

COMP5001 Artificial Intelligence

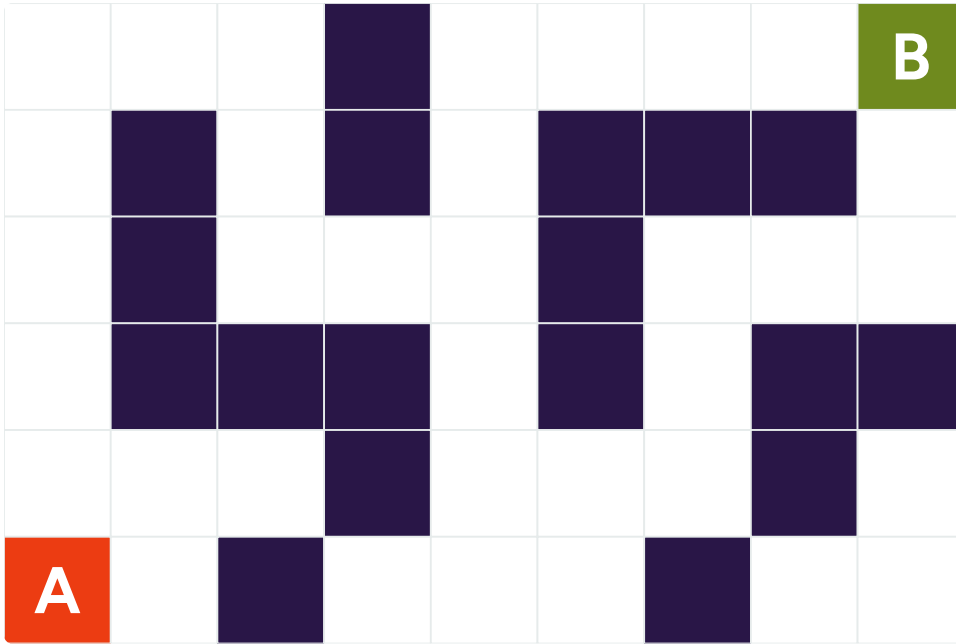
Week 2 — Search

Dr Sakib Anwar

University of Winchester

2026/27

Remember Week 1?



The Week 1 maze. Get from A to B without going through a wall.

? *Get out your maze pseudocode. When it could try several squares next, which one did it pick?*

GROUPS • 5 MIN

- ▶ "The **newest** square I found"
- ▶ "The **oldest** square I found"
- ▶ "The square **closest to B**"

Each of these is a real search algorithm. You meet the first two today and the third next week.

What is a search problem?

? *A sat-nav, a maze and a sliding-tile puzzle. What do they have in common?*

	Start	Goal
Sat-nav	where you are	where you want to go
Maze	square A	square B
15 puzzle	a jumbled board	tiles 1 to 15 in order

In each one, the job is to find the **steps** that get you from the start to the goal.

SEARCH PROBLEM

An **agent** starts in one situation and wants to reach a goal. It must find a sequence of actions that gets it there.

AGENT

Something that perceives its environment and acts on it. In a sat-nav, the agent is the car.

States, actions and results

STATE

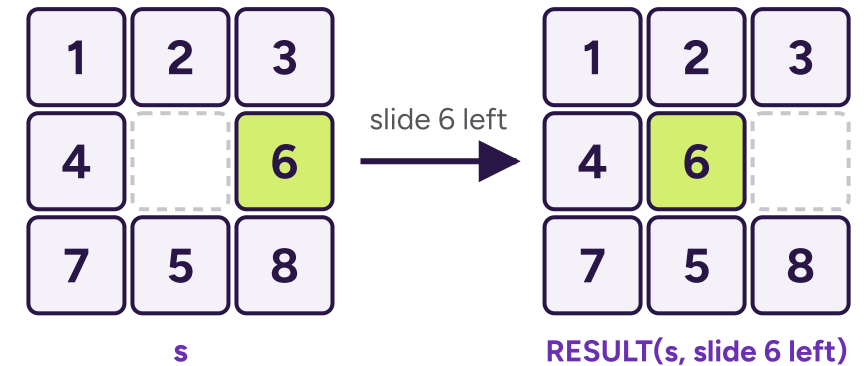
One configuration of the agent and its world. In a sliding puzzle: one arrangement of the tiles. The **initial state** is where the search starts.

ACTIONS

ACTIONS(s) returns the choices you can make in state **s**.

TRANSITION MODEL

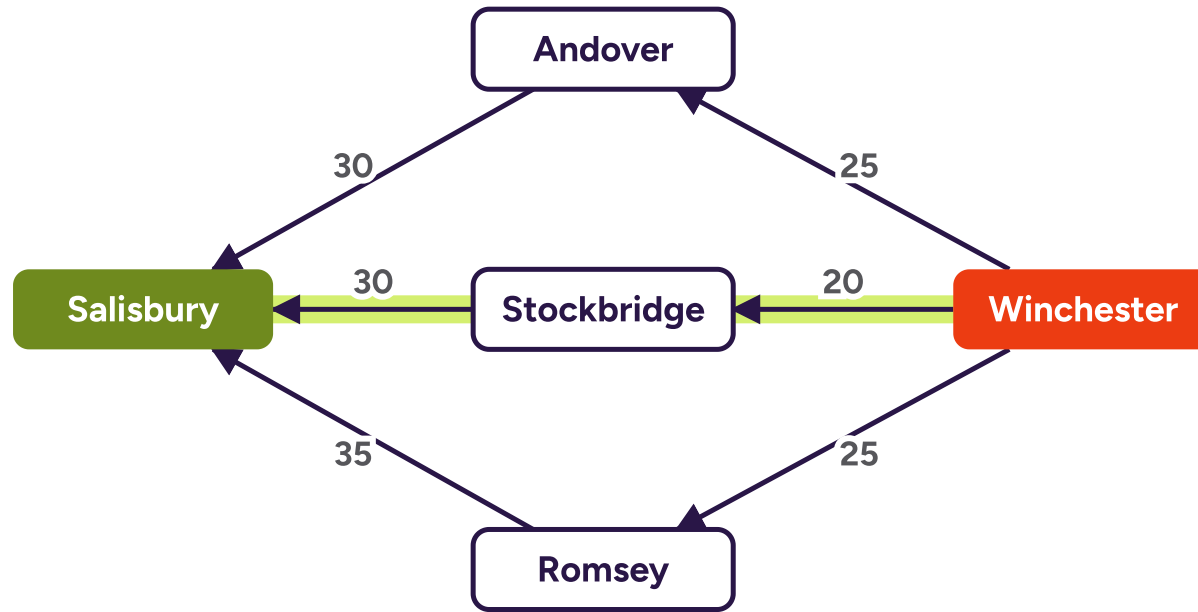
RESULT(s, a) returns the new state you get by doing action **a** in state **s**.



? *How many moves can you make when the gap is in the middle? On an edge? In a corner?*

Middle: **4** · edge: **3** · corner: **2**. **ACTIONS(s)** depends on the state.

State space, goal and path cost



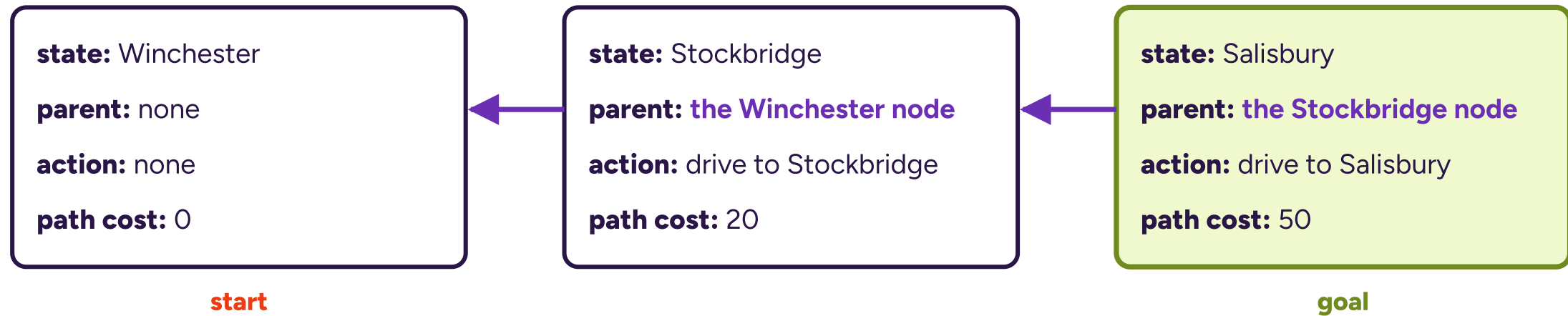
Driving times in minutes (made-up but realistic).

- ▶ **State space:** every state you can reach from the initial state. Draw it as a graph: states are the boxes, actions are the arrows.
- ▶ **Goal test:** is this state a goal? (*Are we in Salisbury yet?*)
- ▶ **Path cost:** a number for each path, here the total minutes.
- ▶ **Solution:** a sequence of actions from the initial state to a goal.
- ▶ **Optimal solution:** the solution with the lowest path cost.

❓ *Three routes to Salisbury. Which one is the optimal solution?*

Via **Stockbridge**: $20 + 30 = 50$ minutes. Andover takes 55, Romsey 60.

Nodes: what the search remembers



NODE

A small record the search keeps for each state it reaches: the **state**, its **parent** (the node it came from), the **action** that got here, and the **path cost** so far.

KEY IDEA

At the goal, follow the parents back: Salisbury → Stockbridge → Winchester. Reverse the actions and you have the solution.

The frontier: a to-do list of nodes

```
frontier = a node with the initial state
explored = an empty set
repeat:
    if the frontier is empty: there is no solution
    remove a node from the frontier
    if the node is a goal: return the solution
    add the node to explored
    expand the node: add each neighbour to the frontier
    unless it is already in the frontier or explored
```

Frontier: the nodes we have found but not looked at yet.

Expand: use **ACTIONS** and **RESULT** to find the neighbours of a node.

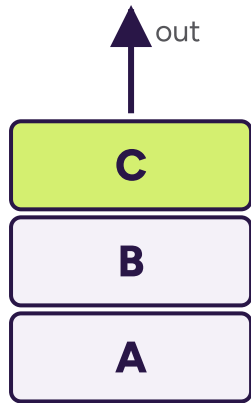
? *Roads go both ways: Winchester → Stockbridge → Winchester → ... What could go wrong?*

The search could go round in circles forever.

Explored set: the states already looked at. Never add them to the frontier again.

Which node comes out next?

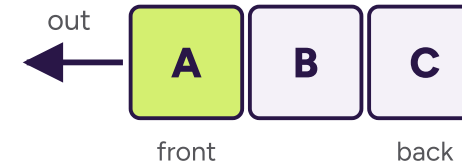
? *The frontier holds three nodes, added in the order A, B, C. Which one should the search take out next?*



Stack

Last in, first out. Like a pile of plates: you take the top one.

DFS depth-first search



Queue

First in, first out.

Like a queue at a café: first come, first served.

BFS breadth-first search

KEY IDEA

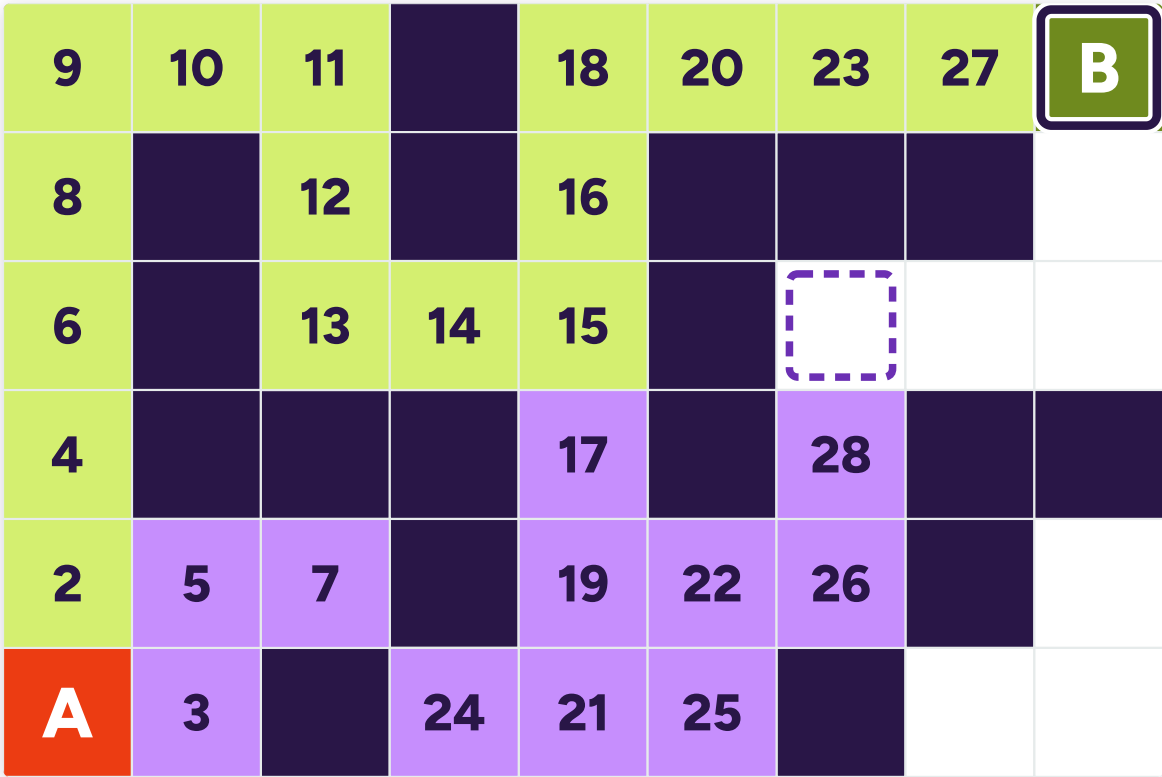
This one choice is the only difference between depth-first and breadth-first search. The rest of the loop is the same.

STACK



9

Breadth-first search QUEUE



waiting in the frontier explored (in order) path

Rule: always take the **oldest** square in the frontier. The frontier is a queue.

Steps 2–3. One step up, then one step right. BFS tries every direction before it goes further.

Steps 16–28. At the fork it takes turns: one step on the top route, one on the lower route, and so on, like a ripple spreading out.

Step 29. B, after exploring 29 squares: more than DFS.

But its path is **17 steps**, the shortest, the same as your Week 1 answer.

So which search is better?

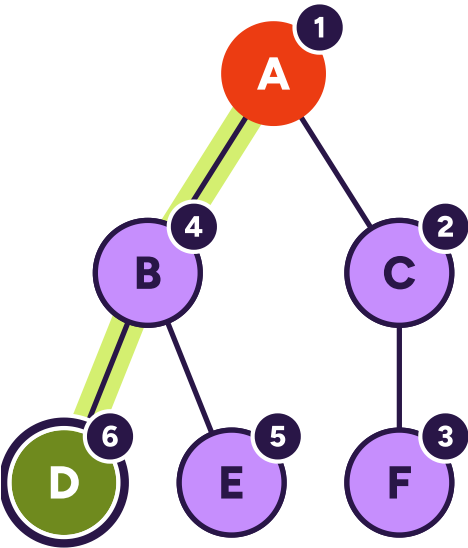
Neither always wins. BFS always finds the shortest path, when every step costs the same. DFS can be quicker, but its path may be long.

Trace it: DFS and BFS

? Start at A, the goal is D. In what order does each search take nodes out of the frontier?

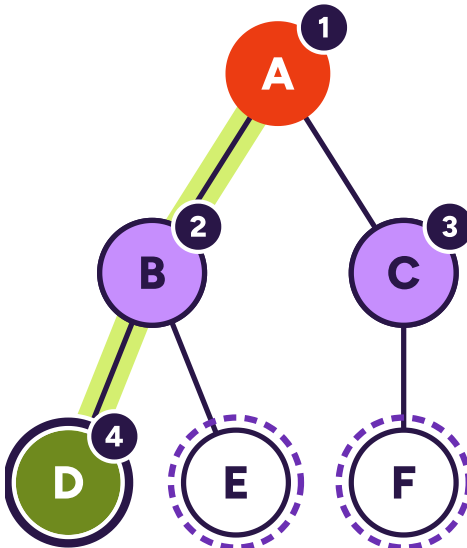
PAIRS · 3 MIN

DFS take the **last** one added (stack)



Step	Take out	Frontier after
1	A	B C
2	C	B F
3	F	B
4	B	D E
5	E	D
6	D	goal ✓

BFS take the **first** one added (queue)



Step	Take out	Frontier after
1	A	B C
2	B	C D E
3	C	D E F
4	D	goal ✓

Children join the right-hand end of the frontier, left child first.

Both return A → B → D. BFS took out 4 nodes; DFS took out 6, because it went down C first.

In code: the only change

DFS `StackFrontier` in `maze.py`: take the **last** node

```
def remove(self):  
    node = self.frontier[-1]  
    self.frontier = self.frontier[:-1]  
    return node
```

BFS `QueueFrontier`: take the **first** node

```
def remove(self):  
    node = self.frontier[0]  
    self.frontier = self.frontier[1:]  
    return node
```

`maze2.txt`, a bigger maze with one way through:

194

squares explored by DFS

77

squares explored by BFS

Both find the same 30-step path.

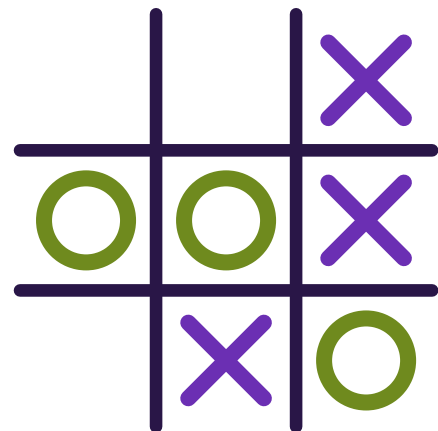
IN THE PRACTICAL

Run `maze.py`, switch `StackFrontier` to `QueueFrontier`, and compare `maze2.txt` and `maze3.txt`.

A game as a search problem

? *In a maze, nobody fights back. What changes when there is an opponent?*

The opponent chooses every other move, and wants the opposite result to you.



X to move. Where should X play?

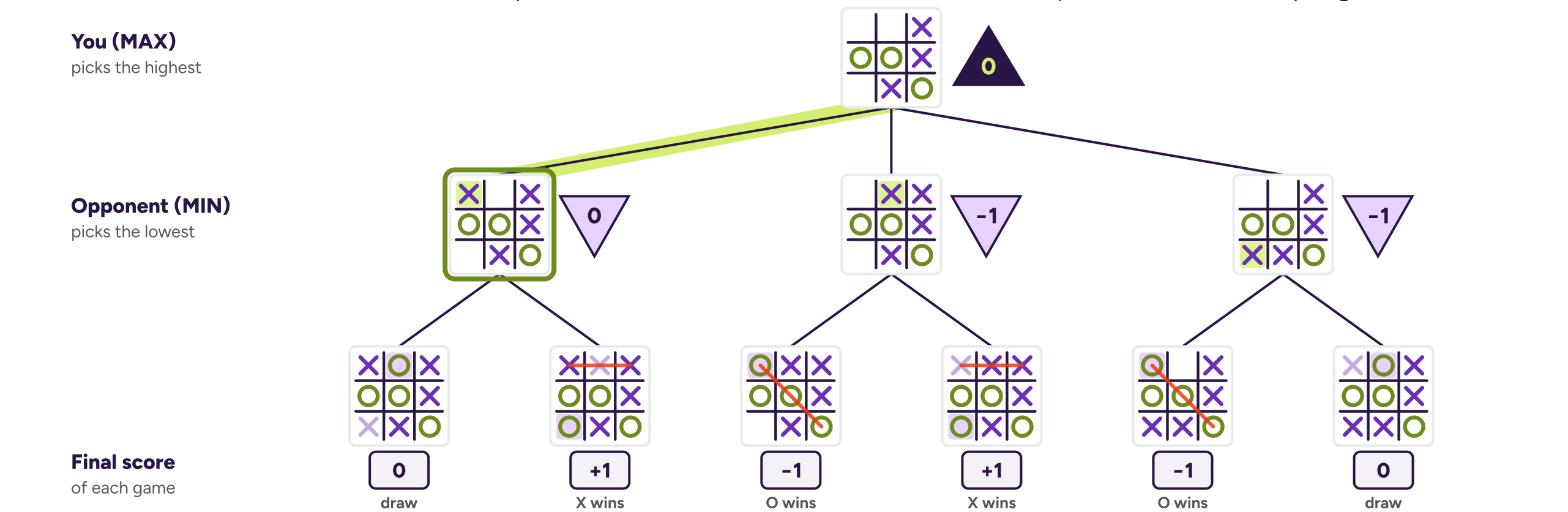
S_0	the initial state: an empty board
$PLAYER(s)$	whose turn it is: X or O
$ACTIONS(s)$	the empty squares
$RESULT(s, a)$	the board after playing in square a
$TERMINAL(s)$	is the game over? (a win or a full board)
$UTILITY(s)$	the score of a finished game: X wins +1 , draw 0 , O wins -1

MAX X wants the score as **high** as possible. **MIN** O wants it as **low** as possible.

Minimax on a real game

? *X to move on the board from the last slide. Which square should X take? Let minimax decide.*

1. Score each finished game: X wins **+1**, draw **0**, O wins **-1**.
2. Under each of X's moves, O (MIN) replies with its **lowest** score.
3. X (MAX) takes the **highest**: block top-left, a draw (0). Top-middle looked tempting, but O wins first.

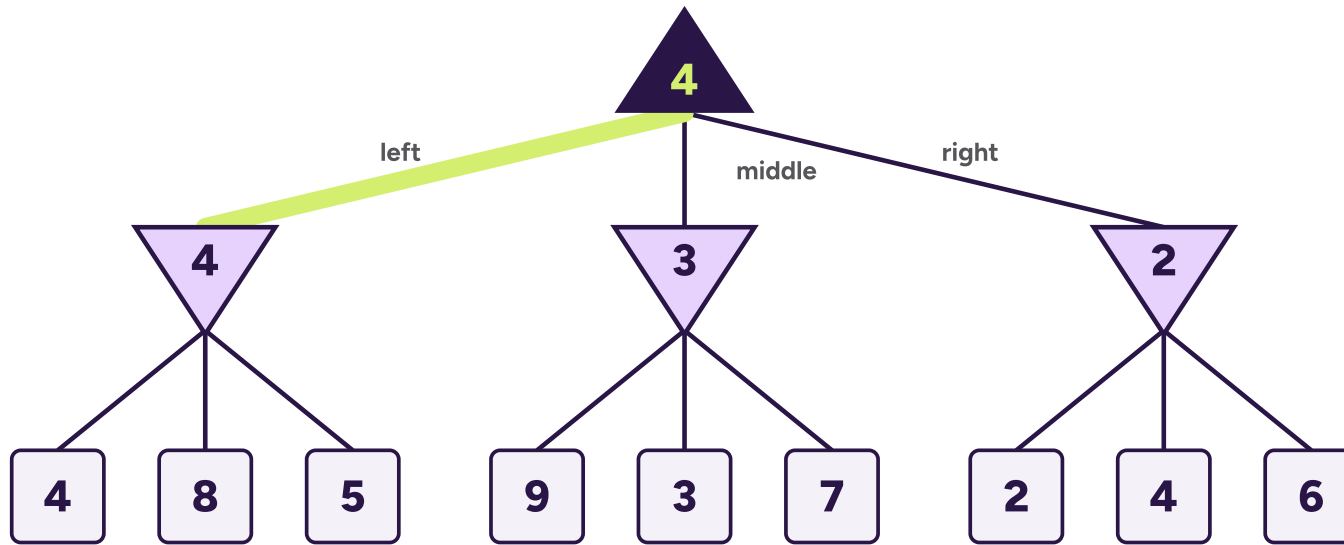


Minimax

You (MAX)
picks the highest

Opponent (MIN)
picks the lowest

Final score
of each game



The same kind of tree as the tic-tac-toe slide, with numbers in place of boards. Each line is a move. The bottom numbers are the scores of finished games, like +1, 0 and -1 but with more values: **higher is better for you**, lower is better for your opponent.

? *You are MAX. MIN always replies with the lowest score it can. Which move should you make?*

1. Work from the bottom up.
Each **MIN** node takes its lowest child: 4, 3, 2.
2. **MAX** takes the highest of those: the **left** move, worth 4.

The middle has a 9, but MIN will never let you get it.

Minimax in pseudocode

```
function MAX-VALUE(state):  
    if TERMINAL(state): return UTILITY(state)  
    v =  $-\infty$   
    for action in ACTIONS(state):  
        v = MAX(v, MIN-VALUE(RESULT(state, action)))  
    return v
```

```
function MIN-VALUE(state):  
    if TERMINAL(state): return UTILITY(state)  
    v =  $+\infty$   
    for action in ACTIONS(state):  
        v = MIN(v, MAX-VALUE(RESULT(state, action)))  
    return v
```

TO CHOOSE A MOVE

MAX picks the move with the **highest** `MIN-VALUE`. MIN picks the move with the **lowest** `MAX-VALUE`. The two functions call each other, all the way to the end of the game.

Pseudocode is a program written in plain words. It is not a real language and does not run. The top block, line by line:

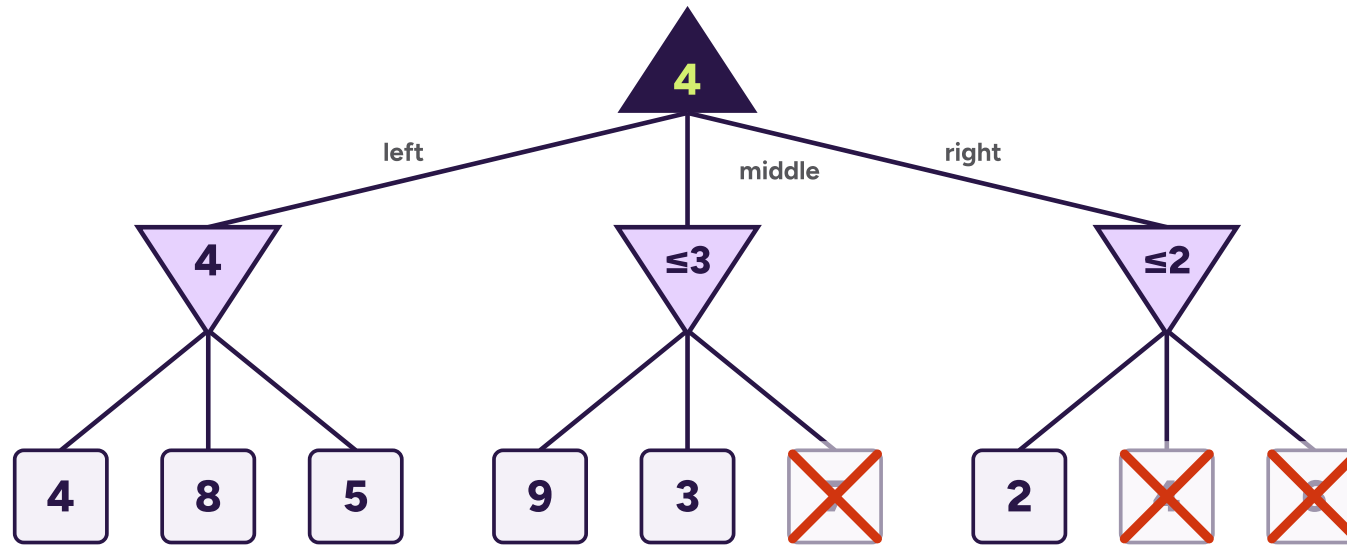
- ▶ **function MAX-VALUE(state):** takes a state and **returns** a score. The indented lines belong to it, as in Python.
- ▶ **if TERMINAL(state):** is the game over? Then the value is just its score.
- ▶ **v = $-\infty$** start below any real score, so the first real score replaces it.
- ▶ **for action in ACTIONS(state):** try each legal move in turn.
- ▶ **v = MAX(v, MIN-VALUE(...))** score the move by asking what MIN would do next, and keep the best score so far.
- ▶ **The bottom block** is the same for MIN: start at $+\infty$ and keep the lowest.

Alpha-beta pruning: skip what cannot matter

You (MAX)
picks the highest

Opponent (MIN)
picks the lowest

Final score
of each game



Each line is a move. The bottom numbers are the scores of finished games, like tic-tac-toe's +1, 0 and -1 but with more values: **higher is better for you**, lower is better for your opponent.

? *You already have a move worth 4.
In the middle, MIN can reach 3.
Do you need to look at the 7?*

No. MIN will pick 3 or less here, so this move is worth at most 3 to you. You already have 4. Stop looking.

On the right, the first leaf is 2. Same story: skip the rest.

Same answer as minimax, but **3 of the 9** leaves were never looked at.

Too big to search to the end

255,168

possible games of tic-tac-toe

10^{29000}

possible games of chess (roughly)

? *A computer can play out every game of tic-tac-toe in a moment. Could it do the same for chess?*

DEPTH-LIMITED MINIMAX

Look only a fixed number of moves ahead, then stop, even though the game is not over.

EVALUATION FUNCTION

Estimates how good a position is without playing to the end. For example, in chess: count the pieces each side has left. The program is only as good as this estimate.

Search in the real world

- ▶ **Sat-navs and map apps:** road junctions are states, roads are actions.
- ▶ **Game engines:** chess and Go programs use adversarial search.
- ▶ **Warehouse robots:** plan paths that avoid shelves and other robots.
- ▶ **Video games:** characters find their way round the map.
- ▶ **Puzzles:** Rubik's cube and sliding-tile solvers.

Discuss

❓ *A sat-nav app picks the “best” route for millions of drivers. Best for whom: fastest, shortest or least fuel? Who gains and who loses?*

GROUPS • 10 MIN

The **path cost** is a design decision. It changes where traffic goes.

Thank you

Practical: `week02-search/maze` in the module code

Next week: smarter search with heuristics (greedy best-first and A*), then logic

Reading: Russell & Norvig, sections 3.1–3.4 and 5.1–5.3; Poole & Mackworth, sections 3.1–3.5 and 14.3

Sakib.Anwar@winchester.ac.uk

Adapted from *CS50's Introduction to Artificial Intelligence with Python* by Brian Yu and David J. Malan, Harvard University, licensed under [CC BY-NC-SA 4.0](#).