

Search

Finding a path from a start to a goal, and choosing a move when someone plays against you

Slides: Week 2 lecture · Seminar: Week 2 sheet · Code: github.com/sakibanwar/comp5001-ai-code, folder **week02-search**

These notes are adapted from the lecture notes of *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, and are shared under the same licence. Sections 7 to 11 add material from the chapter on adversarial search and games in Russell and Norvig's *Artificial Intelligence: A Modern Approach* (4th ed.).

THIS WEEK

By the end of the week you should be able to:

- describe a problem as a search problem: initial state, actions, transition model, goal test and path cost;
- explain how depth-first search and breadth-first search differ, and trace both on a small example;
- use minimax to choose a move in a simple game.

Remember Week 1? You wrote pseudocode for a maze. By the end of Section 4 you will know which search algorithm you wrote.

- | | |
|---|---|
| 1. Search problems | 7. Adversarial search |
| 2. Nodes, the frontier and the explored set | 8. Minimax |
| 3. Depth-first search | 9. Alpha-beta pruning |
| 4. Breadth-first search | 10. Too big to search to the end |
| 5. DFS and BFS side by side | 11. Search in the real world |
| 6. In the code: maze.py | 12. Check yourself, key terms and reading |

1 Search problems

A sat-nav finding a route, a person solving a maze and a program solving a sliding-tile puzzle are all doing the same job. Each starts in one situation, wants to reach a goal, and has to find the steps that get it there. That is a **search problem**.

	Start	Goal	A step is...
Sat-nav	where you are	where you want to go	driving along one road
Maze	square A	square B	moving one square up, down, left or right
15 puzzle	a jumbled board	tiles 1 to 15 in order	sliding one tile into the gap

To make a computer solve these problems, we describe each one with the same few words. Learn these: they come back every week until Week 6.

THE WORDS FOR A SEARCH PROBLEM

Agent

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

State

One configuration of the agent and its environment. *In a sliding puzzle, one arrangement of the tiles; in a maze, the square you are on.*

Initial state

The state where the search starts. *Your current location.*

Actions

ACTIONS(s) returns the set of actions you can take in state **s**. *In a maze: the open squares next to you.*

Transition model

RESULT(s, a) returns the state you reach by doing action **a** in state **s**. *Slide a tile, and you get a new board.*

State space

Every state you can reach from the initial state by any sequence of actions. We draw it as a graph: states are the dots (or boxes), actions are the arrows.

Goal test

A way to check whether a state is a goal. *Are we at the destination yet?*

Path cost

A number for each path, such as total distance or time. *A sat-nav tries to keep it low.*

Example: actions depend on the state

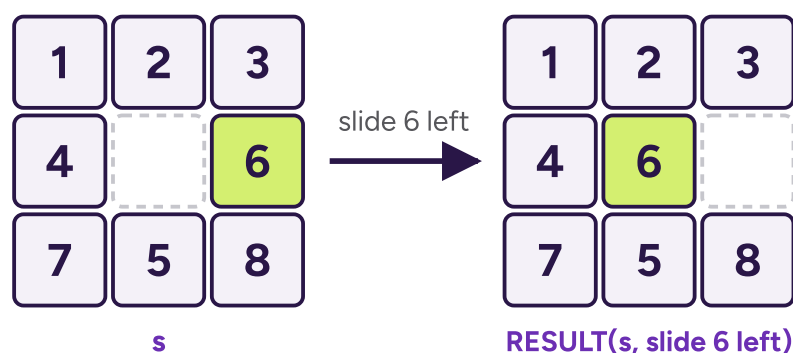
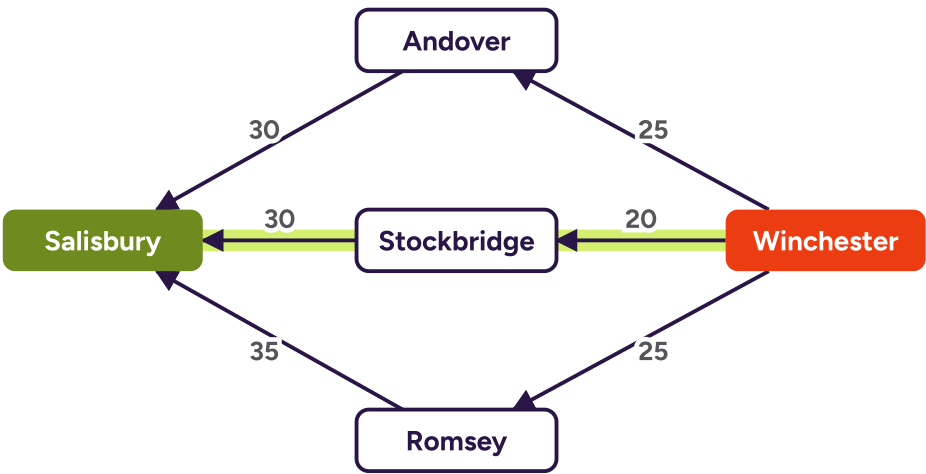


Figure 1. A small sliding puzzle. With the gap in the middle, four tiles (2, 4, 5 and 6) could slide in. Sliding 6 left gives a new state: that new board is **RESULT(s, slide 6 left)**.

How many actions are there? It depends where the gap is. With the gap in the **middle** there are 4 moves, on an **edge** there are 3, and in a **corner** only 2. That is why **ACTIONS** takes the state as its input: different states

allow different actions.

Example: a road map, and what “best” means



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

Figure 2. A state space drawn as a graph. Each box is a state (being in that town) and each arrow is an action (driving to the next town). Winchester is the initial state; Salisbury is the goal. The numbers are driving times in minutes (made up, but realistic). The green route is the optimal solution.

A **solution** is any sequence of actions from the initial state to a goal. An **optimal solution** is a solution with the lowest path cost. Here there are three solutions:

Route	Path cost
Winchester → Andover → Salisbury	25 + 30 = 55 minutes
Winchester → Stockbridge → Salisbury	20 + 30 = 50 minutes (optimal)
Winchester → Romsey → Salisbury	25 + 35 = 60 minutes

Notice that we chose minutes as the path cost. If the path cost were miles or fuel, a different route might be “best”. The seminar comes back to this: choosing the path cost is a design decision, and it affects people.

HOW BIG IS A STATE SPACE?

Real state spaces are far too big to draw. The small 3 × 3 puzzle in Figure 1 has 181,440 reachable boards (9!/2). The 4 × 4 15 puzzle has about 10.5 trillion (16!/2). This is why we need a *systematic* way to search, rather than drawing the whole graph.

2 Nodes, the frontier and the explored set

As a search runs, it keeps a small record for every state it reaches. That record is called a **node**.

NODE

A data structure that keeps track of: a **state**; its **parent** (the node it came from); the **action** that was used to get from the parent to here; and the **path cost** from the initial state to here.

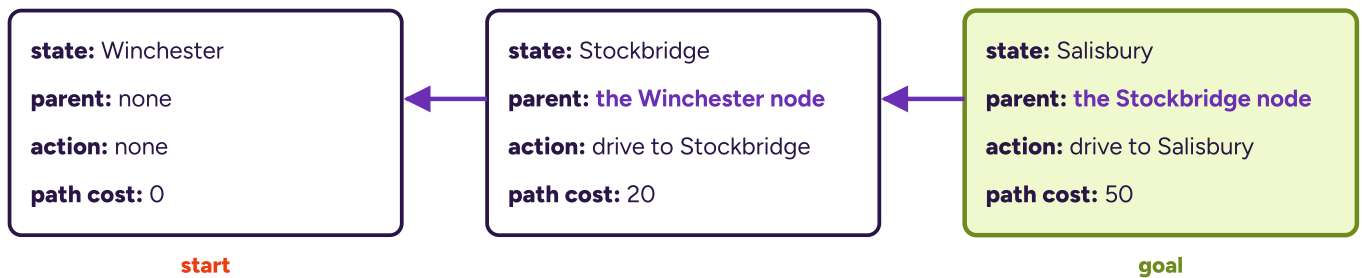


Figure 3. Three nodes on the road map. Each one points back to its parent. A node does not search; it only holds information.

The parent link is what lets us read off the solution at the end. When the search reaches the goal, it follows the parents back (Salisbury → Stockbridge → Winchester) and then reverses the list of actions: “drive to Stockbridge, then drive to Salisbury”.

The frontier

The **frontier** is a to-do list: the nodes we have found but not looked at yet. The loop below is the search algorithm. Every search in Weeks 2 and 3 is this loop with one line changed.

```
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
```

Expanding a node means using **ACTIONS** and **RESULT** to find its neighbours, then adding them to the frontier as new nodes.

SEARCH ALGORITHM

A step-by-step method that takes a search problem and returns a solution (a sequence of actions from the initial state to a goal), or reports that there is none. Depth-first and breadth-first search (this week) and greedy best-first search and A* (Week 3) are all search algorithms. All of them run this loop; they differ only in **which node they take out of the frontier next**.

The search tree

Each new node points back to its parent, so as the loop runs, the nodes it creates join up into a tree.

SEARCH TREE

The tree of nodes that a search builds as it runs. The **root** is the node for the initial state. Expanding a node gives it one **child** for each action, and each child points back to its parent. The root is at **depth 0**, its children at depth 1, and each level down the tree is one step deeper.

The search tree is not the same thing as the state space. The state space (Figure 2) has one box for each *state*. The search tree has one node for each *path* the search has found, so the same state can appear in it more than once. Figure 4 shows the top of the search tree for the road map, with every road usable in both directions.

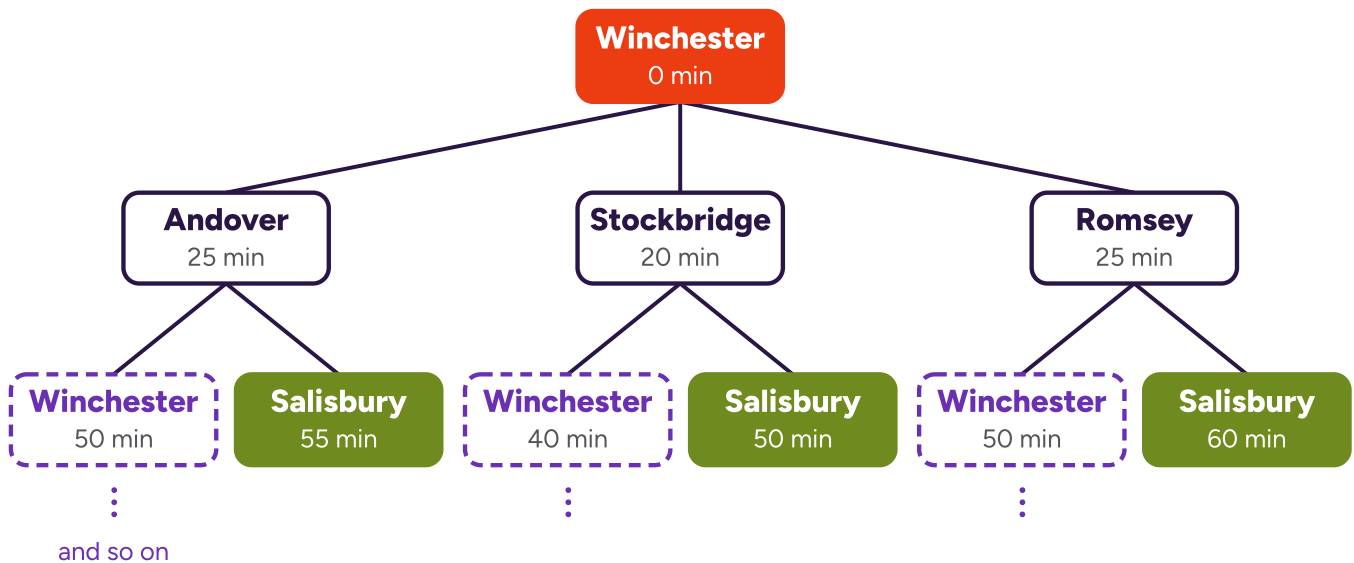


Figure 4. The top of the search tree for the road map in Figure 2, starting from Winchester, with every road usable in both directions. Each box is a node: a town (its state) and the driving time so far (its path cost). The dashed nodes are Winchester again; below each of them the tree starts again.

Three things to notice:

- **The same state can appear in several nodes.** Salisbury appears three times, once for each route, with path costs of 55, 50 and 60 minutes. The state is the same; the paths are different. That is why a node stores its parent and its path cost, not just its state.
- **The search tree can be much bigger than the state space.** Every road goes both ways, so Winchester appears again below each town, and below each of those the whole tree starts again. The map has only five towns, but its search tree never ends.
- **A search does not draw the whole tree first.** It grows the tree one node at a time, and stops as soon as it takes a goal out of the frontier. The frontier is the bottom edge of the part built so far: the nodes found but not yet expanded.

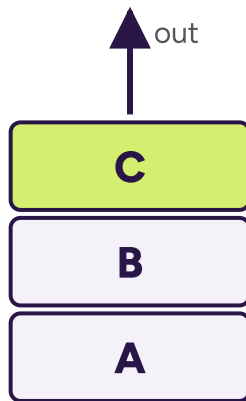
Why the explored set?

Leave out the three highlighted lines and the search can follow the endless branches in Figure 4: from Winchester to Stockbridge, back to Winchester, and so on for ever. The **explored set** remembers the states we have already looked at, so we never add them to the frontier again. Then each state gets at most one node, and in a state space of finite size the loop always stops.

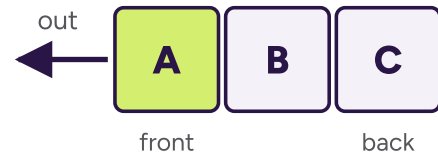
Going further: Russell & Norvig call a search that checks for repeated states in this way a **graph search**, and one that does not a **tree-like search**.

Which node comes out next?

The loop says “remove a node from the frontier”, but which one? There are two simple choices.



Stack: last in, first out, like a pile of plates. C was added last, so C comes out first. This gives **depth-first search**.



Queue: first in, first out, like a queue at a café. A was added first, so A comes out first. This gives **breadth-first search**.

KEY IDEA

This one choice, stack or queue, is the **only** difference between depth-first and breadth-first search. The rest of the loop is exactly the same. The names come from the search tree: depth-first follows one branch down the tree as far as it goes, and breadth-first finishes each level before starting the next.

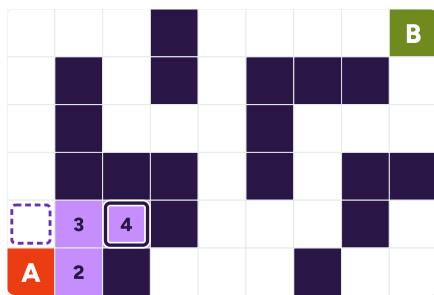
3 Depth-first search

Depth-first search (DFS) uses a stack, so it always takes the **newest** node in the frontier. In practice, it goes as deep as it can down one route, and only turns back when it reaches a dead end.

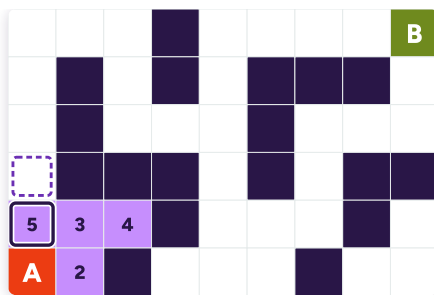
An everyday picture: you are looking for your keys. With depth-first search, you empty every pocket of your jeans before you look anywhere else in the house.

Worked example: DFS on the Week 1 maze

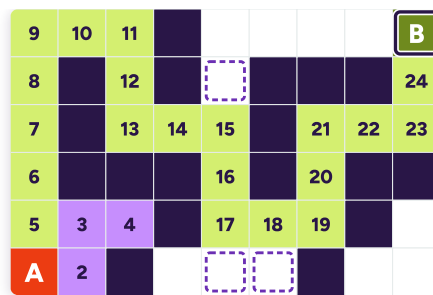
The pictures below use the same rules as the maze program in the practical (Section 6): neighbours are added in the order up, down, left, right. Lilac squares have been taken out of the frontier, with the step number when that happened. Dashed squares are waiting in the frontier. The ringed square is the one just taken out.



After step 4. At step 1, A had two open neighbours: up and right. Right was added last, so DFS took it (steps 2 to 4) and hit a dead end. The square above A has been waiting since step 1.



After step 5. The dead end added nothing new, so the newest square left is the one above A. DFS takes it and starts up the left-hand side.



Finished (step 25). At each later fork DFS again took the newest square. It explored 25 squares and found a **21-step** path (green). Three squares were never looked at.

waiting in the frontier explored (step number) path

In Week 1 you probably found the 17-step path along the top. DFS did not: it stops at the **first** path it finds, and that path went through the middle.

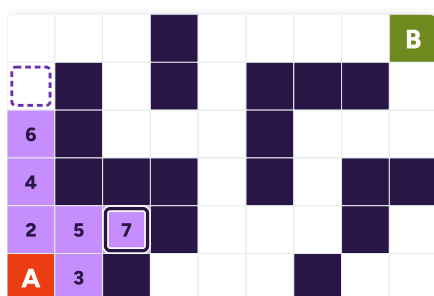
- ✓ If it happens to pick the right way first, DFS can be very fast.
- ✗ The path it finds may not be the shortest one.
- ✗ At worst, it explores almost every state before it finds the goal.

4 Breadth-first search

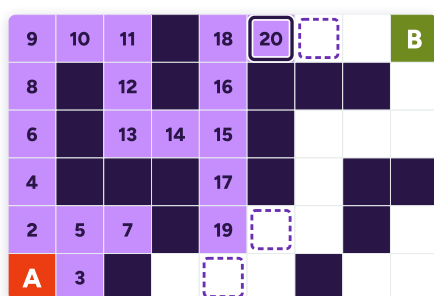
Breadth-first search (BFS) uses a queue, so it always takes the **oldest** node in the frontier. It takes one step in every direction, then two steps, then three, like a ripple spreading out from the start.

The keys again: with breadth-first search you check one pocket, then one drawer, then the table, and only after one quick look everywhere do you go back for the second pocket.

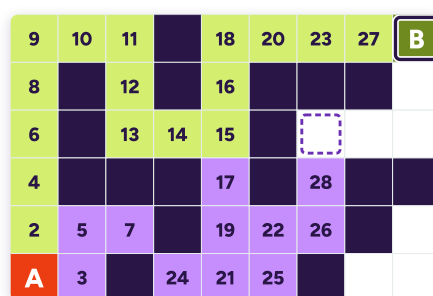
Worked example: BFS on the same maze



After step 7. BFS took turns between the two branches next to A: up (2), right (3), up (4), right (5), and so on. The right-hand branch has just reached its dead end (7).



After step 20. After the long corridor, the maze splits at step 15. Now BFS grows both routes together, one square each in turn: 16 (top), 17 (lower), 18 (top), 19 (lower), 20 (top).



Finished (step 29). The top route reached B first. BFS explored 29 squares, more than DFS, but its path is **17 steps**: the shortest, the same as the Week 1 answer.

waiting in the frontier explored (step number) path

- ✓ BFS always finds the shortest path, when every step costs the same.
- ✗ It often explores many states, because it looks everywhere nearby first.
- ✗ The frontier can grow very large, so it needs a lot of memory.

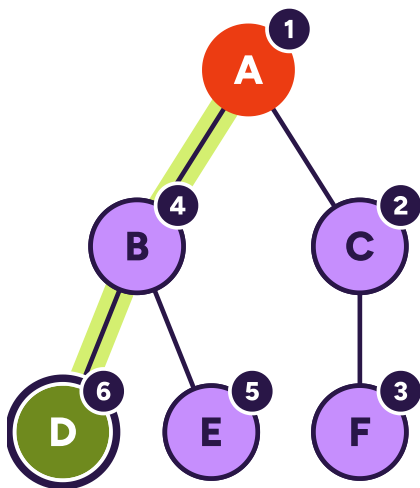
WHY “WHEN EVERY STEP COSTS THE SAME”?

BFS finds the path with the fewest *steps*. In a maze every step costs the same, so fewest steps means shortest. On the road map in Figure 2, steps have different costs. All three routes use two roads, so BFS returns the first one it finds: Winchester adds Andover, Stockbridge and Romsey to the queue in that order, Andover comes out first and adds Salisbury, and BFS returns the route via **Andover (55 minutes)**, not the 50-minute route via Stockbridge. Next week’s A* search takes the costs into account.

Back to Week 1: if your maze pseudocode always took the *newest* square, you wrote depth-first search. If it took the *oldest*, you wrote breadth-first search. If it took the square *closest to B*, you wrote greedy best-first search, which we meet in Week 3.

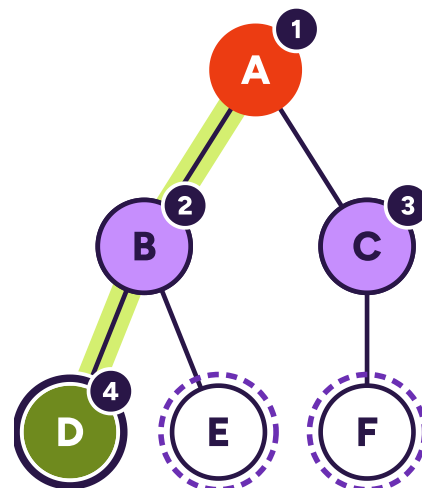
5 DFS and BFS side by side

Here is the search loop from Section 2 (remove a node from the frontier, check it, add it to explored, expand it) run on a small tree instead of a maze, once with a stack and once with a queue. The search starts at A and the goal is D. When a node is expanded, its children join the right-hand end of the frontier, left child first. A stack takes from the right-hand end; a queue takes from the left-hand end.



DFS (stack): badges show the order nodes were taken out.

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 (queue): E and F are still waiting when it finishes.

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

The two searches split at step 2: DFS takes C (added last) and BFS takes B (added first). Both return the path $A \rightarrow B \rightarrow D$, but BFS took out 4 nodes and DFS took out 6, because DFS went down the C branch first. If D had been under C instead, DFS would have been the lucky one.

The table below sums up the two searches. Its last two rows use two more mazes, `maze2.txt` and `maze3.txt`, which come with the maze program in the practical (Section 6).

	Depth-first search	Breadth-first search
Frontier	stack: last in, first out	queue: first in, first out
Explores	one route as deep as it can, then backs up	everything one step away, then two steps, ...
Finds a path if there is one?	yes (with an explored set)	yes
Shortest path?	not guaranteed	yes, when every step costs the same
Week 1 maze	25 squares explored, 21-step path	29 squares explored, 17-step path
<code>maze2.txt</code>	194 squares explored, 30-step path	77 squares explored, 30-step path
<code>maze3.txt</code>	17 squares explored, 16-step path	6 squares explored, 4-step path

KEY IDEA

Neither search is always faster: on the Week 1 maze DFS explored fewer squares, on `maze2` and `maze3` BFS did. Only BFS **guarantees** the shortest path. DFS and BFS are called **uninformed** searches, because they use no knowledge about where the goal is. Week 3 adds that knowledge.

6 In the code: `maze.py`

The practical uses `week02-search/maze/maze.py`. Each part of the loop above is in the code.

```
class Node():
    def __init__(self, state, parent, action):
        self.state = state          # where we are, e.g. (row, column)
        self.parent = parent        # the node we came from
        self.action = action        # "up", "down", "left" or "right"
```

There is no path cost in this `Node`, because every step in a maze costs the same (1). The two frontiers differ in one method only:

```
# StackFrontier (DFS): take the last
def remove(self):
    node = self.frontier[-1]
    self.frontier = self.frontier[:-1]
    return node
```

```
# QueueFrontier (BFS): take the first
def remove(self):
    node = self.frontier[0]
    self.frontier = self.frontier[1:]
    return node
```

In Python, `[-1]` is the last item of a list and `[0]` is the first. The `solve` method is the loop from Section 2: it removes a node, checks for the goal, adds the node to `self.explored`, and adds each neighbour that is not already in the frontier or explored.

TRY IT IN THE PRACTICAL

- Run `python maze/maze.py maze/maze1.txt` and open `maze.png`.
- Change `frontier = StackFrontier()` to `frontier = QueueFrontier()` in `solve`. Compare the squares explored and the path length on `maze2.txt` and `maze3.txt`. The numbers should match the table in Section 5.
- Draw a maze in a text file (`#` for walls, `A` for the start, `B` for the goal) where DFS finds a much longer path than BFS.

7 Adversarial search

In a maze, nobody fights back. In a game, an **opponent** chooses every other move, and wants the opposite result to you. Searching in this setting is called **adversarial search**. The classic example is tic-tac-toe (noughts and crosses).

What kind of game?

This week we look at the simplest kind of game. Tic-tac-toe and chess are both of this kind:

- **two players**, who **take turns**;
- **fully observable**: both players can see the whole board. Game theorists call this **perfect information**;
- **deterministic**: there are no dice or shuffled cards, so a move always has the same result;
- **zero-sum**: what is good for one player is equally bad for the other.

You met “fully observable” and “deterministic” in Week 1, as two of the questions we ask about an environment. In a game, a position on the board is a **state** and a move is an **action**, so the words from Section 1 still work.

ZERO-SUM GAME

A game where one player's gain is exactly the other player's loss, so there is no win-win result. In tic-tac-toe a win for **X** (+1) is a loss for **O** (-1): the two scores always add up to zero.

Games can be sorted with two questions. Does chance play a part? And can each player see everything?

	Deterministic (no chance)	Chance (dice, shuffled cards)
Perfect information (you can see everything)	chess, draughts (checkers), Go, Othello	backgammon, Monopoly
Imperfect information (something is hidden)	Battleships; blind tic-tac-toe, where you cannot see your opponent's marks	bridge, poker, Scrabble

Table 1. Four kinds of two-player game, sorted by the two questions. The green box is the only kind this module covers.

KEY IDEA

In this module we look only at the green box of Table 1: **deterministic games with perfect information**, such as tic-tac-toe and chess. Everything that follows, minimax and alpha-beta pruning, is for these games.

The other three boxes need more than search. With dice, a program has to weigh up what is likely to happen; with hidden cards, it has to guess what the opponent holds. Both need probability, which is the subject of Week 5, but we do not come back to games there.

Describing a game

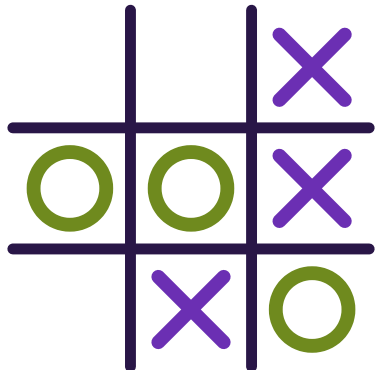


Figure 5. A position with X to move.

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

Compare this with a maze: we still have an initial state, actions and a transition model. The goal test becomes **TERMINAL**, and the path cost becomes **UTILITY**, because what matters is how the game ends, not how long it takes. X is called **MAX**, because X wants the score as high as possible. O is **MIN**, because O wants it as low as possible.

Russell and Norvig use slightly different names for three of these. They write **TO-MOVE(s)** instead of **PLAYER(s)**, and **IS-TERMINAL(s)** instead of **TERMINAL(s)**. Their utility function, **UTILITY(s, p)**, has a second input: the player **p** whose score we want. In a zero-sum game one score is enough, because the other player's score is the same number with the sign changed.

In Figure 5, **X** has three empty squares to choose from. Top-middle looks tempting, because it gives **X** two in the top row. But **O** has the centre and the bottom-right, so **O** is threatening the diagonal through top-left. A program finds the right move by looking ahead at the opponent's replies, which is what minimax does (next section).

The game tree

Start from the initial state and draw every possible move, then every possible reply to each, and so on. The result is a **game tree**: the search tree (Section 2) of the game, followed all the way to the end of every game. The levels take turns. **X** (MAX) moves at the top, **O** (MIN) at the next level down, then **X** again, until the game ends. Each finished game at the bottom has its utility: +1, 0 or -1. Figure 6 shows the top of the tree for tic-tac-toe.

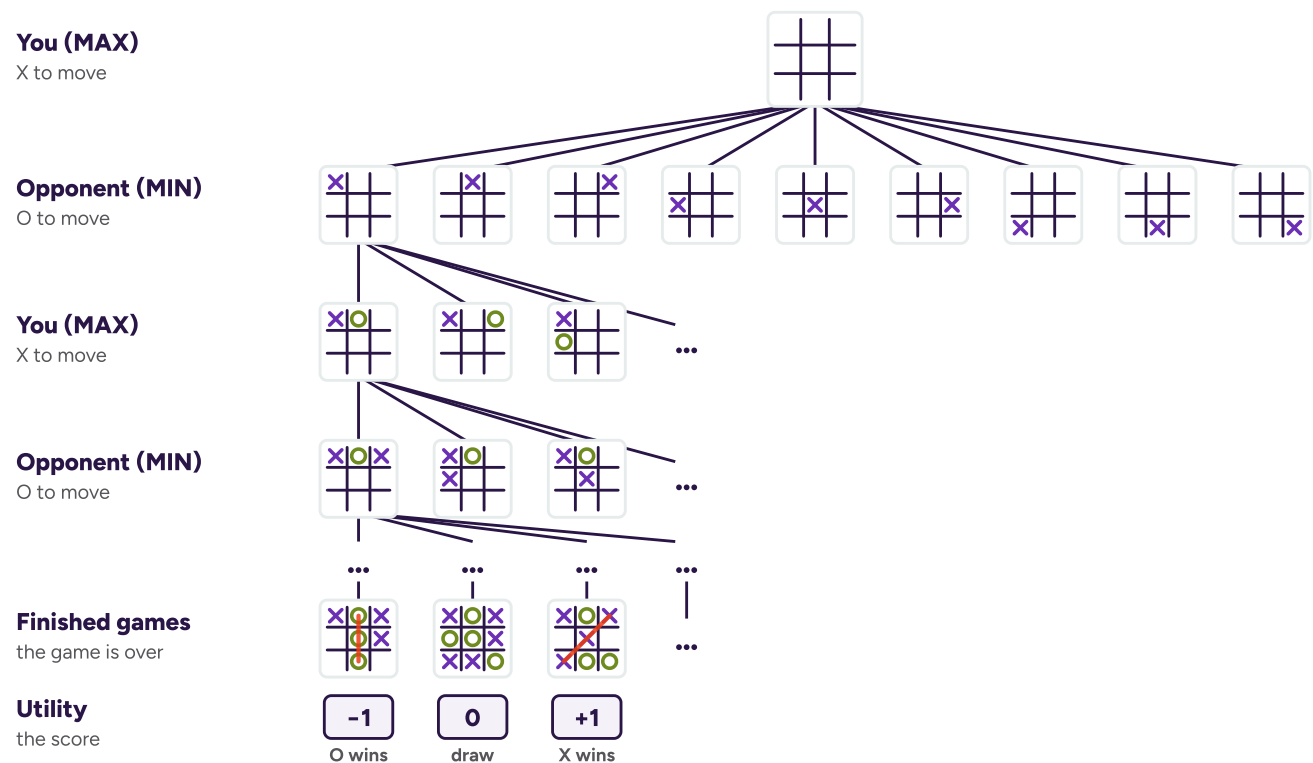


Figure 6. The game tree for tic-tac-toe, after Russell and Norvig. The top row shows all nine first moves. Below that only the left-hand branch is followed, and the dots stand for the parts left out. The bottom row shows three ways the game can end, with their utilities.

To read it, follow the left-hand edge: **X** plays top-left, **O** replies top-middle, **X** plays top-right, and after a few more moves the game ends. The three finished games at the bottom all start that way. In the first, **O** has completed the middle column (-1). The second is a full board with no line, a draw (0). In the third, **X** has completed the diagonal from top-right to bottom-left (+1).

Even this small game has a big tree. On the empty board **X** has 9 squares to choose from, then **O** has 8, then **X** has 7, and so on. That gives at most $9 \times 8 \times 7 \times \dots \times 1 = 362,880$ paths from the top to the bottom. There are fewer real games (255,168), because many games end before the board is full. Section 8 looks at a small part of this tree, starting from the position in Figure 5.

PLY

A **ply** (rhymes with “fly”) is one move by one player. Chess players often say “a move” for a turn by each side, so a single player’s turn is called a ply to avoid confusion. Down the left-hand edge of Figure 6, **X**’s first move is one ply, **O**’s reply is the second and **X**’s next move the third.

A game small enough to draw in full

Tic-tac-toe's tree does not fit on a page, so here is one that does. In this tiny game of **Nim** there is one pile of 3 counters. On your turn you take 1 or 2 counters. Whoever takes the *last* counter loses. Figure 7 is the whole tree.

You (MAX)
to move

```

graph TD
    Root["3 disks"] -- "take 1" --> Node2["2 disks"]
    Root -- "take 2" --> Node1R["1 disk"]
    Node2 -- "take 1" --> Node1L["1 disk"]
    Node2 -- "take 2" --> NoneLeft2["none left"]
    Node1L -- "take 1" --> NoneLeft1["none left"]
    NoneLeft1 --- Loss["-1  
you lose"]
    NoneLeft2 --- Win2["+1  
you win"]
    Node1R -- "take 1" --> NoneLeftR["none left"]
    NoneLeftR --- Win1["+1  
you win"]
  
```

Nim can be described with the same six words as tic-tac-toe (the table beside Figure 5). The initial state is the pile of 3 counters, and any later state is the number of counters left. The players take turns. The actions are “take 1” and “take 2”, and the result of an action is a smaller pile. The game is over (**TERMINAL**) when no counters are left, and the utility is +1 if your opponent took the last one and -1 if you did. Because the whole tree is on the page, you can see every way the game can go: there are only three. In Section 8 you will be able to work out the best first move from this tree, and there is a question on it at the end. A bigger version of Nim comes back in Week 7, as the game a program learns to play by playing against itself.

A game is not a maze

In a maze, the answer is a fixed list of moves: up, up, left, and so on. In a game you cannot write that list in advance, because you do not know what your opponent will do. So the answer to a game is a **strategy** instead.

In a maze, the answer is a fixed list of moves: up, up, left, and so on. In a game you cannot write that list in advance, because you do not know what your opponent will do. So the answer to a game is a **strategy** instead.

STRATEGY

A rule that says which move to make in every position you might reach, whatever the opponent does. Minimax (next section) finds the best strategy, assuming the opponent plays as well as possible too.

Games also have time limits. A chess player has a clock, and a program usually cannot search all the way to the end of the game before its time runs out. So it has to make a good guess about how good a position is. Section 10 shows how.

8 Minimax

Minimax assumes the opponent plays perfectly. MAX imagines each of its moves, then imagines MIN's best reply to it, and so on to the end of the game. Then it works back up: MIN always picks the lowest score, and MAX the highest.

MINIMAX VALUE

The score MAX ends up with from a state if both players play perfectly from there to the end. It is the best score MAX can be sure of, whatever MIN does. It is worked out from the bottom of the tree up:

- if the game is over, it is the utility of the finished game;
- if it is MAX's turn, it is the **highest** minimax value of the states MAX can move to;
- if it is MIN's turn, it is the **lowest** minimax value of the states MIN can move to.

MAX then plays the move that leads to the highest minimax value.

Example: the board in Figure 5

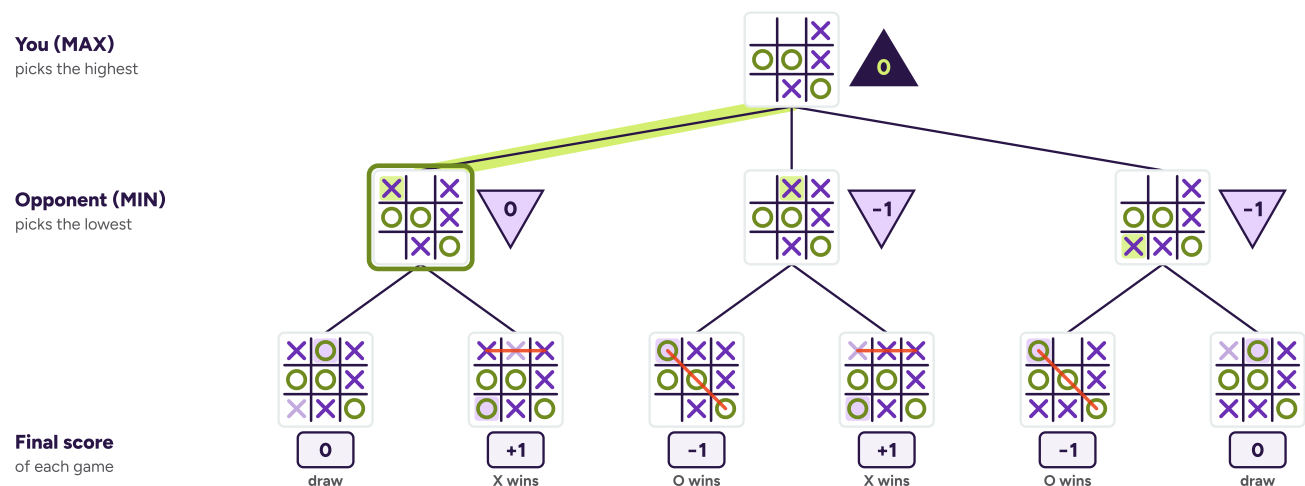


Figure 8. Minimax on the board in Figure 5. Middle row: X's three possible moves (the new X's square is tinted green). Bottom row: O's two replies to each (the new O's square is tinted purple). Where X's last square is forced, it is drawn faint. A red line marks a win.

WORKED EXAMPLE

1. **Score the finished games:** X wins +1, draw 0, O wins -1.

2. **O picks the lowest** under each of **X's** moves. After top-middle or bottom-left, **O** takes top-left and wins: -1. After the block at top-left, **O's** best reply leads to a draw: 0 (its other reply would let **X** complete the top row).
3. **X picks the highest** of -1, -1 and 0: the block at top-left, worth 0. The attack at top-middle looked good, but it ignored **O's** threat.

The same idea with numbers

Real games have many more moves, so we draw the tree with numbers instead of boards. Figure 9 has the same shape: MAX chooses a move, MIN replies, and the bottom row holds the scores of finished games. The scores can be any numbers, not just +1, 0 and -1: higher is better for MAX.

You (MAX)

picks the highest

Opponent (MIN)

picks the lowest

Final score

of each game

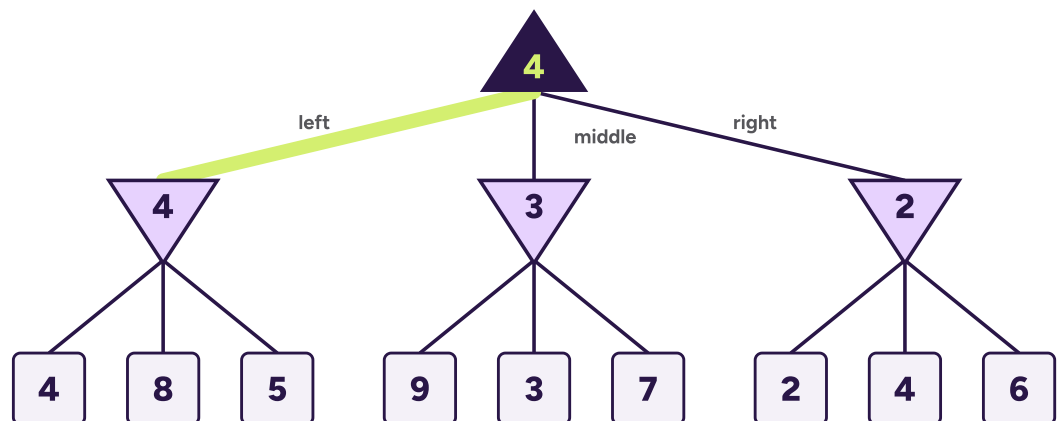


Figure 9. MAX (the dark triangle) has three moves. After each one, MIN (the light triangles) has three replies. The numbers at the bottom are the scores of the finished games.

WORKED EXAMPLE

1. Work from the bottom up. Each MIN node takes the **lowest** of its children: left $\min(4, 8, 5) = 4$; middle $\min(9, 3, 7) = 3$; right $\min(2, 4, 6) = 2$.
2. MAX takes the **highest** of those: $\max(4, 3, 2) = 4$. MAX plays the left move.

The middle move has a 9 under it, but MIN will never let MAX get it: after the middle move, MIN replies with the 3. Minimax chooses the move with the **best worst case**.

In pseudocode, two functions call each other:

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

Given a state **s**, MAX picks the action **a** with the highest **MIN-VALUE(RESULT(s, a))**, and MIN picks the action with the lowest **MAX-VALUE(RESULT(s, a))**. Why start at $-\infty$ in **MAX-VALUE**? So that the first real score is always bigger and replaces it. **MIN-VALUE** starts at $+\infty$ for the same reason. A function that calls itself, here through the other function, is called **recursive**: you need to be able to read it, not write it. Russell and Norvig add a third function for that first step, which returns MAX's best *action*, not just its value.

Tracing it once: for the left MIN node in Figure 9, **MIN-VALUE** starts with $v = +\infty$, then $v = \min(+\infty, 4) = 4$, then $\min(4, 8) = 4$, then $\min(4, 5) = 4$, and returns 4.

How good is minimax?

Russell and Norvig judge every search algorithm with the same four questions. You have met two of them already: the table in Section 5 asked whether DFS and BFS find a path if there is one (is the search **complete**?) and whether that path is the shortest (is it **optimal**?). The other two ask how much time and how much memory the search needs. Two numbers come up in those answers. Suppose every position has about the same number of legal moves, and call that number **b**, the **branching factor**. Suppose a game lasts about **m** ply. Here are the answers for minimax.

Question	Answer for minimax
Complete? Does it always find an answer?	Yes, as long as every game ends (the tree is finite). Chess has rules to make sure games end: for example, a player can claim a draw when the same position has come up three times.
Optimal? Does it find the best move?	Yes, against an opponent who also plays perfectly. See below for a weaker opponent.
Time: how many positions does it look at?	Every position in the tree: about b^m (worked out below).
Space: how much does it keep in memory?	Only about $b \times m$ positions, because it searches depth-first.

What if the opponent is not perfect? Minimax still gets at least its minimax value, because it has planned for the worst. But it may miss a chance to do better. Suppose that in Figure 9, MIN is careless and always plays its left-hand reply. Then the left move would give MAX 4, the middle 9 and the right 2, so the middle move would be best. Minimax still plays left, because it assumes MIN will spot the 3. Playing safe is the right choice when you know nothing about your opponent.

How much work is it? MAX has b moves, MIN has b replies to each, MAX has b replies to each of those, and so on, m times: $b \times b \times \dots \times b = b^m$ games in all. Computer scientists write the time as $O(b^m)$, read "order b to the power m ": it grows like b^m .

Memory is much smaller: about $b \times m$ positions, written $O(bm)$. That is because minimax searches depth-first, like DFS in Section 3. **MAX-VALUE** calls **MIN-VALUE**, which calls **MAX-VALUE**, all the way down one line of play before it tries the next. So it only has to remember the positions on that one line, plus the other moves waiting at each step.

EXAMPLE: MINIMAX FOR CHESS

In chess, a position has about 35 legal moves ($b \approx 35$), and a game of a sensible length is about 100 ply ($m \approx 100$, so 50 moves each). Minimax would have to look at about 35^{100} games, which is about 10^{154} : a 1 followed by 154 zeros. There are only about 10^{80} atoms in the observable universe. Working out perfect chess this way is completely out of reach.

But does minimax really need to look at every game? No: the next section shows how to skip some of them.

9 Alpha-beta pruning

Minimax looks at every possible game. **Alpha-beta pruning** gets the same answer while skipping parts of the tree that cannot change it.

You (MAX)

picks the highest

Opponent (MIN)

picks the lowest

Final score

of each game

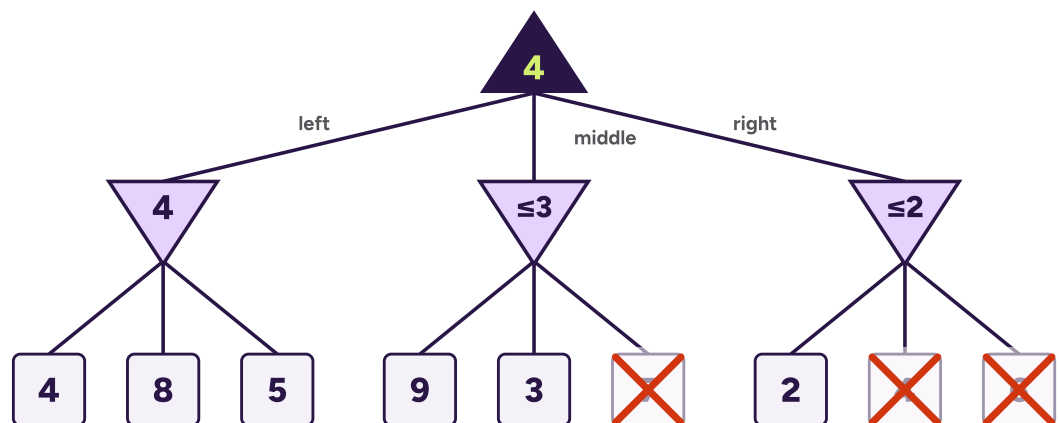


Figure 10. The same tree as Figure 9. The crossed-out leaves are never looked at.

WORKED EXAMPLE

1. The left move is worked out first, as before: it is worth 4 to MAX.
2. Middle move: the first leaf is 9, the second is 3. MIN can already hold MAX to 3 here, so this move is worth **at most 3** to MAX. MAX already has a move worth 4, so it will never choose this one. Whatever the 7 is (10 or -10), it cannot change that. Skip it.
3. Right move: the first leaf is 2, so this move is worth at most 2. Skip the 4 and the 6.

MAX still plays the left move, worth 4, but only 6 of the 9 leaves were looked at.

Pruning never changes the answer; it only saves work. In real games the saving is much bigger than here, because whole branches with thousands of games below them are skipped, not single leaves.

Alpha and beta

To know when it can skip a branch, the algorithm carries two numbers down the tree. They give it its name.

ALPHA (A) AND BETA (B)

Alpha is the best score MAX can already be sure of, from the moves it has looked at so far on the way down. MAX will never settle for less. **Beta** is the best score MIN can already be sure of: MIN will never allow more. At the start nothing has been looked at, so alpha is $-\infty$ and beta is $+\infty$.

The rule: if a branch can give MAX no more than alpha, MAX will avoid it, so stop looking at it. In the same way, if a branch can give MIN no less than beta, MIN will avoid it.

Reading Figure 10 with alpha: after the left move, alpha = 4, because MAX is sure of 4. Under the middle move, MIN finds a 3, so that move is worth at most 3. That is less than alpha, so MAX will never choose it, and the 7 is skipped. Under the right move, the first leaf is 2, also less than alpha, so the 4 and the 6 are skipped.

A second example: the textbook's tree

Russell and Norvig use the tree in Figure 11. It has the same shape as Figure 9, with different scores. Work out the minimax answer yourself first, then read the trace below.

You (MAX)

picks the highest

Opponent (MIN)

picks the lowest

Final score

of each game

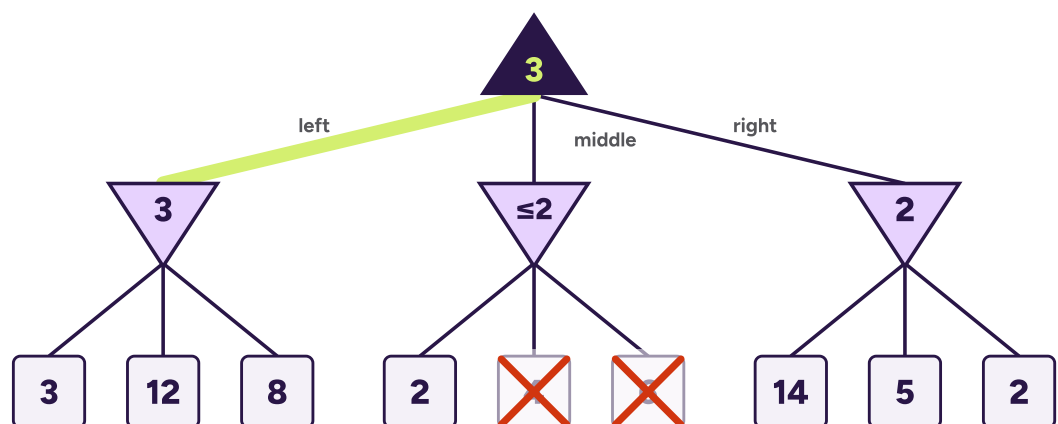


Figure 11. The tree Russell and Norvig use for minimax and alpha-beta (they call the three moves a_1 , a_2 and a_3). MAX plays the left move, worth 3. The crossed-out leaves are never looked at.

WORKED EXAMPLE: ALPHA-BETA, ONE LEAF AT A TIME

Minimax first: the MIN nodes are worth $\min(3, 12, 8) = 3$, $\min(2, 4, 6) = 2$ and $\min(14, 5, 2) = 2$, so MAX plays left, worth 3. Now alpha-beta, which looks at the leaves from left to right:

Leaf	What we now know	So...
3	the left move is worth at most 3	keep looking
12	still at most 3	keep looking
8	the left move is worth exactly 3	alpha = 3: MAX is sure of 3
2	the middle move is worth at most 2	2 is less than alpha: skip the 4 and the 6
14	the right move is worth at most 14	could still beat 3: keep looking
5	the right move is worth at most 5	could still beat 3: keep looking
2	the right move is worth exactly 2	less than 3

Leaf	What we now know	So...
	MAX takes the highest: 3	play the left move

Alpha-beta looked at 7 of the 9 leaves and got the same answer as minimax.

KEY IDEA

The order matters. If the right-hand leaves had come in the order 2, 5, 14, alpha-beta would have stopped after the 2 and skipped two more leaves. Alpha-beta prunes most when it looks at the best moves first.

The alpha-beta algorithm

Alpha-beta is minimax with two extra inputs, alpha and beta, and two extra lines in each function (highlighted).

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

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

How to read the two new lines in **MAX-VALUE**:

- **if $v \geq \beta$: return v** . MAX has found a move worth at least v here. But MIN, higher up the tree, already has a way to hold MAX to β . If v is at least β , MIN will never let the game reach this state, so there is no point looking at MAX's other moves here.
- **$\alpha = \text{MAX}(\alpha, v)$** . If this is the best MAX has found so far, raise alpha, so that the levels below know what MAX is already sure of.

MIN-VALUE is the mirror image. It stops when v is at most α , because MAX already has something at least as good elsewhere, and it lowers beta. The search starts with **MAX-VALUE(initial state, $-\infty$, $+\infty$)**. As with minimax, a third function makes this first call and returns the best action.

How much does alpha-beta save?

- **The answer never changes.** Alpha-beta only skips branches that cannot affect the choice.
- **Order matters.** It prunes most when the best moves are looked at first, as Figure 11 showed. Chess programs try the most promising moves first, such as captures.

- **With perfect ordering** it looks at about $b^{m/2}$ games instead of b^m . In practice this means it can search about **twice as deep** in the same time: if minimax has time to look 4 ply ahead, alpha-beta can look about 8 ply ahead.
- **It is still not enough for chess.** With $b \approx 35$ and $m \approx 100$, perfect ordering cuts 35^{100} games to 35^{50} , which is about 10^{77} . That is still far too many.

Alpha-beta is a simple example of **metareasoning**: reasoning about which calculations are worth doing. Before doing a calculation, the program asks whether its result could change the decision. If it could not, the calculation is skipped.

10 Too big to search to the end

There are 255,168 possible games of tic-tac-toe, so a computer can play them all out in a moment. Chess is different. Section 8 showed that minimax would need about 10^{154} games of a sensible length, and Section 9 that even alpha-beta with perfect ordering would need about 10^{77} . If you count every legal game of chess, including very long ones, there are roughly 10^{29000} : a 1 followed by 29,000 zeros. No computer will ever play out every game of chess, so a chess program has to stop early and guess.

DEPTH-LIMITED MINIMAX

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

EVALUATION FUNCTION

A function that estimates how good a position is without playing to the end. Depth-limited minimax uses it to score the positions where it stops.

EXAMPLE: A SIMPLE CHESS EVALUATION FUNCTION

Give each piece points: pawn 1, knight 3, bishop 3, rook 5, queen 9. The score of a position is White's points minus Black's points. If White has an extra rook, the score is about +5. Real chess programs add much more (which squares are attacked, how safe the king is), but the idea is the same. The program can only be as good as its evaluation function.

Depth-limited minimax gives up the guarantee of the perfect move: the evaluation function is only a guess, so the move it picks is only as good as that guess. In return, it can answer within the time limit, and with alpha-beta it can look deeper in the same time.

11 Search in the real world

- **Sat-navs and map apps:** road junctions are states and roads are actions.
- **Game engines:** chess and Go programs use adversarial search, with an evaluation function. IBM's Deep Blue beat the world chess champion, Garry Kasparov, in 1997 with alpha-beta search, a carefully built evaluation function and very fast hardware. Go has about 250 legal moves in a position, too many for

alpha-beta, so AlphaGo (2016) used a different search that plays out many sample games, guided by neural networks (Week 8).

- **Warehouse robots:** plan paths that avoid shelves and other robots.
- **Video games:** characters find their way round the map.
- **Puzzle solvers:** Rubik's cube and sliding-tile puzzles.

In the seminar you describe a sat-nav, a chess engine and a warehouse robot as search problems, and discuss who decides what the “best” route is. Remember Figure 2: the answer depends on the path cost, and someone has to choose it.

Where these ideas came from

Most of this week's ideas about games are older than the name “artificial intelligence” (1956, Week 1).

Year	Who	Idea
1846	Charles Babbage	A machine could play a game by working through the possible lines of play.
1912	Ernst Zermelo	Perfect play in a game like chess can be worked out, in principle.
1944	John von Neumann and Oskar Morgenstern	Game theory, including minimax. Economists still use it.
1945–50	Konrad Zuse, Norbert Wiener, Claude Shannon	Look a fixed number of moves ahead, then estimate the score: depth-limited minimax with an evaluation function.
1951	Alan Turing	One of the first chess programs. No computer could run it yet, so Turing worked through its moves by hand.
1952–57	Arthur Samuel	A draughts (checkers) program that improved its own evaluation function by playing games: an early example of machine learning (Week 7).
1956	John McCarthy	Pruning, so that a program can search deeper: the idea behind alpha-beta.

Check yourself

Try these without looking back. The answers are at the end.

1. In a 4×4 sliding puzzle, how many actions are there when the gap is in a corner? Why does **ACTIONS** take the state as its input?
2. Describe a sat-nav as a search problem: give its initial state, actions, transition model, goal test and path cost.
3. What could go wrong if the search loop had no explored set?
4. What is the difference between the state space and the search tree? In Figure 4, why does Salisbury appear three times?

5. The frontier holds P, Q and R, added in that order. Which node does DFS take out next? Which does BFS take out?
6. On the tree in Section 5, make **F** the goal instead of D. In what order does each search take nodes out of the frontier?
7. Why does BFS not find the 50-minute route on the road map in Figure 2?
8. Put each game in Table 1: Connect Four, Ludo, Battleships and poker. Which of them could plain minimax play?
9. Use minimax (Section 8) on the Nim tree in Figure 7. What is the minimax value of each state, and should you take 1 or 2 counters first?
10. In Figure 9, change the 3 in the middle branch to a 6. Which move does MAX choose now, and what is it worth?
11. With that change, can alpha-beta pruning still skip the 7? Can it still skip anything?
12. In Figure 11, suppose the leaves under the right move came in the order 2, 5, 14. How many of the nine leaves does alpha-beta look at now?
13. A game has 10 legal moves in every position and always lasts 20 ply. About how many games does minimax look at? About how many does alpha-beta look at with perfect move ordering?
14. Why can a chess program not use plain minimax, and what does it do instead?

Key terms

Search problem	Find a sequence of actions from an initial state to a goal.
Agent	Something that perceives its environment and acts on it.
State / initial state	One configuration of the agent and its environment / the one the search starts from.
Actions, <code>ACTIONS(s)</code>	The choices available in state <code>s</code> .
Transition model, <code>RESULT(s, a)</code>	The state you reach by doing <code>a</code> in <code>s</code> .
State space	All states reachable from the initial state; drawn as a graph.
Goal test	Checks whether a state is a goal.
Path cost	A number for each path, such as time or distance.
Solution / optimal solution	Actions from the initial state to a goal / a solution with the lowest path cost.
Node	A record of a state, its parent, the action that reached it and the path cost.
Search tree	The tree of nodes a search builds as it runs; one node per path, so a state can appear more than once.
Frontier	The nodes found but not yet looked at.
Explored set	The states already looked at; stops the search going round in circles.

Search algorithm	A method that returns a solution to a search problem, or reports that there is none (DFS, BFS, and in Week 3 greedy and A*).
Stack / queue	Last in, first out / first in, first out.
Depth-first search	Takes the newest node (stack); goes deep first; path may not be shortest.
Breadth-first search	Takes the oldest node (queue); spreads out evenly; shortest path when every step costs the same.
Uninformed search	A search that uses no knowledge about where the goal is, such as DFS and BFS.
Adversarial search	Search when an opponent chooses some of the moves.
Zero-sum game	One player's gain is the other player's loss; there is no win-win result.
Perfect information	Both players can see the whole game (it is fully observable).
Game tree / ply	Every possible move and reply, drawn as a tree / one move by one player.
Strategy	Which move to make in every position you might reach, whatever the opponent does.
Terminal state / utility	A state where the game is over / its score (for tic-tac-toe: +1, 0 or -1).
Minimax	MAX picks the highest value and MIN the lowest, working up from the end of the game.
Minimax value	The score MAX gets from a state if both players play perfectly from there.
Branching factor (b)	The number of legal moves in a position. Minimax looks at about b^m games, for games m ply long.
Alpha-beta pruning	Skips branches that cannot change the minimax answer.
Alpha (α) / beta (β)	The best score MAX / MIN can already be sure of.
Move ordering	Looking at the most promising moves first, so that alpha-beta can prune more.
Depth-limited minimax	Minimax that stops a fixed number of moves ahead.
Evaluation function	Estimates how good a position is without playing to the end.

Reading and next week

- Russell & Norvig, *Artificial Intelligence: A Modern Approach* (4th ed.): sections 3.1 to 3.4 (search problems, uninformed search) and 5.1 to 5.3 (games, minimax, alpha-beta). In the Global Edition, games are in Chapter 6.
- Poole & Mackworth, *Artificial Intelligence: Foundations of Computational Agents* (3rd ed., free at artint.info): sections 3.1 to 3.5 and 14.3.

Next week: smarter search. A *heuristic* estimates how far a state is from the goal, and greedy best-first search and A* search use it to look in the right direction first. Then **logic**: writing knowledge down precisely, and letting a computer work out what follows from it.

Answers to Check yourself

- Two: only the two tiles next to a corner can slide into it. In the middle of an edge there are 3 actions, and with the gap in one of the four middle squares there are 4. **ACTIONS** takes the state as input because the moves allowed depend on where the gap is.
- Initial state: the car's current location. Actions: the roads you can drive along from the current junction. Transition model: driving along a road takes you to the junction at its end. Goal test: are you at the destination? Path cost: total travel time (or distance, or fuel).
- The search could go round in circles for ever, adding the same states again and again (for example Winchester → Stockbridge → Winchester → ...). Even when it does finish, it would waste time looking at the same states many times.
- The state space has one box for each state; the search tree has one node for each path the search has found, so a state can appear in it many times. Salisbury appears three times because there are three routes to it (via Andover, Stockbridge and Romsey), each with its own parent and path cost (55, 50 and 60 minutes).
- DFS takes **R** (the newest, from the stack). BFS takes **P** (the oldest, from the queue).
- DFS: A, C, F (goal). After A the frontier is B C; DFS takes C, adds F, then takes F: lucky, only 3 nodes. BFS: A, B, C, D, E, F (goal), all 6 nodes, because F is on the deepest level and BFS finishes each level first.
- BFS counts steps, not minutes. All three routes use two roads, so BFS returns the first route it completes. Andover was added to the queue first, so it comes out first and adds Salisbury with Andover as its parent. BFS returns Winchester → Andover → Salisbury (55 minutes). To find the 50-minute route, the search has to use the path cost, which A* does.
- Connect Four: deterministic, perfect information. Ludo: chance (dice), perfect information. Battleships: deterministic, imperfect information (the ships are hidden). Poker: chance and imperfect information. Only **Connect Four** is in the green box of Table 1, so it is the only one plain minimax could play.
- Work up from the bottom. With 1 counter and you to move, you must take the last counter: -1. With 2 counters and your opponent to move, they choose $\min(-1, +1) = -1$: they take 1 and leave you the last one. With 1 counter and your opponent to move, they must take the last: +1. At the start, $\max(-1, +1) = +1$: **take 2**, leaving 1, so your opponent has to take the last counter.
- The middle MIN node becomes $\min(9, 6, 7) = 6$. MAX now takes $\max(4, 6, 2) = 6$: the **middle** move, worth 6.
- No, the 7 cannot be skipped. After the 9 and the 6, the middle move is worth at most 6, and that is better than the 4 MAX already has, so MAX must look at the 7 to find the real value (6). But on the right, the first leaf is 2: that move is worth at most 2, which is worse than 6, so the 4 and the 6 on the right are still skipped.
- Five: all three on the left (3, 12, 8), the 2 in the middle, and the 2 on the right. After that 2, the right move is worth at most 2, which is less than $\alpha = 3$, so the 5 and the 14 are skipped.
- Minimax: $b^m = 10^{20}$, a hundred billion billion. Alpha-beta with perfect ordering: about $b^{m/2} = 10^{10}$, ten billion. That is still a lot, but it is ten billion times fewer.

14. There are far too many possible games (roughly 10^{29000}) to play every one out to the end. Instead it uses depth-limited minimax: it looks a few moves ahead and scores the positions it reaches with an evaluation function, such as counting the pieces.

COMP5001 Artificial Intelligence, University of Winchester, 2026/27 · Dr Sakib Anwar ·

Sakib.Anwar@winchester.ac.uk

Adapted from the lecture notes of *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. Shared under the same licence. Sections 7 to 11 also draw on Russell and Norvig, *Artificial Intelligence: A Modern Approach*, 4th ed., the chapter on adversarial search and games.