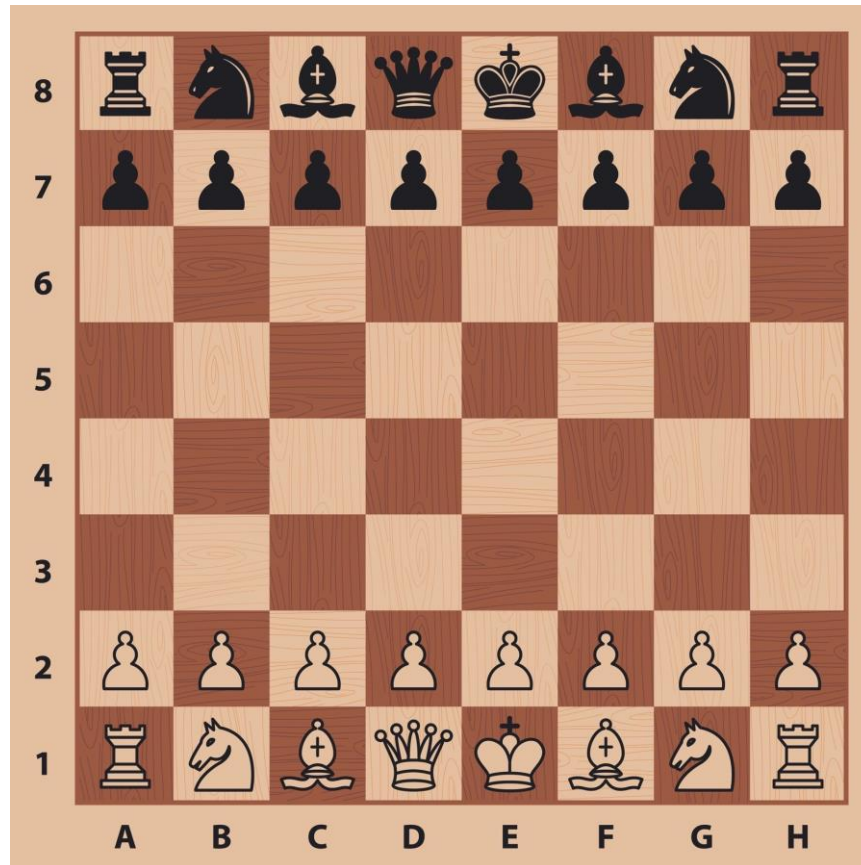


Artificial Intelligence

Game Playing



Artificial intelligence : Games

- Games are a big deal in AI, Tictactoe, backgammon etc.
- Games are interesting to AI because they are too hard to solve
- One of the first examples of AI is the computerized game of [Nim](#) made in 1951
- https://www.archimedes-lab.org/game_nim/play_nim_game.html

Artificial intelligence : Games

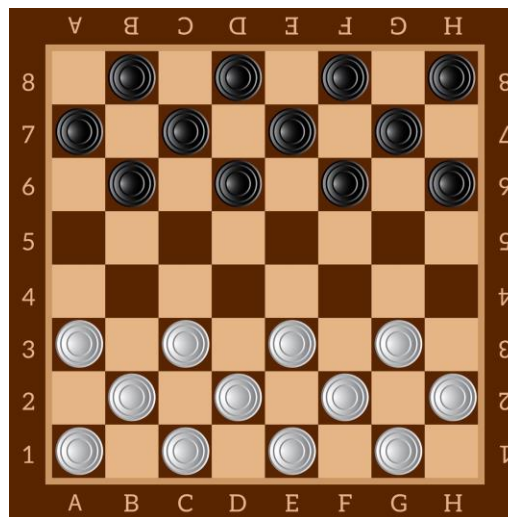
- One of the first examples of AI is the computerized game of [Nim](#) made in 1951
- https://www.archimedes-lab.org/game_nim/play_nim_game.html
- In 1951, using the [Ferranti Mark 1](#) machine of the [University of Manchester](#), [Christopher Strachey](#) wrote a [checkers](#) program and [Dietrich Prinz](#) wrote one for [chess](#)
- [Arthur Samuel](#)'s checkers program developed in 1960s introduced several concepts on game playing - search tree, pruning, heuristics
- Coined the term machine learning – “*rote learning* (or generalization learning)”
- https://www.youtube.com/watch?v=iT_Un3xo1qE



Game playing machines

Checkers:

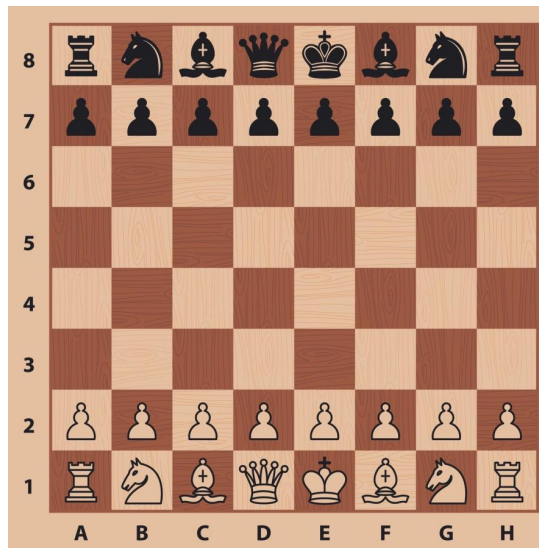
- **Chinook** ended 40-year-reign of human world champion Marion Tinsley in 1994.
- Used an endgame database defining perfect play for all positions involving 8 or fewer pieces on the board, a total of 443,748,401,247 positions.
- <https://cardgames.io/checkers/>



Game playing machines

Chess:

- In 1949, Claude E. Shannon in his paper “Programming a Computer for Playing Chess”, suggested *Chess* as an AI problem for the community.
- Deep Blue defeated human world champion Gary Kasparov in a six-game match in 1997.
- In 2006, Vladimir Kramnik, the undisputed world champion, was defeated 4-2 by Deep Fritz.



Deep Blue : Man vs Machine

Deep Thought : chess playing machine project started in CMU

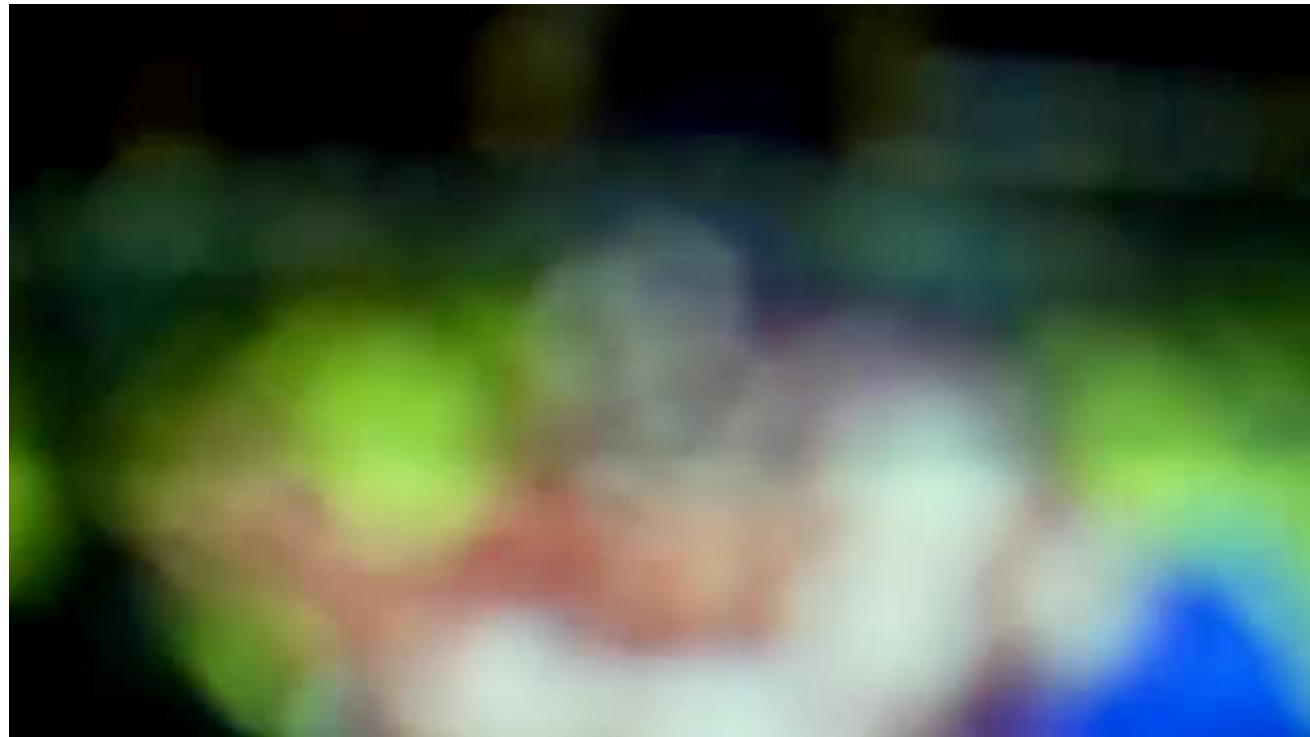
Deep Blue : First computer to beat a chess champion in 1996

Kasparov said that he sometimes saw deep intelligence and creativity in the machine's moves



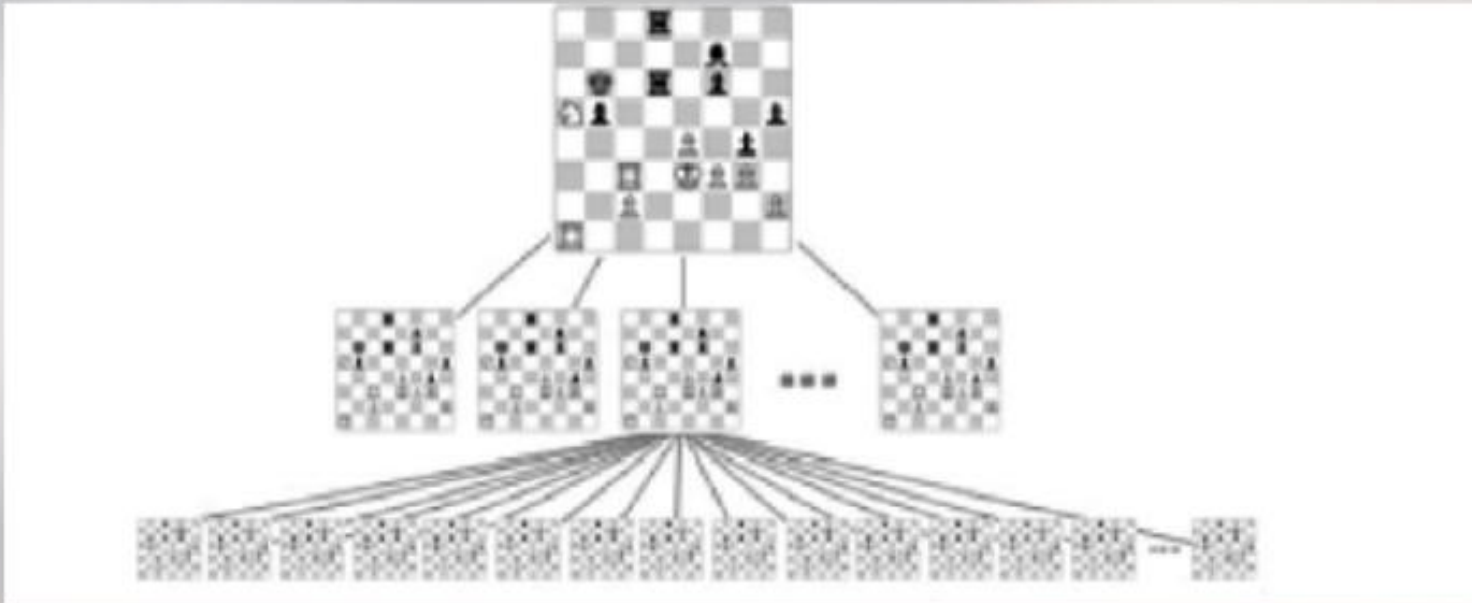
Game playing machines

Go: $b > 300!$ Google Deep mind Project AlphaGo. In 2016, AlphaGo beat both Fan Hui, the European Go champion and Lee Sedol the worlds best player.



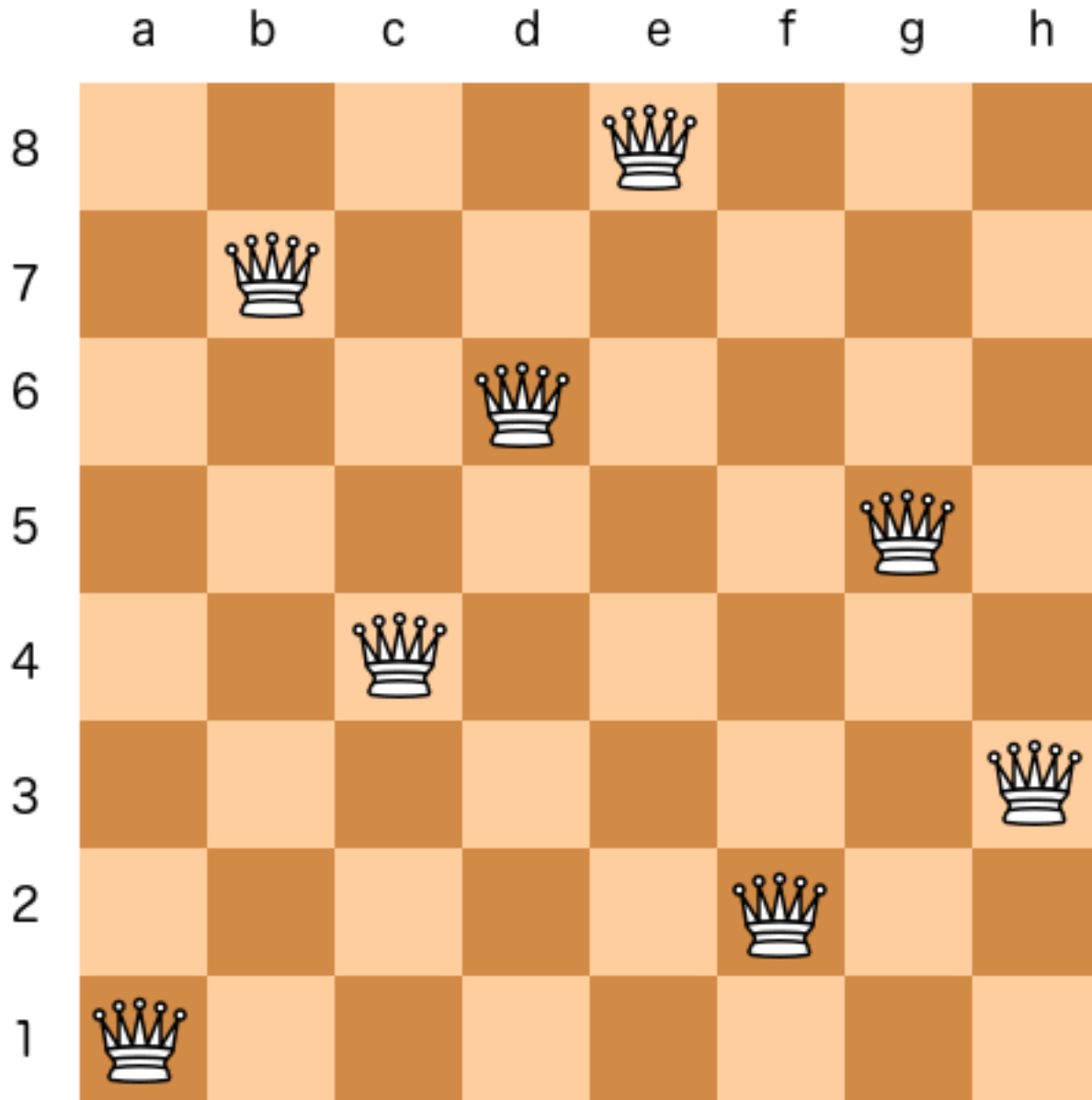
Game playing : Tree search

Tree Search



"Programming a Computer for Playing Chess"

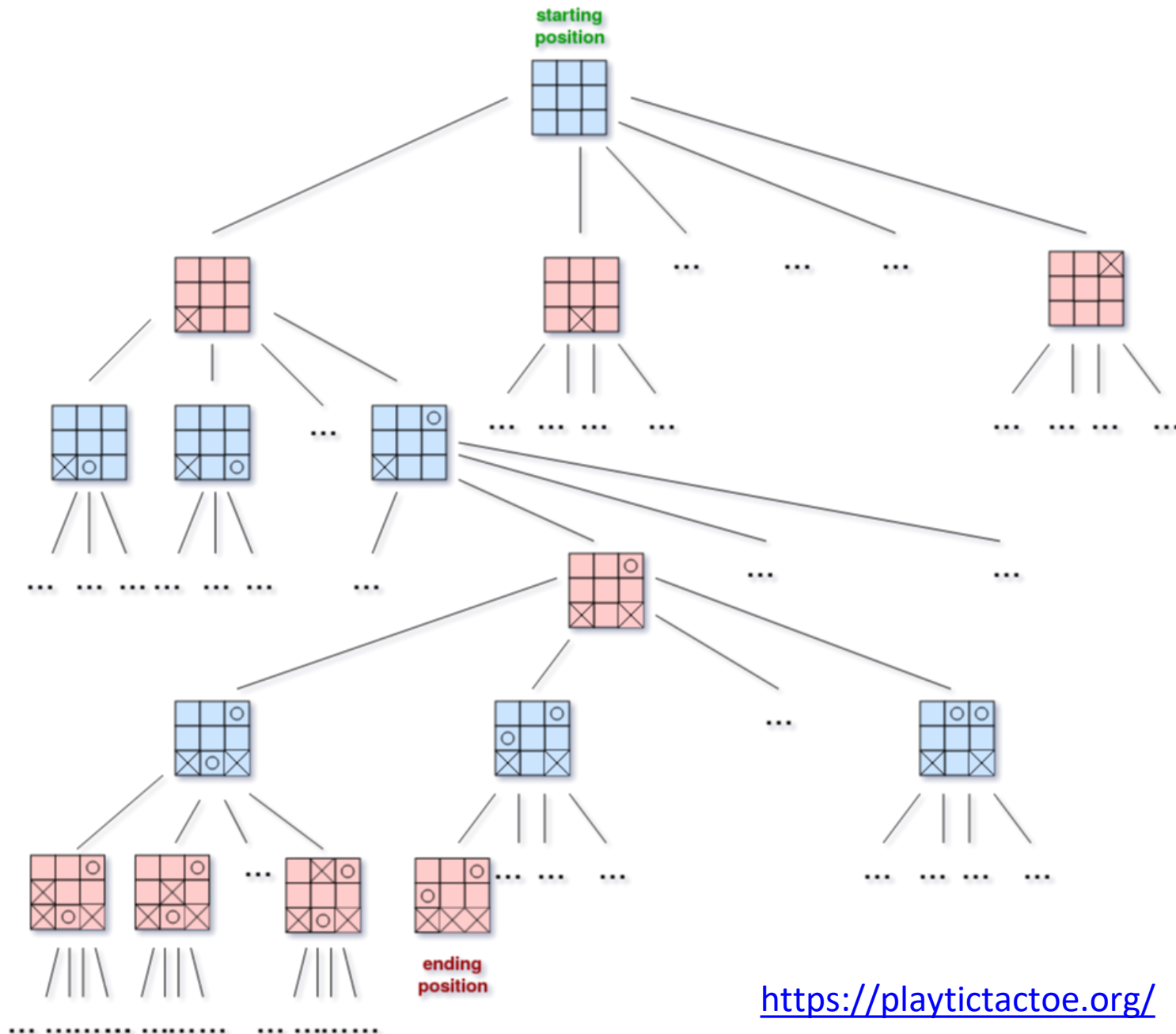
Game playing : 8 queens



<https://mypuzzle.org/sliding>

<https://www.brainmetrix.com/8-queens/>

Game Playing : Tic Tac Toe



<https://playtictactoe.org/>

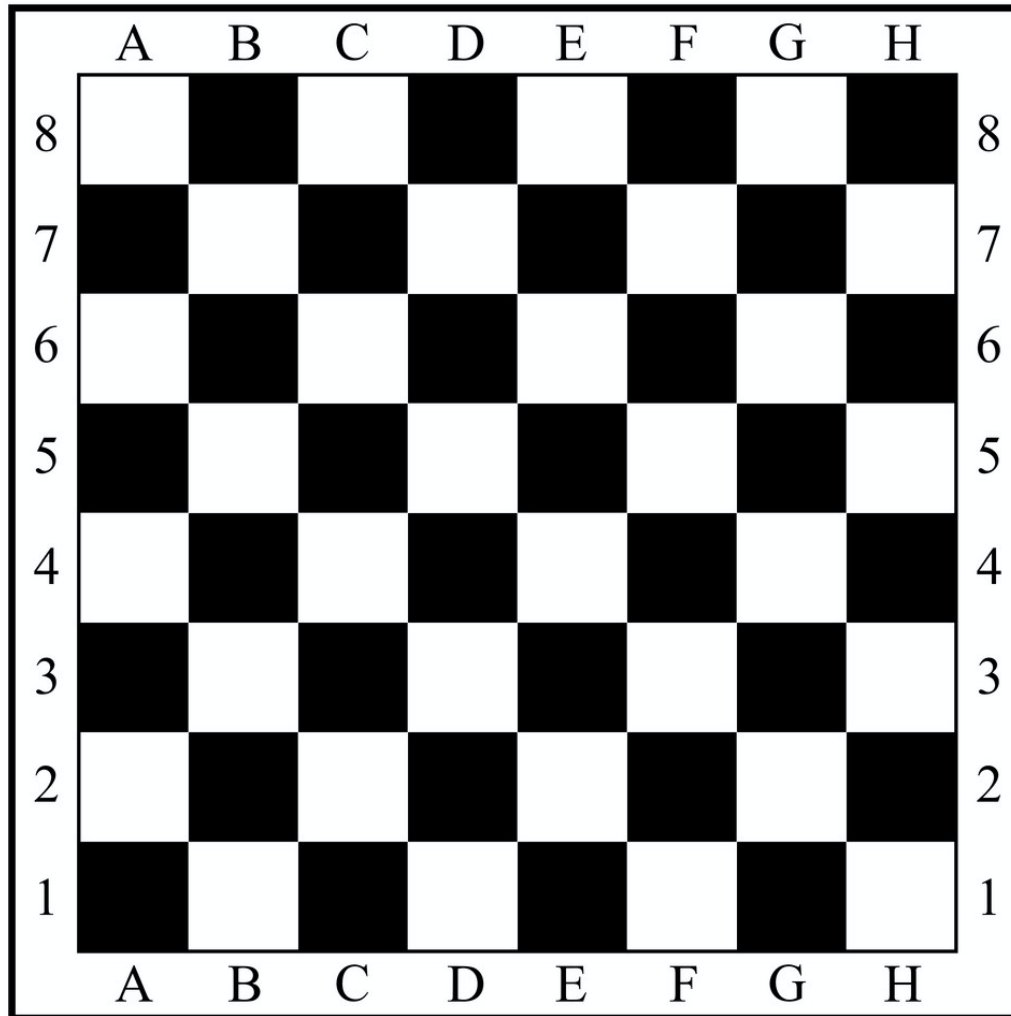
“Search” Solutions !

- **Goal formulation**, based on the current situation and the agent's performance measure, is the first step in problem solving.
- **Problem formulation** is the process of deciding what actions and states to consider, given a goal.
- The process of looking for a sequence of actions that reaches the goal is called **search**.
- A search algorithm takes a problem as input and returns a solution in the form of an action sequence. Once a solution is found, the actions it recommends can be carried out. This is called the **execution phase**.
- Intelligent agents use the design philosophy “**formulate, search, execute**” design

8 Queens Puzzle

Goal of the 8-queens problem is to place eight queens on a chessboard such that no queen attacks any other. (A queen attacks any piece in the same row, column or diagonal.)

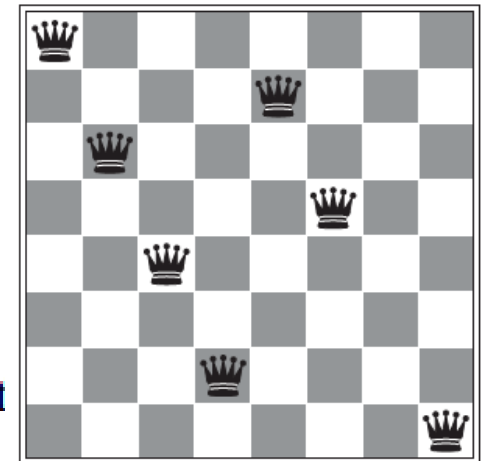
<http://brainmetrix.com/8-queens/>



8 Queens Puzzle

- **States:** Any arrangement of 0 to 8 queens on the board is a state.
- **Initial state:** No queens on the board.
- **Actions:** Add a queen to any empty square.
- **Transition model:** Returns the board with a queen added to the specified square.
- **Goal test:** 8 queens are on the board, none attacked.

$64 \cdot 63 \cdots 57 \approx 1.8 \times 10^{14}$ possible sequences to investigate.



- **States:** All possible arrangements of n queens ($0 \leq n \leq 8$), one per column in the leftmost n columns, with no queen attacking another.
- **Actions:** Add a queen to any square in the leftmost empty column such that it is not attacked by any other queen.

8 puzzle problem

• <http://mypuzzle.org/sliding>

- **States:** A state description specifies the location of each of the eight tiles and the blank in one of the nine squares.
- **Initial state:** Any state can be designated as the initial state. Note that any given goal can be reached from exactly half of the possible initial states (Exercise 3.4).
- **Actions:** The simplest formulation defines the actions as movements of the blank space *Left*, *Right*, *Up*, or *Down*. Different subsets of these are possible depending on where the blank is.
- **Transition model:** Given a state and action, this returns the resulting state; for example, if we apply *Left* to the start state in Figure 3.4, the resulting state has the 5 and the blank switched.
- **Goal test:** This checks whether the state matches the goal configuration shown in Figure 3.4. (Other goal configurations are possible.)
- **Path cost:** Each step costs 1, so the path cost is the number of steps in the path.

7	2	4
5		6
8	3	1

Start State

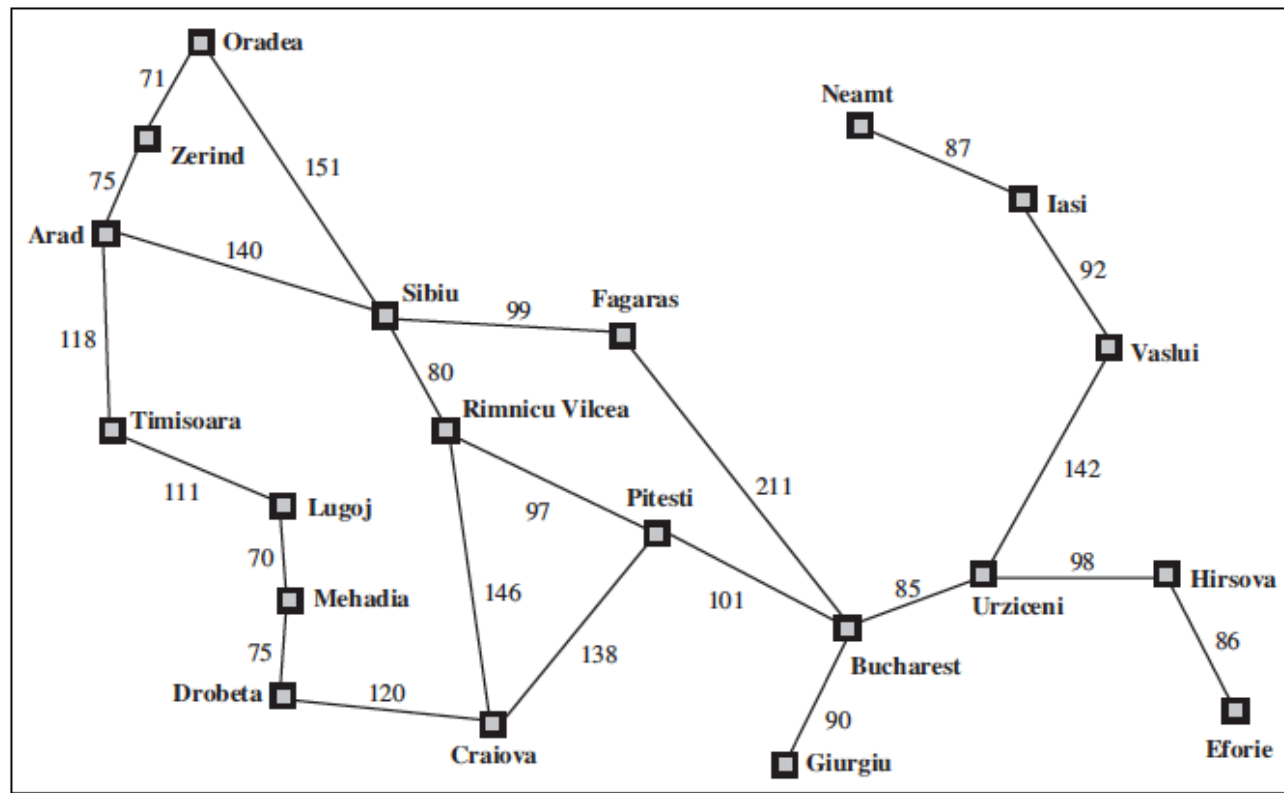
	1	2
3	4	5
6	7	8

Goal State

Travelling Problem

- The **initial state** that the agent starts : In(Arad).
- **ACTIONS(s)** returns the set of actions From In(Arad), the applicable
 - actions are {Go(Sibiu), Go(Timisoara), Go(Zerind)}.
- **TRANSITION MODEL** model, specified by a function **RESULT(s, a)** that returns the state that results from action a in state
 - $\text{RESULT}(\text{In}(\text{Arad}), \text{Go}(\text{Zerind})) = \text{In}(\text{Zerind})$.
- **GOAL STATE** {In(Bucharest)}.
- **PATH COST** cost function that assigns a numeric cost to each path.

Find the path from Arad to Bucharest

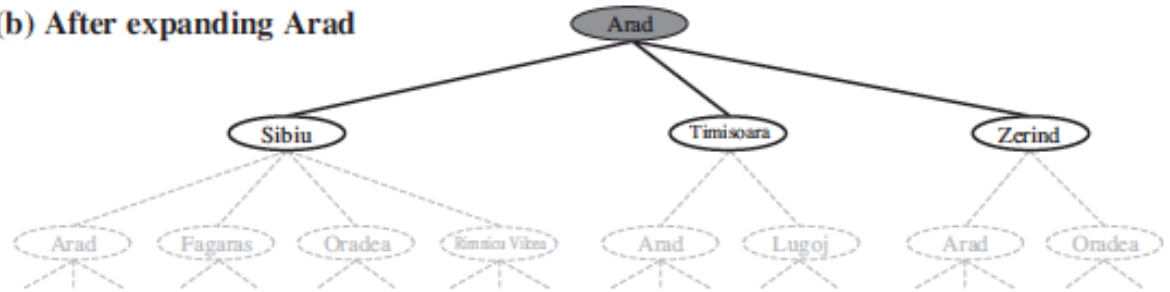


Search Trees

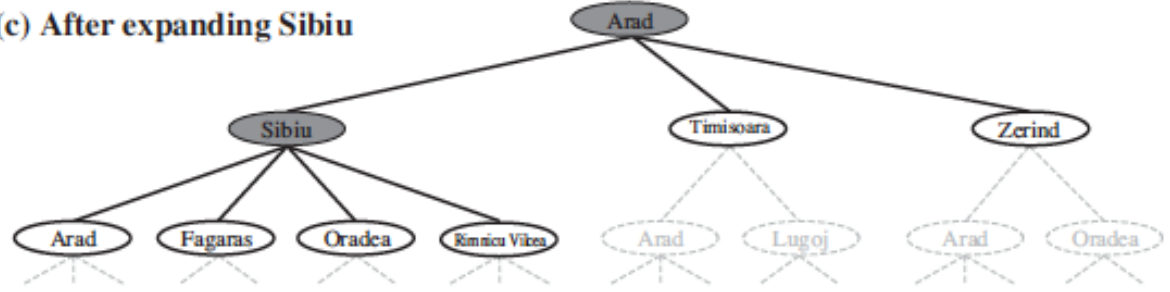
(a) The initial state



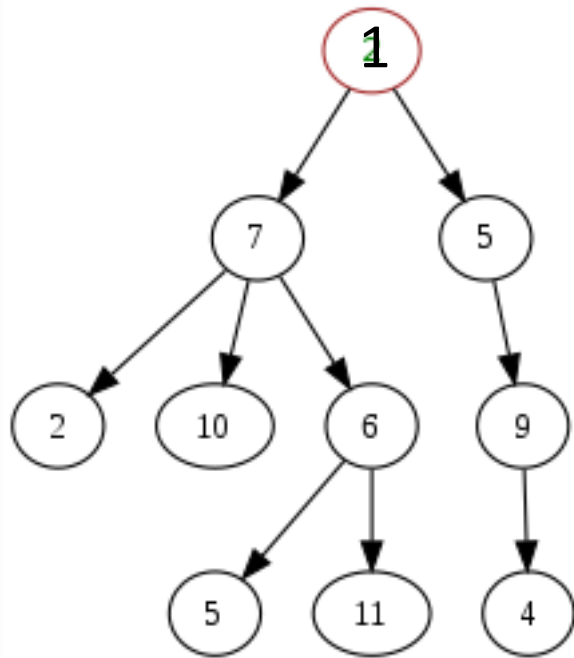
(b) After expanding Arad



(c) After expanding Sibiu



Tree Terminology



Node — denotes some state/value

Root — The top node in a tree, the prime ancestor.

Child — A node directly connected to another node when moving away from the root, an immediate descendant.

Parent — The converse notion of a child, an immediate ancestor.

Leaf — A node with no children.

Internal node — A node with at least one child.

Edge — The connection between one node and another.

Depth — The distance between a node and the root.

Level — the number of edges between a node and the root + 1

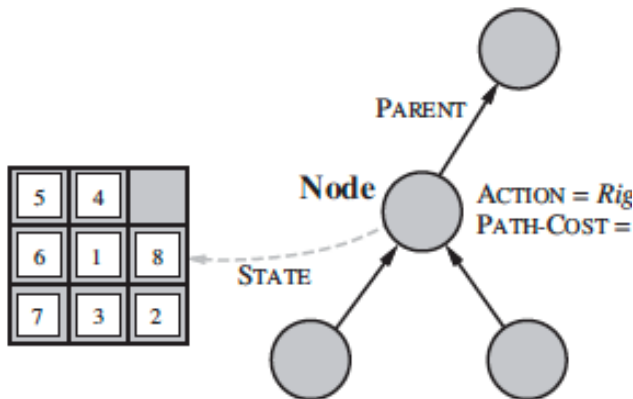
Height — The number of edges on the longest path between a node and a descendant leaf.

Breadth — The number of leaves.

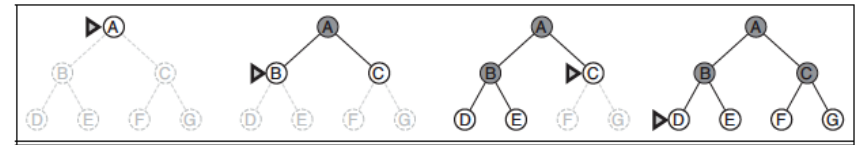
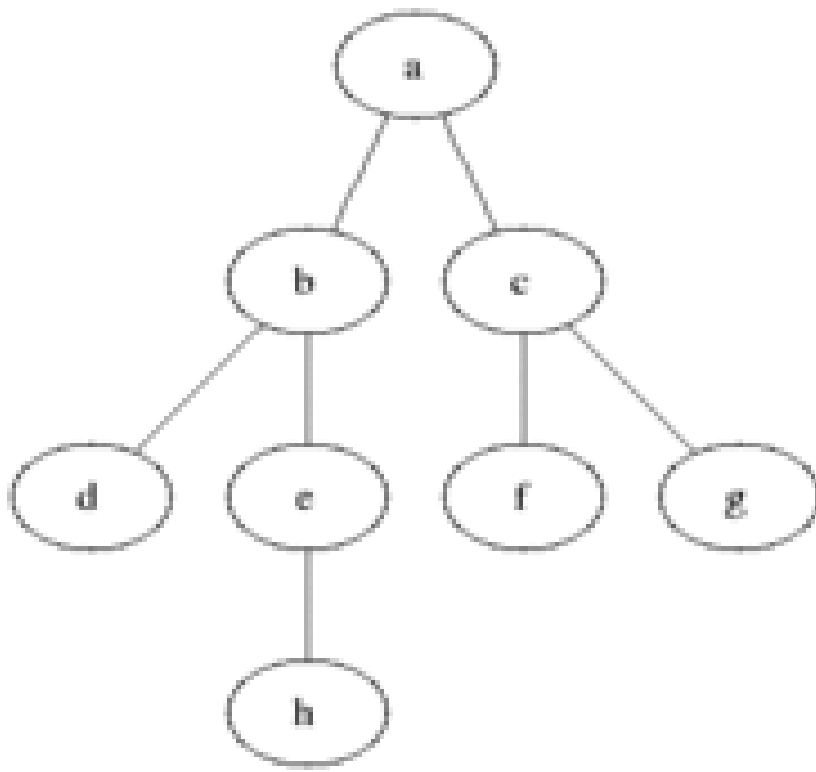
Sub Tree — A tree T is a tree consisting of a node in T and all of its descendants in T .

Binary Tree — is a tree data structure in which each node has at most two children, which are referred to as the left child and the right child.

Branching Factor — maximum number of branches for any node



Breadth first search



Breadth first search

Depth	Nodes	Time	Memory
2	110	.11 milliseconds	107 kilobytes
4	11,110	11 milliseconds	10.6 megabytes
6	10^6	1.1 seconds	1 gigabyte
8	10^8	2 minutes	103 gigabytes
10	10^{10}	3 hours	10 terabytes
12	10^{12}	13 days	1 petabyte
14	10^{14}	3.5 years	99 petabytes
16	10^{16}	350 years	10 exabytes

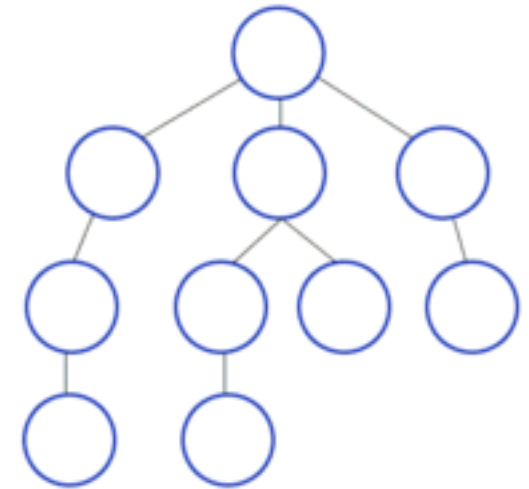
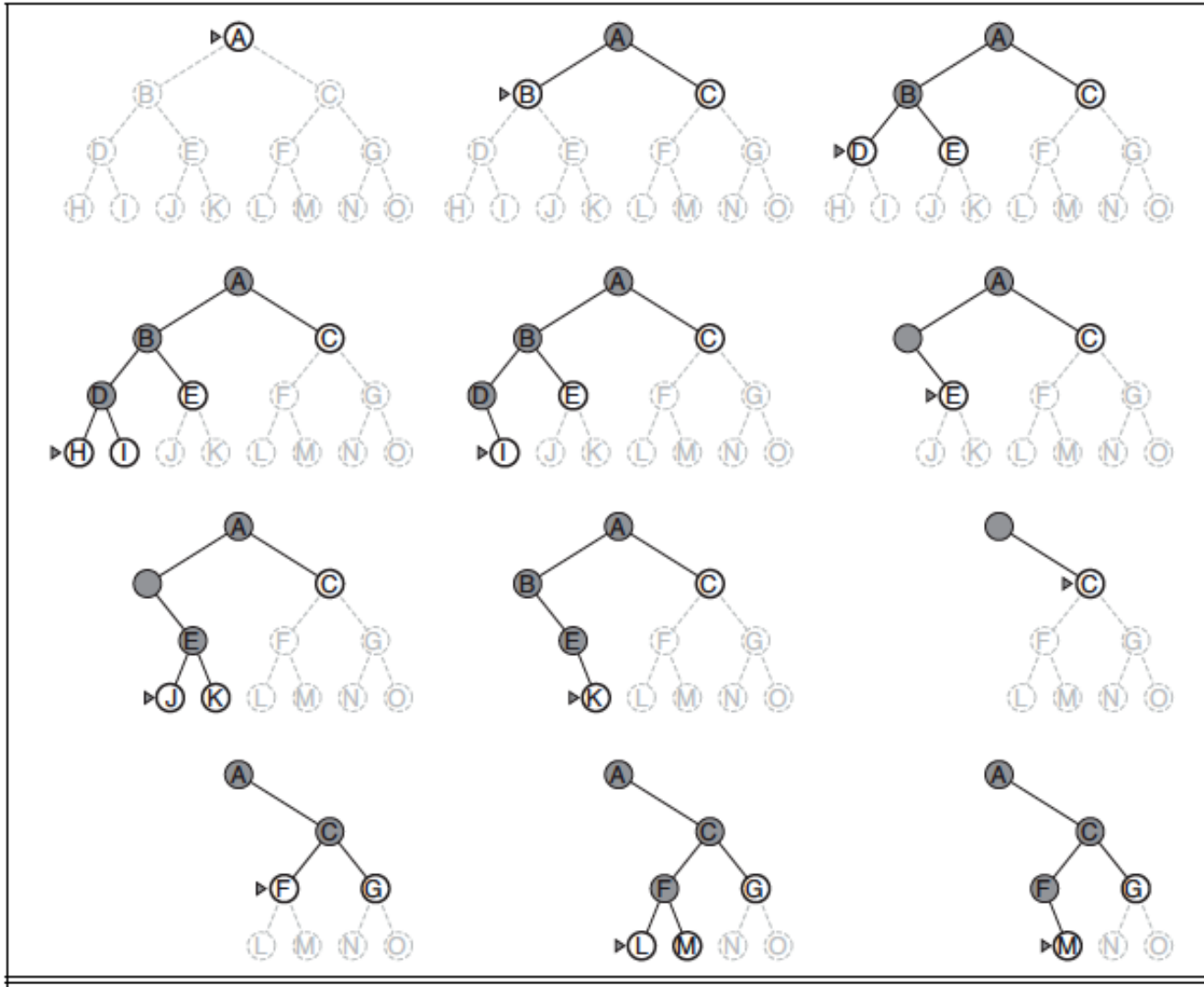
Figure 3.13 Time and memory requirements for breadth-first search. The numbers shown assume branching factor $b = 10$; 1 million nodes/second; 1000 bytes/node.

memory requirements are a bigger problem for breadth-first search than is the execution time.

$$b + b^2 + b^3 + \dots + b^d = O(b^d) .$$

exponential-complexity search problems cannot be solved by uninformed methods for any but the smallest instances.

Depth First Search

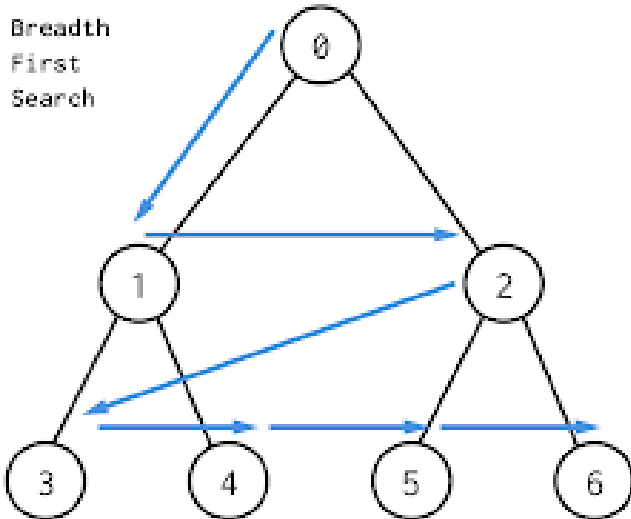


depth-first search requires storage of only $O(bd)$ nodes, depth-first search would require 156 kilobytes instead of 10 exabytes at depth $d = 16$, a factor of 7 trillion times less space.

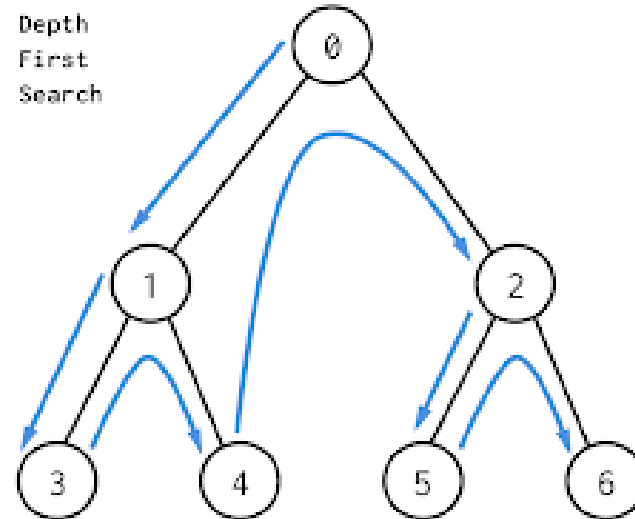
Time complexity : $O(b^d)$

BFS vs DFS

Breadth
First
Search



Depth
First
Search



DFS



BFS



search



Informed Search : Best first search

a node is selected for expansion based on an

evaluation function, $f(n)$.

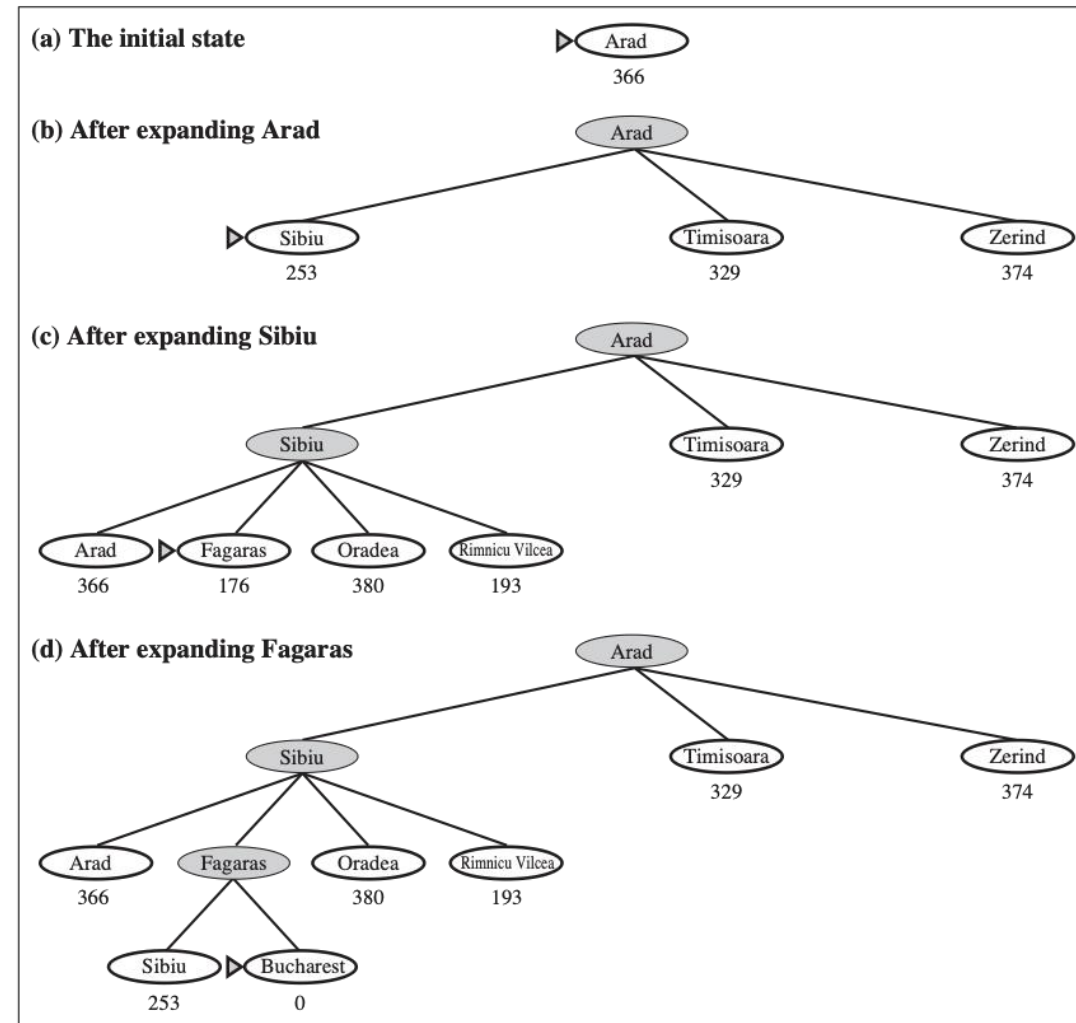
a component of f a **heuristic function**, denoted $h(n)$:

$h(n)$ = estimated cost of the cheapest path from the state at node n to a goal state. if n is a goal node, then

$h(n) = 0$.

straight- line distance heuristic,

in Romania, one might estimate the cost of the cheapest path from Arad to Bucharest via the straight-line distance from Arad to Bucharest.



Informed Search : A* search

Evaluates nodes by combining

$g(n)$: the cost to reach the node, and

$h(n)$: the cost to get from the node to the goal:

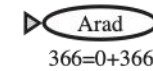
$$f(n) = g(n) + h(n)$$

$f(n)$ = estimated cost of the cheapest solution through n .

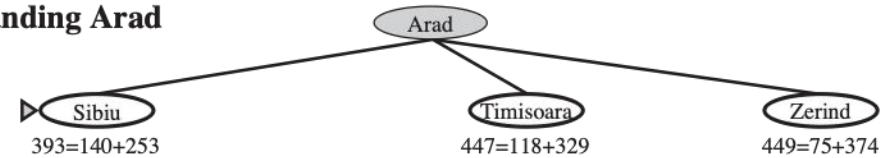
Try first is the node with the lowest value of

$$g(n) + h(n)$$

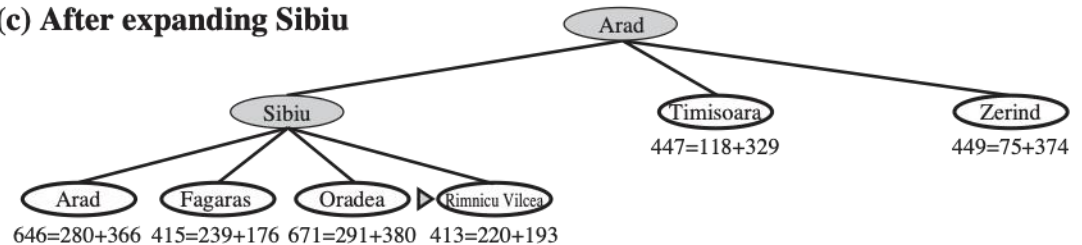
(a) The initial state



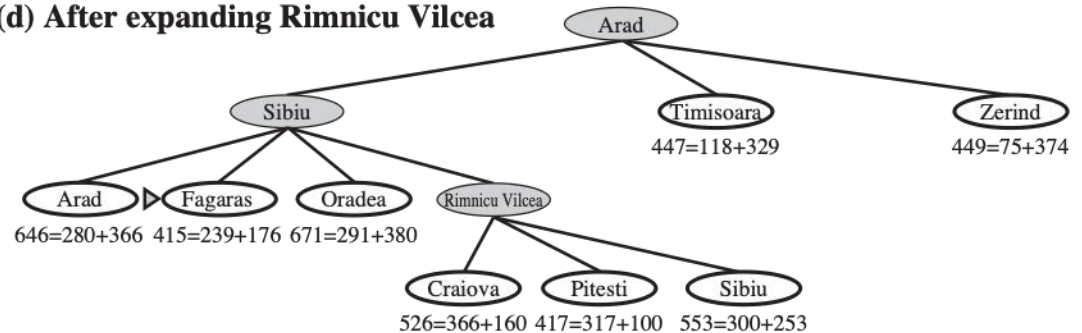
(b) After expanding Arad



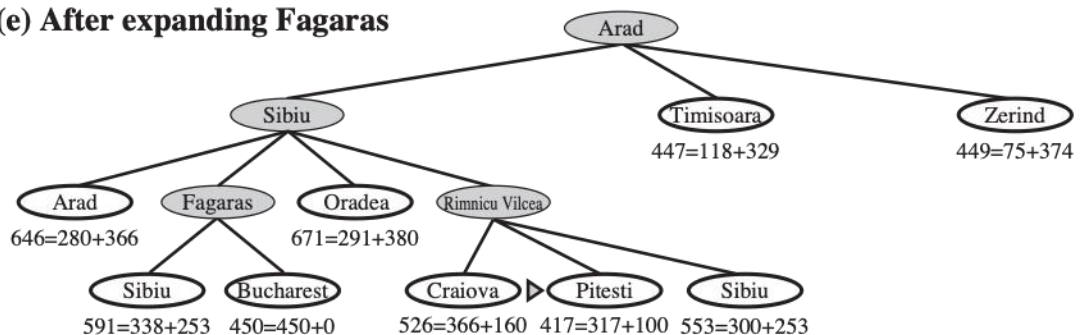
(c) After expanding Sibiu



(d) After expanding Rimnicu Vilcea



(e) After expanding Fagaras



Types of games

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

We are mostly interested in deterministic games, fully observable environments, zero-sum, where two agents act alternately.

Zero-sum Games

- Adversarial: Pure competition.
- Agents have different values on the outcomes.
- One agent maximizes one single value, while the other minimizes it.
- Each move by one of the players is called a “ply.”

One function: one agents maximizes it and one minimizes it!

Embedded thinking...

Embedded thinking or backward reasoning!



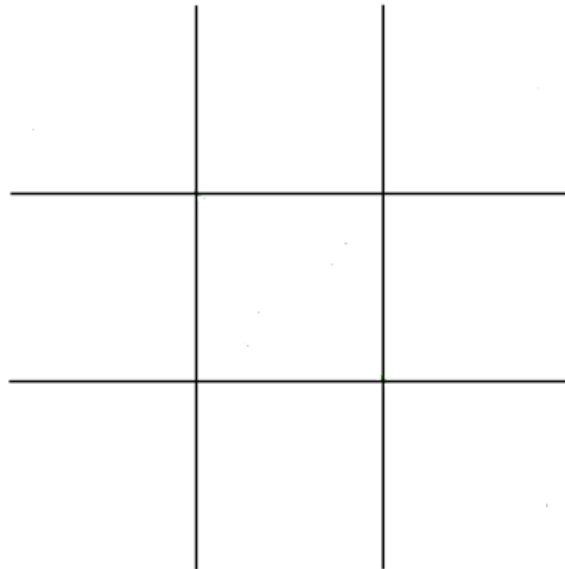
- One agent is trying to figure out what to do.
- How to decide? He thinks about the consequences of the possible actions.
- He needs to think about his opponent as well...
- The opponent is also thinking about what to do etc.
- Each will imagine what would be the response from the opponent to their actions.
- This entails an embedded thinking.

Adversarial Search

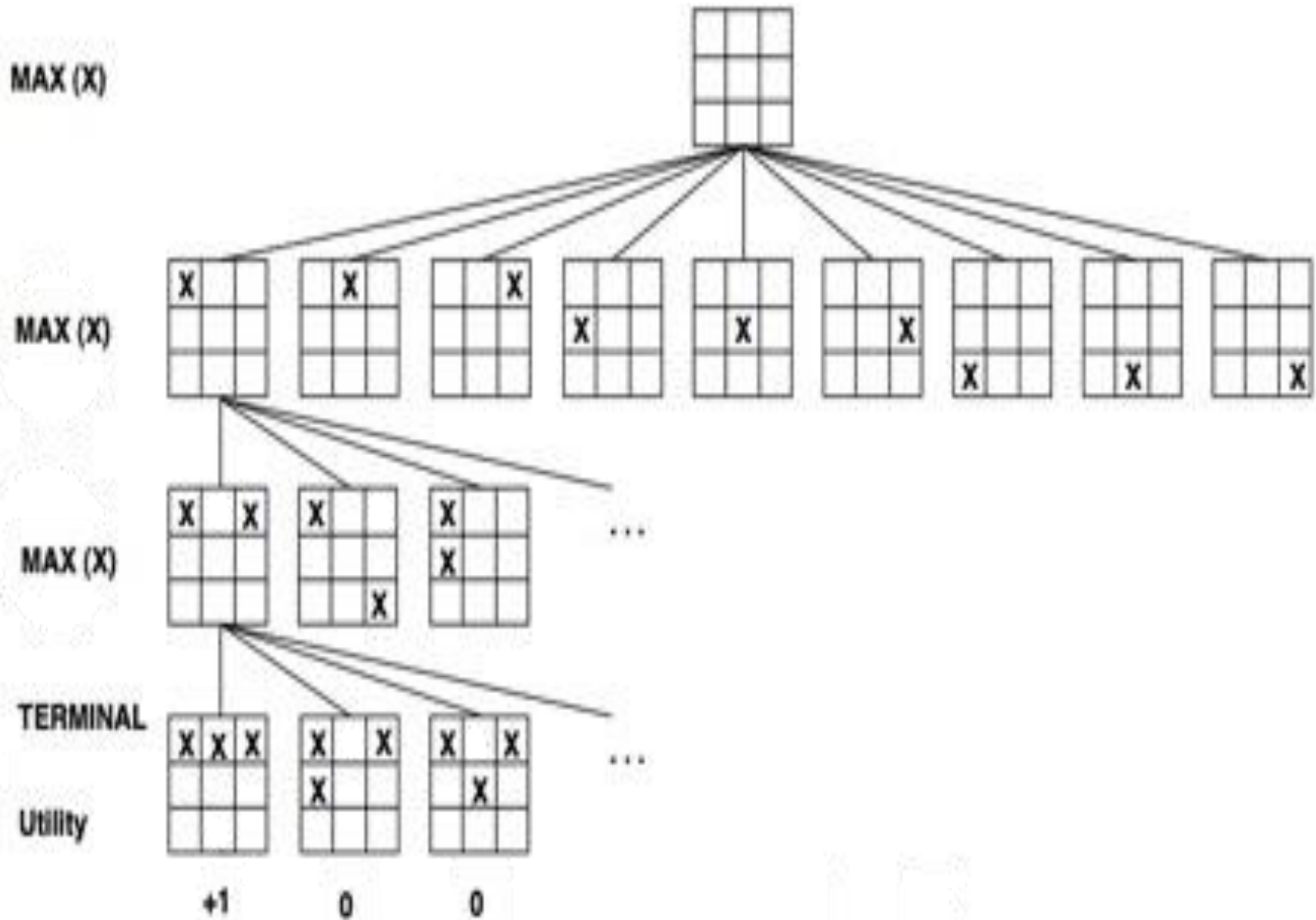
- There is an **opponent** we can't control planning again us!
- Game: optimal solution is not a sequence of actions but a **strategy** (policy) If opponent does a, agent does b, else if opponent does c, agent does d, etc.
- Tedious and fragile if hard-coded (i.e., implemented with rules)
- Good news: Games are modeled as **search problems** and use **heuristic evaluation** functions.

Single player...

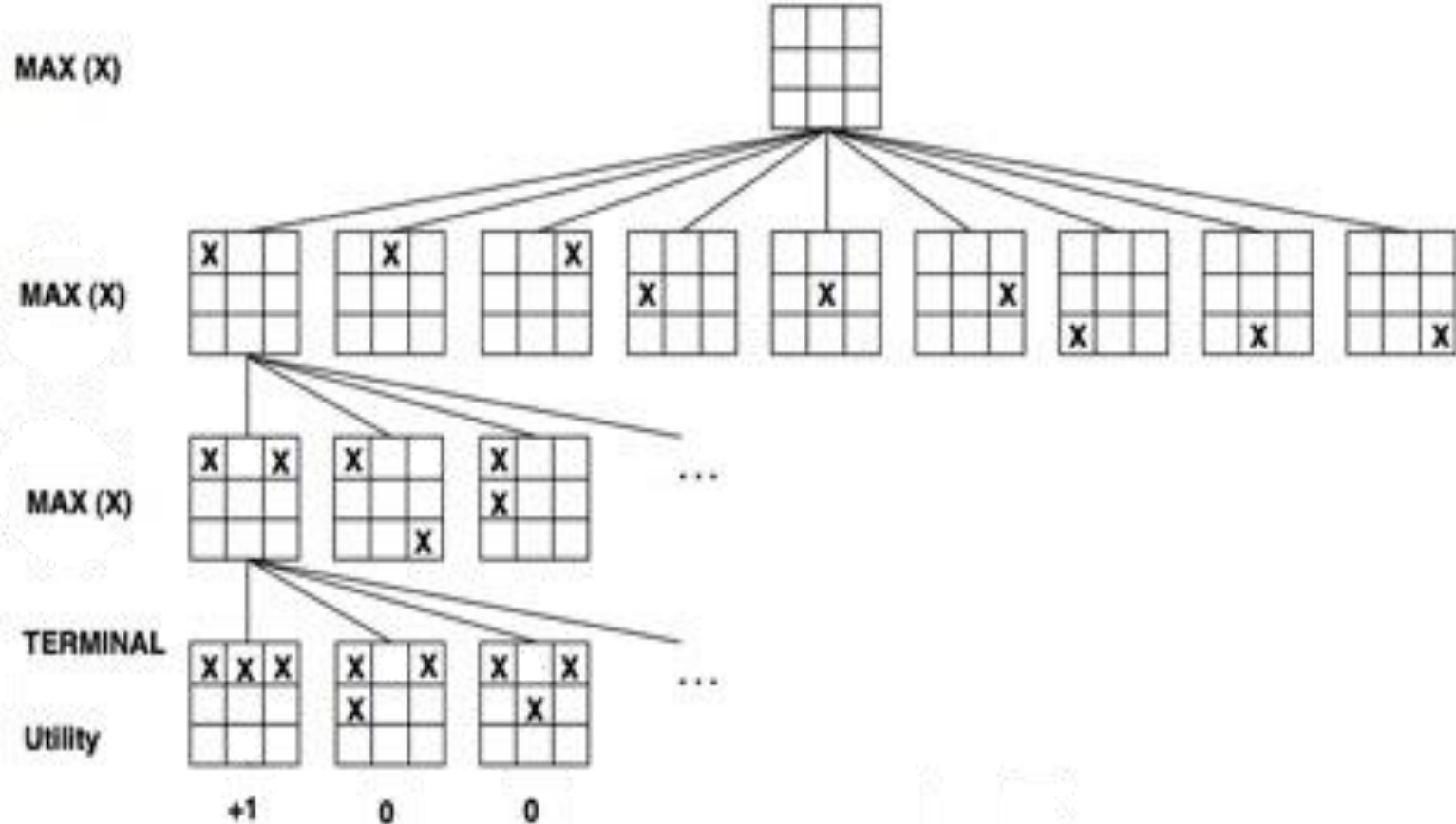
Assume we have a tic-tac-toe with one player.
Let's call him Max and have him play three moves only
for the sake of the example.



Single player...




Single player...

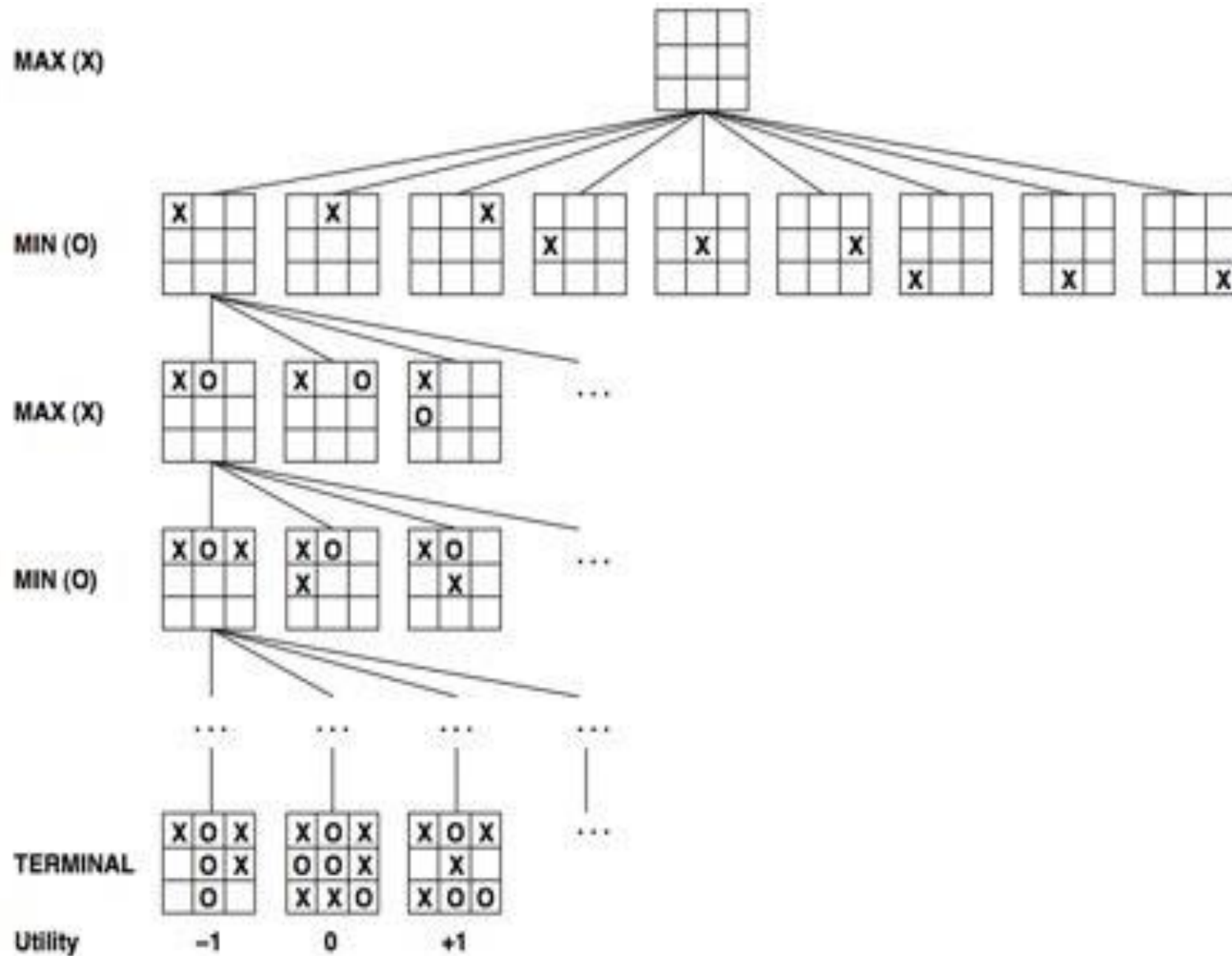


In the case of one player, nothing will prevent Max from winning (choose the path that leads to the desired utility here 1), unless there is another player who will do everything to make Max lose, let's call him Min (the Mean :))

Adversarial search: minimax

- Two players: Max and Min
- Players alternate turns
- Max moves first
- Max maximizes results 
- Min minimizes the result
- Compute each node's minimax value's the best achievable utility against an optimal adversary
- Minimax value = best achievable payoff against best play

Minimax example



Formalization

- The **initial state**
- $P\ player(s)$: defines which player has the move in state s . Usually taking turns.
- $Actions(s)$: returns the set of legal moves in s
- **Transition** function: $s * A \rightarrow s$ defines the result of a move
- **Terminal test**: True when the game is over, False otherwise. States where game ends are called **terminal states**
- **Utility(s,p)**: **utility function** or objective function for a game that ends in terminal state s for player p . In Chess, the outcome is a win, loss, or draw with values $+1$, 0 , $1/2$. For tic-tac-toe we can use a utility of $+1$, -1 , 0 .

Adversarial search: minimax

- Find the optimal strategy for Max:
 - Depth-first search of the game tree
 - An optimal leaf node could appear at any depth of the tree
 - Minimax principle: compute the utility of being in a state assuming both players play optimally from there until the end of the game
 - Propagate minimax values up the tree once terminal nodes are discovered

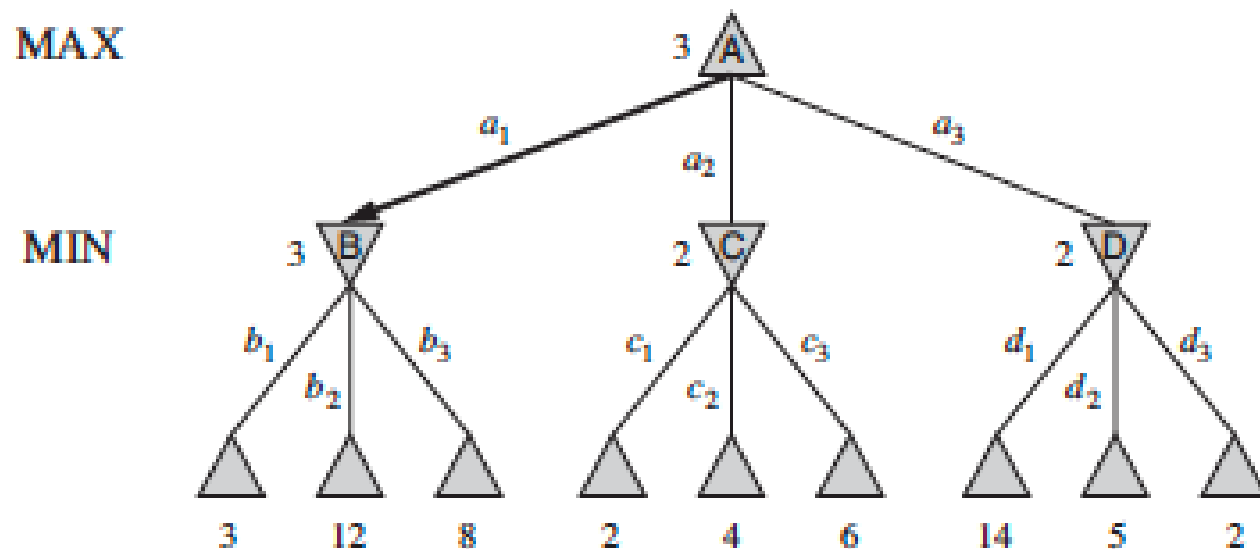
Adversarial search: minimax

- If state is terminal node: Value is utility(state)
- If state is MAX node: Value is highest value of all successor node values (children)
- If state is MIN node: Value is lowest value of all successor node values (children)

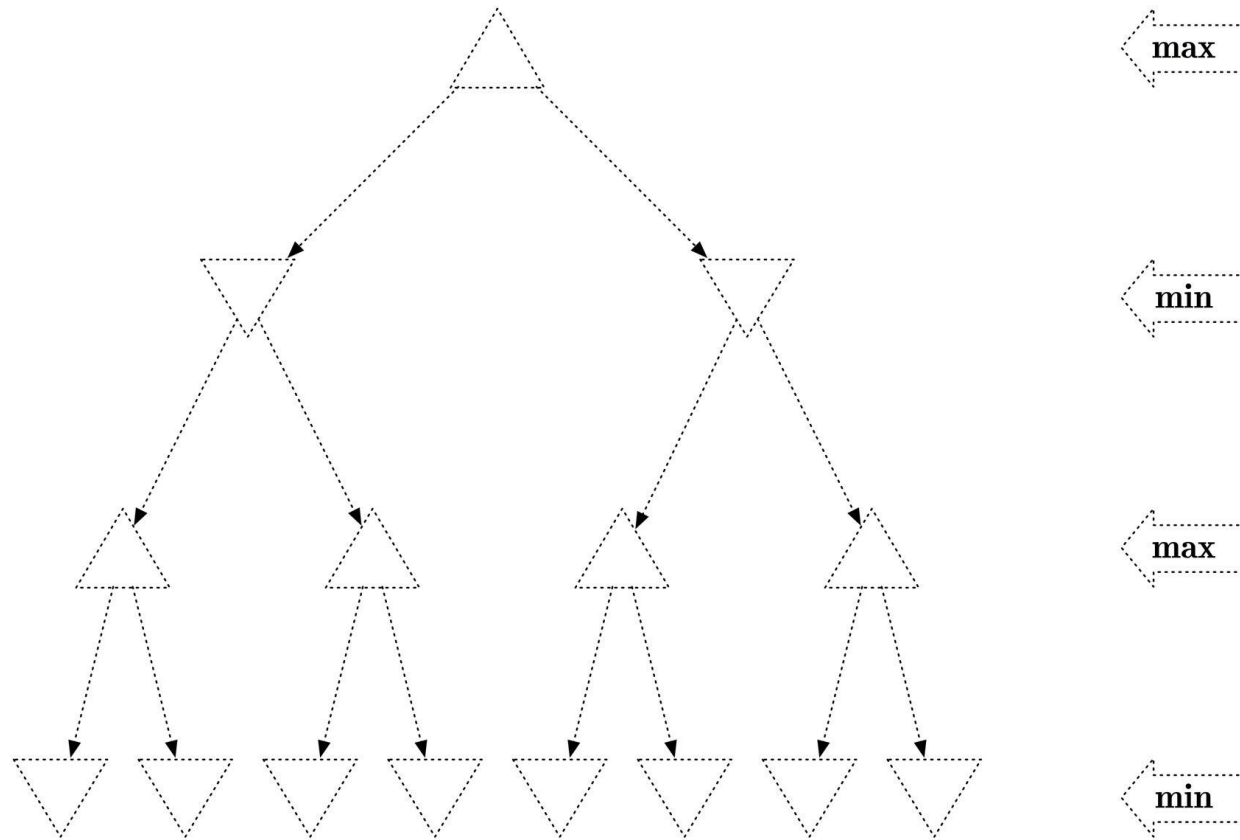
Adversarial search: minimax

For a state s $\text{minimax}(s) =$

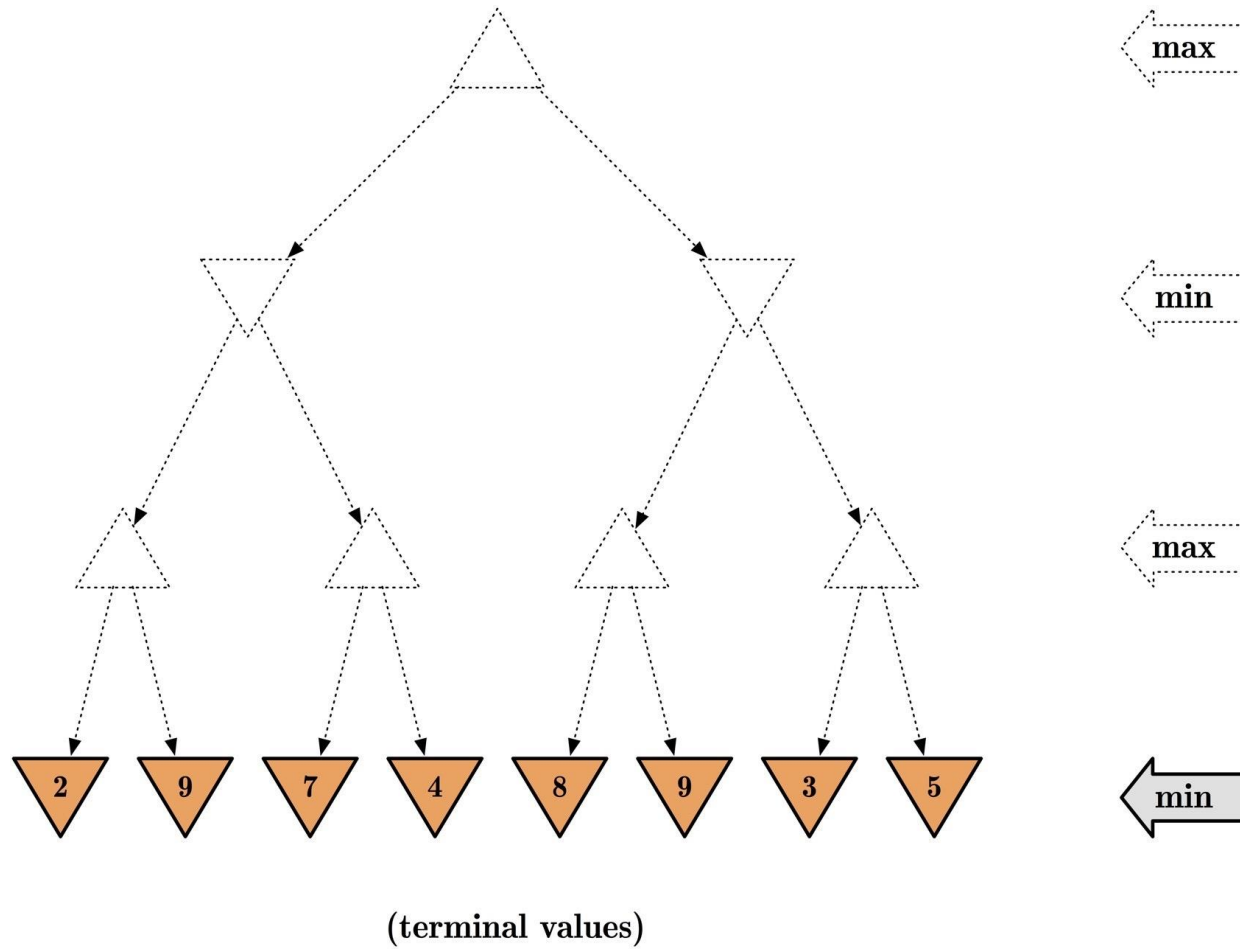
$$\begin{cases} \text{Utility}(s) & \text{if Terminal-test}(s) \\ \max_{a \in \text{Actions}(s)} \text{minimax}(\text{Result}(s,a)) & \text{if Player}(s) = \text{Max} \\ \min_{a \in \text{Actions}(s)} \text{minimax}(\text{Result}(s,a)) & \text{if Player}(s) = \text{Min} \end{cases}$$



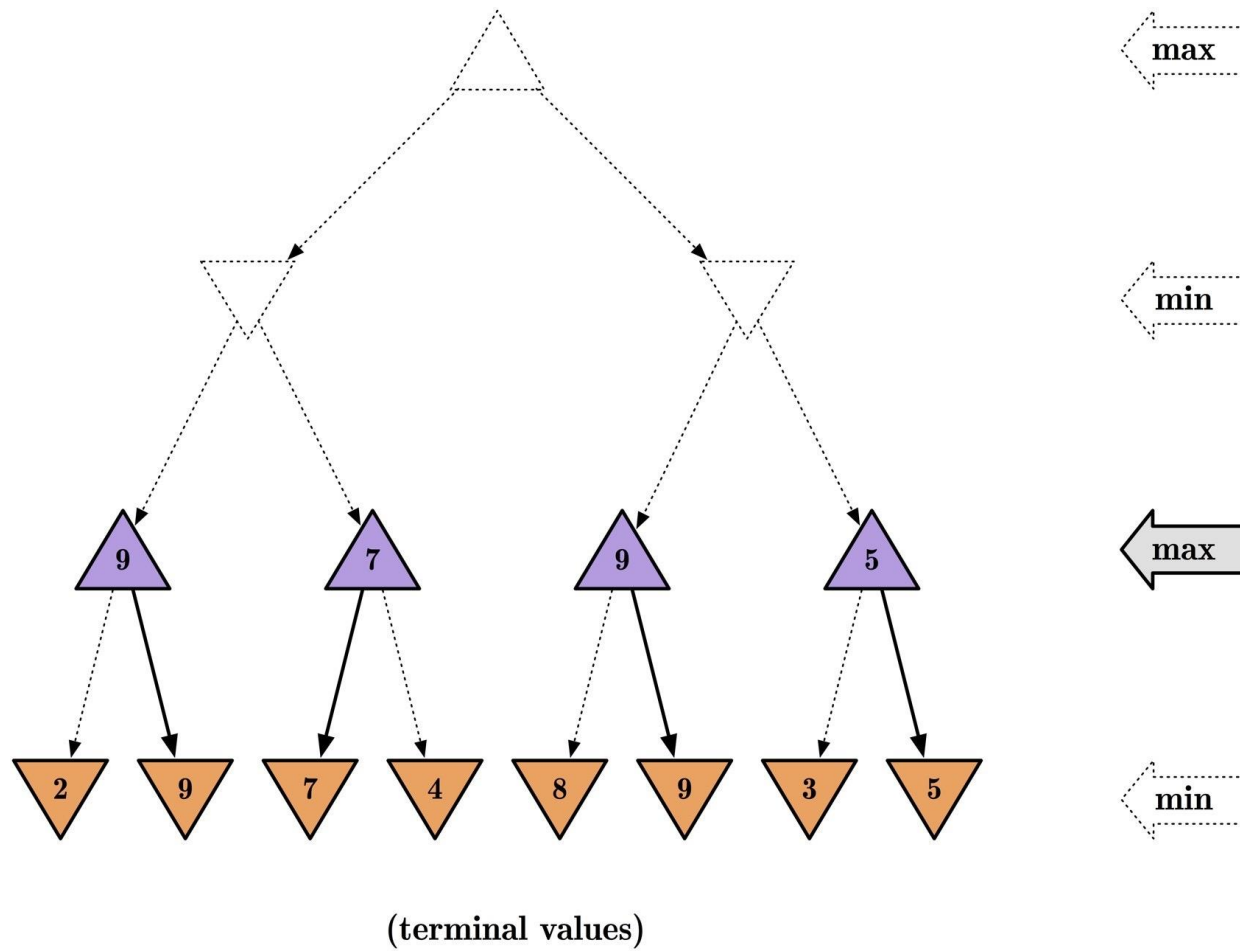
Minimax example



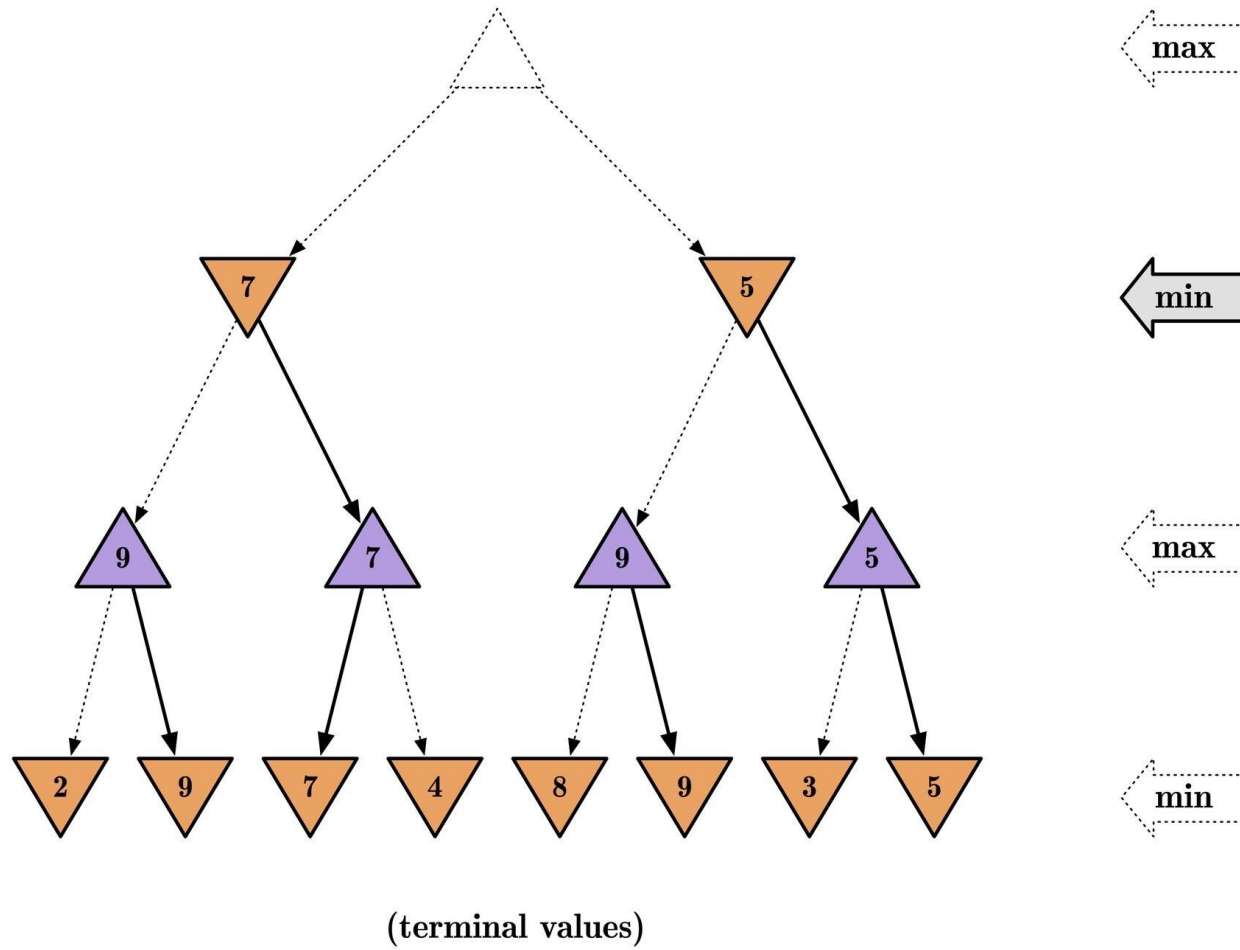
Minimax example



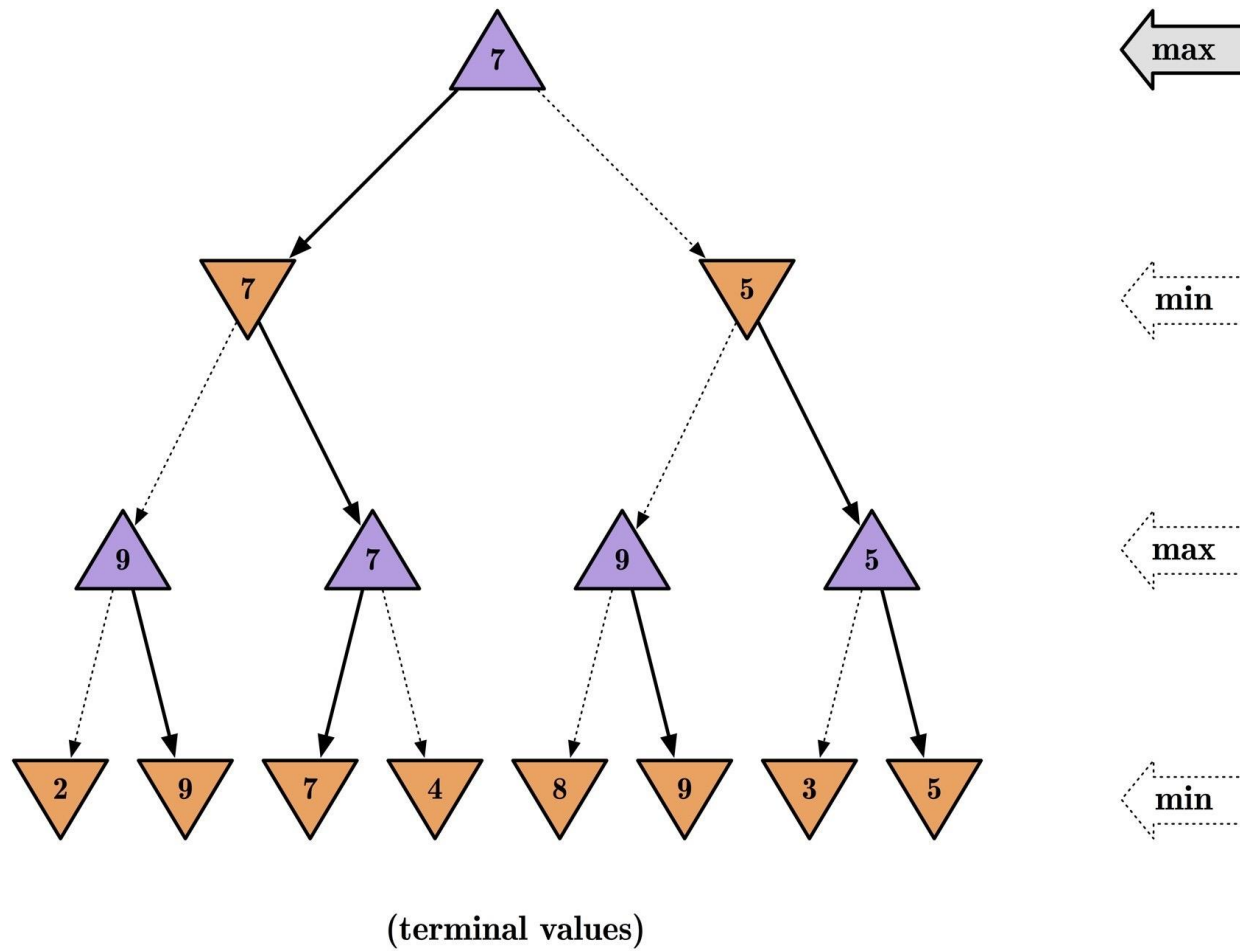
Minimax example



Minimax example

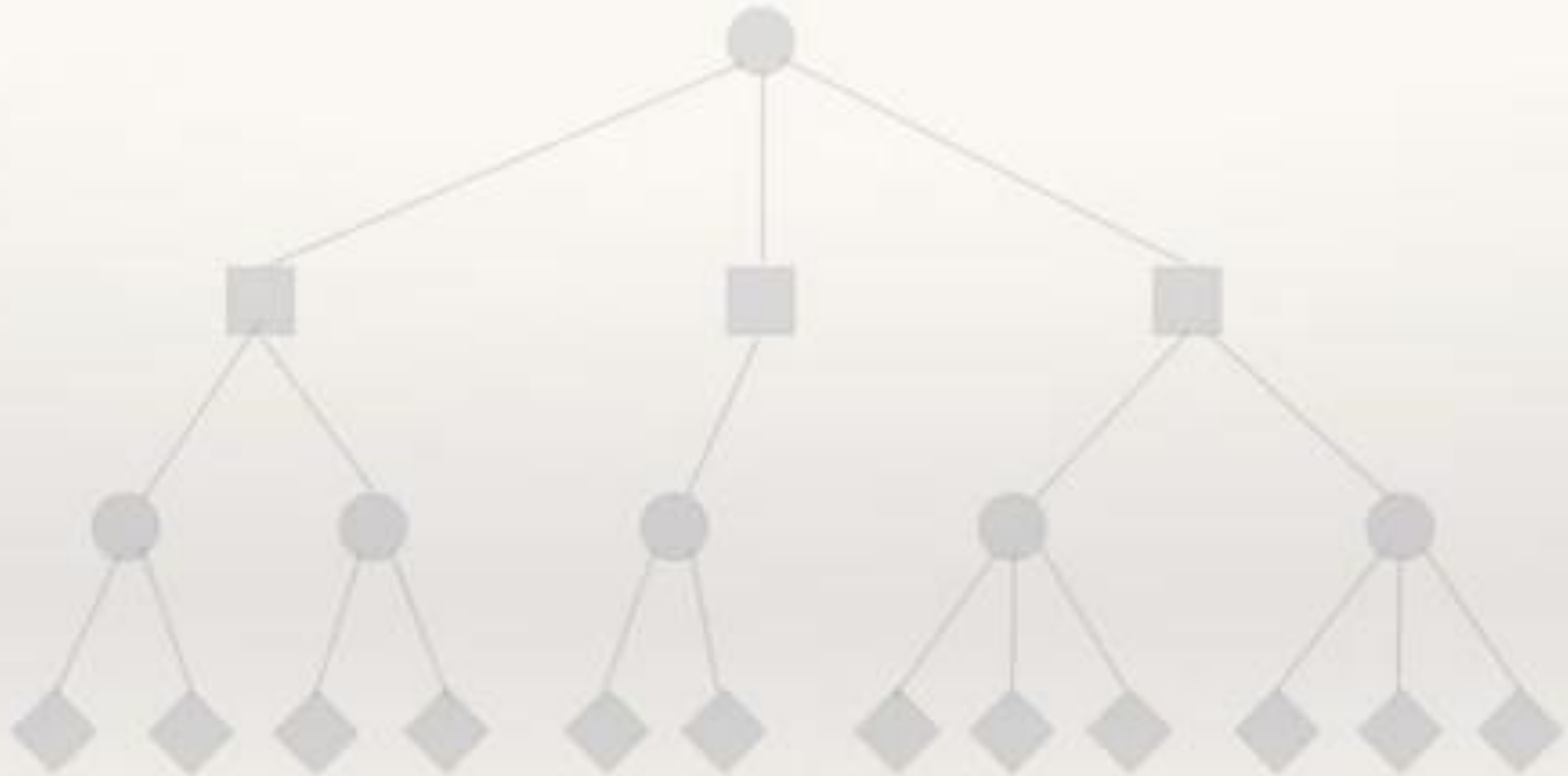


Minimax example



Minimax algorithm

Animation of the Minimax algorithm



Properties of minimax

- Optimal (opponent plays optimally) and complete (finite tree)
- DFS time: $O(b^d)$
- DFS space: $O(bd)$

- **Tic-Tac-Toe**

- 5 legal moves on average, total of 9 moves (9 plies).

- $5^9 = 1,953,125$

- $9! = 362,880$ terminal nodes

- **Chess**

- $b \sim 35$ (average branching factor)

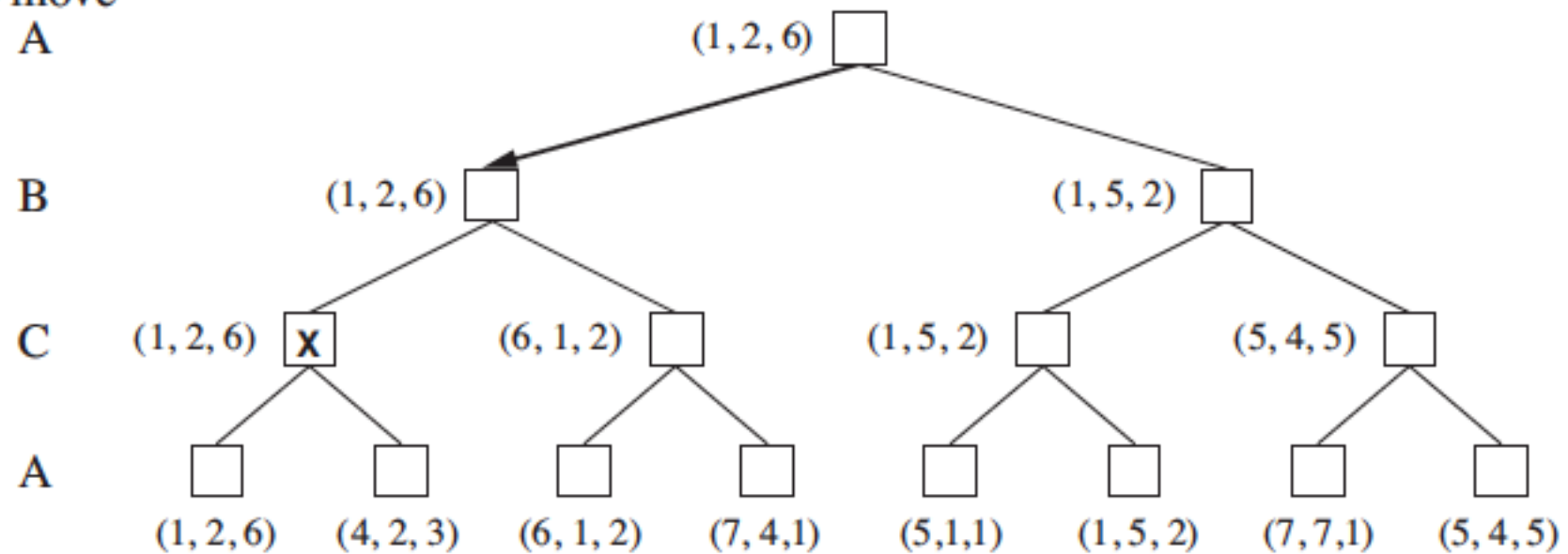
- $d \sim 100$ (depth of game tree for a typical game)

- $\rightarrow b^d \sim 35^{100} \sim 10^{154}$ nodes

- **Go** branching factor starts at 361 (19X19 board)

Multiplayer Games

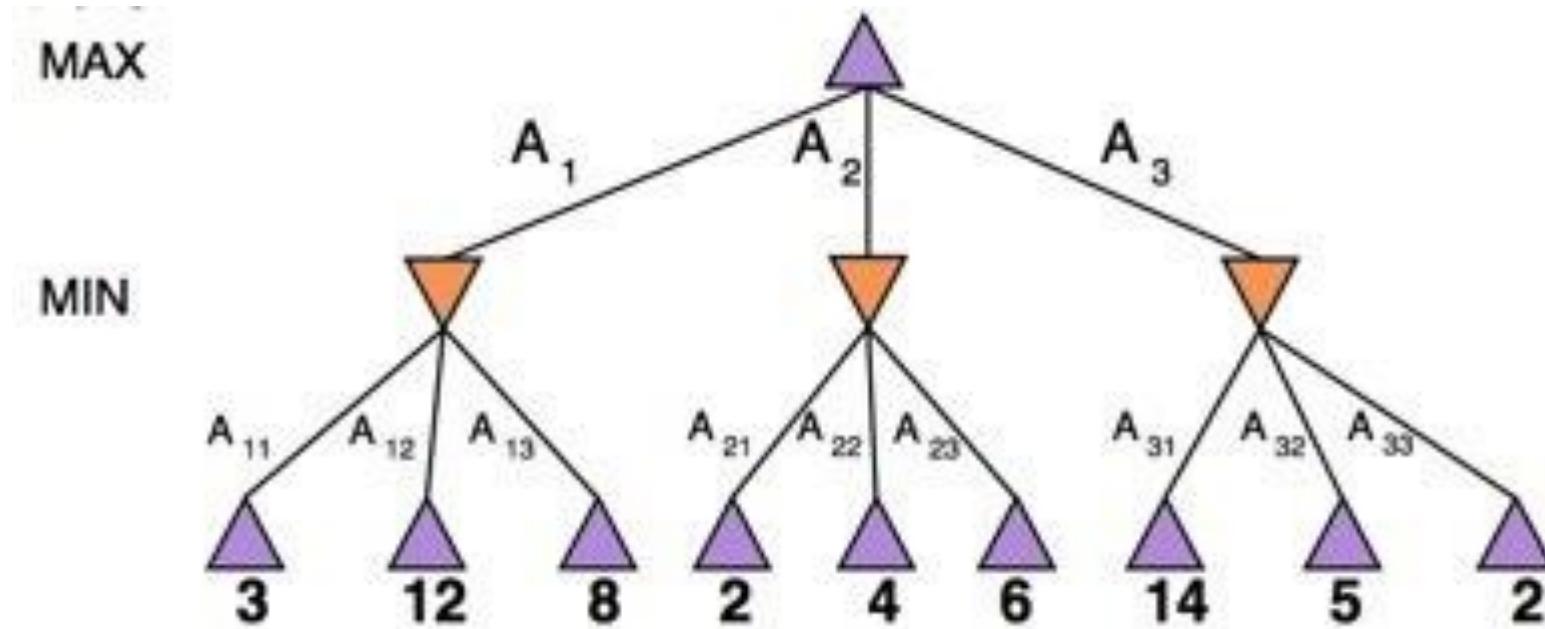
to move
A



Case of limited resources

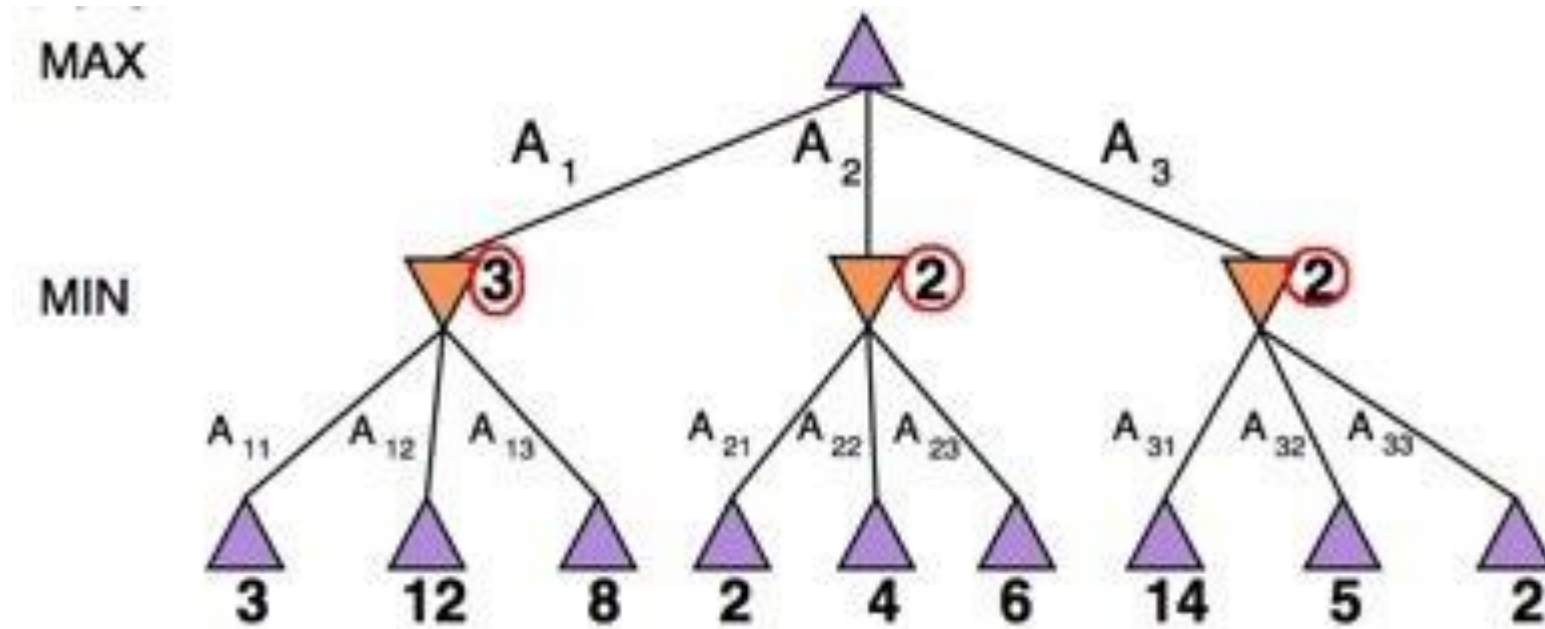
- **Problem:** In real games, we are limited in time, so we can't search the leaves.
- To be practical and run in a reasonable amount of time, min- max can only search to some depth.
- More plies make a big difference.
- **Solution:**
 1. Replace terminal utilities with an evaluation function for non-terminal positions.
 2. Use pruning: eliminate large parts of the tree.

$\alpha - \beta$ pruning

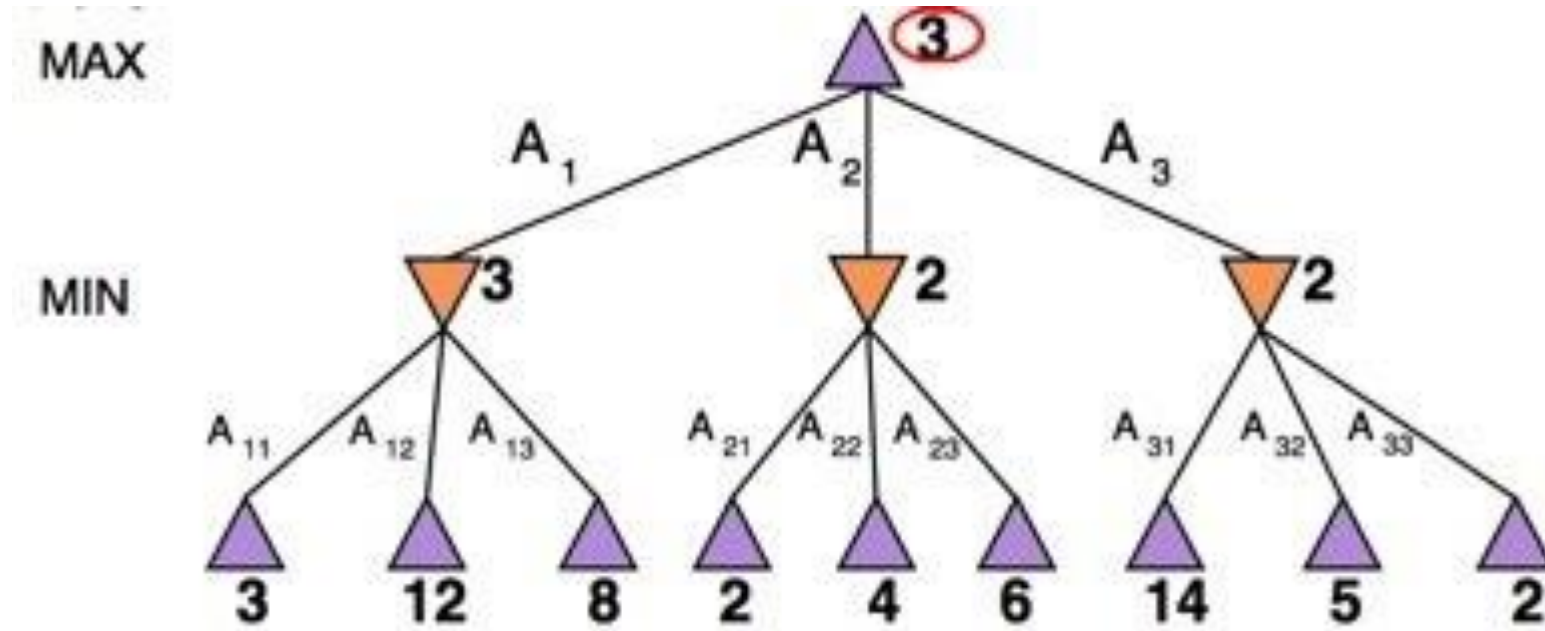


A two-ply game tree.

$\alpha - \beta$ pruning

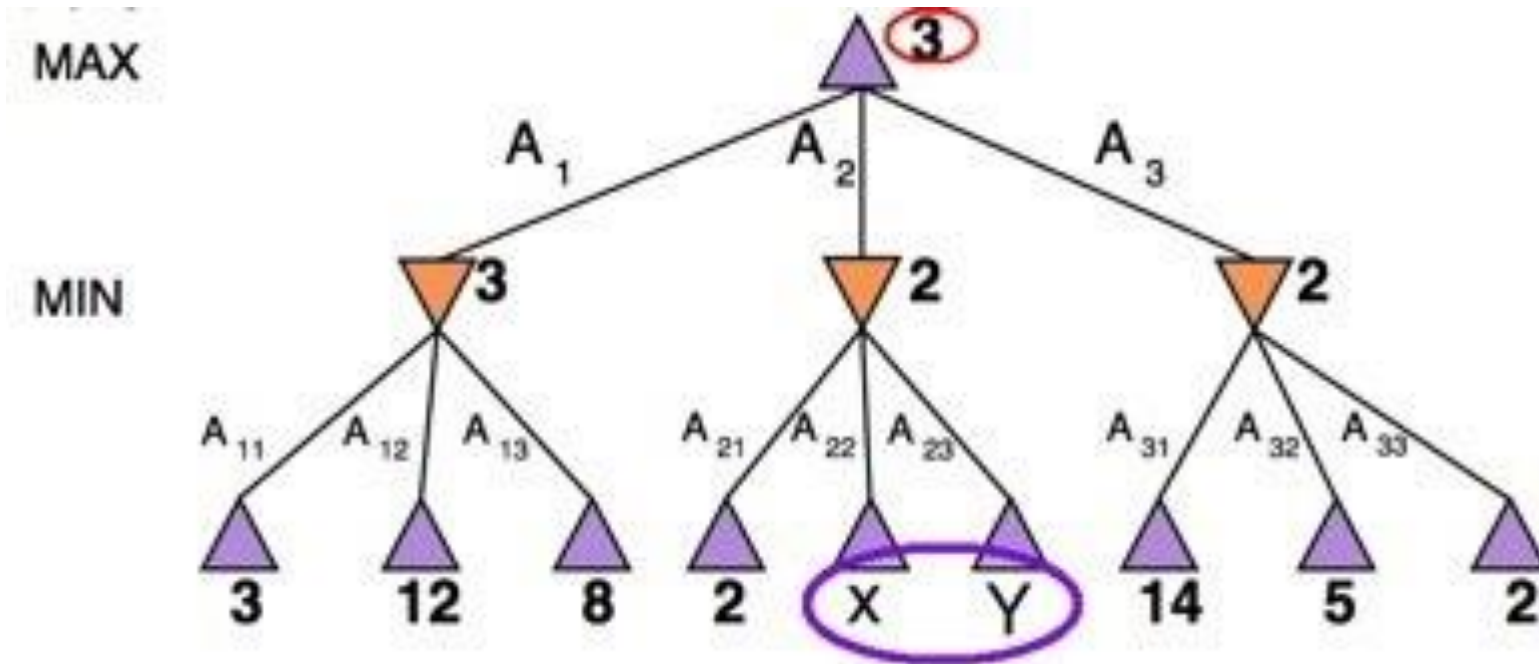


$\alpha - \beta$ pruning

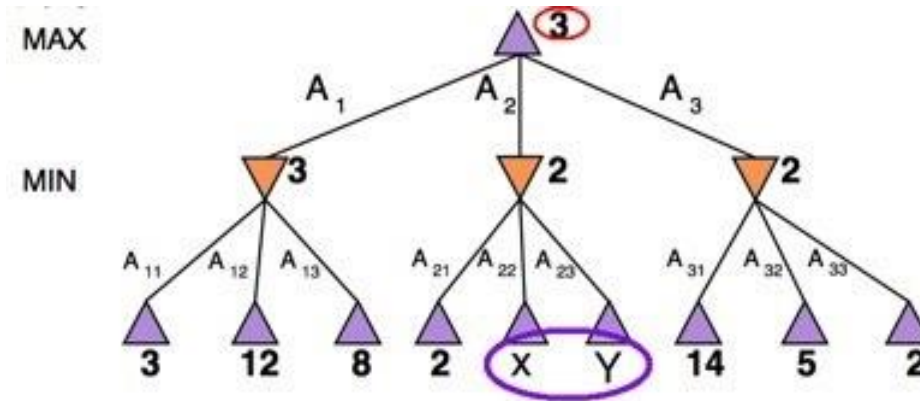


$\alpha - \beta$ pruning

Which values are necessary?

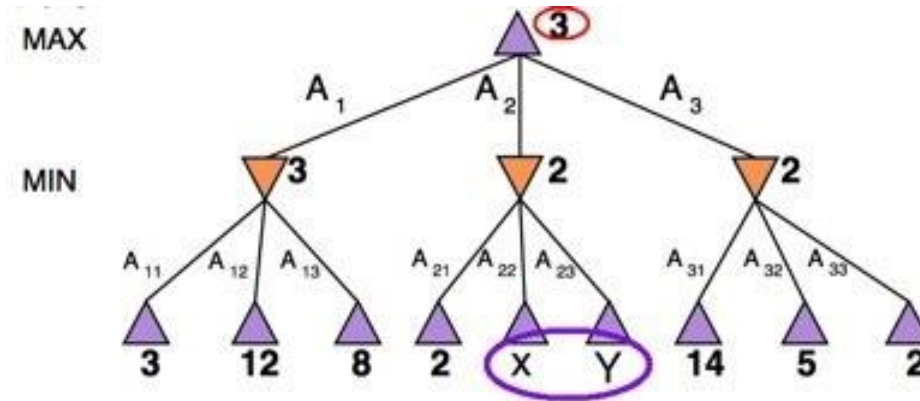


$\alpha - \beta$ pruning



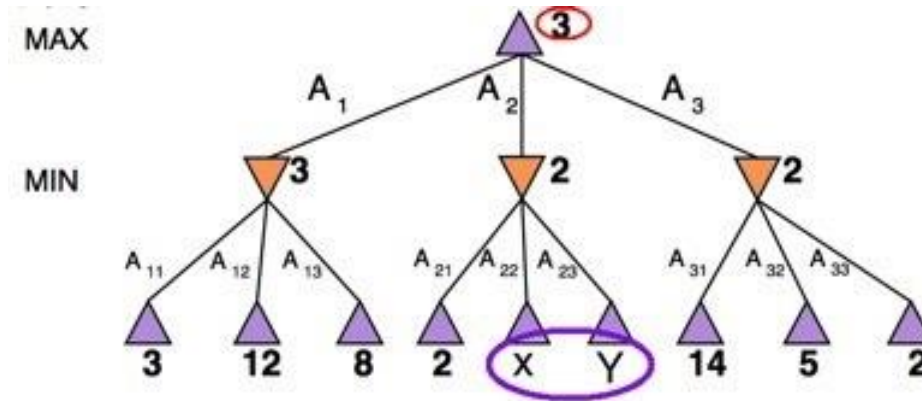
$$\text{Minimax}(\text{root}) = \max(\min(3, 12, 8), \min(2, X, Y), \min(14, 5, 2))$$

$\alpha - \beta$ pruning



$$\begin{aligned}\text{Minimax}(\text{root}) &= \max(\min(3, 12, 8), \min(2, X, Y), \min(14, 5, 2)) \\ &= \max(3, \min(2, X, Y), 2)\end{aligned}$$

$\alpha - \beta$ pruning



$$\begin{aligned}\text{Minimax}(\text{root}) &= \max(\min(3, 12, 8), \min(2, X, Y), \min(14, 5, 2)) \\ &= \max(3, \min(2, X, Y), 2) \\ &= \max(3, Z, 2) \quad \text{where } Z = \min(2, X, Y) \leq 2\end{aligned}$$

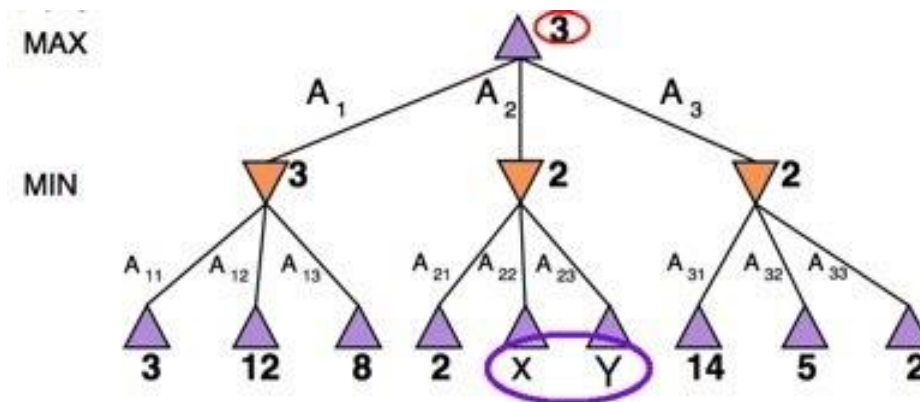
$\alpha - \beta$ pruning

$$\text{Minimax}(\text{root}) = \max(\min(3, 12, 8), \min(2, X, Y), \min(14, 5, 2))$$

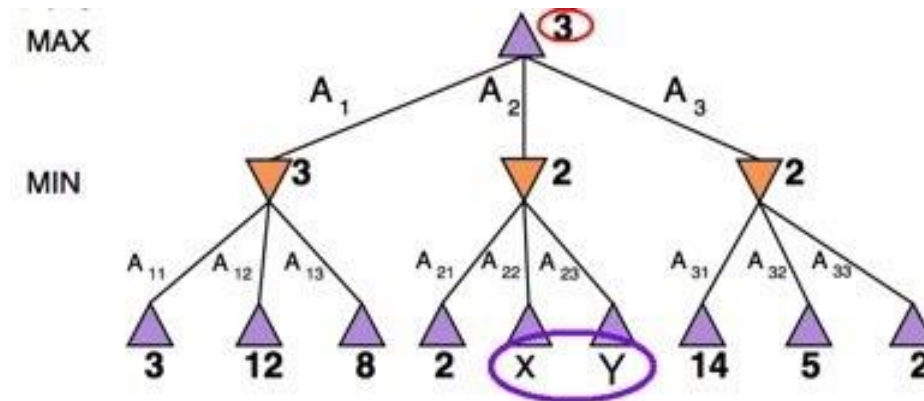
$$= \max(3, \min(2, X, Y), 2)$$

$$= \max(3, Z, 2) \quad \text{where } Z = \min(2, X, Y) \leq 2$$

$$= 3$$



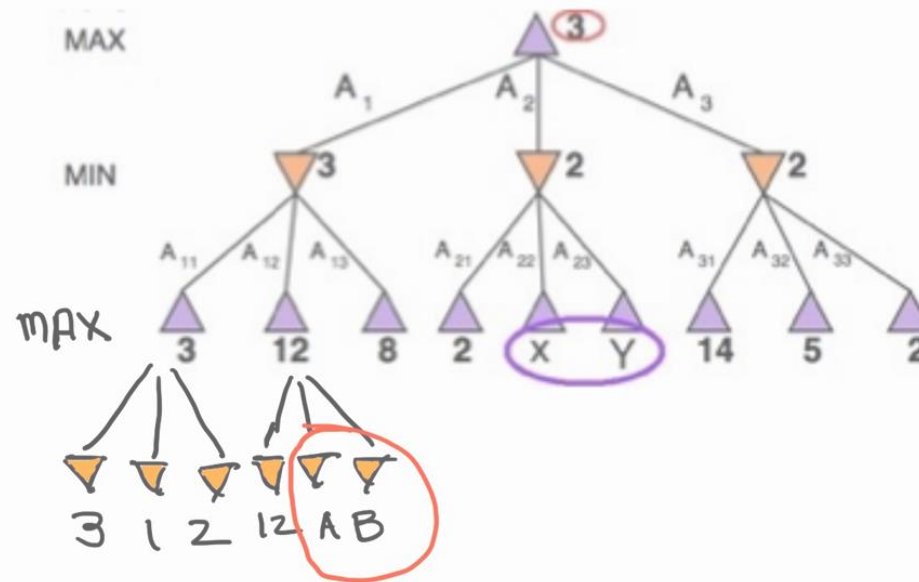
$\alpha - \beta$ pruning



$$\begin{aligned}\text{Minimax}(\text{root}) &= \max(\min(3, 12, 8), \min(2, X, Y), \min(14, 5, 2)) \\ &= \max(3, \min(2, X, Y), 2) \\ &= \max(3, z, 2) \quad \text{where } z = \min(2, X, Y) \leq 2 \\ &= 3\end{aligned}$$

Minimax decisions are independent of the values of X and Y .

$\alpha - \beta$ pruning



\leq

$$\text{Minimax}(\text{MIN}) = \min(\max(3, 1, 2), \max(12, A, B))$$

$$= \min(3, \max(12, A, B))$$

$$= \min(3, Z) \quad \text{where } Z \geq \max(12, A, B) \quad 2$$

$$= 3$$

Minimax decisions are independent of the values of A and B.

$\alpha - \beta$ pruning

- **Strategy:** Just like minimax, it performs a DFS.

$\alpha - \beta$ pruning

- **Strategy:** Just like minimax, it performs a DFS.
- **Parameters:** Keep track of two bounds
 - **alpha** : largest value for Max across seen children (current lower bound on MAX's outcome).
 - **beta** : lowest value for MIN across seen children (current upper bound on MIN's outcome).

$\alpha - \beta$ pruning

- **Strategy:** Just like minimax, it performs a DFS.
- **Parameters:** Keep track of two bounds
 - **alpha**: largest value for Max across seen children (current lower bound on MAX's outcome).
 - **beta**: lowest value for MIN across seen children (current upper bound on MIN's outcome).
- **Initialization:** $\alpha = -\infty, \beta = \infty$

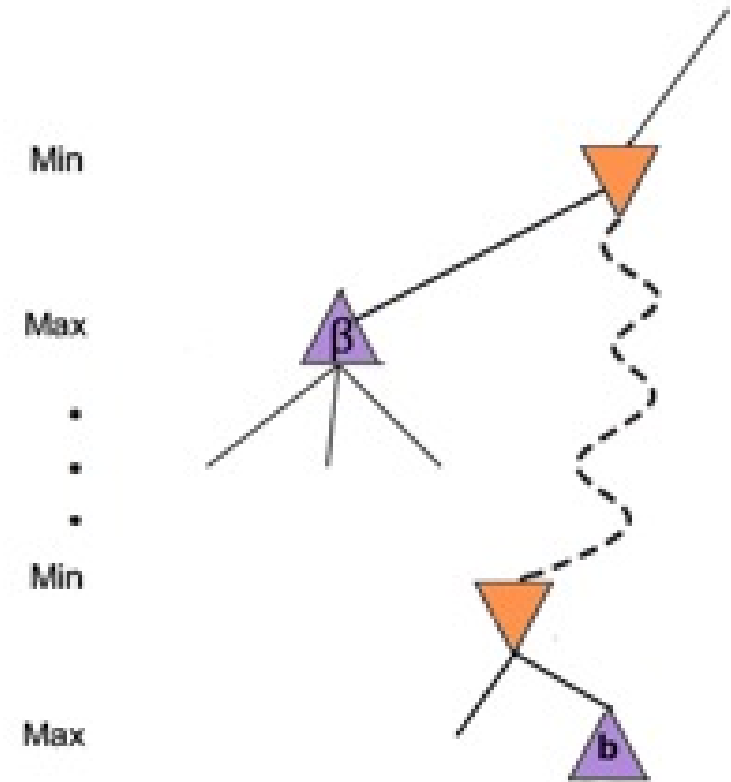
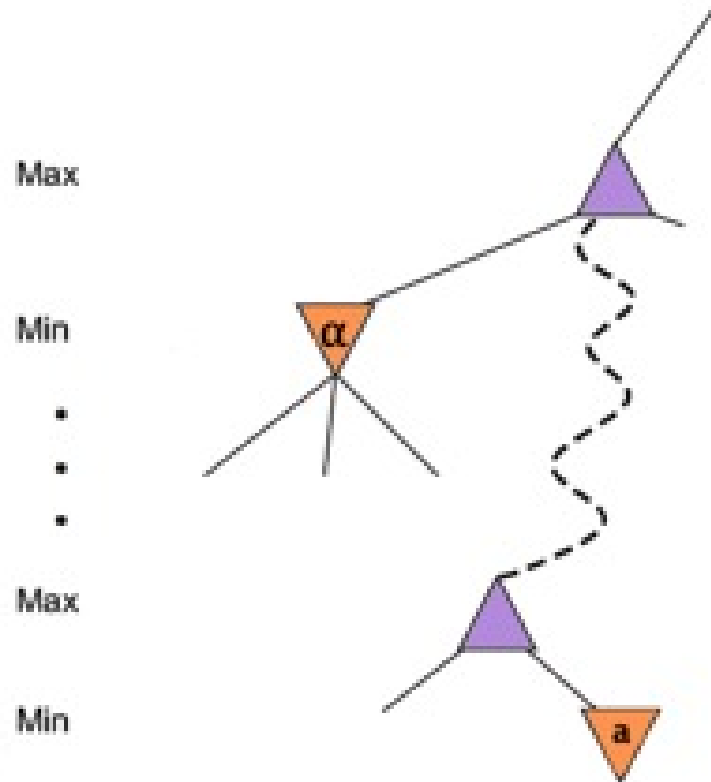
$\alpha - \beta$ pruning

- **Strategy:** Just like minimax, it performs a DFS.
- **Parameters:** Keep track of two bounds
 - α : largest value for Max across seen children (current lower bound on MAX's outcome).
 - β : lowest value for MIN across seen children (current upper bound on MIN's outcome).
- **Initialization:** $\alpha = -\infty$, $\beta = \infty$
- **Propagation:** Send α , β values *down* during the search to be used for pruning.
 - Update α , β values by *propagating upwards* values of terminal nodes.
 - Update α only at Max nodes and update β only at Min nodes.

$\alpha - \beta$ pruning

- **Strategy:** Just like minimax, it performs a DFS.
- **Parameters:** Keep track of two bounds
 - α : largest value for Max across seen children (current lower bound on MAX's outcome).
 - β : lowest value for MIN across seen children (current upper bound on MIN's outcome).
- **Initialization:** $\alpha = -\infty$, $\beta = \infty$
- **Propagation:** Send α , β values *down* during the search to be used for pruning.
 - Update α , β values by *propagating upwards* values of terminal nodes.
 - Update α only at Max nodes and update β only at Min nodes.
- **Pruning:** Prune any remaining branches whenever $\alpha \geq \beta$.

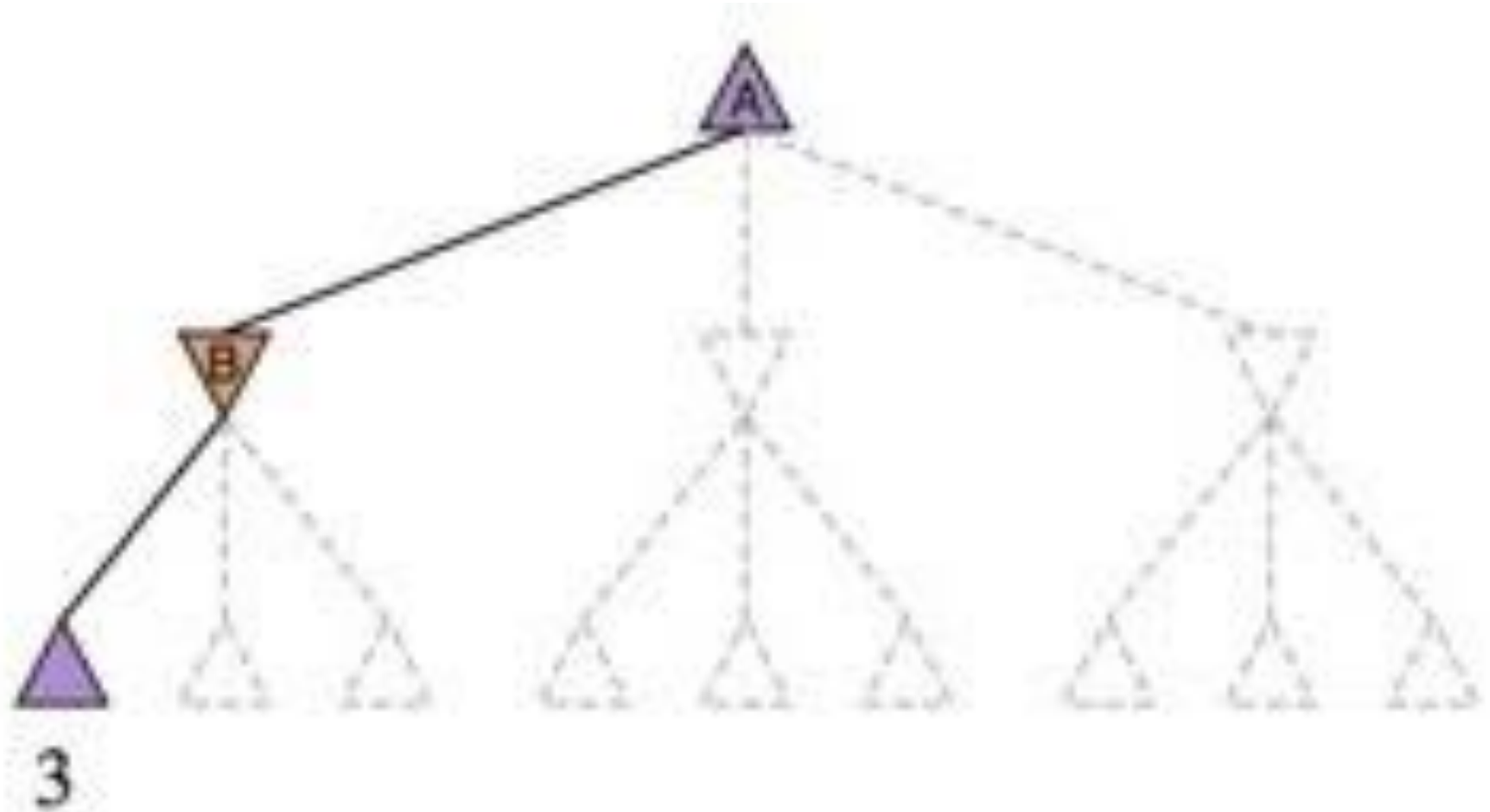
$\alpha - \beta$ pruning



- If α is better than a for Max, then Max will avoid it, that is prune that branch.
- If β is better than b for Min, then Min will avoid it, that is prune that branch.

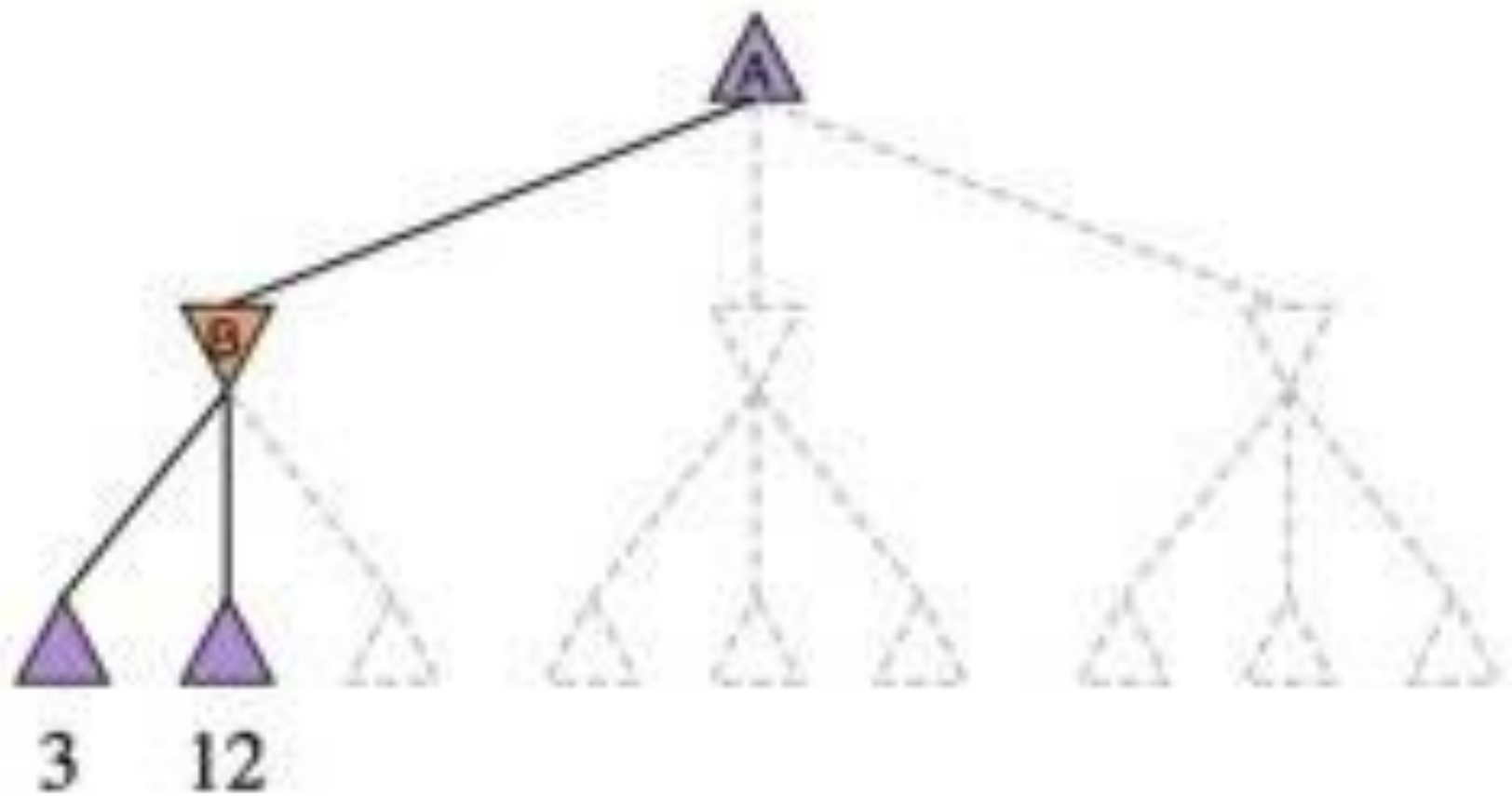
$\alpha - \beta$ pruning

(a)



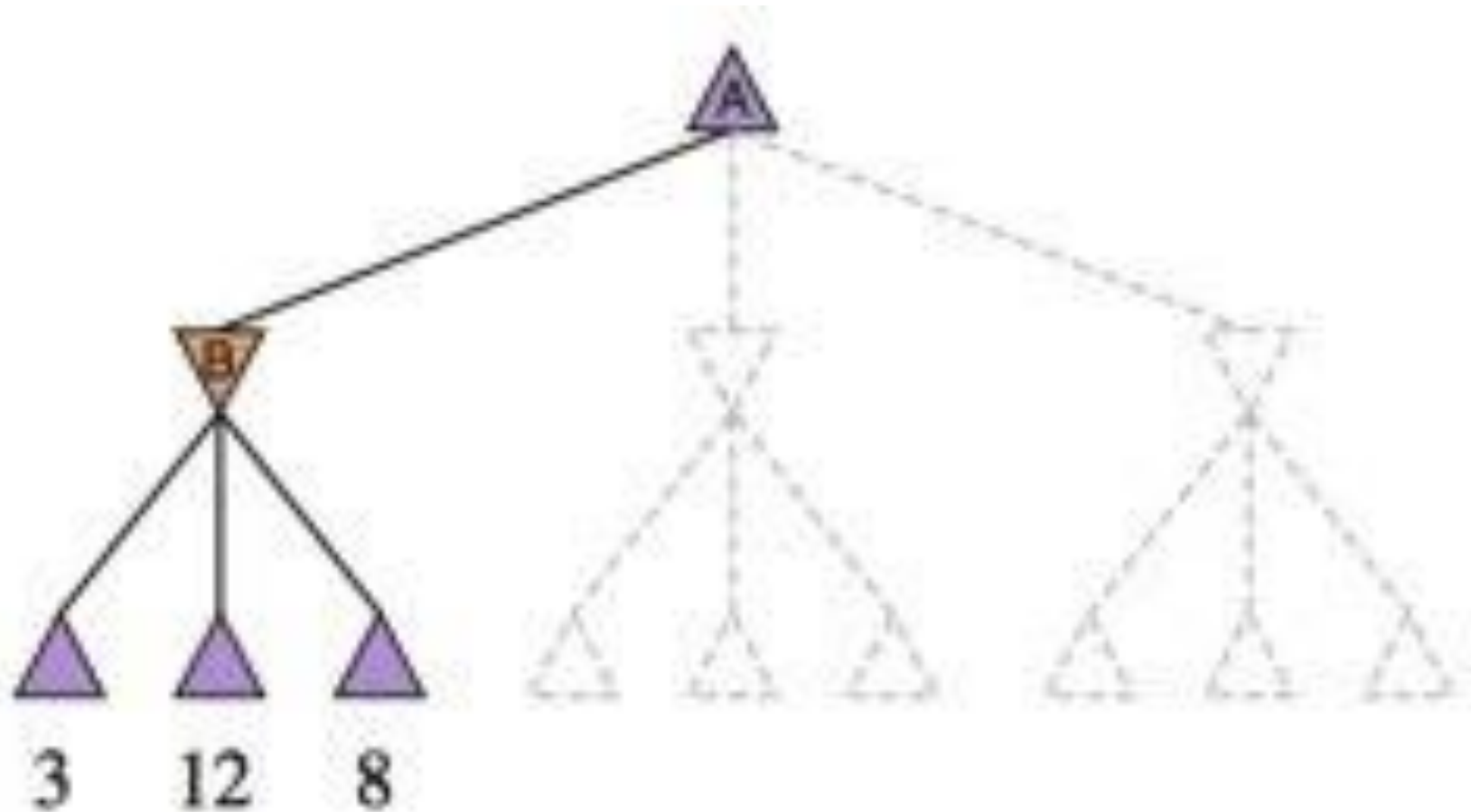
$\alpha - \beta$ pruning

(b)



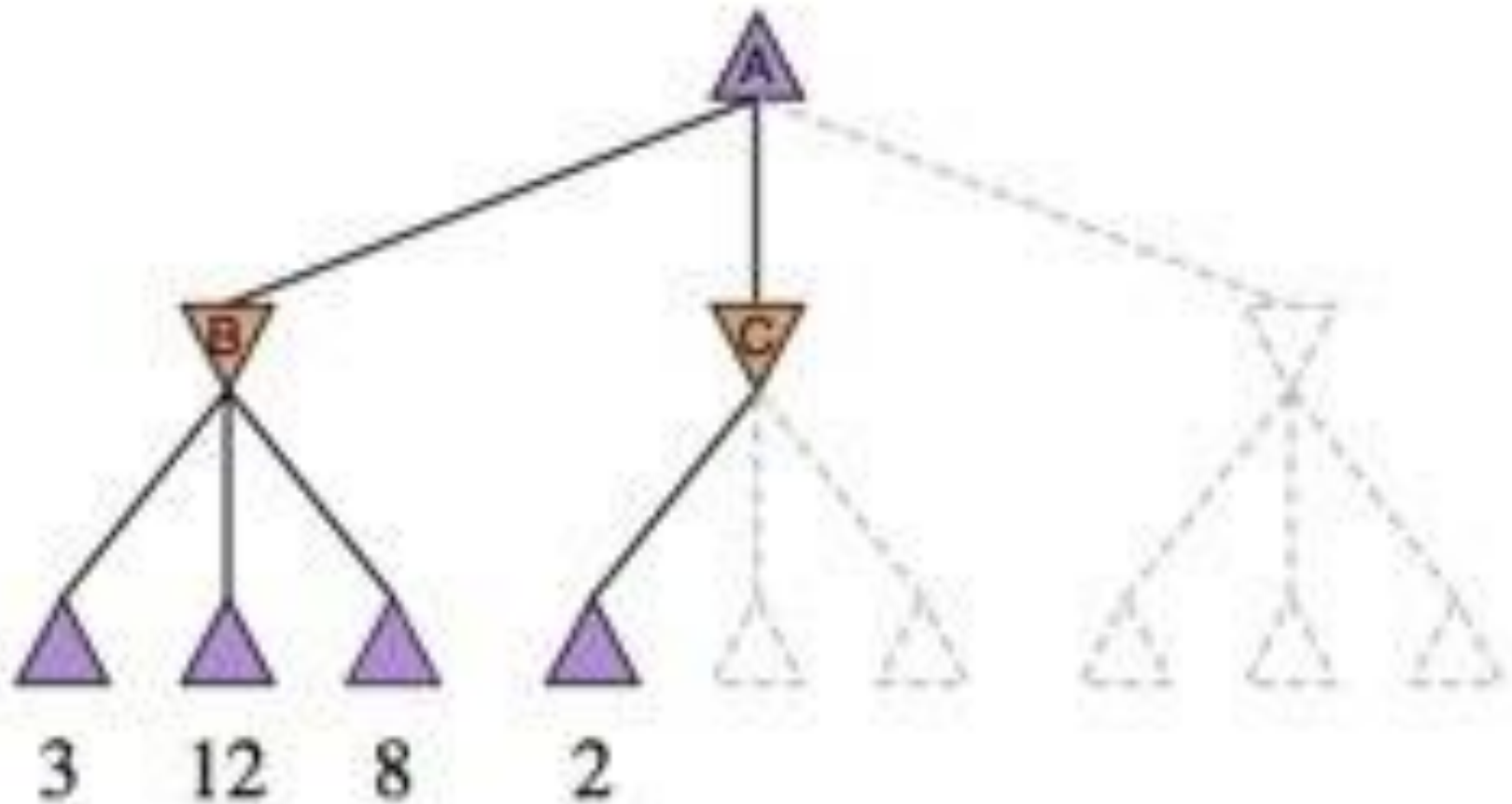
$\alpha - \beta$ pruning

(c)



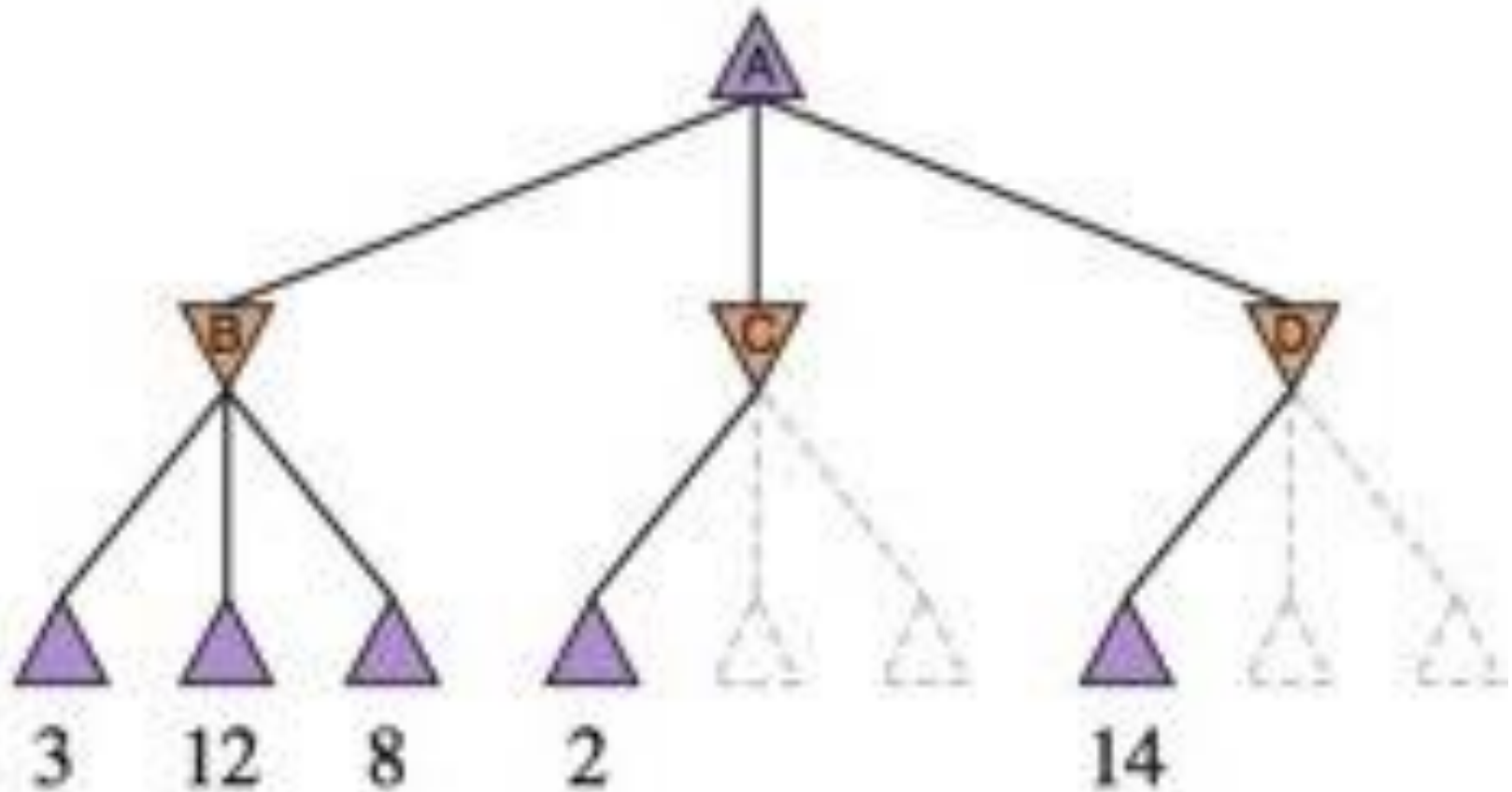
$\alpha - \beta$ pruning

(d)



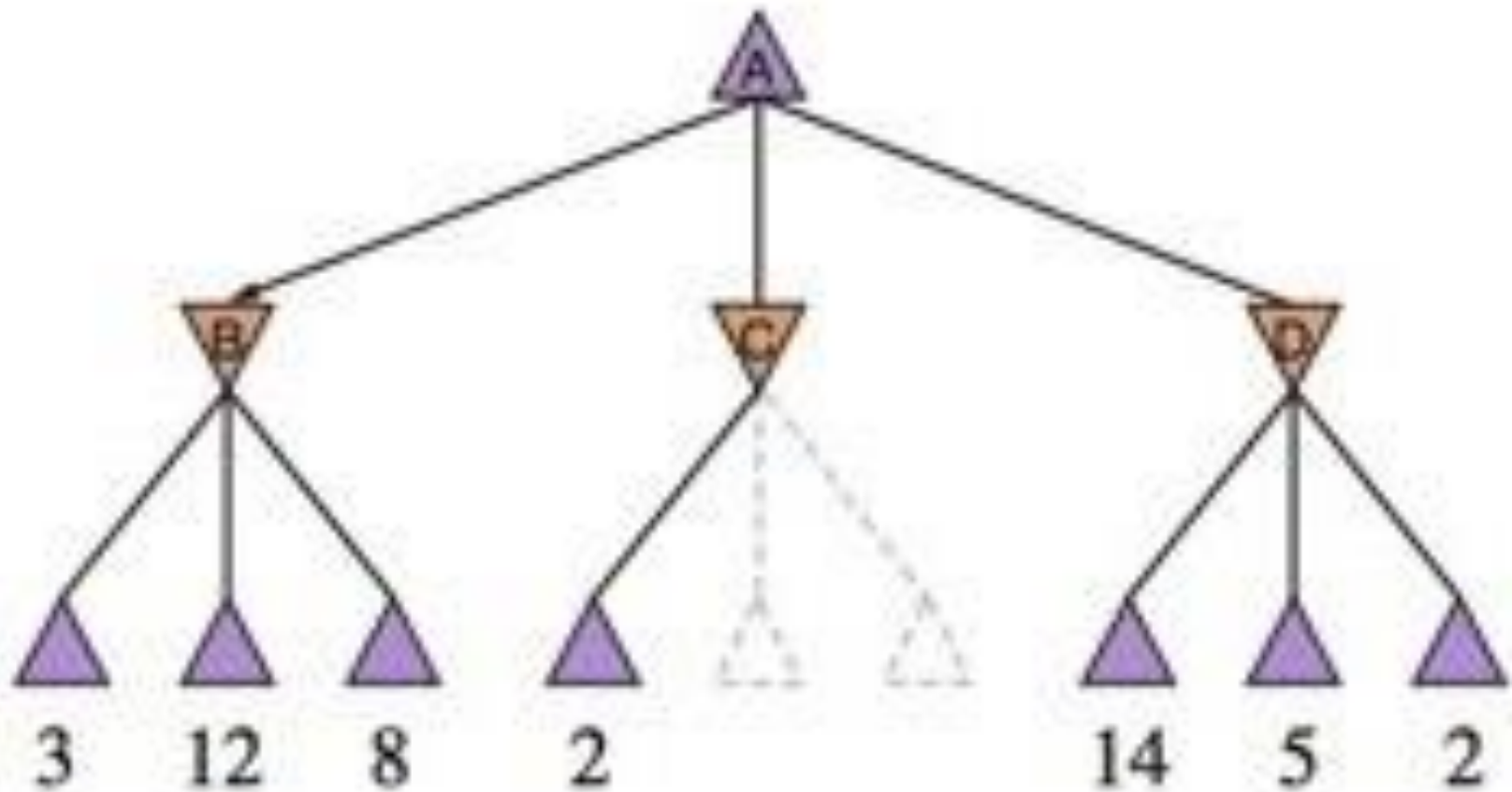
$\alpha - \beta$ pruning

(e)

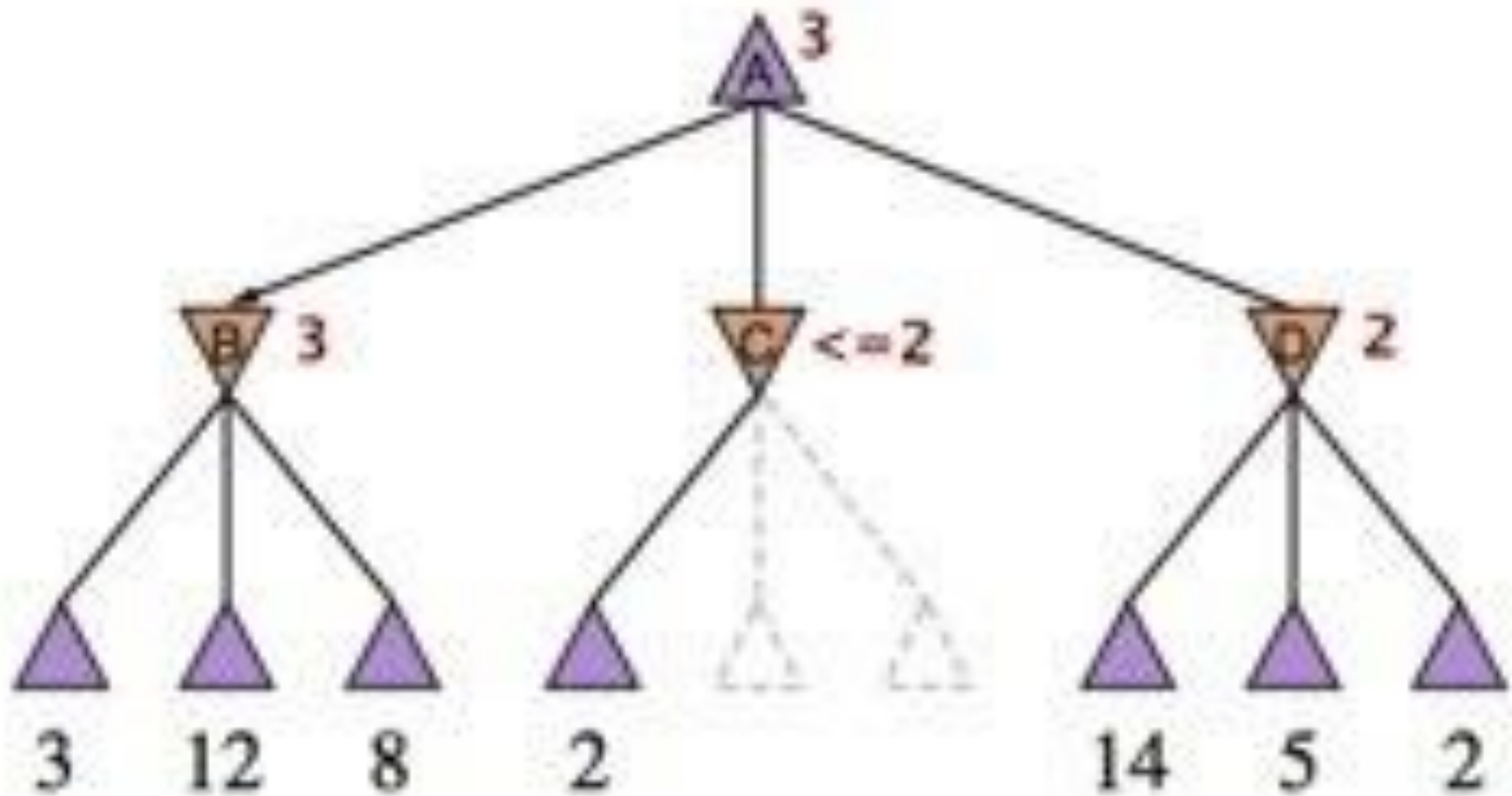


$\alpha - \beta$ pruning

(f)



$\alpha - \beta$ pruning



Real-time decisions

- Minimax: generates the entire game search space
- ~~alpha - beta~~ algorithm: prune large chunks of the trees
- BUT alpha - beta still has to go all the way to the leaves
- Impractical in real-time (moves has to be done in a reasonable amount of time)
- Solution: bound the depth of search (cut off search) and **replace** $utility(s)$ **with** $eval(s)$, an evaluation function to **estimate** value of current board configurations

Real-time decisions

- $eval(s)$ is a heuristic at state s
 - E.g., Othello: white pieces - black pieces
 - E.g., Chess: Value of all white pieces - Value of all black piecesturn non-terminal nodes into terminal leaves!
- An ideal evaluation function would rank terminal states in the same way as the true utility function; but must be fast
- Typical to define features, make the function a linear weighted sum of the features
- Use domain knowledge to craft the best and useful features.

Real-time decisions



- How does it work?
 - Select useful features f_1, \dots, f_n e.g., Chess: # pieces on board, value of pieces (1 for pawn, 3 for bishop, etc.)
 - Weighted linear function
$$eval(s) = \sum_{i=1}^n w_i f_i(s)$$
 - Learn w_i from the examples
 - Deep blue uses about 6,000 features!

Deep Blue

Predictions were in Kasparov's favor, many experts predicted the champion to score at least four points out of six. Kasparov was coming off a superb performance at the Linares tournament and his rating was at an all-time high of 2820. In a 2003 movie, he recalled his early confidence: "I will beat the machine, whatever happens. Look at Game One. It's just a machine. Machines are stupid." Later Kasparov said that he sometimes saw deep intelligence and creativity in the machine's moves



History of Chess playing machines

The Chess 4.5 program in the late 1970s first demonstrated that an engineering approach emphasizing hardware speed might be more fruitful.

Belle,² a special purpose hardwired chess machine from Bell Laboratories, became the first national master program in the early 1980s.

Cray Blitz³ running on a Cray supercomputer, and Hitech,⁴ another specialpurpose chess machine, became the top programs in the mid-1980s.

ChipTest (1986-1987), Deep Thought (1988-1991), and Deep Thought II (1992-1995)—claimed spots as the top chess programs in the world.

In 1988 the Deep Thought team won the second Fredkin Intermediate Prize for Grandmaster-level performance for Deep Thought's 2650-plus rating on the USCF's scale over 25 consecutive games.

Deep Blue's 1996 debut in the first Kasparov versus Deep Blue match in Philadelphia finally eclipsed Deep Thought II.

Deep Blue

Winning a match against the human World Chess Champion under regulation time control. The games had to play out no faster than three minutes per move.

In 1996 Deep Blue was even with Kasparov after four games in the first match. Kasparov pinpointed Deep Blue's weaknesses and won the remaining two games easily. Computation speed alone apparently didn't suffice.

What caused the problems for Cray Blitz and Deep Blue in the 1984 and 1996 matches? First, they played adaptable human opponents.

Humans learn !

Make the weights associated with the positional features individually adjustable from software.

System configuration

The 1997 version of Deep Blue included 480 chess chips. Since each chess chip could search 2 to 2.5 million chess positions/s, the “guaranteed not to exceed” system speed reached about one billion chess positions per second, or 40 tera operations.

For a program searching close to 40 billion positions for each move—as the 1997 Deep Blue did—the alpha-beta algorithm increases the search speed by up to 40 billion times.

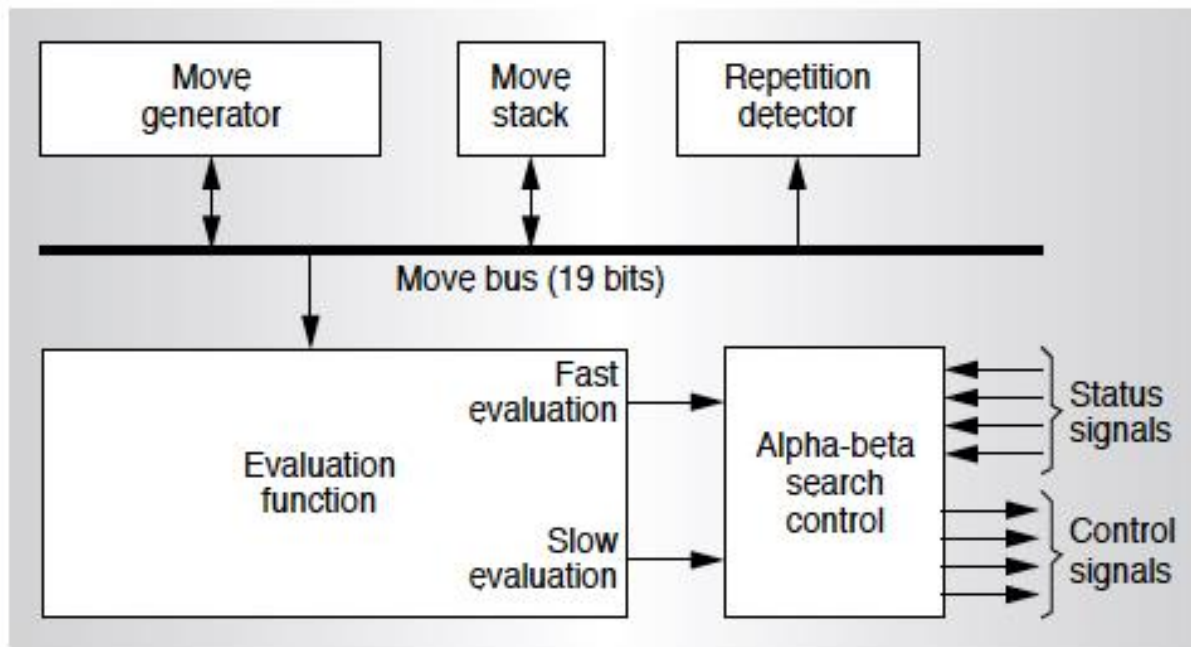


Figure 1. Block diagram of the chess chip.

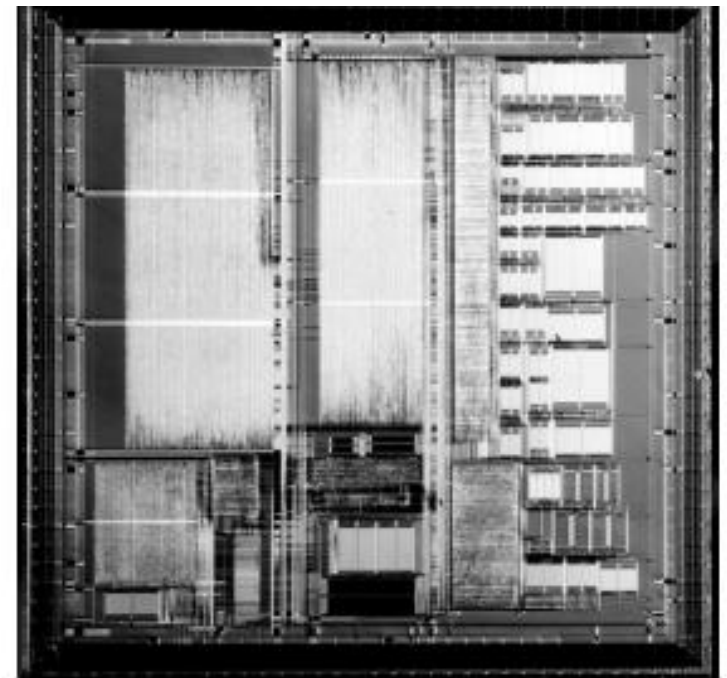


Figure 2. Die photo of the chess chip.

Deep Blue evaluation functions

Evaluation Function



- Measures the "goodness" of a given chess position
 - Deep blue's evaluation function had been split into 8000 parts, many of them designed for special positions
e.g. how important is a safe king position compared to a space advantage in the center
 - The optimal values for these parameters were then determined by the system itself, by analyzing thousands of master games.
-
- Deep Blue searched the game tree as far as possible, usually to a depth of 12 ply (Good human chess players look roughly 10 ply)
 - Use an evaluation function to evaluate the quality of the nodes at that depth.

Deep Blue : computing resource

- Raw computing power : More computing power means the game tree can be searched to a greater depth, which leads to better estimates by the evaluation function.
 - Massively parallel, 30-node, RS/6000, SP-based computer system enhanced with 480 special purpose VLSI chess chips.
 - Capable of evaluating 200,000,000 positions per second
 - 259th fastest supercomputer
- Quality of the evaluation function : Fine tuned by Grand masters

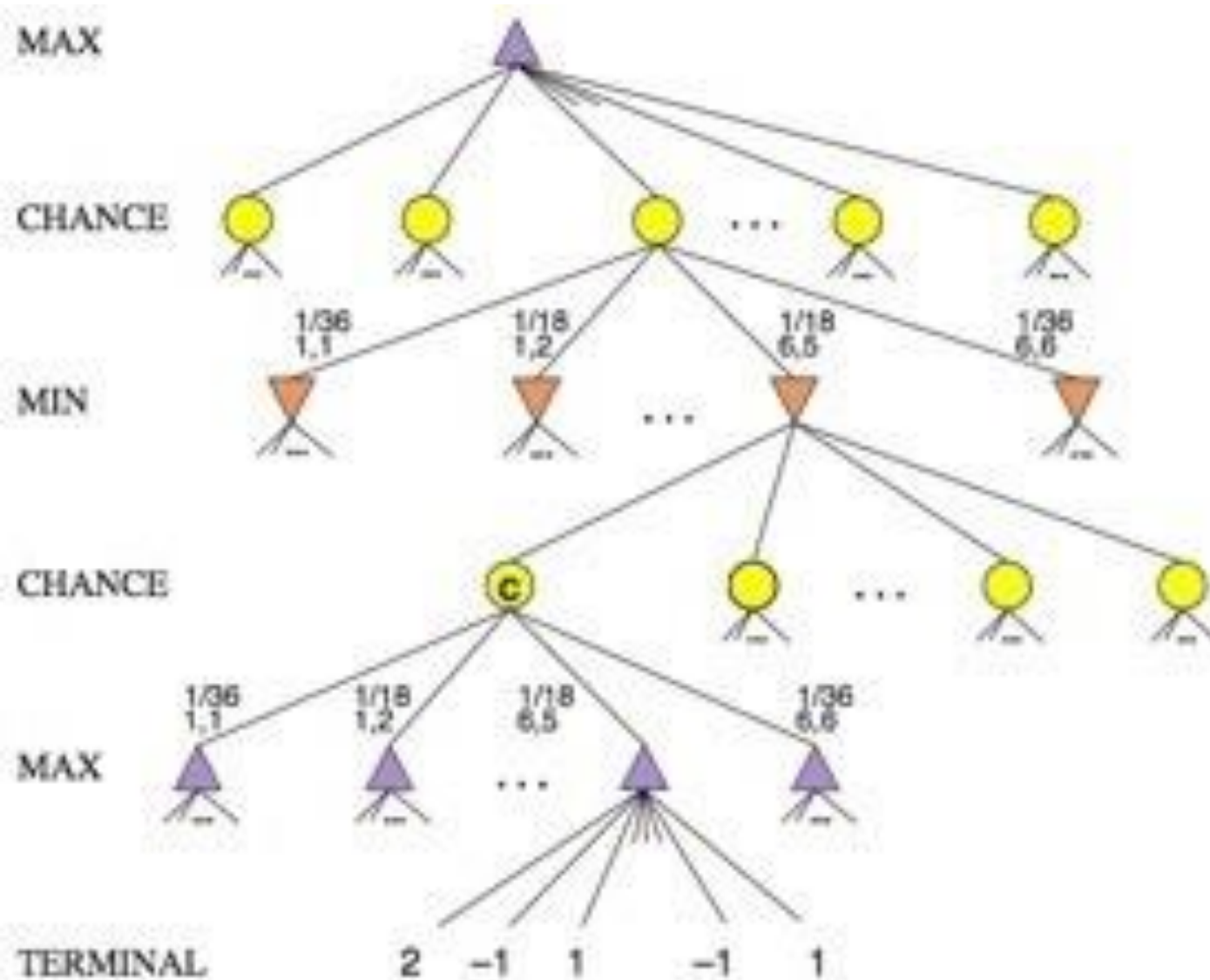
Stochastic games

- Include a random element (e.g., throwing a die).
- Include chance nodes.
- Backgammon: old board game combining skills and chance.
- The goal is that each player tries to move all of his pieces off the board before his opponent does.



Ptkfgr [Public domain], via Wikimedia Commons

Stochastic games



Partial game tree for Backgammon.

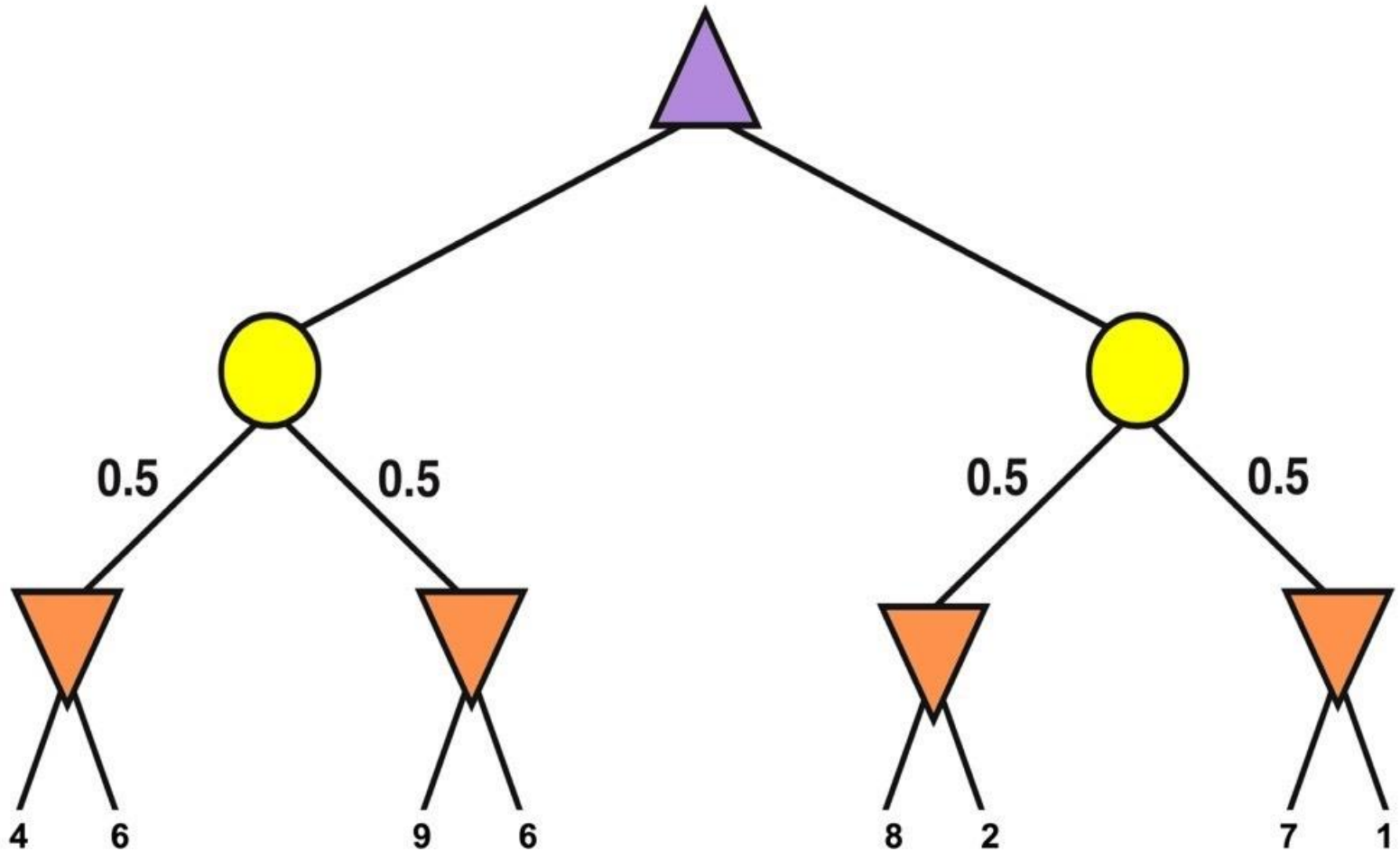
Stochastic games

Algorithm **Expectiminimax** generalized Minimax to handle chance nodes as follows:

- If state is a Max node then
return the highest Expectiminimax-Value of Successors(state)
- If state is a Min node then
return the lowest Expectiminimax-Value of Successors(state)
- If state is a chance node then
return average of Expectiminimax-Value of Successors(state)

Stochastic games

Example with coin-flipping:



Expectiminimax

For a state s :

Expectiminimax(s) =

$$\left\{ \begin{array}{ll} \text{Utility}(s) & \text{if Terminal-test}(s) \\ \max_{a \in \text{Actions}(s)} \text{Expectiminimax}(\text{Result}(s,a)) & \text{if Player}(s) = \text{Max} \\ \min_{a \in \text{Actions}(s)} \text{Expectiminimax}(\text{Result}(s,a)) & \text{if Player}(s) = \text{Min} \\ \sum_r P(r) \text{Expectiminimax}(\text{Result}(s,r)) & \text{if Player}(s) = \text{Chance} \end{array} \right.$$

Where r represents all chance events (e.g., dice roll), and $\text{Result}(s,r)$ is the same state as s with the result of the chance event is r .

Games: conclusion

- Games are modeled in AI as a search problem and use heuristic to evaluate the game.
- Minimax algorithm chooses the best move given an optimal play from the opponent.
- Minimax goes all the way down the tree which is not practical given game time constraints.
- Alpha-Beta pruning can reduce the game tree search which allows to go deeper in the tree within the time constraints.
- Pruning, bookkeeping, evaluation heuristics, node re-ordering and IDS are effective in practice.

Games: conclusion

- Games is an exciting and fun topic for AI.
- Devising adversarial search agents is challenging because of the huge state space.
- We have just scratched the surface of this topic.
- Further topics to explore include partially observable games (card games such as bridge, poker, etc.).
- Except for robot football (a.k.a. soccer), there was no much interest from AI in physical games.
(see <http://www.robocup.org/>).
- Interested in chess? check out the evaluation functions in Claude Shannon's paper.