COMP3620 / 6320 – S1 2023

This lectu will be recorded

Turn off your camera if you do not want to be in the recording

If you cannot see my whole slide, go to **View Options -> Zoom Ratio** and select **Fit to Window**

If you want to ask a question, either:

- Raise your hand and I will open your mic, or
- Type the question in the chat box

Outline for Today

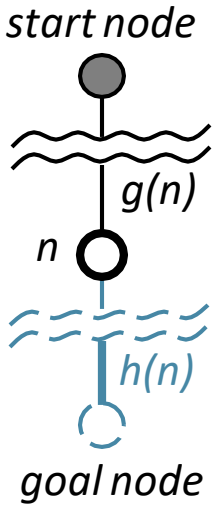
- Finish A*
- Generalizing Tree Search to Graph Search
- How to construct (Admissible) Heuristics
- Adversarial Search

Recap: A* Search

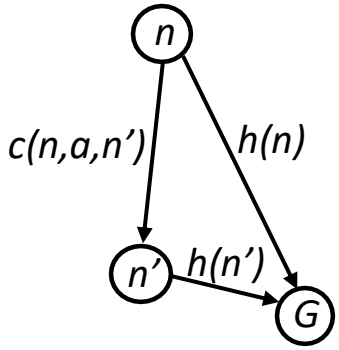
- Idea: avoid expanding paths that are already expensive
- Evaluation function $f(n) = g(n) + h(n)$
  - $g(n)$ = cost so far to reach n
  - $h(n)$ = estimated cost from n to the closest goal
  - $f(n)$ = estimated total cost of path through n to goal

- Admissible heuristic:
  - $\forall n\ h(n) \leq h^*(n)$ where $h^*(n)$ is the true cost from n
  - Also require $h(n) \geq 0$, so $h(G) = 0$ for any goal G

- Theorem: if h is admissible, A* search finds the optimal solution



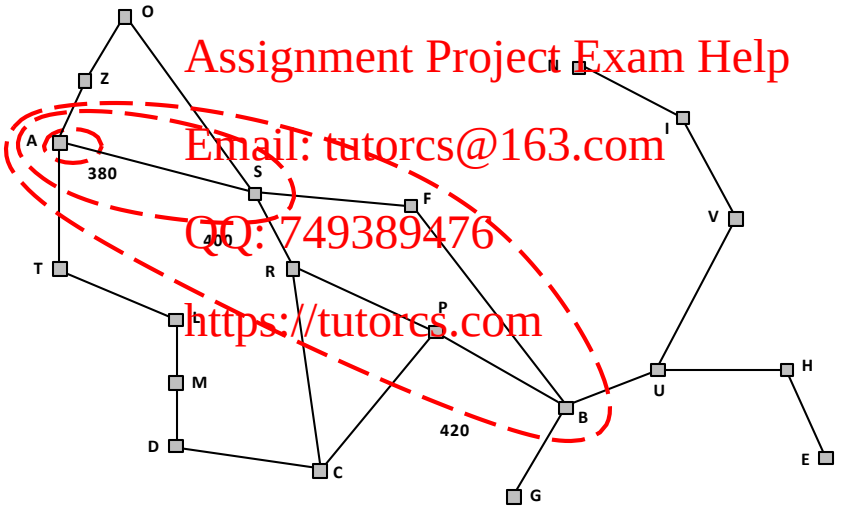
Consistency



- A heuristic is **consistent** if ***h(n) ≤ c(n, a, n') + h(n')***
- If *h* is consistent, then
  - *h* is admissible, and
  - *f*(*n*) is nondecreasing along any path: $f(n') = g(n') + h(n')$
    $= g(n) + c(n, a, n') + h(n')$
    $\geq g(n) + h(n)$
    $= f(n)$
  - ➢ Consequently, when expanding a node, we cannot get a node with a smaller *f*, and so **the value of the best node on the frontier will never decrease**.
- When we dequeue a node labelled with a new state, we found the optimal path to that state:
  - ➢ any other node *m* labelled with the same state satisfies $f(n) \leq f(m)$ and $h(n) = h(m)$, hence $g(n) \leq g(m)$.

Optimality of A* (based on consistency)

- **Consistency**: A* expands nodes in order of increasing f value
- Gradually expands "f-contours" of nodes
  - Contour i has all nodes with $f = f_i$, where $f_i < f_{i+1}$
  - (breadth-first expands layers, uniform-cost expands g-contours)



Properties of A*

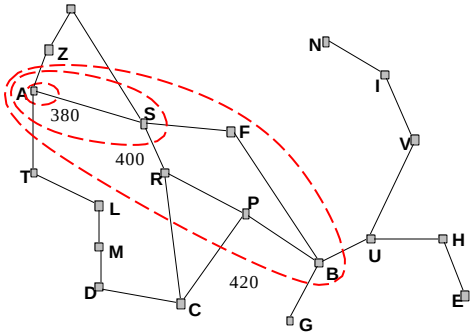
- **A* expands**
  - **all nodes** with $f(n) < C^*$ (i.e. optimal solution cost)
  - **some nodes** with $f(n) = C^*$ (only one goal node)
  - **no nodes** with $f(n) > C^*$

- Complete? Yes, unless there are infinitely many nodes with $f \leq C^*$

- Time? Exponential in [relative error in h × length of solution]

- Space? Exponential

- Optimal? Yes! Cannot expand f_{i+1} until f_i is finished



IDA* is a version of iterative deepening with a cutoff on f

Tree Search and Repeated States

- For many problems, the state space is a graph rather than a tree
- Cycles can prevent termination
- Failure to detect repeated states can turn a linear problem into an exponential one!

Graph search (simple)

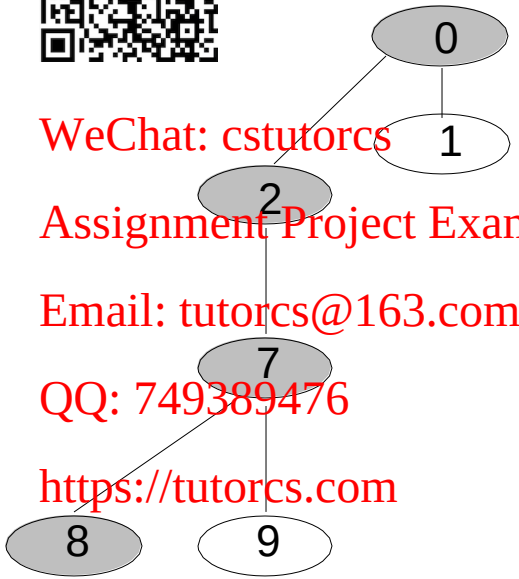

```
function GRAPH-SEARCH( problem, frontier) returns a solution, or failure
    explored ← an empty s
    frontier ← INSERT(M                INITIAL-STATE[problem]), frontier)
    loop do
        if frontier is emp                rn failure
        node ← REMOVE-FRONT(frontier)
        if GOAL-TEST(problem, STATE[node]) then return node
        add node to explored
        frontier ← INSERTNODES(EXPAND(node, problem), frontier)

function INSERTNODES( nodes, frontier) returns updated frontier
    for each n in nodes do
        if ∄ m in explored ∪ frontier : STATE[m] = STATE[n] then
            add n to frontier
    return frontier
```


- At most one instance of each state in *explored* ∪ *frontier*
- All expanded nodes kept in memory!

Graph search and optimality

- When seeking optimal solutions, multiple paths to the same state may need to be explored and compared to find the optimal

Number inside the nodes represents $g(n)$, i.e., the path cost



Graph search and optimality

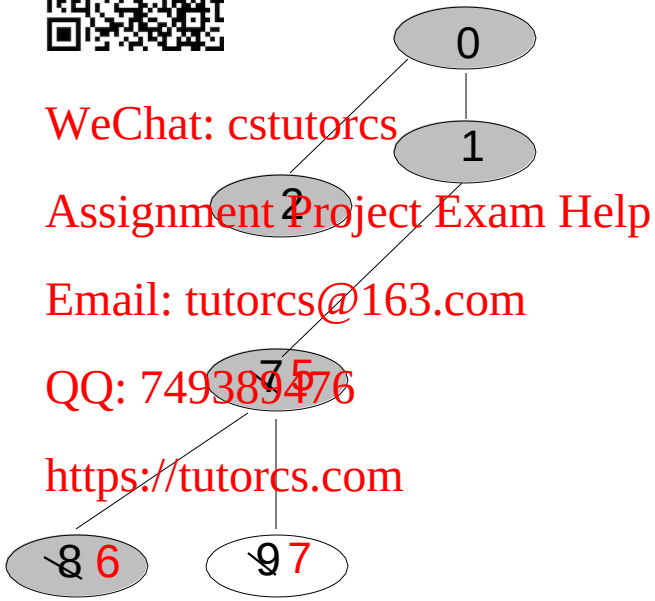
- When seeking optimal solutions, multiple paths to the same state may need to be explored and compared to find the optimal

Number inside the nodes represents $g(n)$, i.e., the path cost

Graph search and optimality

- We may need to keep the better node and update the depths and path-costs of the descendants of the worst one.

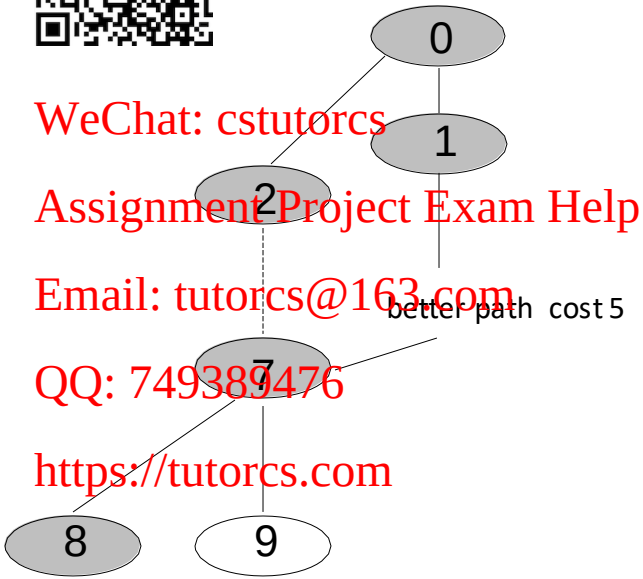
Number inside the nodes represents $g(n)$, i.e., the path cost



Graph search and optimality

- Trick to avoid updating descendants: re-open the explored node; its descendants will be updated if it is re-expanded.

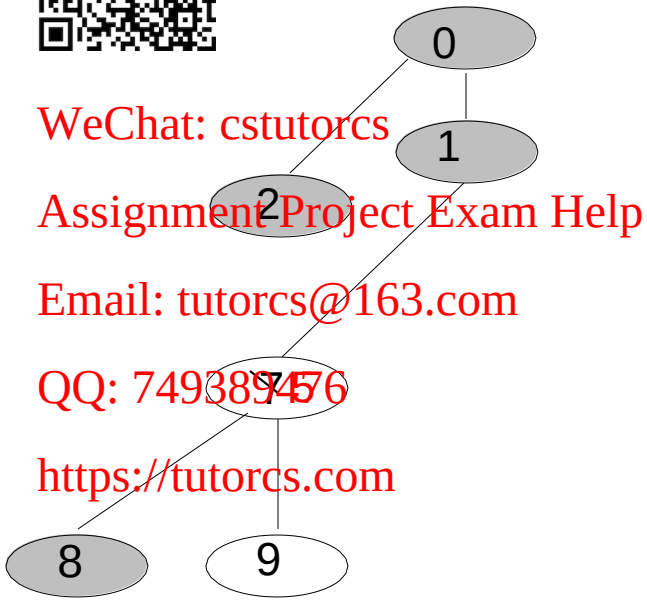
Number inside the nodes represents $g(n)$, i.e., the path cost



Graph search and optimality

- Trick to avoid updating descendants: re-open the explored node; its descendants will be updated if it is re-expanded.

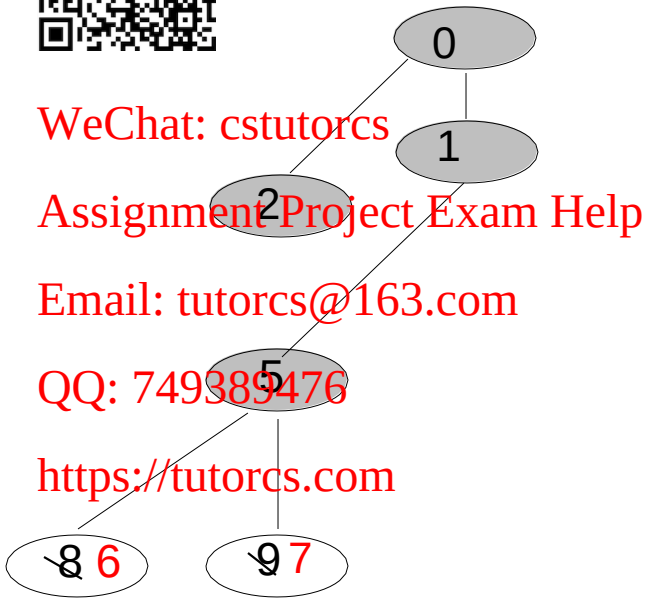
Number inside the nodes represents $g(n)$, i.e., the path cost



Graph search and optimality

- Trick to avoid updating descendants: re-open the explored node; its descendants will be updated when it is re-expanded.

Number inside the nodes represents $g(n)$, i.e., the path cost



Graph Search (optimal)


```
function INSERT-NODES( nodes, frontier) returns updated frontier
  for each n in nodes do
    if ∄ m in explored ∪ frontier such that STATE[m] = STATE[n] then
      add n to frontier
    else if PATH-COST[n] < PATH-COST[m]
      PATH-COST[m] ← PATH-COST[n]
      PARENT[m] ← PARENT[n]
      ACTION[m] ← ACTION[n]
      DEPTH[m] ← DEPTH[n]
      if m in explored then
        move m back to frontier // reopen m
  return frontier
```

Same as before

Found a cheaper path to *m*

- The internal block (else if) is needed with A*
  - *Uniform cost*: it also needs this block (unless all step costs are equal)
- **If *h* is consistent** (not just admissible), **no re-opening** (last 2 lines) is needed.
  - *Uniform cost*: no reopening is needed because *h* = 0 is consistent

Which algorithm and strategy to use

strategy	solution	useful when	ier	algorithm and state space
DFS	arbitrary	many solutions exist	O	• tree-search for finite acyclic graphs • recursive algorithm for finite acyclic graphs • add cycle detection for finite graphs
BFS	shortest	shallow solutions exist	FIFO	• tree search • graph search (simple) may improve performance
UC	optimal	good admissible heuristic lacking	priority queue ordered by g	• tree search for trees • graph-search (simple) for equal step costs • graph-search (optimal) for arbitrary step costs (no reopening needed)
A*	optimal	good admissible heuristics exist	priority queue ordered by f	• tree search for trees and admissible heuristics • graph-search (optimal) for admissible heuristics (no reopening needed for consistent heuristics)
greedy search	arbitrary but maybe good	good (inadmissible) heuristics exist	priority queue ordered by h	• tree search for finite acyclic graphs • graph search (simple) for finite graphs

Admissible heuristics – Example

8-puzzle:

- $h_1(n)$ = number of misplaced tiles
  - $h_1(start\ state\ above) = 6$
- $h_2(n)$ = total Manhattan distance
  (i.e., no. of squares from desired location of each tile)
  - $h_2(start\ state\ above) = 4+0+3+3+1+0+2+1 = 14$

Dominance

d: depth of the shallowest solution

- Given two admissible heuristics h_a, h_b, if $h_b(n) \geq h_a(n)$ for all n then
  - h_b dominates h_a and is b[illegible] search
  - In the 8-puzzle h_2 (Manhatt[illegible]minates h_1 (# misplaced). Typical search costs:

d = 14	IDS = 3,473,941 nodes	d = 24	IDS ≈ 54,000,000,000 nodes
	$A^*(h_1)$ = 539 nodes		$A^*(h_1)$ = 39,135 nodes
	$A^*(h_2)$ = 113 nodes		$A^*(h_2)$ = 1,641 nodes

- There is a trade-off between the accuracy of h and the time to compute h
- Given two admissible heuristics h_a and h_b
  $h(n) = \max(h_a(n), h_b(n))$ is also admissible and dominates h_a and h_b
- Is $h_a(n) + h_b(n)$ admissible? No! It can double count work

Relaxed Problems

- Admissible heuristics can be derived from the optimal solution cost of a relaxed version of the problem
  - **Key point**: the optimal solution cost of a relaxed problem is no greater than the optimal solution cost of the real problem

- Rules of the 8-puzzle:

> a tile can move from square A to square B if A is adjacent to B and B is blank; get all tiles in their correct positions.

- If we relax the rules so that a tile can move anywhere
  - then $h_1(n)$ gives the shortest solution
- If we relax the rules so that a tile can move to any adjacent square
  - then $h_2(n)$ gives the shortest solution

Relaxed Problems

Rules of the 8-puzzle:

a tile can move from square A to square B if A is adjacent to B and B is blank; get all tiles in their correct positions.

- Relaxing the rules so that only some tiles need to get in their correct positions and solving the relaxed problem optimally yields another admissible heuristic

– Heuristics derived from the cost of an optimal solution to a smaller subproblem are used in pattern databases to store solutions for every possible subproblem up to a given size.

Relaxed Problems

- Well-known example: travelling salesperson problem (TSP), i.e., find the least-cost tour visiting all cities exactly once

NP-Hard

- Minimum spanning tree can be computed in $O(n^2)$
  - and the sum of the edge costs in an MST is a lower bound on the optimal (open) tour cost

Announcements

- **No class on Monday!** (Canberra Day)
- **First Tutorials and Labs next Monday tutorial week after next.**
- **Quiz:**
  - **during your tutorial the tutors will give you a password for the quiz**
    - After the tutorial, your quiz password will stop working
  - open book!
  - If you have an in-person tutorial: **you must bring a device** (e.g., laptop or tablet) to do the quiz. Smartphone is OK
- **Mid-semester exam will be on March 28**, 12pm-midnight on Wattle. (1.5h+15min reading time). It uses Proctorio, please make sure in advance that it works for you
- **Assignment 1 due March 31.** Due to the exam should we move it to a week later?

Adversarial search (game playing)

Chapter 5

Outline

- Games
- Perfect play
  - minimax decisions
  - α–β pruning
- Imperfect decisions in real time

Adversarial Search Problems (Games)

- Arise in competitive multi-a[illegible]vironments
- In Game Theory, a multi-ag[illegible]ronment is called a game
- In AI, a game is often a deterministic, turn-taking, two-player, zero-sum game of perfect information:
  - deterministic
  - two agents
  - whose action alternate
  - utility values are opposite, e.g. (+1,-1)
  - fully observable
- We will write algorithms that play such games against an opponent.

Games Definition

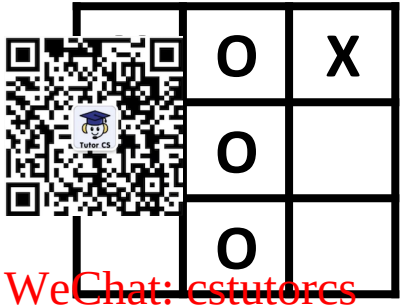
A Game consists of:

- sets of players P, states S (b[illegible]d player to play), and moves M
- an initial state $s_0 \in S$ which [illegible] how the game is set up
- Player(s) $\in P$: defines the player to move in state s
- Moves(s) $\in 2^M$: defines the set of legal moves in state s
- Result(s,m) $\in S$: defines the result of performing move m in state s
- Terminal(s) $\in$ {T,F}: the terminal test says whether the game is over
- Utility(s,p) $\in \mathbb{R}$: the utility function gives a numeric value to terminal states from the point of view of a given player, e.g.:
  - {+1, −1, 0} for chess
  - {−192, . . . , 192} for backgammon

Strategy

- "Unpredictable" opponent
  ⇒ solution for a player is not a sequence of actions but a strategy
- A strategy for a player: a function mapping the player's states to (legal) moves.
- A winning strategy always lead the player to a win from s_0

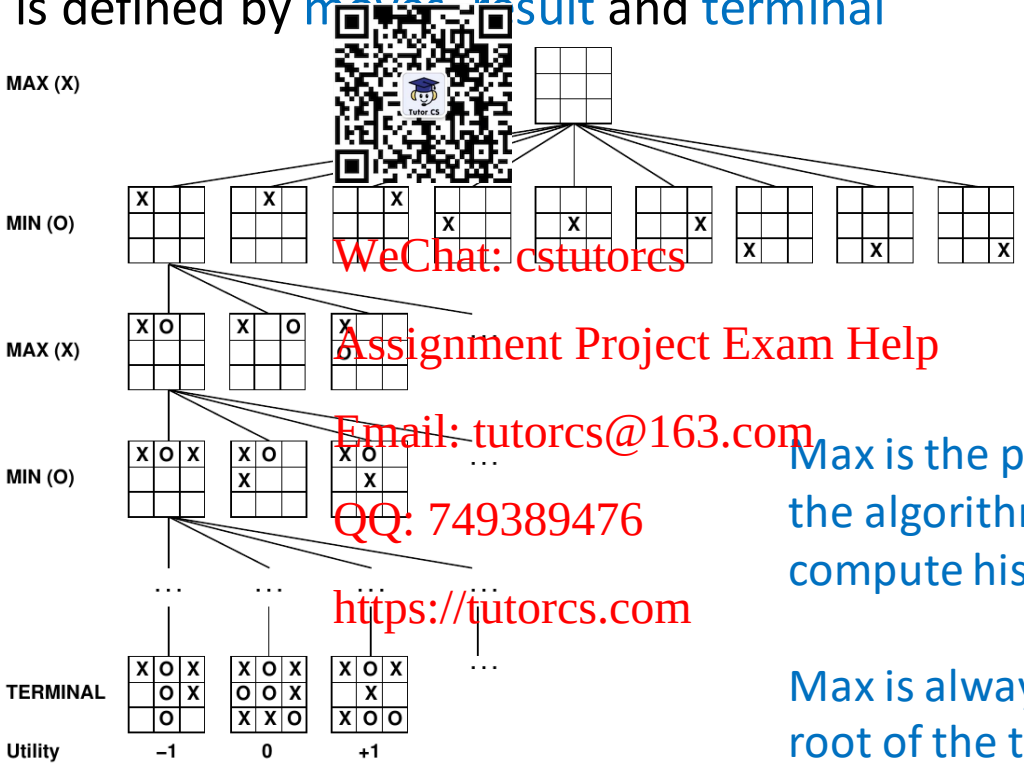
Example: Tic-tac-toe



- States? content of each cell {X, O, empty}, player to play
- Moves? an empty cell
- Result? content of chosen cell is X or O depending on the player playing; next player
- Terminal test? are 3 Os or 3 Xs aligned or is the board full?
- Utility function? for a given player gives +1 if player has aligned 3 tokens, -1 if his opponent has, and 0 otherwise. There is a draw strategy.

Game tree (2-player, deterministic, turns)

The game tree is defined by moves, result and terminal



Max is the player using the algorithm to compute his strategy

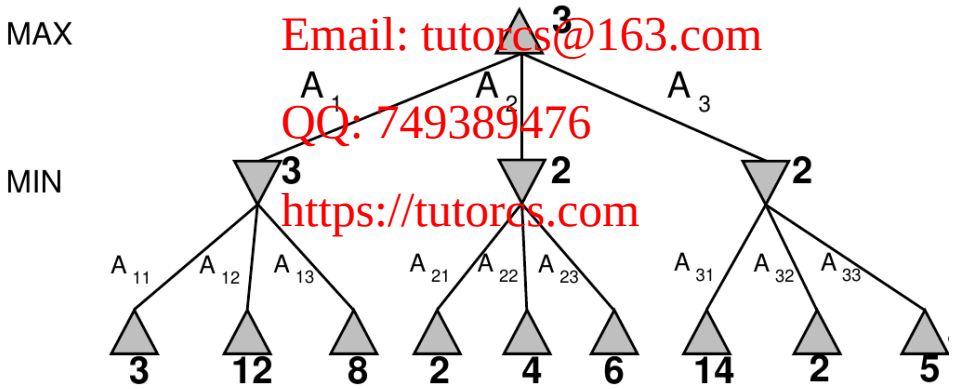
Max is always at the root of the tree

Minimax

- **Perfect play** for deterministic, two-player, zero-sum, perfect-information games
- **Idea**: choose move to position with highest minimax value
  - ➢ best achievable utility against best possible opponent

$$\text{MINIMAX-VALUE}(s) = \begin{cases} \text{UTILITY}(s, \text{MAX}) & \text{if TERMINAL}(s) \\ \max_{m \in \text{MOVES}(s)} \text{MINIMAX-VALUE}(\text{RESULT}(s, m)) & \text{if PLAYER}(s) = \text{MAX} \\ \min_{m \in \text{MOVES}(s)} \text{MINIMAX-VALUE}(\text{RESULT}(s, m)) & \text{if PLAYER}(s) = \text{MIN} \end{cases}$$

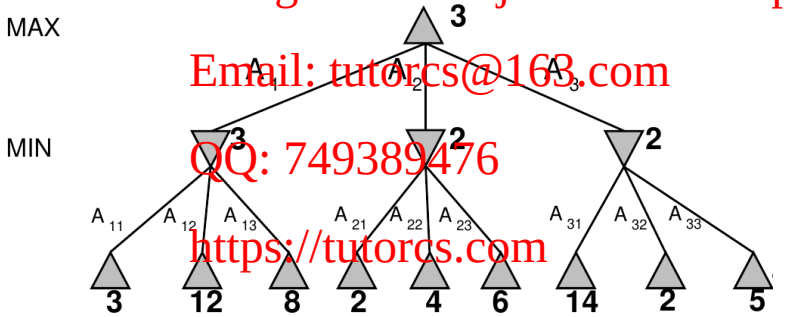
E.g. 2-ply game:



Minimax

Computing the Minimax value:

- Apply utility function to each leaf of the game tree
- Back-up values from the leaves through inner nodes up to the root:
  - **min** node: compute the min of its children values
  - **max** node: compute the max of its children values
- At the root: choose the move leading to the child of highest value



- Better method: use a depth-first like approach to save space

Minimax Algorithm


```
function MINIMAX-DECISION(state) returns a move
  inputs: state, curre                    me
  v ← MAX-VALUE(st
  return the move m                       ate) with value v
```

```
function MAX-VALUE(state) returns a utility value
  if TERMINAL(state) then return UTILITY(state,MAX)
  v ← −∞
  for m in MOVES(state) do
    v ← MAX(v, MIN-VALUE(RESULT(state,m)))
  return v
```

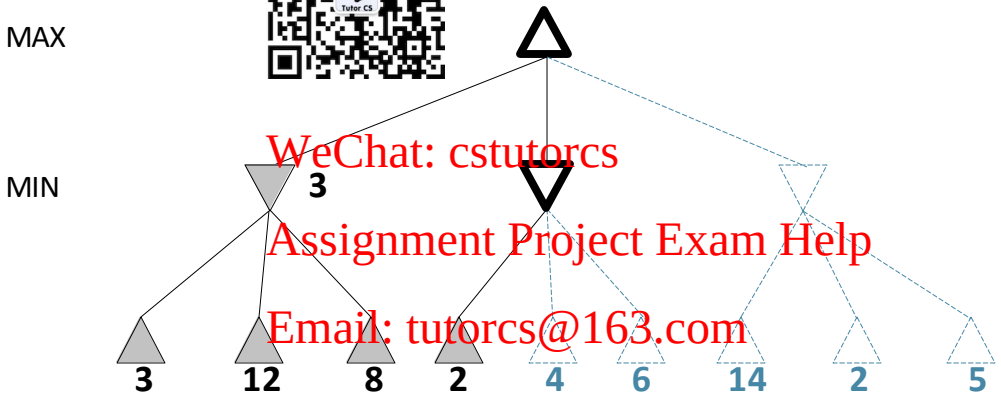
```
function MIN-VALUE(state) returns a utility value
  if TERMINAL(state) then return UTILITY(state,MAX)
  v ← +∞
  for m in MOVES(state) do
    v ← MIN(v, MAX-VALUE(RESULT(state,m)))
  return v
```

Properties of Minimax

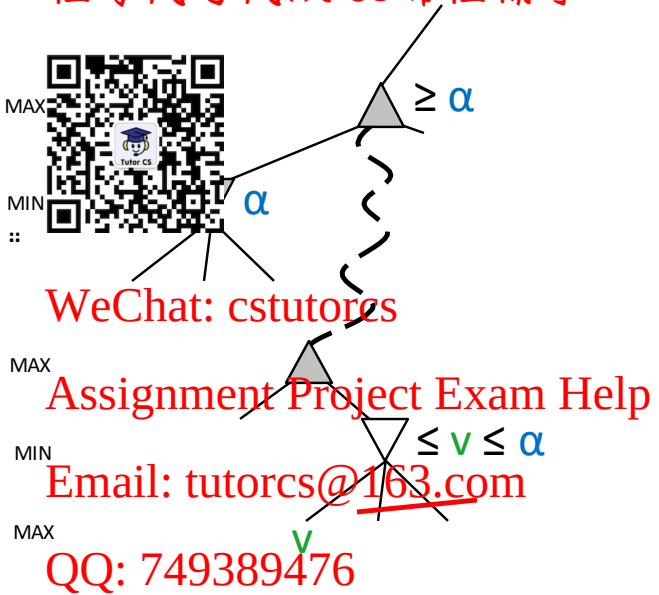
- Complete? Yes, if tree is finite
- Optimal? Yes, against an optimal opponent. Otherwise?
- Time complexity? $O(b^m)$
- Space complexity? $O(bm)$ (depth-first exploration)
  - For chess, $b \approx 35$, $m \approx 100$ for "reasonable" games
    ⇒ exact solution completely infeasible
  - But do we need to explore every path?

b: max. branching factor
d: depth of the shallowest solution
m: max. depth of the state space

Do we need to explore every path?

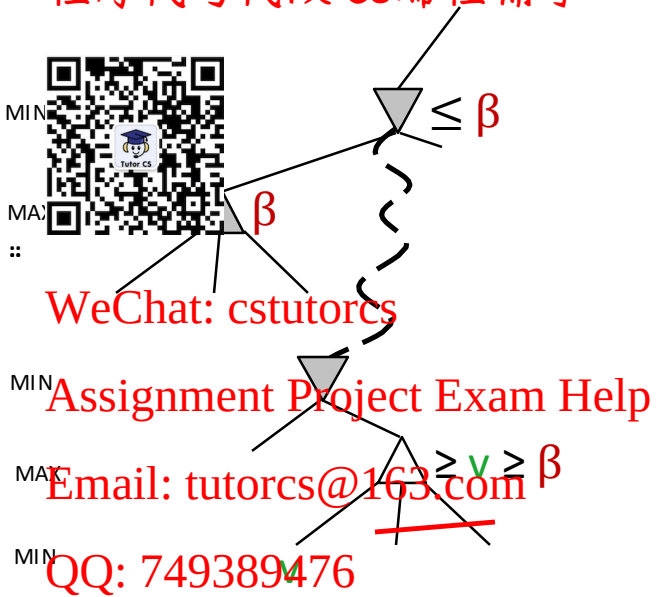


α–β pruning



- α is the best value (to max, i.e. highest) found so far
- If v is not better (greater) than α, max will avoid it
  - ➢ prune that branch
- Define β similarly for min

α–β pruning



- β is the best value (to min, i.e. lowest) found so far
- If v is not worse (lower) than β, min will avoid it
  - ➢ prune that branch

α–β pruning

- Idea:
  - α is the best (largest) value fo[illegible]ax on path to current node
  - β is the best (lowest) value fo[illegible]in on path to current node
  - **Some values** outside the interval [α, β] can be pruned

- Algorithm:
  - Node passes its current values for α and β to its children in turn
  - Child passes back up its value to Node
  - Node updates its current value v (max or min with child's value)
  - Node checks whether v ≤ α (for min node) or v ≥ β (for max node)
    - **If so**, child's siblings can be **pruned** and v returned to Parent
    - **Otherwise** β (min) or α (max) is **updated**

The α–β algorithm


```
function ALPHA-BETA-DECISION(state) returns a move
    v ← MAX-VALUE(sta
    return the move m ir                te) with value v

function MAX-VALUE(                 eturns a utility value
    inputs: state, current state in game
            α, the value of the best choice for MAX so far
            β, the value of the best choice for MIN so far
    if TERMINAL(state) then return UTILITY(state, MAX)
    v ← −∞
    for m in MOVES(state) do
        v ← MAX(v, MIN-VALUE(RESULT(m,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
    same as MAX-VALUE but with roles of α, β reversed
```