
  - the better the heuristics h, the narrower the contour around the optimal path

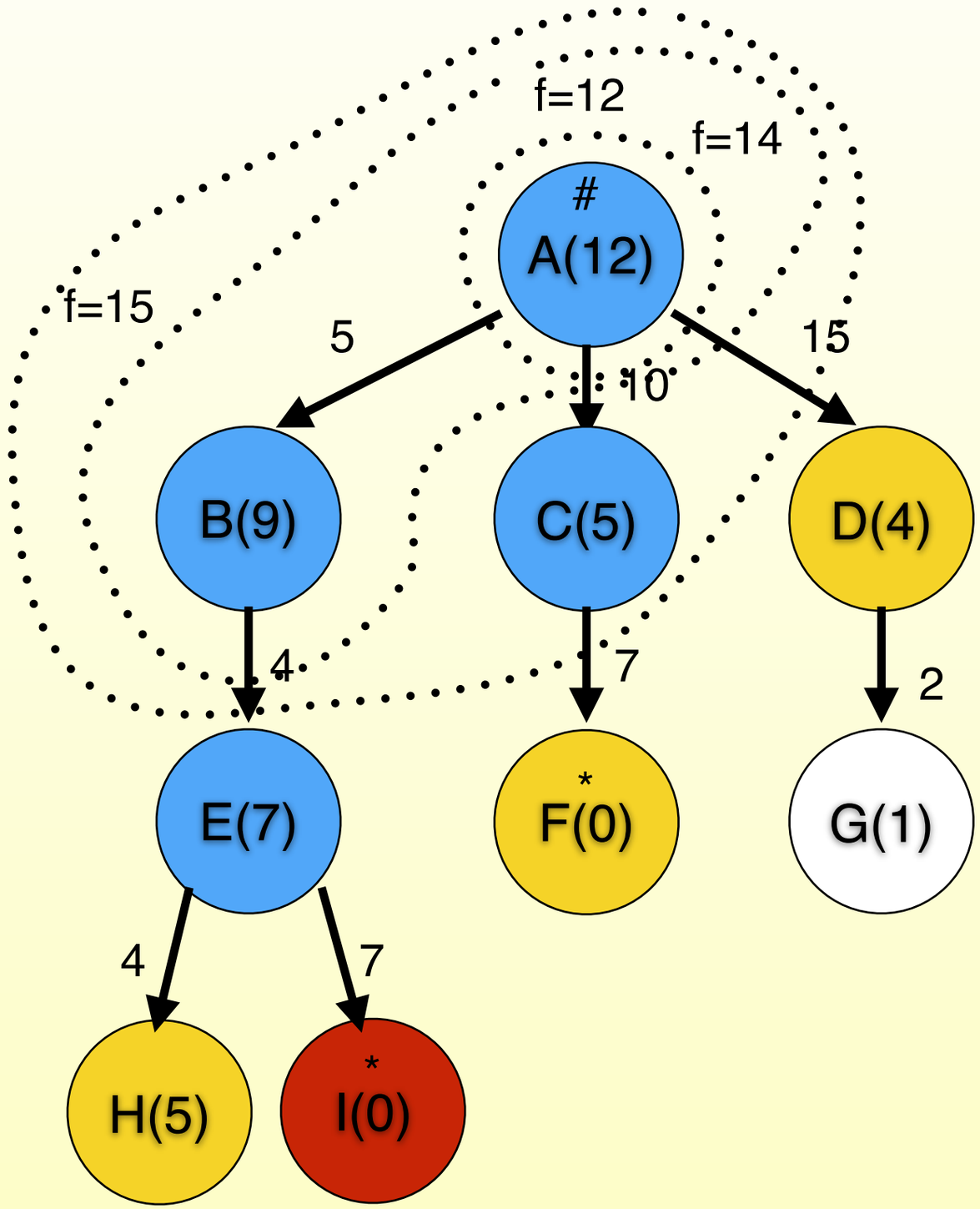
Visualizing contours



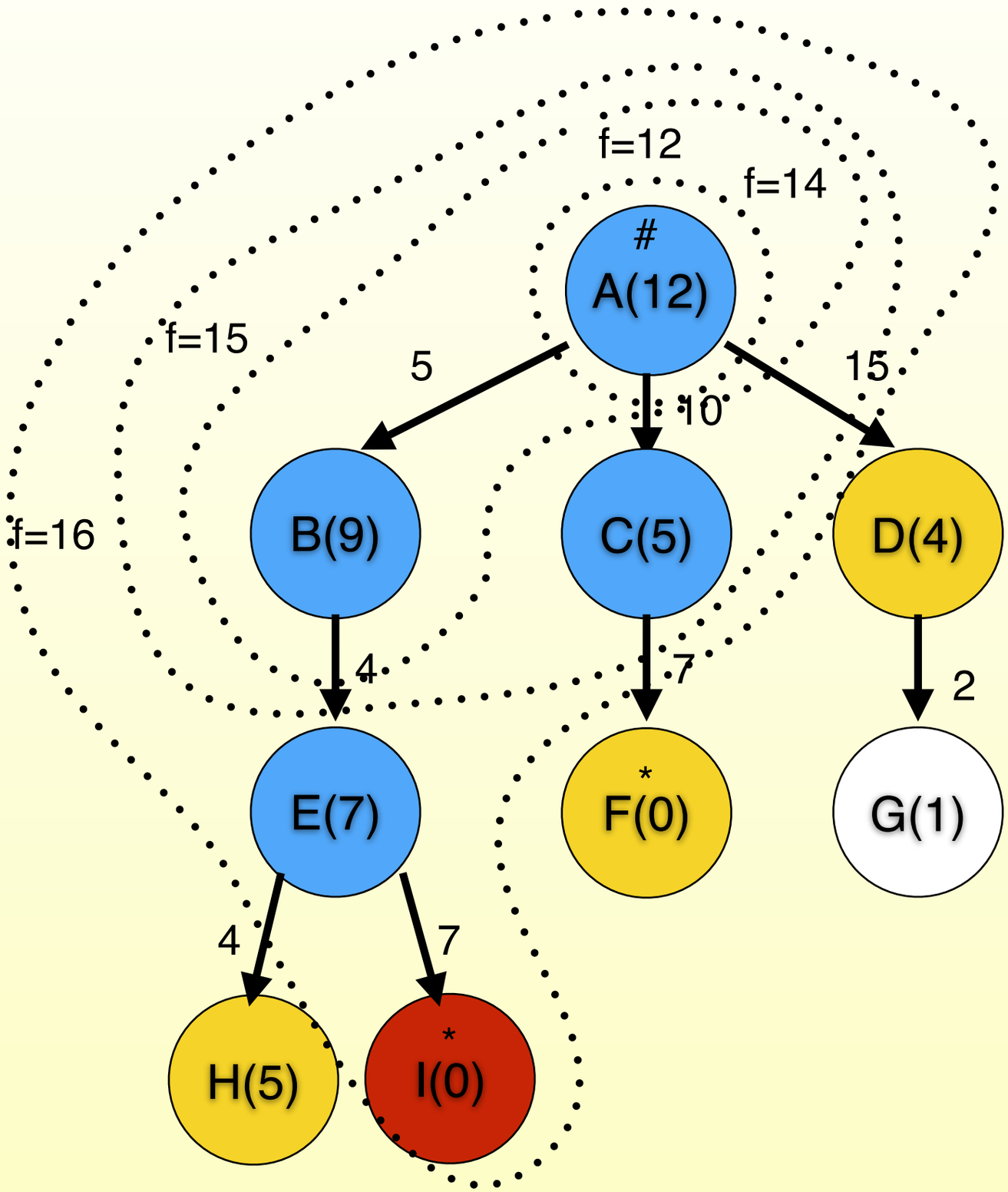
Visualizing contours



Visualizing contours



Visualizing contours



Complexity of A*

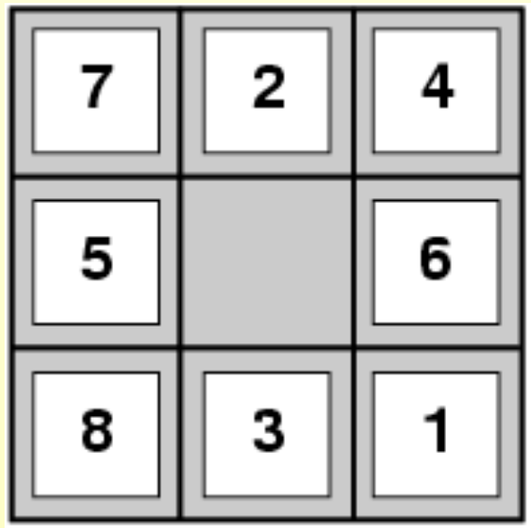
- **the number of nodes within the goal contour search space is still exponential**
  - with respect to the length of the solution
  - better than other algorithms, but still problematic
- **frequently, space complexity is more severe than time complexity**
  - A* keeps all generated nodes in memory
  - Memory-bounded variations (e.g. Iterative deepening A*) can alleviate the problem

Heuristics for Searching

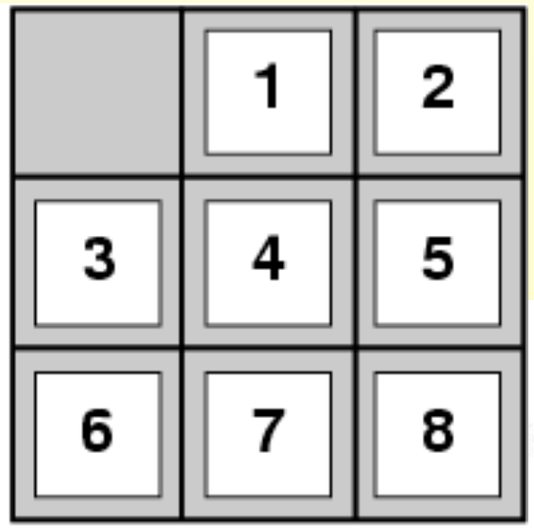
- **for many tasks, a good heuristic is the key to finding a solution**
  - prune the search space
  - move towards the goal
- **In general, a heuristic with higher (more accurate) values yields better search efficiency than a heuristic with lower values**
  - As long as they are both admissible / consistent
  - If for all n, h1(n) <= h2 (n), we say h2 *dominates* h1
  - This means A* using h2 expands <= nodes than A* using h1

8-Puzzle Heuristics

- **level of difficulty**
  - around 20 steps for a typical solution
  - branching factor is about 3
    - exhaustive search would be 3^20 = 3.5 * 10^9
  - 9!/2 = 181,440 different reachable states
    - distinct arrangements of 9 squares
- **candidates for heuristic functions**
  - number of tiles in the wrong position
  - sum of distances of the tiles from their goal position
    - city block or Manhattan distance



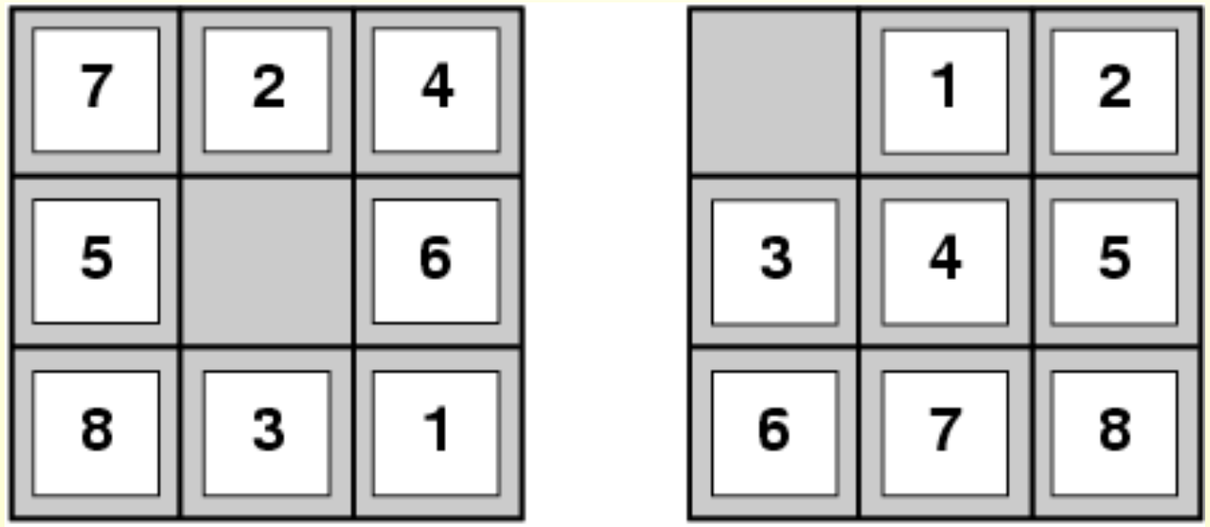
Start State



Goal State

Admissible heuristics

- **E.g., for the 8-puzzle:**
  - h1(n) = number of misplaced tiles
  - h2(n) = total Manhattan distance
  - (i.e., no. of squares from desired location of each tile)



  - h1(S) = 8
    - there are only 8 tiles, the empty space doesn't count as tile
  - h2(S) = 3+1+2+2+2+3+3+2 = 18
  - Note: h2 dominates h1

How to generate heuristics?

- **From relaxed versions of the problem**
  - This means we ignore some restrictions of the problem
  - Equivalent to adding edges between previously non-neighboring states in the state space
  - Examples: h1, h2 from the previous slide

- **From subproblems**
  - This means we consider some non-goal states as if they were goal states
  - Example: have 1,2,3,4 in the right places in 8-Puzzle, no matter the location of 5,6,7,8
  - Exact cost to subproblems can be calculated exactly and stored in a **pattern database**

- **By combining other heuristics**
  - If hand h' are admissible, then max(h , h') is also admissible
  - In some cases (but not always), we can also add heuristics (see **disjoint pattern databases**)

- **From experience**
  - Human expertise
  - Machine learning

Adversarial Search

Adversarial Search

Minimax and Alpha-Beta pruning

Multiplayer games

source: https://www.ssbwiki.com/File:8-Player_Smash_Original_8.jpg (fair use)

Multiplayer games

Multiplayer games

- Games with 2+ players are harder to analyze

Multiplayer games

- Games with 2+ players are harder to analyze
- Reward/goal is no longer defined by a single value

Multiplayer games

- Games with 2+ players are harder to analyze
- Reward/goal is no longer defined by a single value
- We need to keep track of the current player

Multiplayer games

- Games with 2+ players are harder to analyze
- Reward/goal is no longer defined by a single value
- We need to keep track of the current player
- Actions that don’t affect a player may affect others

Multiplayer games

- Games with 2+ players are harder to analyze
- Reward/goal is no longer defined by a single value
- We need to keep track of the current player
- Actions that don’t affect a player may affect others
- 3+ players: alliances, king making, diplomacy etc.

Multiplayer games

- Games with 2+ players are harder to analyze
- Reward/goal is no longer defined by a single value
- We need to keep track of the current player
- Actions that don’t affect a player may affect others
- 3+ players: alliances, king making, diplomacy etc.
- Simplification: focus on 2-player, zero-sum (or constant-sum) games

Zero-Sum Games

Zero-Sum Games

- Rather than classifying each node as goal/not a goal, each terminal node yields a certain utility (score) to each player

Zero-Sum Games

- Rather than classifying each node as goal/not a goal, each terminal node yields a certain utility (score) to each player
- With n players, **utility(s) = [u1,u2,…,un]**

Zero-Sum Games

- Rather than classifying each node as goal/not a goal, each terminal node yields a certain utility (score) to each player
    - With n players, **utility(s) = [u1,u2,…,un]**
- In zero-sum games, the utility of both players always adds to zero

Zero-Sum Games

- Rather than classifying each node as goal/not a goal, each terminal node yields a certain utility (score) to each player
    - With n players, **utility(s) = [u1,u2,…,un]**
- In zero-sum games, the utility of both players always adds to zero
    - Simplest case: single winner game (1 for win, -1 for loss, 0 for draw)

Constant-Sum Games

Constant-Sum Games

- A slightly more general type of game is **constant sum**

Constant-Sum Games

- A slightly more general type of game is **constant sum**
- Here all utilities add up to a constant

Constant-Sum Games

- A slightly more general type of game is **constant sum**
- Here all utilities add up to a constant
  - Analogy: players are competing for a share of a fixed “pie”

Constant-Sum Games

- A slightly more general type of game is **constant sum**
- Here all utilities add up to a constant
  - Analogy: players are competing for a share of a fixed “pie”
- Can always be reduced to zero-sum by adding a “fee” to enter

Constant-Sum Games

- A slightly more general type of game is **constant sum**
- Here all utilities add up to a constant
    - Analogy: players are competing for a share of a fixed “pie”
- Can always be reduced to zero-sum by adding a “fee” to enter
    - For example, if all utilities add up to 10, a fee of 5 to enter turns the game into zero-sum

Constant-Sum Games

- A slightly more general type of game is **constant sum**
- Here all utilities add up to a constant
    - Analogy: players are competing for a share of a fixed “pie”
- Can always be reduced to zero-sum by adding a “fee” to enter
    - For example, if all utilities add up to 10, a fee of 5 to enter turns the game into zero-sum
    - This means we can assume the game is zero-sum rather than constant sum.

Minimax intuition

Minimax intuition

- With general utilities, each player tries to maximize their own reward

Minimax intuition

- With general utilities, each player tries to maximize their own reward
    - In particular, this creates situations where one player is indifferent among outcomes that another player cares about.

Minimax intuition

- With general utilities, each player tries to maximize their own reward
    - In particular, this creates situations where one player is indifferent among outcomes that another player cares about.
    - It also creates situations where one player playing sub optimally results in worse outcomes for a different player

Minimax intuition

- With general utilities, each player tries to maximize their own reward
    - In particular, this creates situations where one player is indifferent among outcomes that another player cares about.
    - It also creates situations where one player playing sub optimally results in worse outcomes for a different player
    - Hard to model!

Minimax intuition

- With general utilities, each player tries to maximize their own reward
    - In particular, this creates situations where one player is indifferent among outcomes that another player cares about.
    - It also creates situations where one player playing sub optimally results in worse outcomes for a different player
        - Hard to model!
- With zero-sum, we can model the outcome of the game as a single value **u**

Minimax intuition

- With general utilities, each player tries to maximize their own reward
    - In particular, this creates situations where one player is indifferent among outcomes that another player cares about.
    - It also creates situations where one player playing sub optimally results in worse outcomes for a different player
        - Hard to model!
- With zero-sum, we can model the outcome of the game as a single value **u**
    - One player (max) tries to maximize **u**, the other (min) tries to minimize **u**

Minimax intuition

- With general utilities, each player tries to maximize their own reward
    - In particular, this creates situations where one player is indifferent among outcomes that another player cares about.
    - It also creates situations where one player playing sub optimally results in worse outcomes for a different player
    - Hard to model!
- With zero-sum, we can model the outcome of the game as a single value **u**
    - One player (max) tries to maximize **u**, the other (min) tries to minimize **u**
    - This removes both problems above. Easier to model

Minimax intuition

- With general utilities, each player tries to maximize their own reward
    - In particular, this creates situations where one player is indifferent among outcomes that another player cares about.
    - It also creates situations where one player playing sub optimally results in worse outcomes for a different player
    - Hard to model!
- With zero-sum, we can model the outcome of the game as a single value **u**
    - One player (max) tries to maximize **u**, the other (min) tries to minimize **u**
    - This removes both problems above. Easier to model
    - Easier to model

Minimax Value

Minimax Value

- **Definition: minimax value of a state s with successors s_i**
  - Minimax(s) :=
    - Utility (s), if s is terminal
    - Max_{s_i} (minimax (s_i)) if player = MAX
    - Min_{s_i} (minimax (s_i)) if player = MIN

Minimax Value

- **Definition: minimax value of a state s with successors s_i**
  - Minimax(s) :=
    - Utility (s), if s is terminal
    - Max_{s_i} (minimax (s_i)) if player = MAX
    - Min_{s_i} (minimax (s_i)) if player = MIN

- **It can be shown that if the two players play optimally from a state s, the final utility of the game will be Minimax(s)**

Minimax Value

- **Definition: minimax value of a state s with successors s_i**
  - Minimax(s) :=
    - Utility (s), if s is terminal
    - Max_{s_i} (minimax (s_i)) if player = MAX
    - Min_{s_i} (minimax (s_i)) if player = MIN

- **It can be shown that if the two players play optimally from a state s, the final utility of the game will be Minimax(s)**

- **Strategy outline: compute Minimax recursively, choose node with best Minimax value**

Minimax Value

- **Definition: minimax value of a state s with successors s_i**
  - Minimax(s) :=
    - Utility (s), if s is terminal
    - Max_{s_i} (minimax (s_i)) if player = MAX
    - Min_{s_i} (minimax (s_i)) if player = MIN

- **It can be shown that if the two players play optimally from a state s, the final utility of the game will be Minimax(s)**

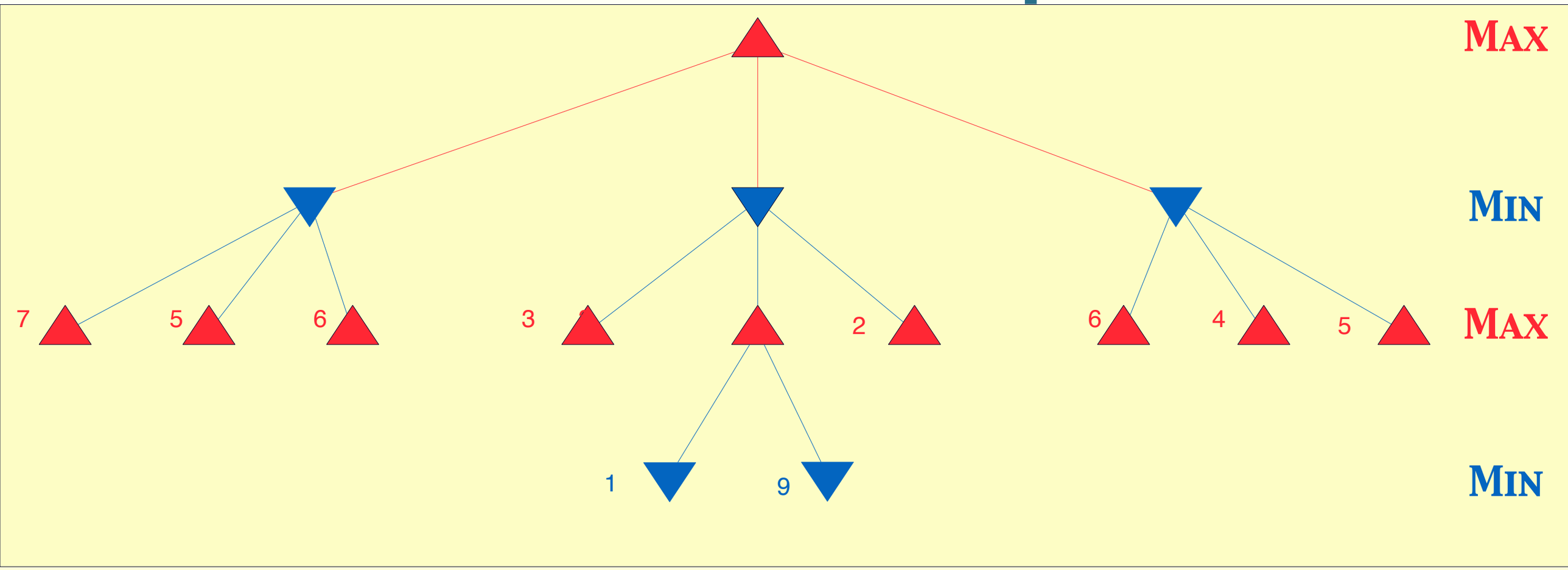
- **Strategy outline: compute Minimax recursively, choose node with best Minimax value**

- **For two-player zero-sum games, it can be shown this corresponds to optimal play (Nash equilibrium)**

Minimax Pseudocode

```
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 Example



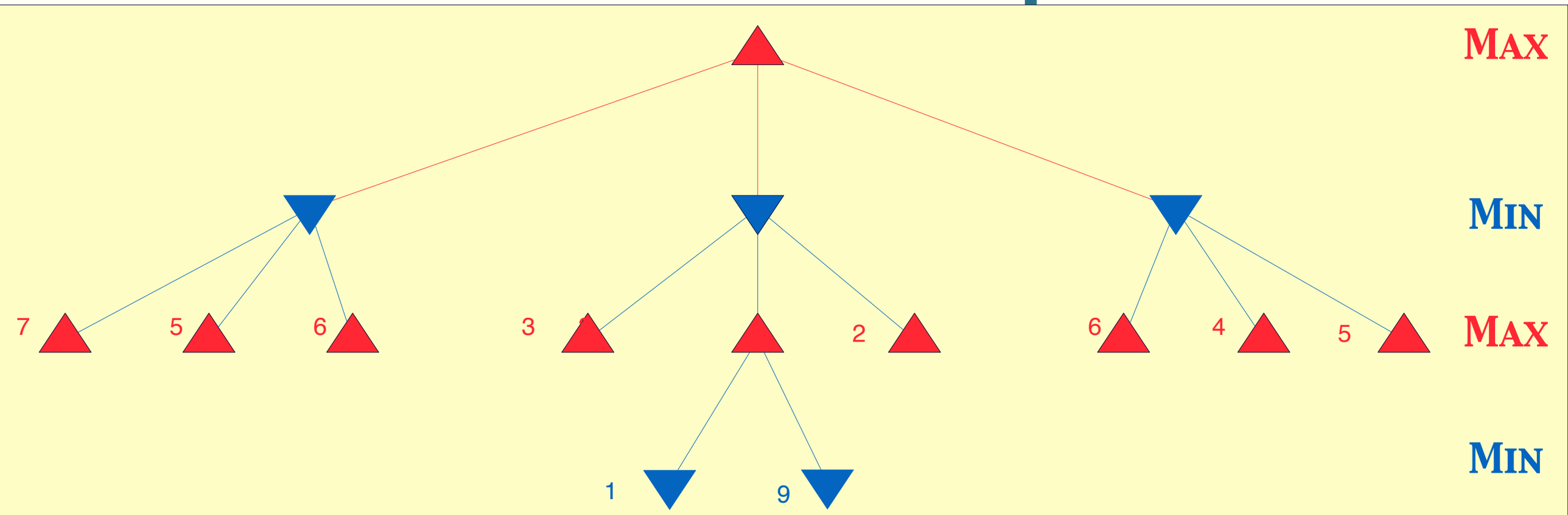
Legend:

Max node

Min node

1-9 Utility for the MAX player

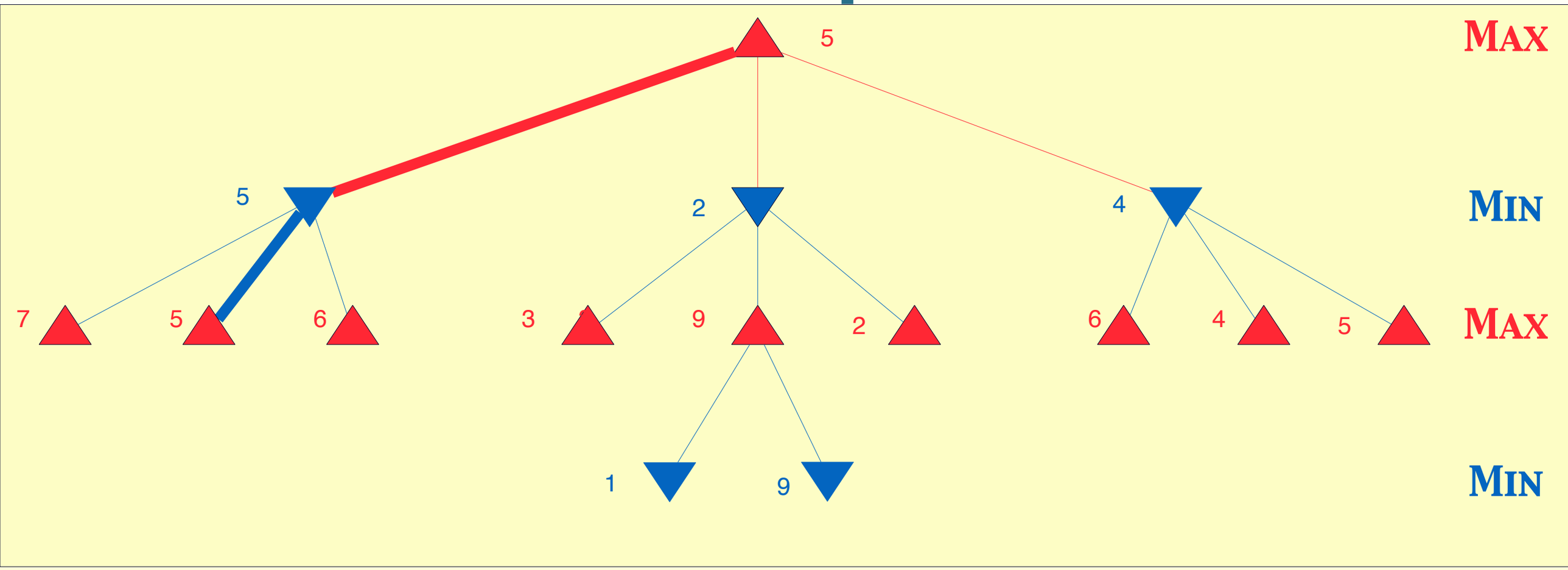
Minimax Example



Exercise (5 minutes):

- **Fill in the minimax value of every non-terminal node**
- **Highlight the path resulting from both player's actions**

Minimax Example Solution



Note: on a visual example, it's easy to go by "layers" from bottom to top, but it's important to remember Minimax is explores the tree Depth-First

Minimax and Optimality

Minimax and Optimality

- Minimax maximizes the worst-case outcome for MAX

Minimax and Optimality

- **Minimax maximizes the worst-case outcome for MAX**
- **If MIN does not play optimally and the game is 2-player zero-sum game, MAX does even better**
  - But not necessarily as well as possible versus that specific sub-optimal player

Minimax and Optimality

- **Minimax maximizes the worst-case outcome for MAX**
- **If MIN does not play optimally and the game is 2-player zero-sum game, MAX does even better**
  - But not necessarily as well as possible versus that specific sub-optimal player
- **Minimax also does not distinguish between tied nodes**
  - But as long as both players are playing optimally, this doesn't matter (any tied node will lead to the minimax score)
  - Additional heuristics may need to be used if we assume one of the players is not optimal

MiniMax Properties

MiniMax Properties

- **based on depth-first**
  - recursive implementation

MiniMax Properties

- **based on depth-first**
  - recursive implementation
- **time complexity is $O(b^m)$**
  - exponential in depth of the tree

MiniMax Properties

- **based on depth-first**
  - recursive implementation
- **time complexity is $O(b^m)$**
  - exponential in depth of the tree
- **space complexity is O(b*m)**
  - inherited from depth-first
  - If only one successor is generated at a time, can be O(m) instead

MiniMax Properties

- **based on depth-first**
  - recursive implementation
- **time complexity is $O(b^m)$**
  - exponential in depth of the tree
- **space complexity is O(b*m)**
  - inherited from depth-first
  - If only one successor is generated at a time, can be O(m) instead

b branching factor (possible moves - can be high for some games)
m maximum depth of the search tree (can be infinite, but often rules prohibit this)

Irrelevant Nodes in MiniMax

Irrelevant Nodes in MiniMax

- **the values of some of the leaf nodes are irrelevant for decisions at the next level**
  - this also holds for decisions at higher levels

Irrelevant Nodes in MiniMax

- **the values of some of the leaf nodes are irrelevant for decisions at the next level**
  - this also holds for decisions at higher levels
- **as a consequence, under certain circumstances, some parts of the tree can be disregarded**
  - it is possible to still make an optimal decision without considering those parts
  - these parts can be “pruned”

Pruning

Pruning

- **discards parts of the search tree**
  - guaranteed not to contain good moves
  - guarantee that the solution is not in that branch or sub-tree
    - if both players make optimal (rational) decisions, they will never end up in that part of the search tree
    - sub-optimal moves by the opponent may lead into that part

Pruning

- **discards parts of the search tree**
  - guaranteed not to contain good moves
  - guarantee that the solution is not in that branch or sub-tree
    - if both players make optimal (rational) decisions, they will never end up in that part of the search tree
    - sub-optimal moves by the opponent may lead into that part
- **results in substantial time and space savings**
  - as a consequence, longer sequences of moves can be explored
  - the leftover part of the task may still be exponential, however

Alpha-Beta Pruning

Alpha-Beta Pruning

- **certain moves are not considered**
  - new states won't result in a better evaluation value than a move further up in the tree
  - they would lead to a less desirable outcome

Alpha-Beta Pruning

- **certain moves are not considered**
  - new states won't result in a better evaluation value than a move further up in the tree
  - they would lead to a less desirable outcome
- **applies to moves by both players**
  - α indicates the best choice for Max so far
    - never decreases
  - β indicates the best choice for Min so far
    - never increases

Alpha-Beta Pruning

- **certain moves are not considered**
  - new states won't result in a better evaluation value than a move further up in the tree
  - they would lead to a less desirable outcome

- **applies to moves by both players**
  - α indicates the best choice for Max so far
    - never decreases
  - β indicates the best choice for Min so far
    - never increases

- **extension of the minimax approach**
  - results in the same sequence of moves as minimax, but with less overhead
  - prunes uninteresting parts of the search tree

Alpha-Beta Pseudocode

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

  $v \leftarrow$ MAX-VALUE($state, -\infty, +\infty$)
  return the *action* in SUCCESSORS(*state*) with value v

---

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

Alpha-Beta Pseudocode (cont.)

function MIN-VALUE($state, \alpha, \beta$) **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*)

$v \leftarrow +\infty$

for a, s in SUCCESSORS(*state*) **do**

  $v \leftarrow$ MIN(v, MAX-VALUE(s, α, β))

  if $v \leq \alpha$ **then return** v

  $\beta \leftarrow$ MIN(β, v)

return v

Alpha-Beta Pseudocode (cont.)

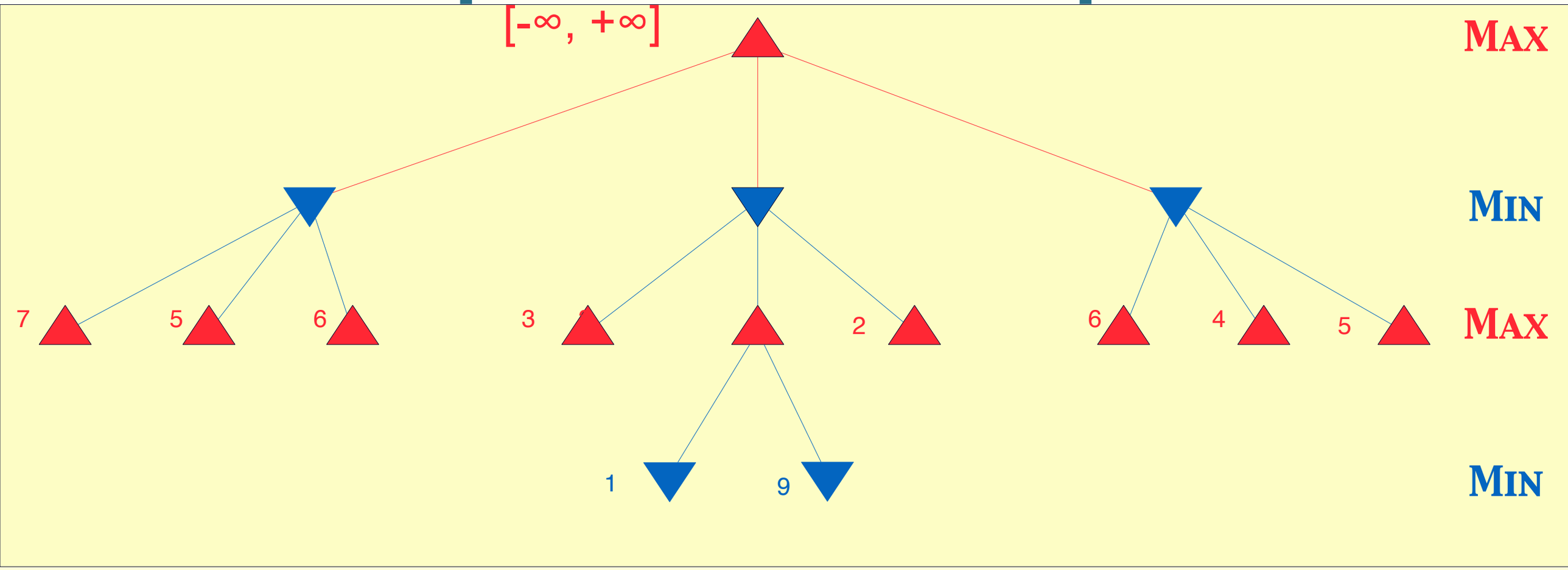
```
function ALPHA-BETA-SEARCH(state) returns an action
    inputs: state, current state in game
    v ← MAX-VALUE(state, −∞, +∞)
    return the action in SUCCESSORS(state) with value v
```

```
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
```

```
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
```

http://aima.eecs.berkeley.edu/slides-ppt/

Alpha-Beta Example



Legend:

Max node

Min node

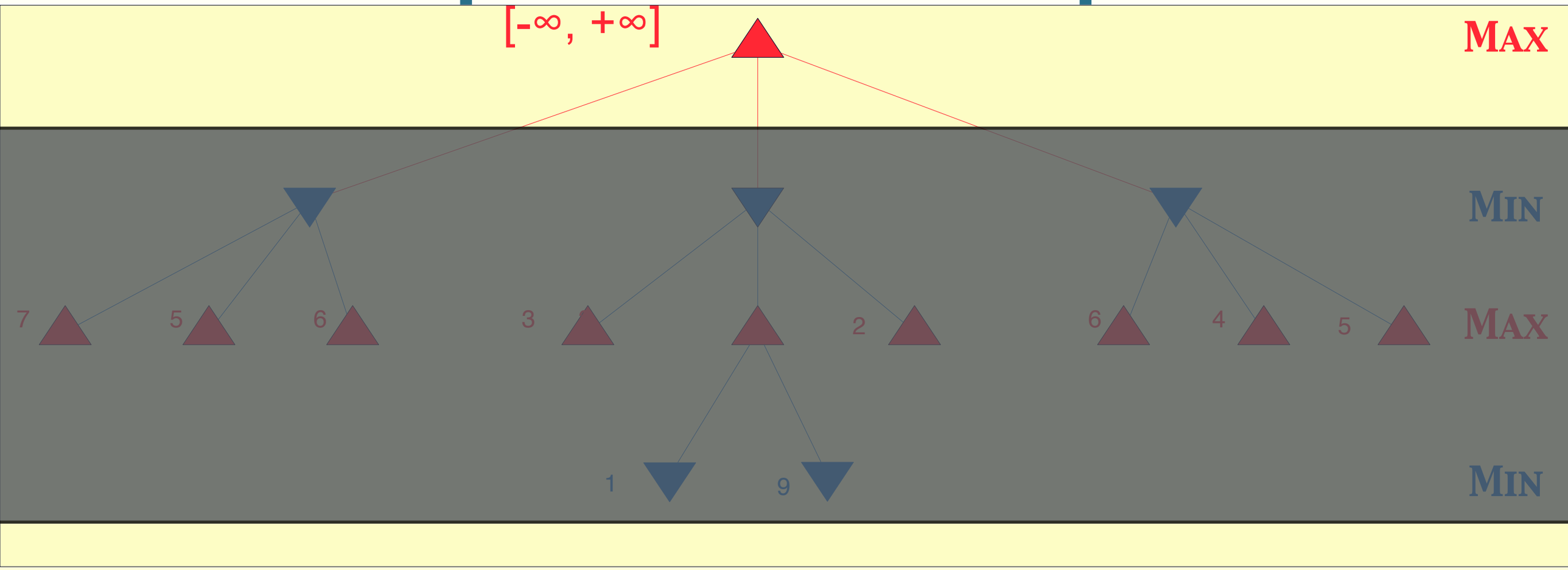
[α, β] Alpha and Beta values

1-9 Utility

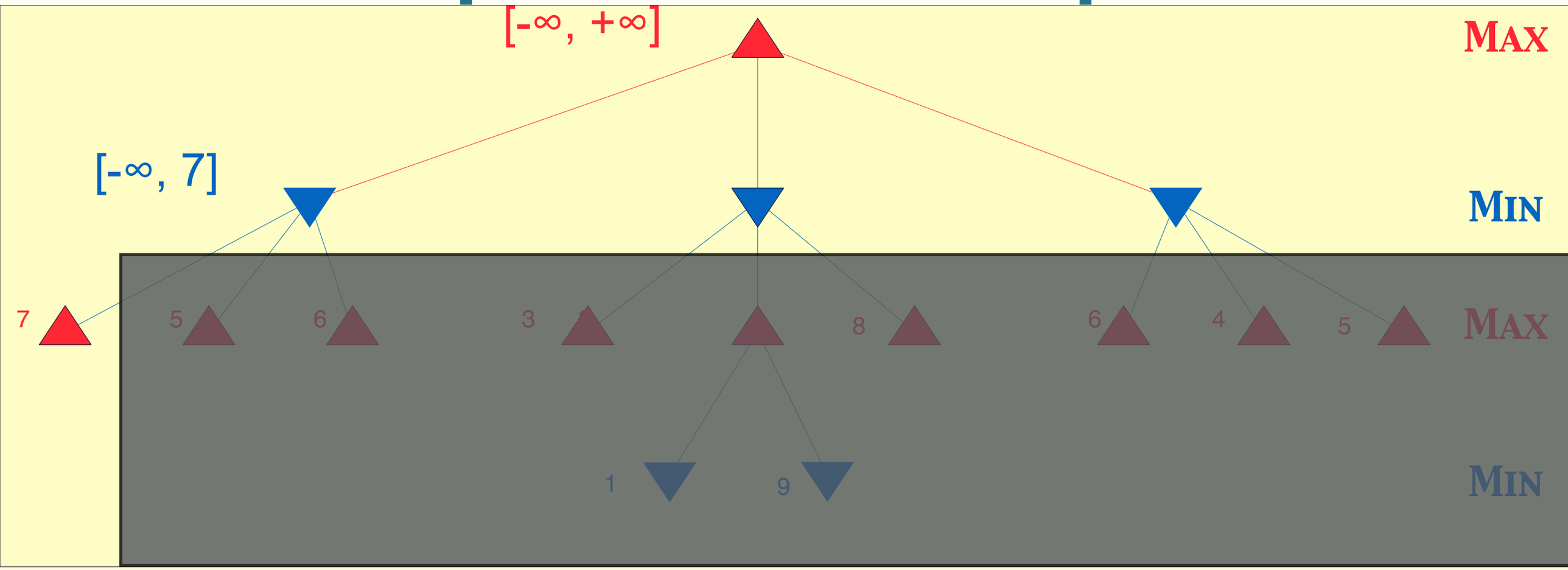
CAL POLY

ignite your passion

Alpha-Beta Example

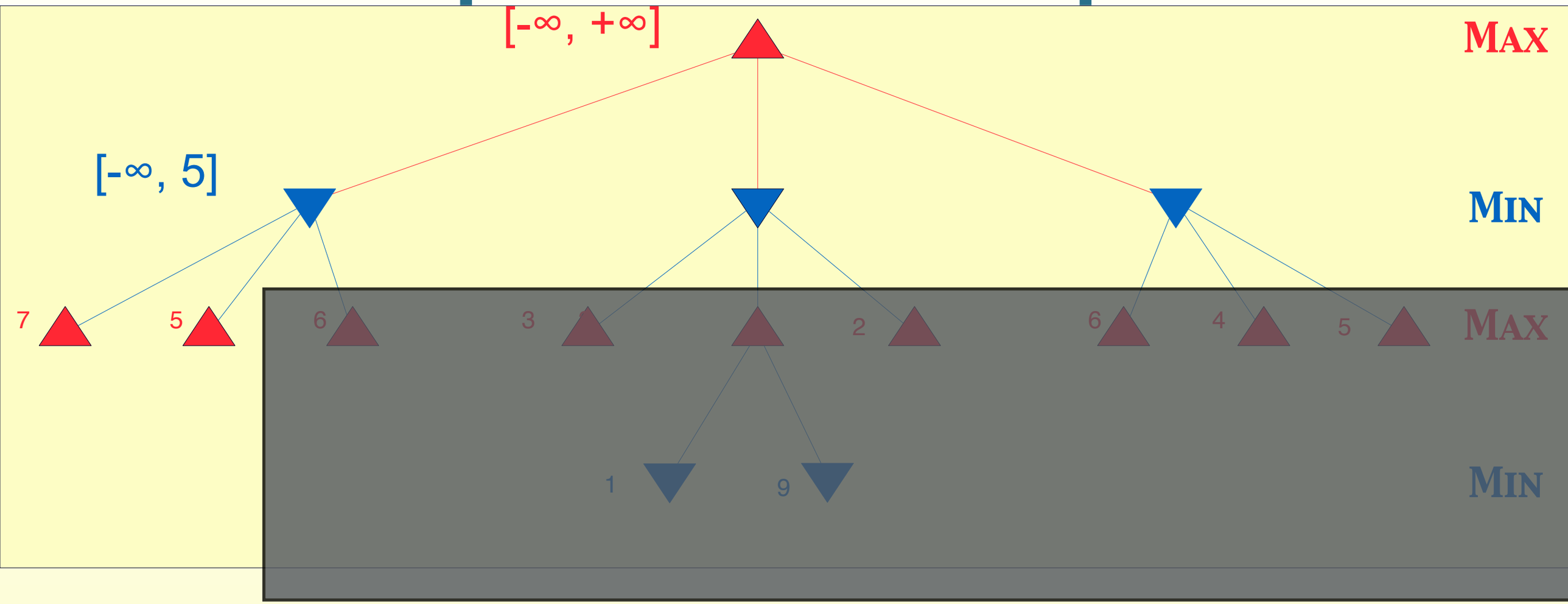


Alpha-Beta Example



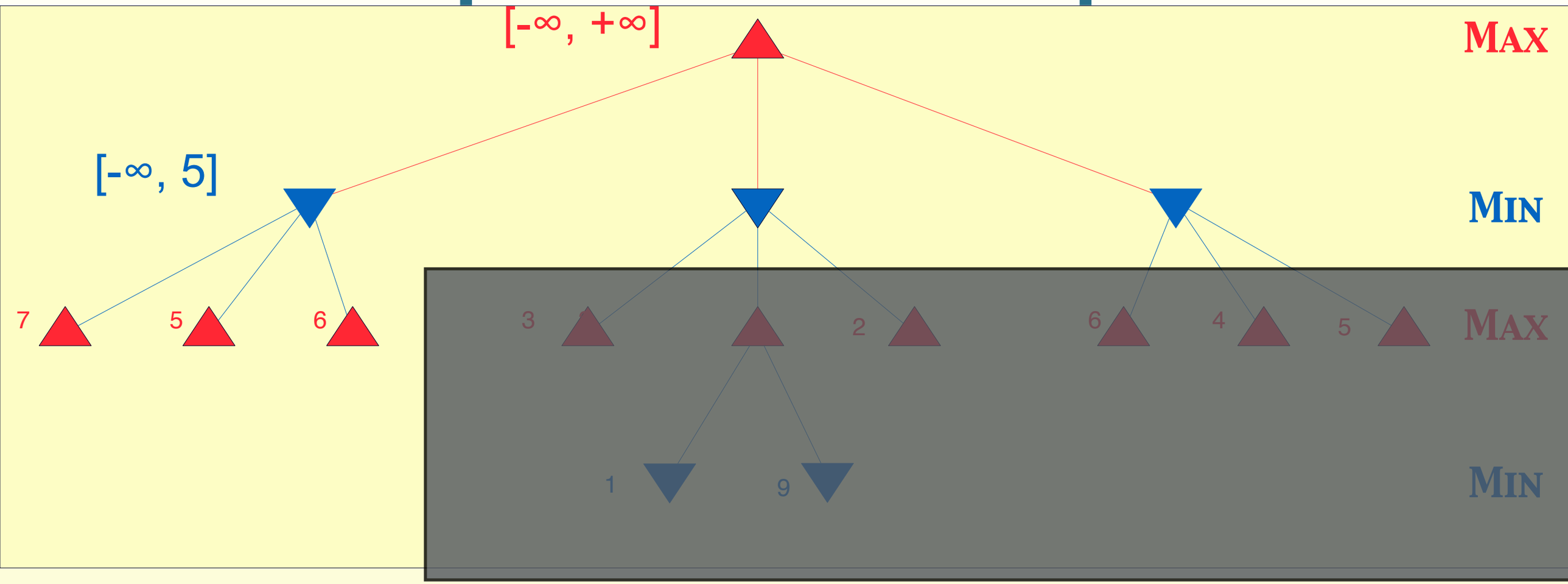
Regardless of the remaining children of the first MIN node, MIN will never choose < 7 at that node

Alpha-Beta Example



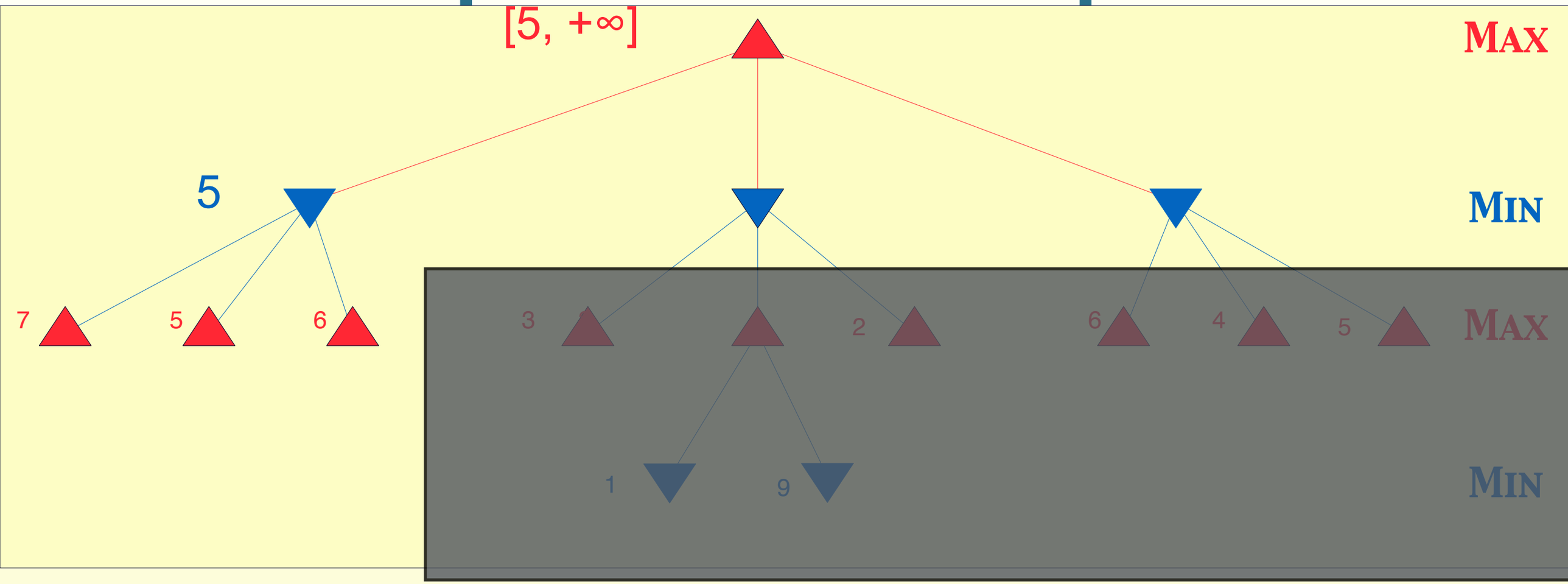
MIN node improves (reduces) its β value to 5

Alpha-Beta Example



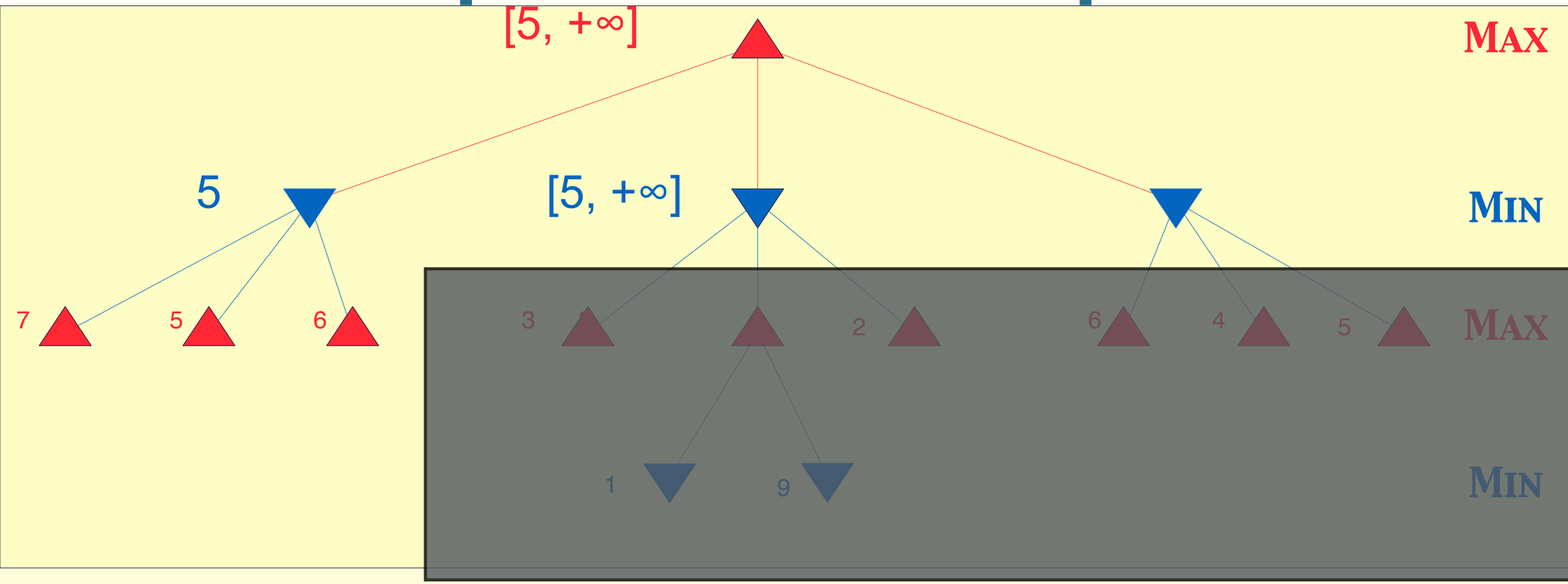
We've finished exploring the first MIN node, and β stays at 5

Alpha-Beta Example



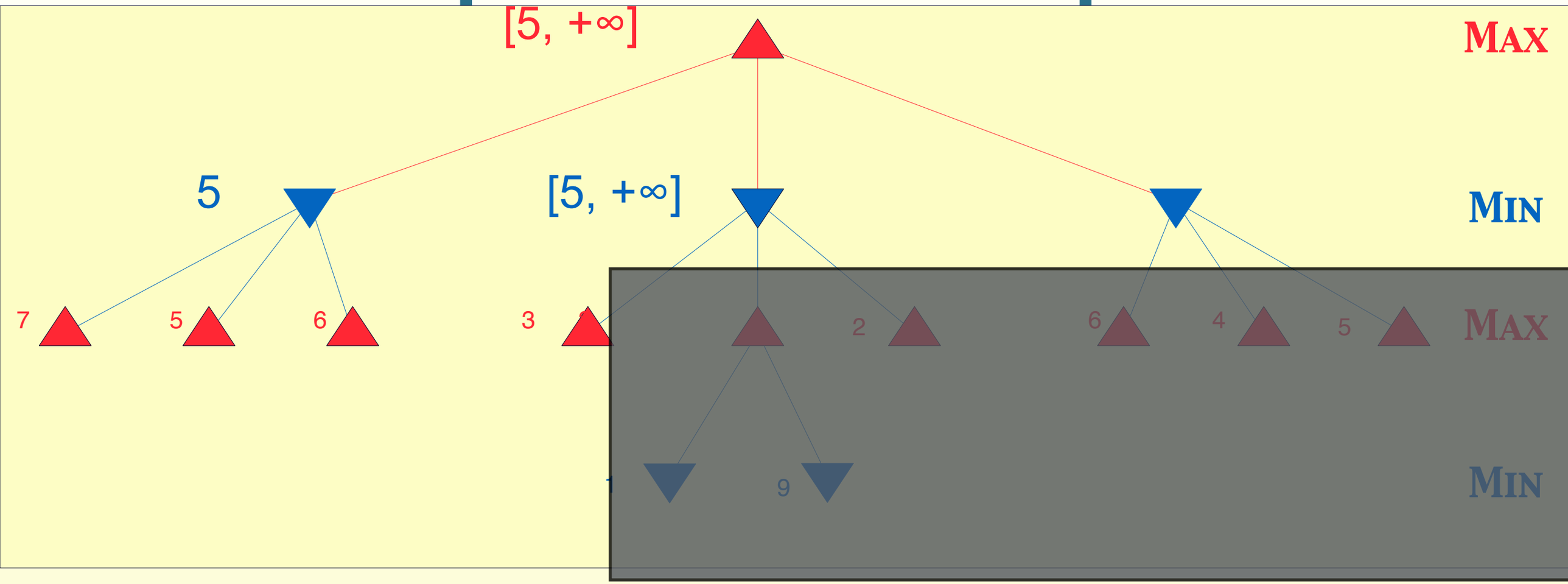
We update α at the starting node to 5: MAX will never pick a bide worse than 5

Alpha-Beta Example



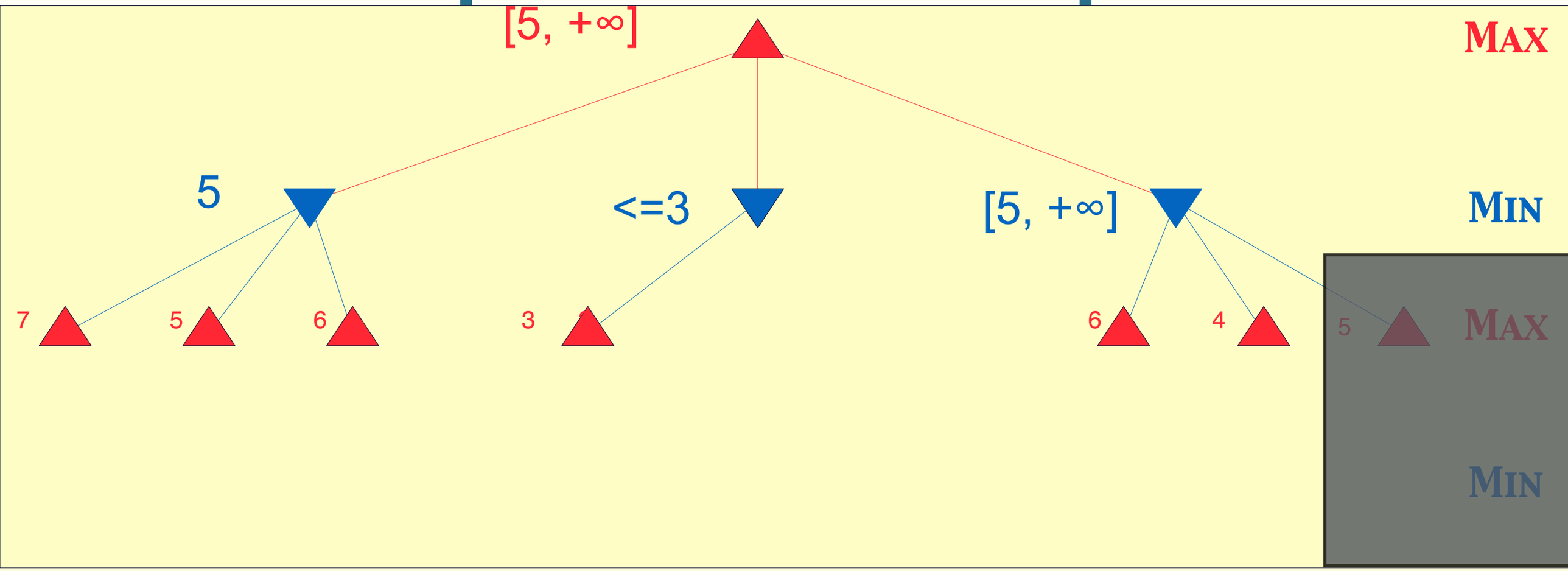
We propagate α=5 to the next MIN node

Alpha-Beta Example



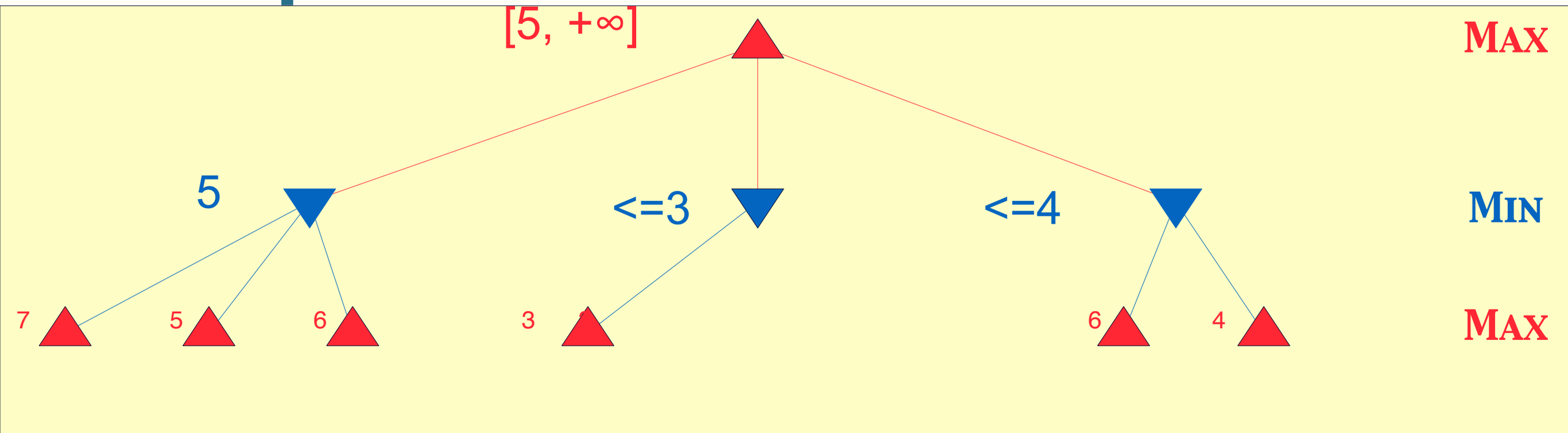
Min would like to update β to 3. But since this is < α, we can stop here and prune the rest of the subtree

Alpha-Beta Example



Once again, we have a value for MIN that is <= α, so we can stop

Alpha-Beta Final Search Tree



MIN

Note that in the middle MIN node, we “got lucky” and only expanded one child (and pruned an entire subtree). In the rightmost MIN node, we got less lucky and only pruned one leaf node

Note on alpha-beta efficiency

Note on alpha-beta efficiency

- The efficiency of alpha-beta search depends on how many nodes are pruned in a practical scenario

Note on alpha-beta efficiency

- **The efficiency of alpha-beta search depends on how many nodes are pruned in a practical scenario**
- **The order in which children of a node are visited matters**
  - For MAX nodes, you ideally want children with _____ value visited first
  - For MIN nodes, you ideally want children with _____ value visited first

Note on alpha-beta efficiency

- **The efficiency of alpha-beta search depends on how many nodes are pruned in a practical scenario**
- **The order in which children of a node are visited matters**
  - For MAX nodes, you ideally want children with **greater** value visited first
  - For MIN nodes, you ideally want children with **smaller** value visited first
- **Heuristics can help!**

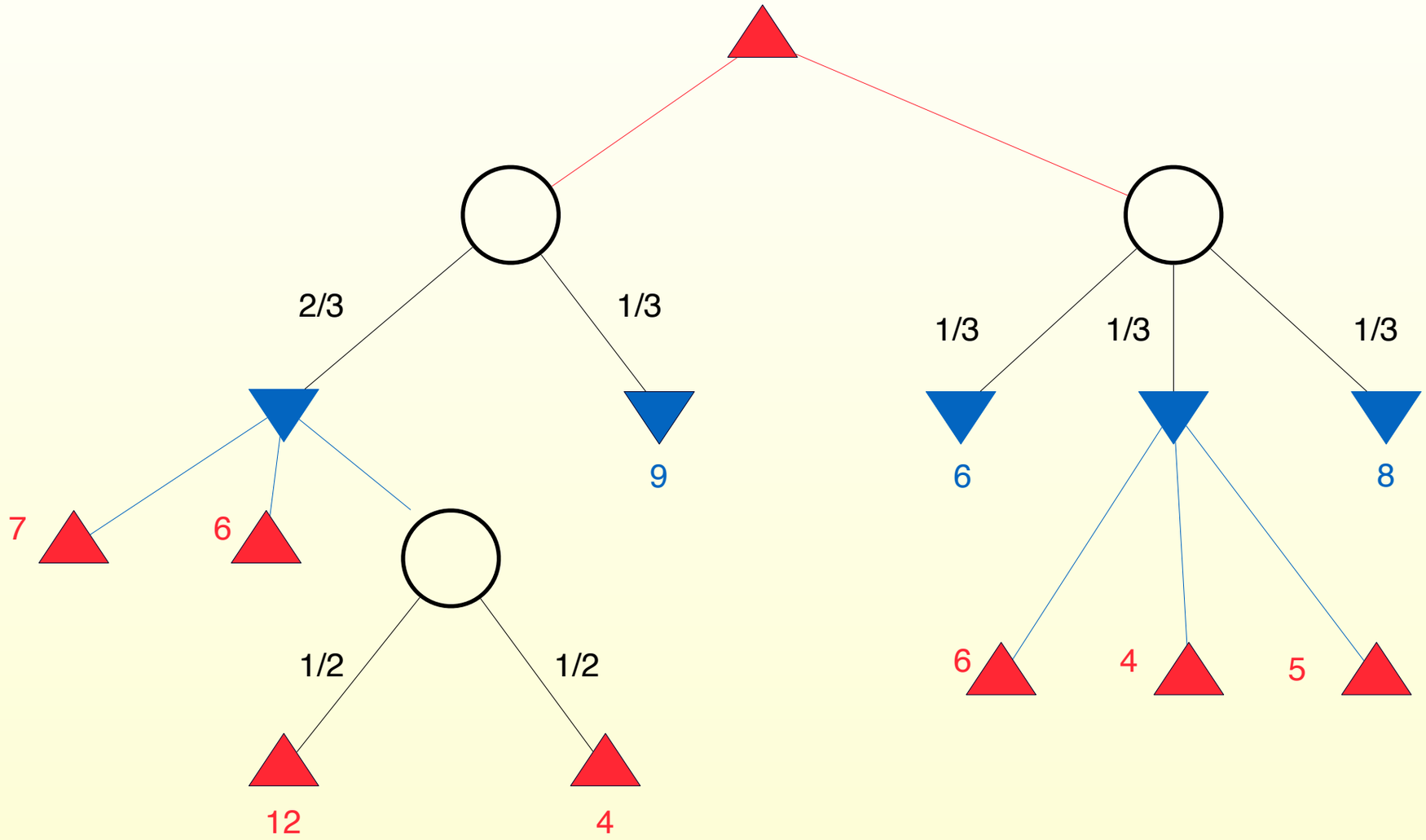
Expectiminimax

- Variation of Minimax to address randomness or incomplete information

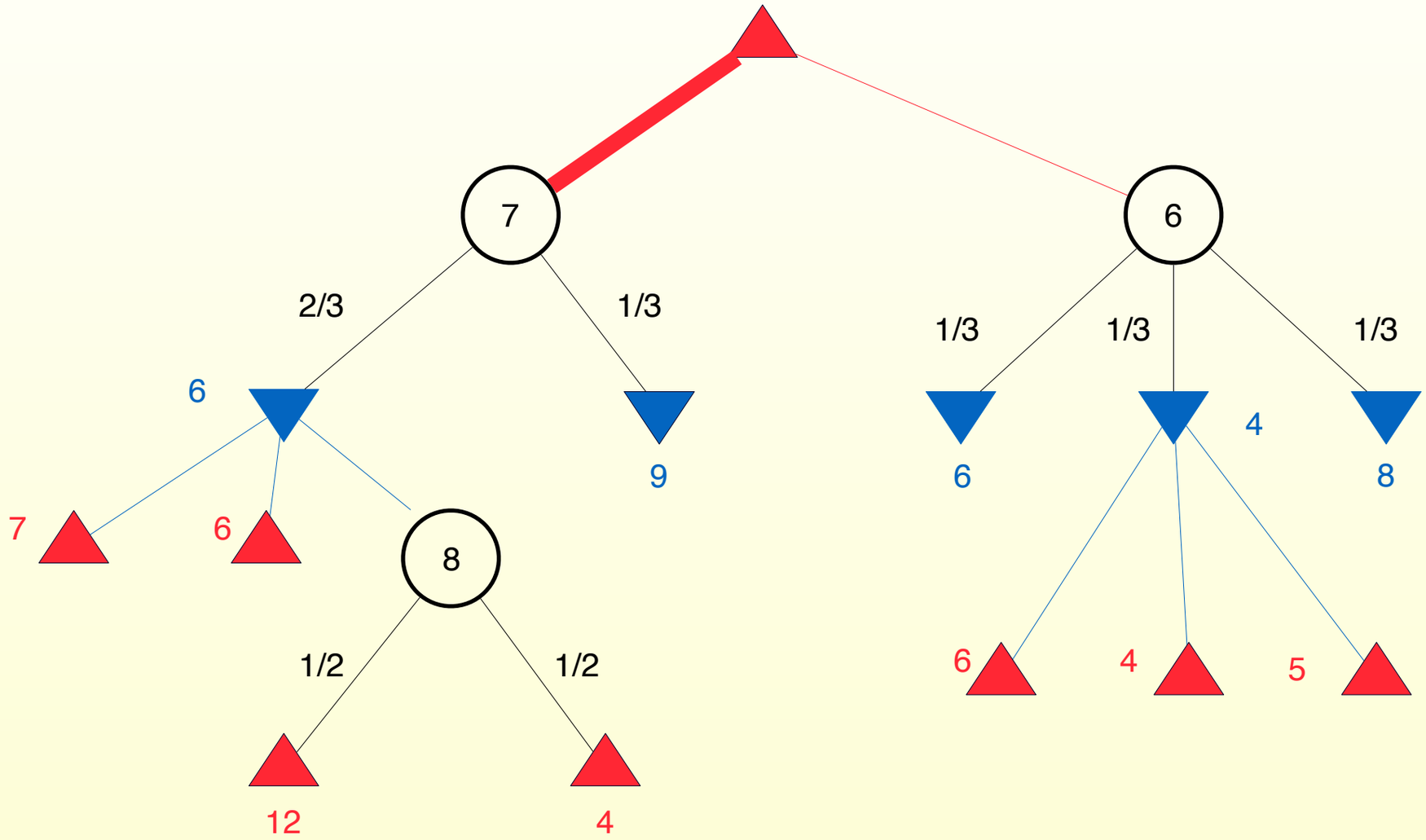
- In addition to min nodes, max nodes and terminal nodes, **chance nodes** are introduced

- A chance node return the expected value of its children (rather than the min or max)

Expectiminimax example



Expectiminimax example



Monte-Carlo Tree Search (MCTS)

The problem with tree search

The problem with tree search

- Most tree search algorithms: $O(b^d)$

The problem with tree search

- Most tree search algorithms: $O(b^d)$

b = breadth
d = depth

The problem with tree search

- Most tree search algorithms: $O(b^d)$

b = breadth
d = depth

- Intractable for many problems

The problem with tree search

- Most tree search algorithms: $O(b^d)$

b = breadth
d = depth

- Intractable for many problems
- Two approaches:

The problem with tree search

- Most tree search algorithms: $O(b^d)$

b = breadth
d = depth

- Intractable for many problems
- Two approaches:
    - Reduce effective *breadth*

The problem with tree search

- Most tree search algorithms: $O(b^d)$

b = breadth
d = depth

- Intractable for many problems
- Two approaches:
    - Reduce effective *breadth*

Node selection and pruning
Avoid expanding all nodes

The problem with tree search

- Most tree search algorithms: $O(b^d)$

b = breadth
d = depth

- Intractable for many problems
- Two approaches:
  - Reduce effective *breadth*
  - Reduce effective *depth*

Node selection and pruning
Avoid expanding all nodes

The problem with tree search

- Most tree search algorithms: $O(b^d)$

b = breadth
d = depth

- Intractable for many problems

- Two approaches:
  - Reduce effective *breadth*
  - Reduce effective *depth*

Node selection and pruning
Avoid expanding all nodes

Node evaluation
Cut search at a certain depth with approximate value

How to improve search?

- Key ideas:
  - Replace heuristic evaluation function with random **rollouts** - reduces depth
  - Use a heuristic (**UCT value**) to choose which nodes to expand next - reduces breadth

Example: DeepBlue

- Reduces *breadth* with several heuristics for node selection and pruning
- Reduces *depth* with heuristics for node evaluation

Limitations

- Designing good expansion and evaluation heuristics can be hard!

- Many chess experts contributed to the heuristics in DeepBlue

- In certain problems (like the game of Go), the best human-crafted heuristics still didn't perform well

Key ideas in MCTS

- Evaluation heuristic can be replaced by a random *rollout*
    - Reduces effective *depth*
- Expand nodes balancing *exploration* of new nodes with *exploitation* of promising solutions
    - Asymmetrically reduces effective *breadth*
- No domain-specific heuristics needed!

Related problem: Multi-Armed Bandits

Source: Wikipedia (CC BY-SA 3.0)

Related problem: Multi-Armed Bandits

- Agent can pull one of n levers, each giving a reward from some random distribution

Source: Wikipedia (CC BY-SA 3.0)

Related problem: Multi-Armed Bandits

- Agent can pull one of n levers, each giving a reward from some random distribution
- How to maximize expected return over time?

Source: Wikipedia (CC BY-SA 3.0)

Related problem: Multi-Armed Bandits

- Agent can pull one of n levers, each giving a reward from some random distribution
- How to maximize expected return over time?
- Some ideas
  - Pull at random (pure exploration)
  - Pull each lever k times, then only pull the best (exploitation)
  - Pull random lever with small probability, otherwise pull best lever
  - Is there a more principled approach?

Source: Wikipedia (CC BY-SA 3.0)

Which arm to pull?

Flat Monte Carlo

Share trials uniformly between arms

ε-Greedy

P(1- ε) – Best arm so far

P(ε) – Random arm

UCB1 (Auer et al (2002)).

Choose arm j so as to maximise:

$$\bar{X}_j + \sqrt{\frac{2 \log n}{T_j(n)}}$$

Mean so far

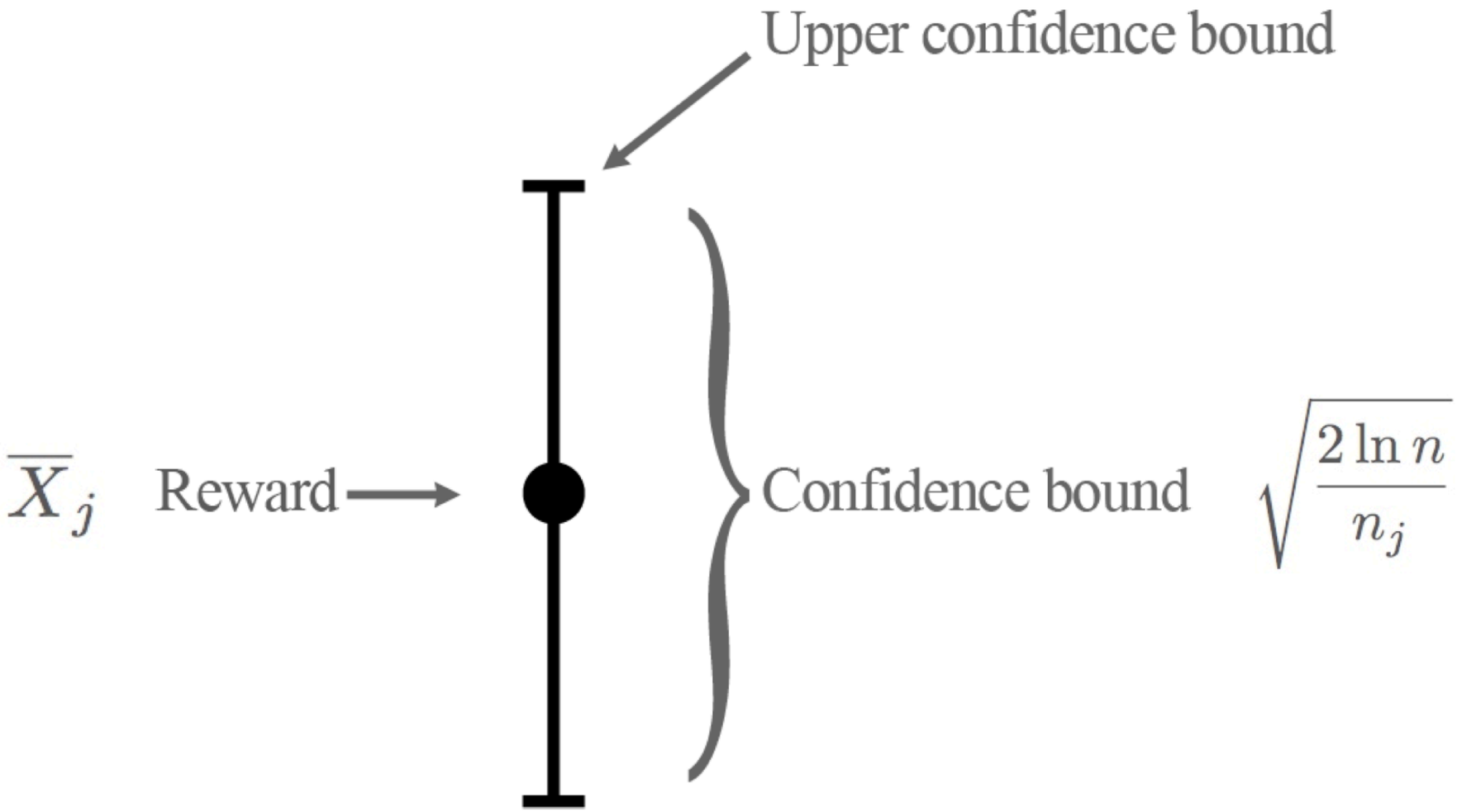
Upper bound on variance

n = number of plays so far

Tj(n) = number of times arm j was pulled

Confidence Bounds

- Confidence in the reward's accuracy

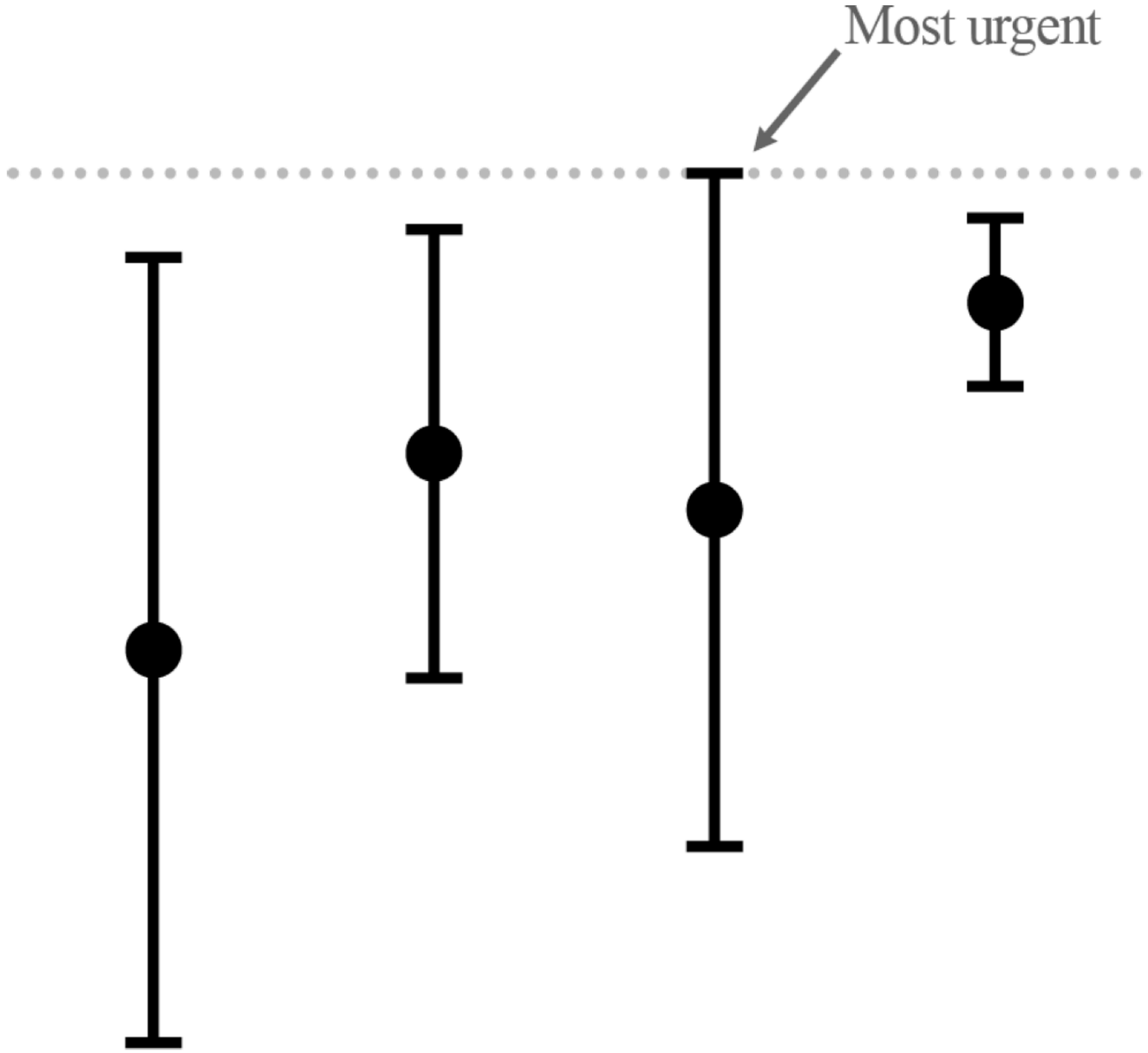


- More visits = tighter bound

Browne, C. (n.d.). Monte Carlo Tree Search

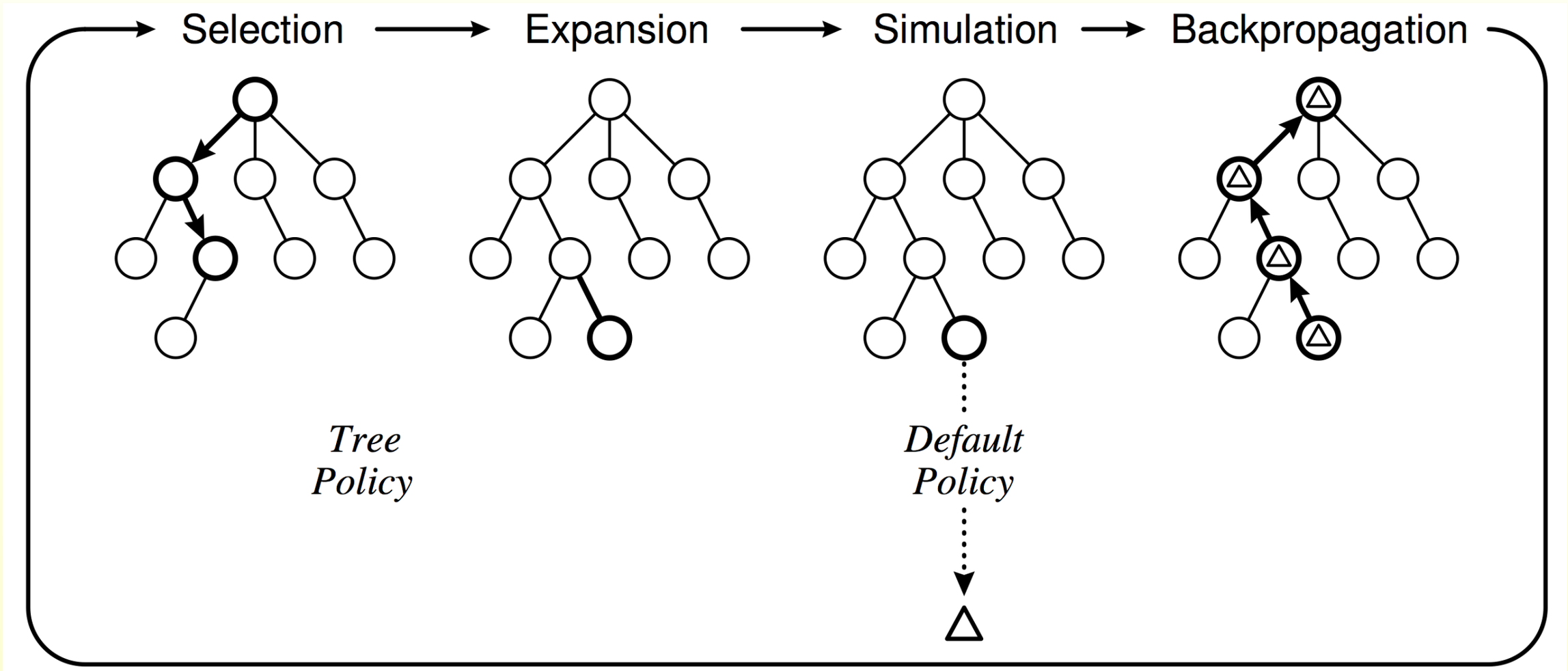
Most Urgent

- Most urgent node has the highest UCB
- **Not** highest reward
- **Not** widest spread



Browne, C. (n.d.). Monte Carlo Tree Search

MCTS Process



Steinmetz, E., & Gini, M. (n.d.). Improving Play of Monte-Carlo Engines in the Game of Go | Computer Science & Engineering. Retrieved May 3, 2016, from https://www.cs.umn.edu/research/technical_reports/view/14-019

MCTS outline

MCTS outline

- Iteratively build a tree from top to bottom

MCTS outline

- Iteratively build a tree from top to bottom
- Start each iteration by visiting the root

MCTS outline

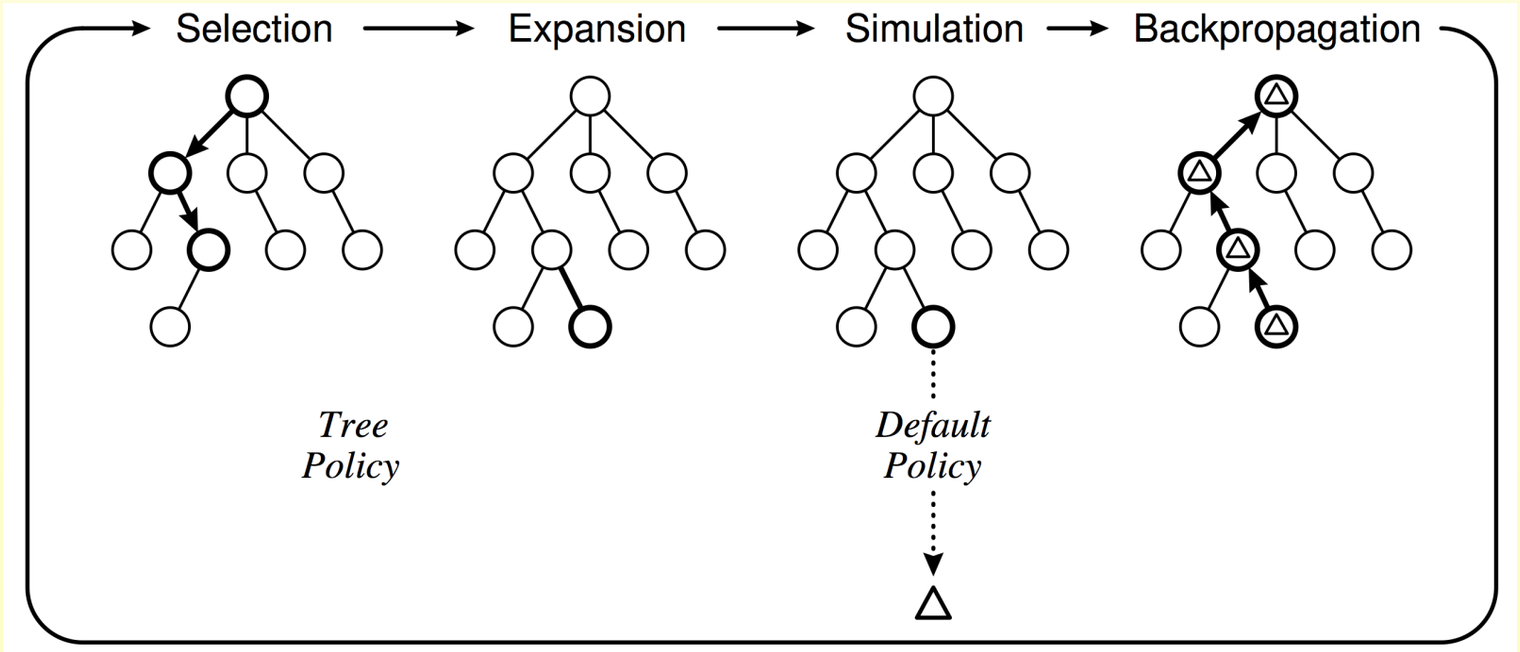
- **Iteratively build a tree from top to bottom**
- **Start each iteration by visiting the root**
- **If all children of visited node have been added to the tree:**
  - Use UCB1 formula to choose which node recursively visit (Selection)

MCTS outline

- **Iteratively build a tree from top to bottom**
- **Start each iteration by visiting the root**
- **If all children of visited node have been added to the tree:**
  - Use UCB1 formula to choose which node recursively visit (Selection)
- **If a node has some children not yet on the tree:**
  - Add a random unexplored child node to the tree (Expansion)
  - Perform rollouts from that node to a terminal state (Default Policy)
  - Back propagate the final score of the rollout up the tree (Backpropagation)

MCTS outline

- **Iteratively build a tree from top to bottom**
- **Start each iteration by visiting the root**
- **If all children of visited node have been added to the tree:**
  - Use UCB1 formula to choose which node recursively visit (Selection)
- **If a node has some children not yet on the tree:**
  - Add a random unexplored child node to the tree (Expansion)
  - Perform rollouts from that node to a terminal state (Default Policy)
  - Back propagate the final score of the rollout up the tree (Backpropagation)



CAL POLY

ignite your passion

MCTS outline

- **Iteratively build a tree from top to bottom**
- **Start each iteration by visiting the root**
- **If all children of visited node have been added to the tree:**
  - Use UCB1 formula to choose which node recursively visit (Selection)
- **If a node has some children not yet on the tree:**
  - Add a random unexplored child node to the tree (Expansion)
  - Perform rollouts from that node to a terminal state (Default Policy)
  - Back propagate the final score of the rollout up the tree (Backpropagation)

- **Note: Tree Policy = Selection + Expansion**
  - Returns the next node to add to the tree and perform rollout from

MCTS - Pseudocode

Algorithm 1 General MCTS approach.

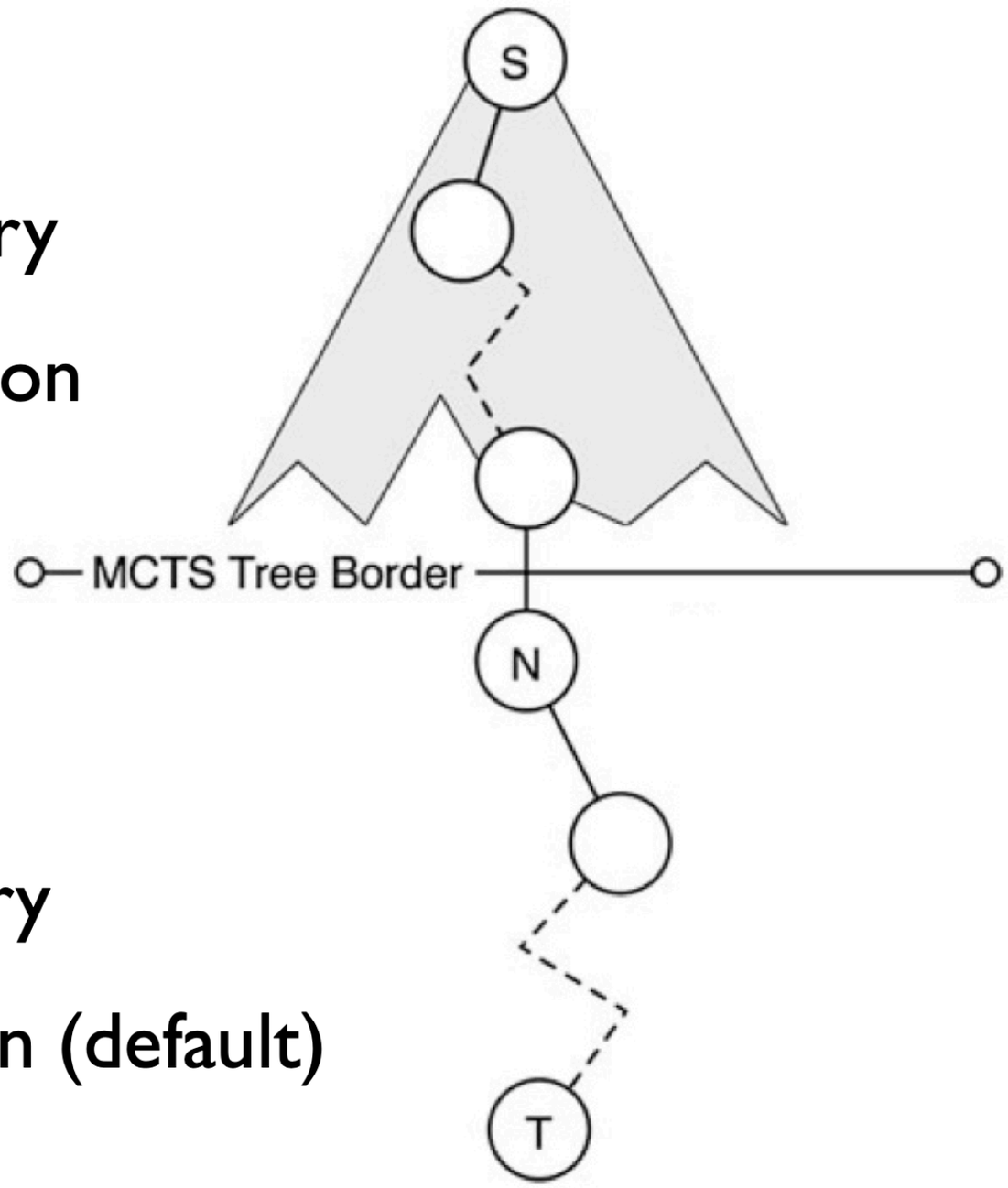
function MCTSSEARCH(s_0)
  create root node v_0 with state s_0
  while within computational budget **do**
    $v_l \leftarrow$ TREEPOLICY(v_0)
    $\Delta \leftarrow$ DEFAULTPOLICY($s(v_l)$)
    BACKUP(v_l, Δ)
return a(BESTCHILD(v_0))

By prof. Julian Togelius at NYU. Used with permission.
Original pseudocode on all slides by Browne, C. (n.d.). Monte Carlo Tree Search

Policies

Tree Policy

- Above the tree boundary
- Intelligent action selection



Finnsson & Bjornsson (2008)

Default policy

- Below the tree boundary
- Random action selection (default)

Tree policy

function TREEPOLICY(v)
  while v is nonterminal **do**
    if v not fully expanded **then**
      return EXPAND(v)
    else
      $v \leftarrow$ BESTCHILD(v, Cp)
  return v

Note that node selected for expansion does not need to be a leaf of the tree (the nonterminal test refers to the game state)

By prof. Julian Togelius at NYU. Used with permission.

Tree expansion

function EXPAND(v)

choose $a \in$ untried actions from $A(s(v))$

add a new child v' to v

with $s(v') = f(s(v), a)$

and $a(v') = a$

return v'

By prof. Julian Togelius at NYU. Used with permission.

Best child (UCT)

function BESTCHILD(v, c)

$$\textbf{return} \quad \underset{v' \in \text{children of } v}{\arg\max} \frac{Q(v')}{N(v')} + c\sqrt{\frac{2\ln N(v)}{N(v')}}$$

- Standard UCT equation (compare UCB)
- Higher values of c lead to more exploration

By prof. Julian Togelius at NYU. Used with permission.

Default policy (rollout)

function DEFAULTPOLICY(s)
  while s is non-terminal **do**
    choose $a \in A(s)$ uniformly at random
    $s \leftarrow f(s, a)$
  return reward for state s

- Each time a node is added to the tree, the default policy plays out until the terminal state of the game
- The standard is to do this uniformly randomly

By prof. Julian Togelius at NYU. Used with permission.

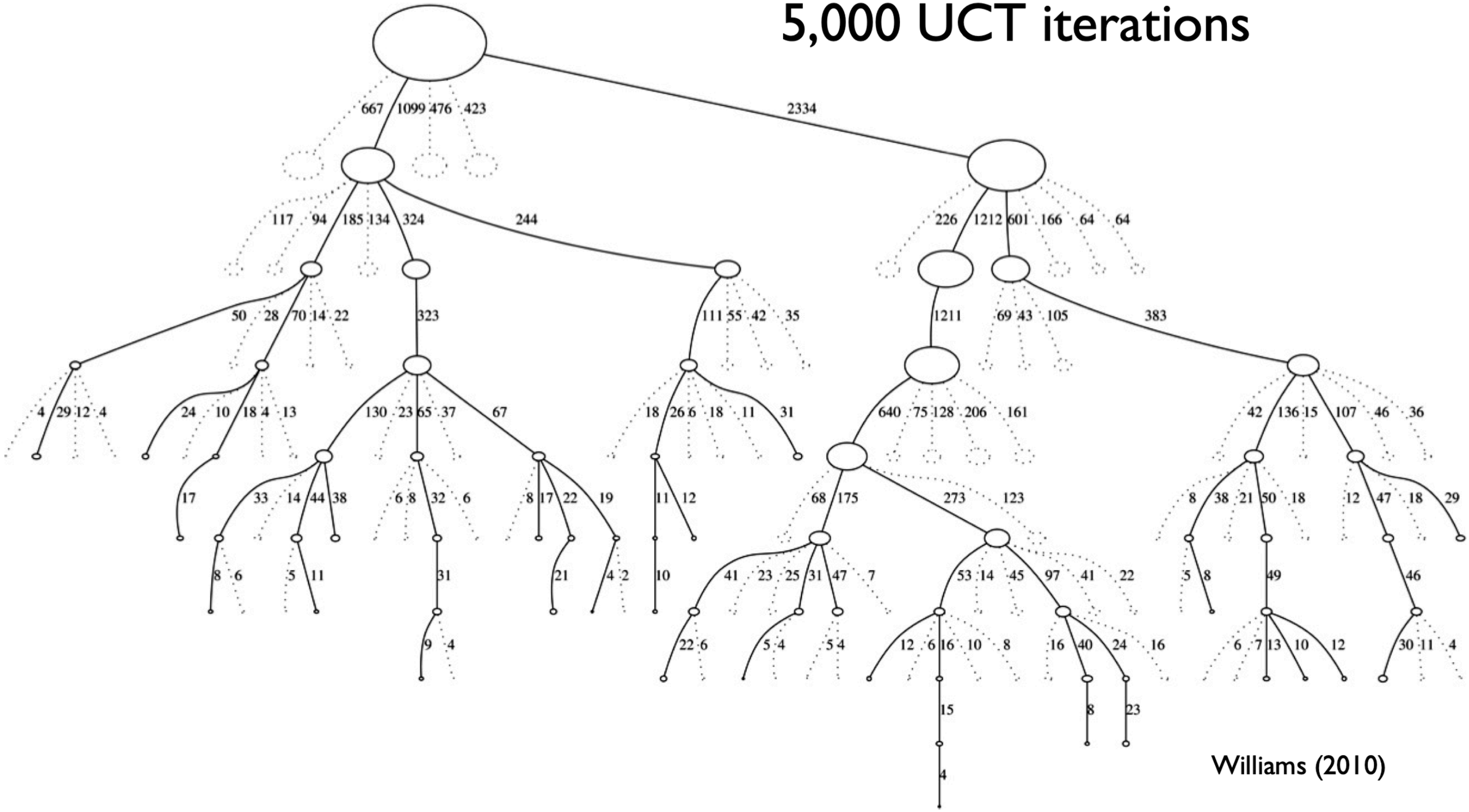
Backup

function BACKUP(v, Δ)
  while v is not null **do**
    $N(v) \leftarrow N(v) + 1$
    $Q(v) \leftarrow Q(v) + \Delta(v, p)$
    $v \leftarrow$ parent of v

- v is the new node added to the tree by the tree policy
- Back up the values from the added node up the tree to the root

MCTS Tree Growth

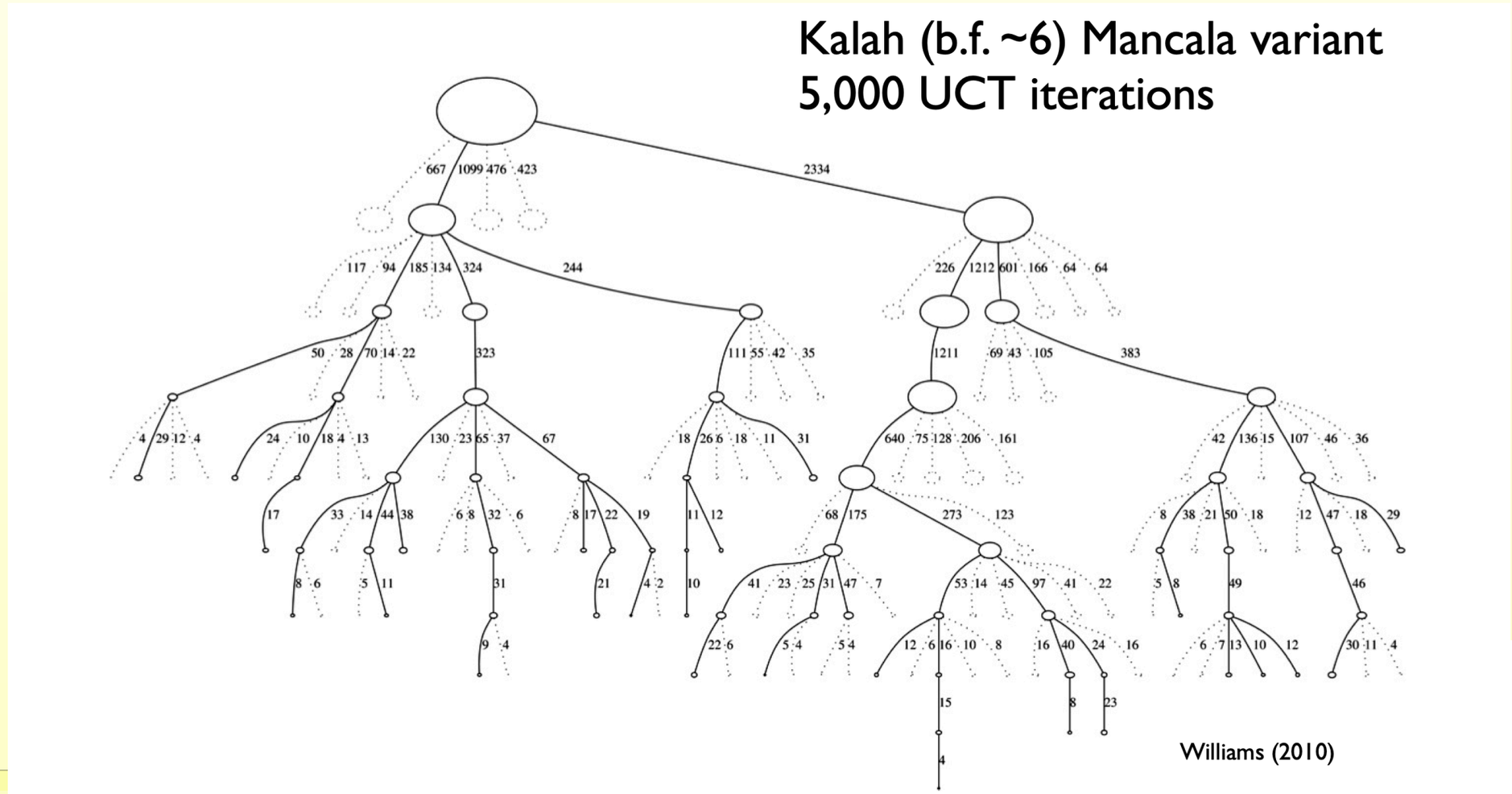
Kalah (b.f. ~6) Mancala variant
5,000 UCT iterations



Williams (2010)

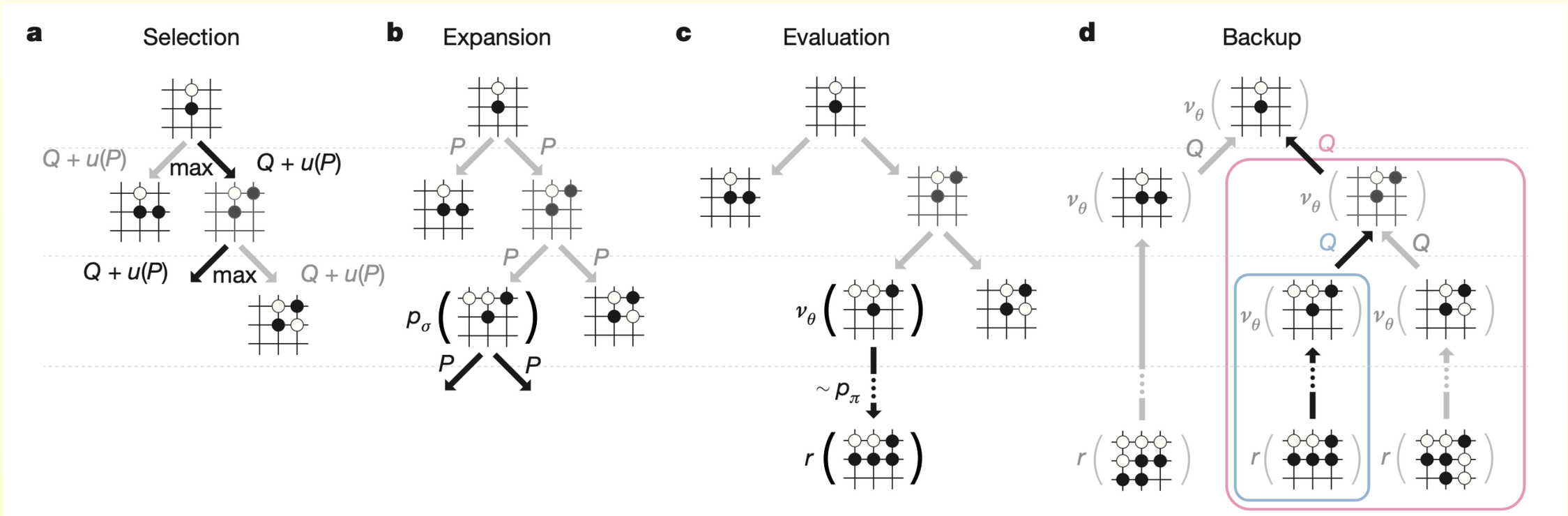
MCTS Tree Growth

- **search focusses on more interesting parts of the tree**
  - Search is Asymmetric
- **“anytime” graceful exit**
  - search can be stopped anytime and delivers the best outcome so far



Alpha Go

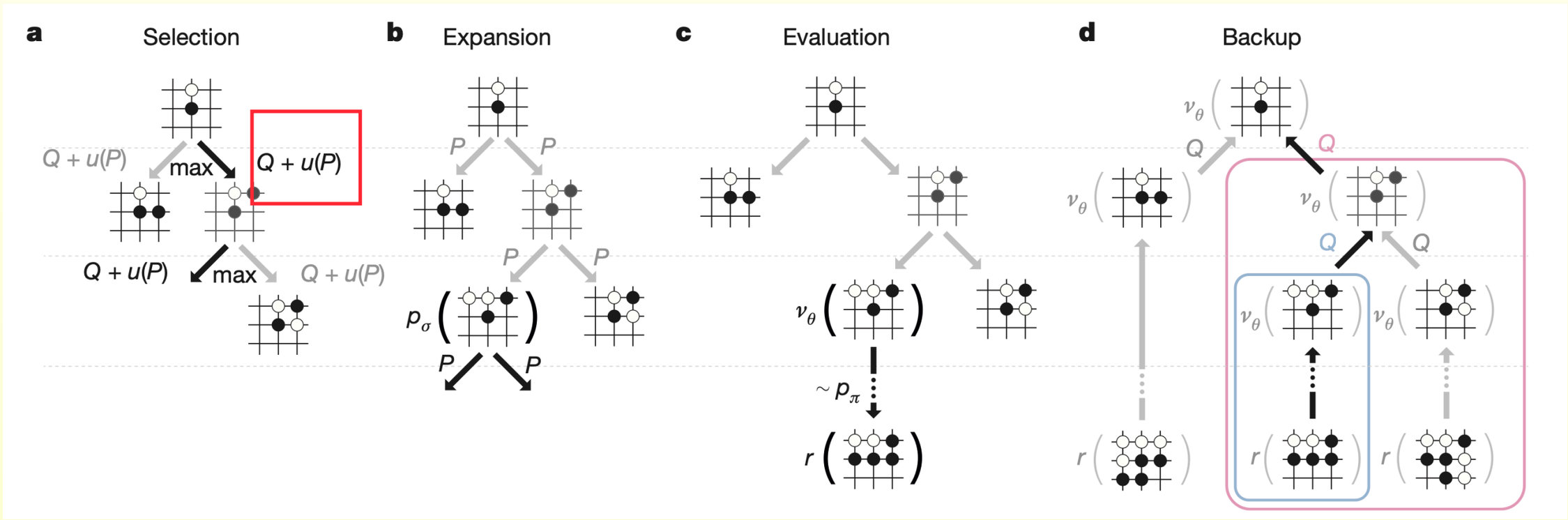
- Combines MCTS with trained neural networks
- During selection, a modified UCB formula has the "exploration factor" combined with the output of a "policy" neural network.



- During rollouts, the value of the rollout is averaged with the output of a "value" neural network

Alpha Go

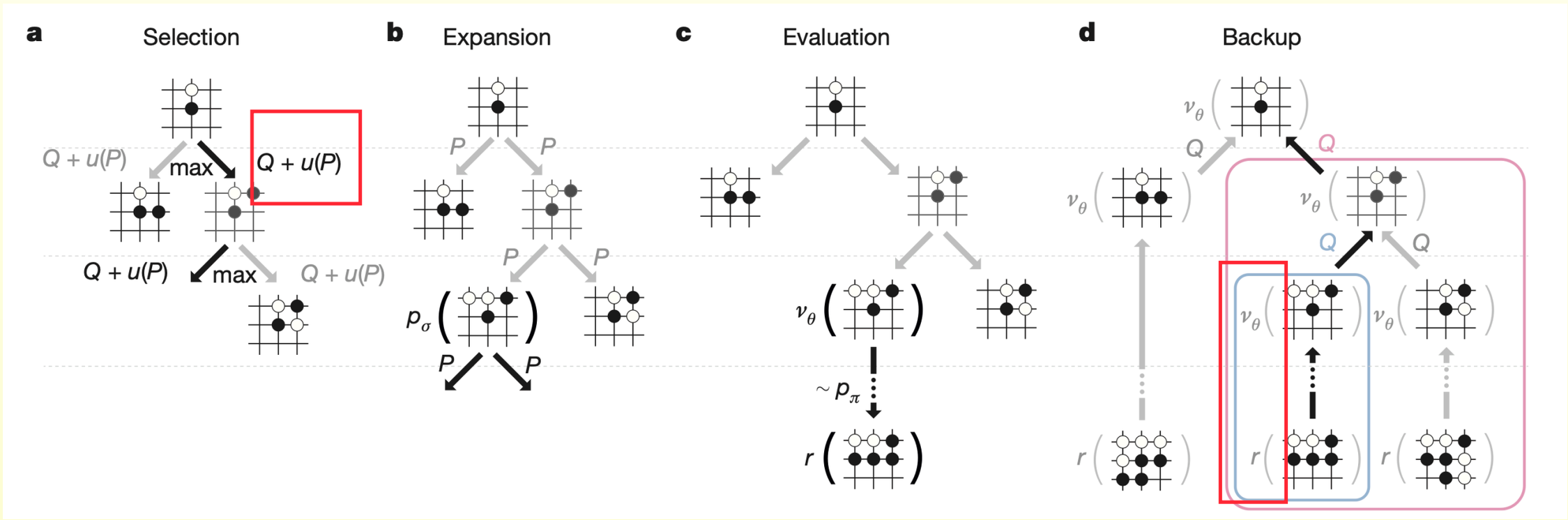
- Combines MCTS with trained neural networks
- During selection, a modified UCB formula has the “exploration factor” combined with the output of a “policy” neural network.



- During rollouts, the value of the rollout is averaged with the output of a “value” neural network

Alpha Go

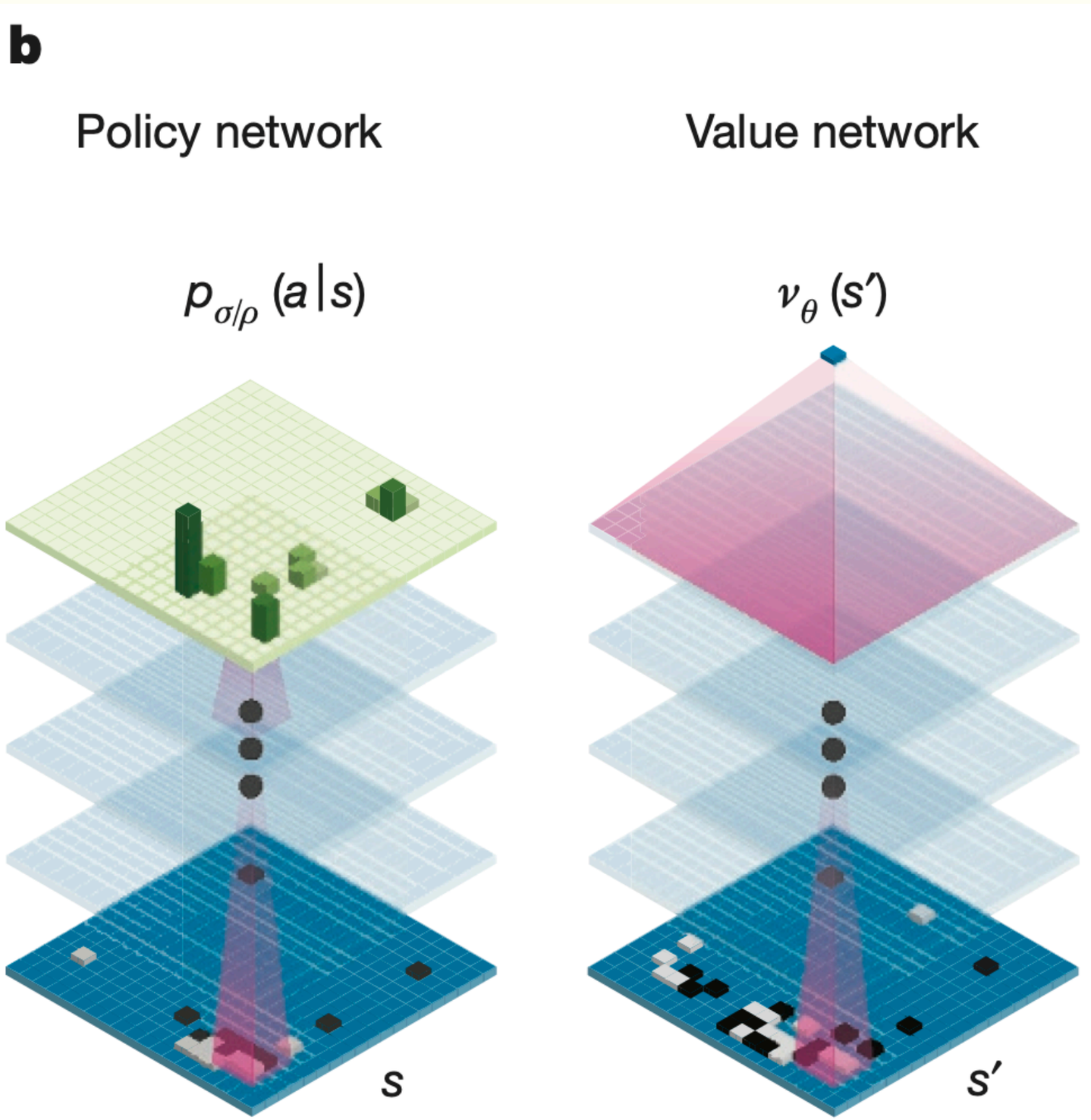
- Combines MCTS with trained neural networks
- During selection, a modified UCB formula has the “exploration factor” combined with the output of a “policy” neural network.



- During rollouts, the value of the rollout is averaged with the output of a “value” neural network

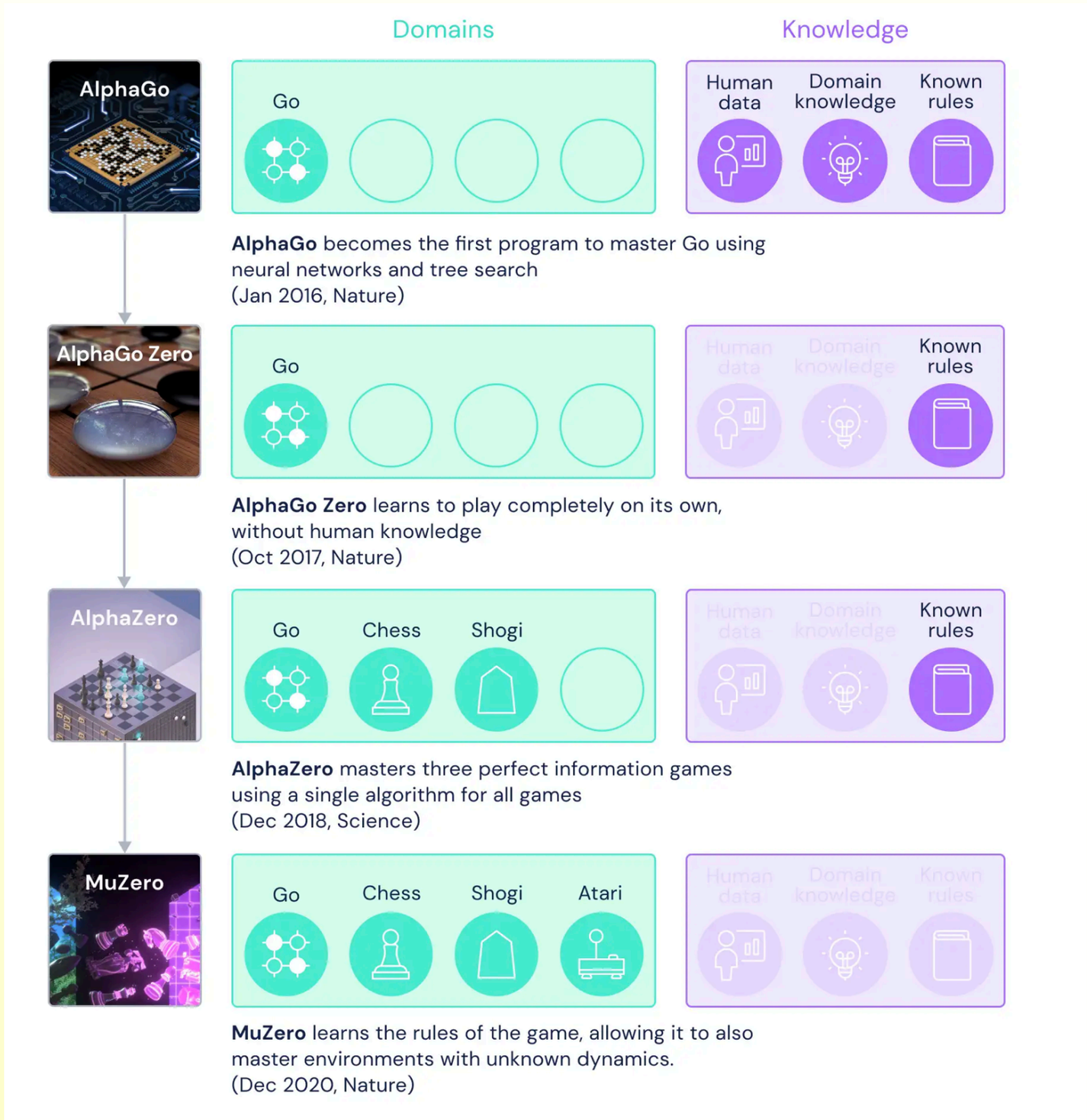
Alpha Go

- Policy and Value network are kickstarted by supervised learning from human demonstrations
- They are then further trained by reinforcement learning via self-play



CAL POLY

Successors of Alpha Go



Source: MuZero: Mastering Go, chess, shogi and Atari without rules (DeepMind Blog, access Oct 21, 2021)

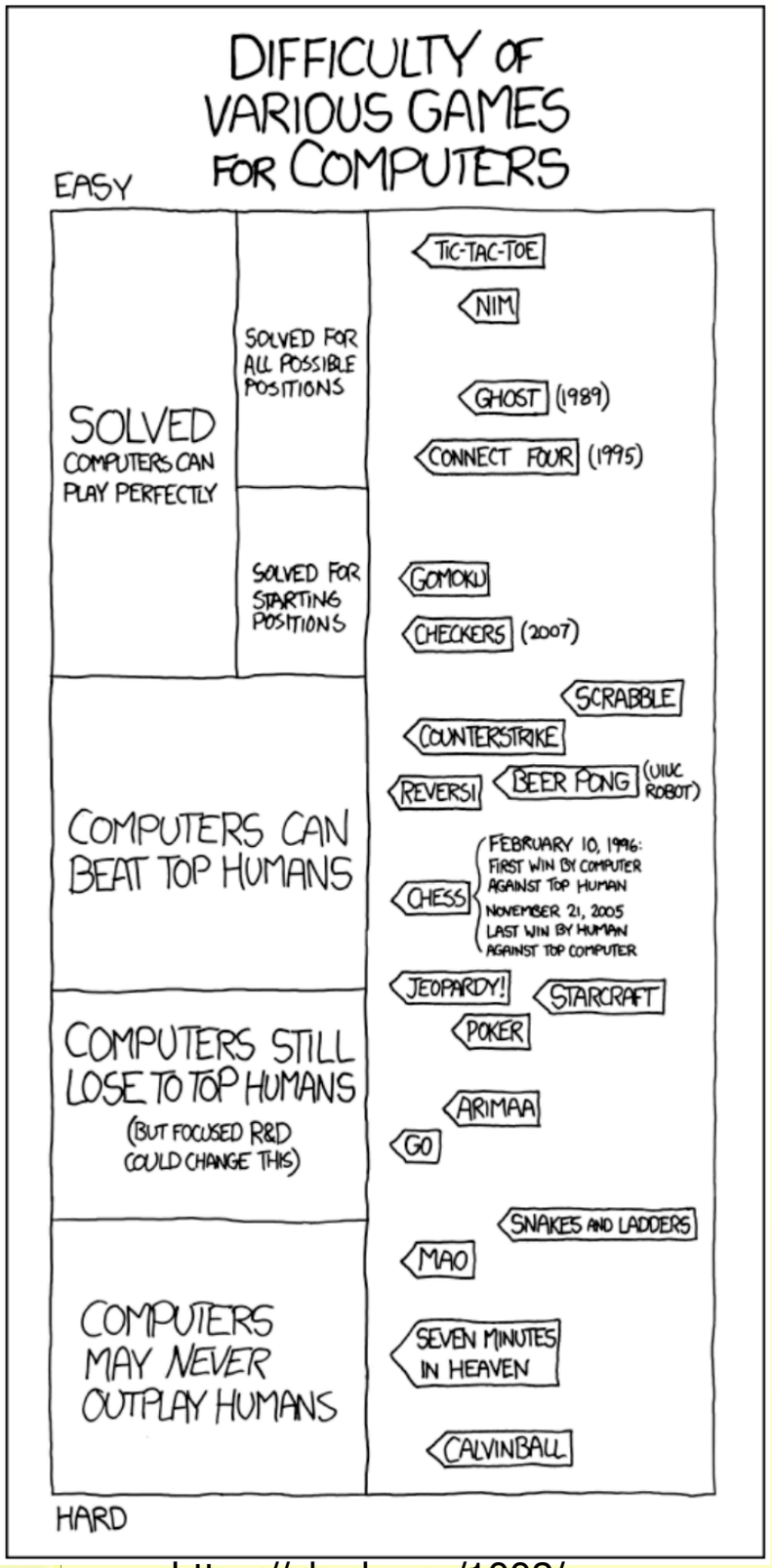
Successors of Alpha Go

- **Alpha Go Zero:**
  - Single neural network combines the role of policy and value networks
  - Trained via self-play alone, no human demonstrations
  - Does not use rollouts (relies purely on network for value estimation)

- **Alpha Zero:**
  - Similar to Alpha Go Zero, but achieves superhuman performance on various games
    - Go
    - Chess
    - Shogi
  - Trained individually for each of these game types

- **Mu-Zero**
  - Does not know game rules in advance
  - Does not use an explicit forward model
  - Environment dynamics are also learned and modeled by a neural network
  - Also plays Atari

XKCD 1002



https://xkcd.com/1002/

- **A bit outdated now…**
  - Go and Poker bots have beaten top humans
  - Starcraft (and Dota2) bots have beaten professional humans
    - Not too consistently
    - With some limitations

MCTS Takeaways

- **Applicable to vastly larger state spaces than most other methods**
- **Does not require heuristics**
  - But heuristics can be incorporated
- **“Anytime” graceful exit**
  - search can be stopped anytime and delivers the best outcome so far
- **Transitions and probabilities do not need to be known in advance**
  - As long as a “forward model” (simulator) is available
- **Drawback: requires lots of memory**
  - Every node must be kept in memory
- **Drawback: no guarantee of completeness or optimality**