

AI Notes for Handwriting, W1 to W3

Topics: W1 Intelligent Agents, W2 Problem Formulation, W3 Search Methods.

Exam focus

- Environment properties: deterministic, stochastic, nondeterministic, single or multi agent. See W1 section 5.
- Problem formulation and total path cost. See W2 sections 2 and 3.
- A* search. See W3 section 6.
- Minimax. See W3 section 9.
- A medical case solved with search. See the last section, "Medical worked examples".

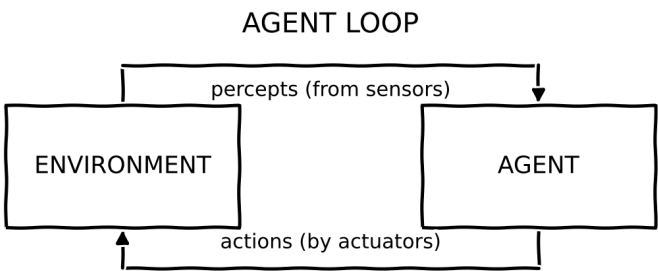
W1. Intelligent Agents

1. What is AI

- AI = machines that do tasks needing human intelligence.
 - Slide wording: a branch of computer science that simulates intelligent behaviour in computers. The capability of a machine to mimic intelligent human behaviour.
 - Google Translate: AI, because translating needs an understanding of language.
 - Calculator: not AI, because it only follows one fixed formula.
- 4 capabilities:
 - NLP = talk in human language. Siri understanding "set an alarm for 6" is NLP, because the input is ordinary speech.
 - Knowledge representation = store what it knows. Google Maps holding the road network is this, because the roads are stored facts it can look up.
 - Automated reasoning = draw new conclusions from what it knows. A medical expert system concluding "flu" from "fever + cough" is this, because nobody stored that answer for this patient.
 - Machine learning = adapt and find patterns. A spam filter is this, because it improves from the mails you mark as spam.

2. Agent

- Agent = anything that senses its environment through sensors and acts on it through actuators.
 - Sprinkler A reads soil moisture, then waters: agent, because its action depends on what it senses.
 - Sprinkler B runs on a timer: not an agent, because it senses nothing.
- Loop: environment -> percept -> agent -> action -> environment.



- Percept = what the agent senses right now. Vacuum reads [A, Dirty].
- Percept sequence = everything sensed so far. [A, Dirty], [A, Clean], [B, Dirty].
- Agent function = the rule that maps every percept sequence to an action. The vacuum table below is one, because it is a rule on paper and not yet code.
- Agent program = the code that runs that rule on a machine. REFLEX-VACUUM-AGENT in section 6 is one.
- Agent = architecture + program. The architecture is the machine with its sensors and actuators. A robot vacuum's body is the architecture, and its cleaning code is the program.
- Vacuum world: squares A and B, each clean or dirty. Percept [location, status]. Actions Left, Right, Suck.
 - [A, Clean] -> Right. [A, Dirty] -> Suck. [B, Clean] -> Left. [B, Dirty] -> Suck.

3. Rational agent

- Rational agent = for each percept sequence, picks the action expected to maximise its performance measure.
 - You look both ways, the road is empty, you cross, and a falling tree hits you: still rational, because the choice was the best one given what you could sense.
 - So rational is not all-knowing. It maximises the expected result, not the actual one.
- Slide wording: a rational agent does the right thing. Every entry in the table for the agent function is filled out correctly.
- The performance measure evaluates a sequence of environment states. For the vacuum: how many squares are clean over time.
- Rational depends on 4 things. For the vacuum:
 - Performance measure: squares cleaned.
 - Prior knowledge: there are two squares.
 - Actions: Left, Right, Suck.
 - Percept sequence: what it has sensed so far.

4. Specifying the Task Environment: PEAS

- Task environment = the problem to which a rational agent is the solution. For a vacuum robot it is the dirty room.
- P = Performance measure, E = Environment, A = Actuators, S = Sensors.
- Vacuum robot (from the slides):

- P: fast, cleanness, energy consumption.
 - E: floor, room layout, furniture.
 - A: display, wheel, sweeper, battery, switch.
 - S: ultrasonic, camera.
- Amazon Alexa (from the slides):
 - P: fast, information accuracy.
 - E: customer.
 - A: lights, motor, Wi-Fi controller.
 - S: sound sensor.
- Self-driving car (blank in the slides):
 - P: safe, fast, legal, comfortable trip.
 - E: roads, other traffic, pedestrians.
 - A: steering, accelerator, brake, signal, horn.
 - S: cameras, GPS, speedometer, radar.
- Medical diagnosis system:
 - P: healthy patient, correct diagnosis, low cost.
 - E: patient, hospital, staff.
 - A: screen that shows questions, tests, diagnoses and treatments.
 - S: typed-in symptoms, test results, the patient's answers.

5. Property of Task Environment (6)

- Fully Observable vs Partially Observable. Fully = the sensors give the complete state at every moment. Partially = the sensors give only part of it. Unobservable = the agent has no sensors.
 - Tic-tac-toe AI: fully, because it can see every square of the board.
 - Voice recognizer: fully, because the microphone delivers all the audio it has to judge.
 - Dirt-cleaner: partially, because it detects dirt only in its own square and cannot tell whether the other square is dirty.
 - Self-driving car: partially, because cameras cannot see around a corner or behind a truck.
 - Medical diagnosis: partially, because the system sees symptoms and test results, not the disease itself.
- Single Agent vs Multi Agent. Single Agent = only one agent. Multiple Agent = more than one agent.
 - Crossword solver: single, because it solves the puzzle by itself.
 - Chess player: multi (two agents), because the opponent also chooses moves.
 - Dirt-cleaner: single, because nothing else in the room makes decisions.
 - Voice recognizer: multi, because the speaker is another agent.
 - Self-driving car: multi, because of other drivers and pedestrians.
 - Medical diagnosis: single, because the disease does not choose moves to beat the system.
- Multi-agent comes in two types.
 - Competitive = one agent's gain is the other's loss. Chess is competitive, because my win is your loss.
 - Cooperative = the agents gain together. Cars avoiding a collision are cooperative, because every driver is better off with no crash.
- When does B count as an agent, and not just an object? When B's behaviour is best described as maximising a performance measure whose value depends on A's behaviour.
 - A chess opponent: agent, because it tries to win, and whether it wins depends on my moves.
 - Dirt on the floor: object, because it is not trying to achieve anything.
- Deterministic vs Stochastic. Deterministic = the next state is completely determined by the current state and the agent's action.
 - 8-puzzle: deterministic, because sliding a tile always gives the same board.
 - Chess: deterministic, because a move always puts the piece where you said.
- Stochastic = the next state is not fully determined. Chance is involved, and the chances can be given as probabilities.
 - Dirt-cleaner: stochastic, because suction can miss and new dirt can appear.
 - Voice recognizer: stochastic, because noise and accents make the same word sound different.
 - Self-driving car: stochastic, because it cannot predict what other drivers will do.
 - Medical: "this drug cures 80% of patients" is stochastic, because the outcome is uncertain and comes with a probability.
- Nondeterministic = actions are described by their possible outcomes, but no probabilities are attached.
 - Medical: "this drug may cure the patient or may cause a rash" is nondeterministic, because the outcomes are listed with no percentages.
- Slide statements to quote: a fully observable environment must be deterministic. If the environment is partially observable, it could be stochastic.
 - The slide table still lists the voice recognizer as fully observable and stochastic. If asked about a task in that table, answer with the table.
- Episodic vs Sequential. Episodic = the experience is split into atomic episodes, and the next episode does not depend on earlier actions. Sequential = the current decision could affect all future decisions.
 - Voice recognizer: episodic, because each sentence is recognised on its own.
 - Dirt-cleaner: sequential, because where it moves now decides what it can clean next.
 - Self-driving car: sequential, because braking late now leaves less room later.
 - Medical diagnosis: sequential, because the test ordered now decides which treatment is possible next.
- Static vs Dynamic. Static = the environment does not change while the agent is deciding, so the agent need not keep looking at the world or worry about time. Dynamic = the environment can change while the agent deliberates. If it has not decided yet, that counts as deciding to do nothing.
 - Crossword: static, because the grid does not change while you think.
 - Dirt-cleaner: dynamic, because dirt can fall while it decides.
 - Voice recognizer: dynamic, because the speaker keeps talking.
 - Self-driving car: dynamic, because traffic keeps moving.
 - Medical diagnosis: dynamic, because the patient's condition changes while the system decides.

- Continuous vs Discrete. Discrete = every possible state is known from a countable set of percepts and actions. Continuous = states and time sweep smoothly through a range of values.
 - Chess: discrete, because the moves and board states can be counted.
 - Dirt-cleaner: discrete, because there are 2 squares and 3 actions.
 - Taxi driving: continuous, because speed and location change smoothly over time.
 - Voice recognizer: continuous, because sound is a smooth wave.
 - Medical diagnosis: continuous, because temperature and blood pressure can take any value.
- Summary:
 - Dirt-cleaner: partial, single, stochastic, sequential, dynamic, discrete.
 - Voice recognizer: fully, multi, stochastic, episodic, dynamic, continuous.
 - Self-driving car (blank in the slides): partial, multi, stochastic, sequential, dynamic, continuous. This is the hardest combination.
 - Medical diagnosis: partial, single, stochastic, sequential, dynamic, continuous.

6. Structure of Agents: agent types

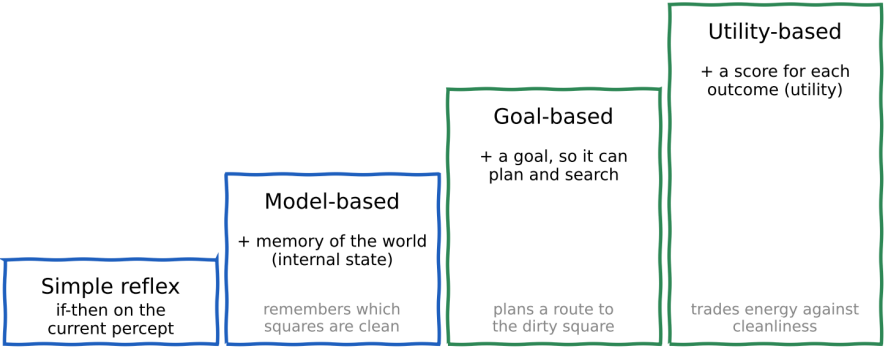
- Each type adds one ingredient to the one before it. The vacuum is the example throughout.
- Table-driven = looks up the whole percept sequence in a table and returns the stored action.
 - A vacuum with one row for every possible history. Impractical, because the number of rows explodes.
- Simple Reflex Agent = selects actions on the basis of the current percept only, ignoring the rest of the percept history. It uses condition-action rules (if-then).
 - "Dirty? Suck." Works only when the environment is fully observable, because it has no memory.

```
function REFLEX-VACUUM-AGENT([location, status])
  if status = Dirty then return Suck
  else if location = A then return Right
  else if location = B then return Left
```

- Model-based reflex = also keeps an internal state, a memory of how the world works.
 - It remembers that B is already clean, so it can act on what it cannot see right now.
- Goal-based = also has a goal and picks actions that reach it.
 - It plans a route to the dirty square. This is the agent that searches (W2, W3).
- Utility-based = also scores how good each outcome is.
 - Two routes reach the dirty square, and it takes the one that uses less battery.
- Learning Agent = any type above plus 4 parts. Learning lets it work in an unknown environment and become more competent than its initial knowledge. For a self-driving car:
 - Performance element picks the actions: it drives.
 - Critic compares behaviour with a standard: "that stop was too hard".
 - Learning element uses the feedback to improve: it brakes earlier next time.
 - Problem generator suggests new things to try: a different route, to gain experience.
- Agentic AI (LLM-based) = Natural Language Input -> Interpretation and Reasoning -> Workflow Generation -> Workflow Execution -> Output. Learning and course correction feed back into the loop.
 - "Book me a flight to Bali": it plans the steps, searches, fills in the form and returns the booking.

AGENT LADDER: each step adds one ingredient

Any step can also LEARN: critic + learning element + problem generator.
 Search (W2, W3) is what a goal-based agent does.



W1 quick recall

- Agent = sensors + actuators + environment. Rational = maximise the expected performance measure.
- PEAS specifies the task environment. 6 properties classify it.
- Reflex -> model-based (+ memory) -> goal-based (+ goal) -> utility-based (+ score). Any of them can learn.

W2. Problem Formulation

1. Three steps

- The example is a drive from Denpasar to Padangbai.
- Goal formulation = decide which states count as success. In(Padangbai).
 - A goal limits what the agent tries to achieve, so it limits the actions to consider. A tourist in Denpasar whose goal is "get to Lombok on time" can ignore sightseeing.

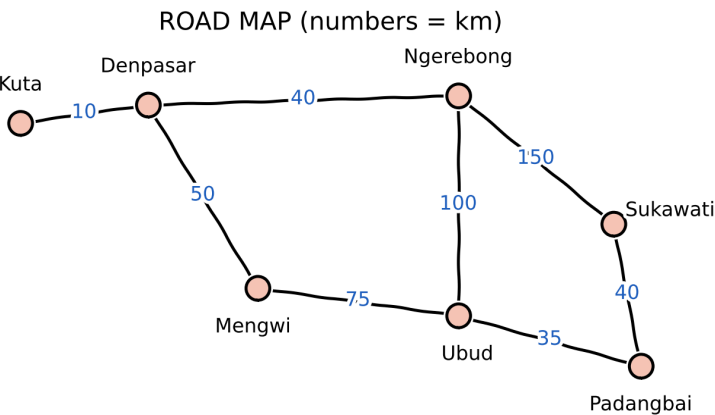
- Problem formulation = decide which states and actions to consider. Towns are states, drives between towns are actions.
- Search = the process of looking for a sequence of actions that reaches the goal. A search algorithm takes a problem as input and returns a solution, an action sequence. The actions are then carried out in the execution phase.

2. Component of Problem (5)

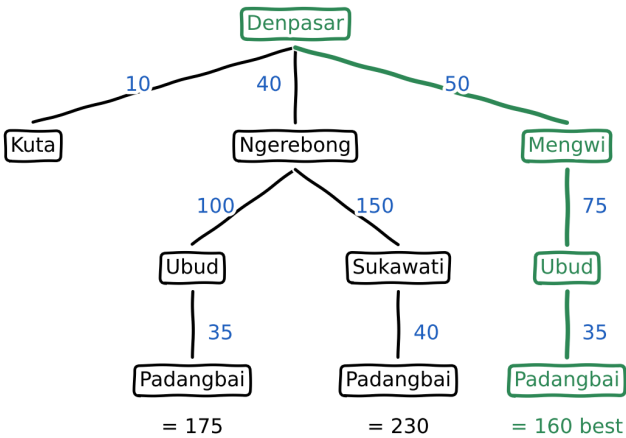
- Initial state = where the agent starts. In(Denpasar).
- ACTION(s) = actions available in state s. In Denpasar: Go(Kuta), Go(Ngerebong), Go(Mengwi).
- RESULT(s, a) = the transition model: the state reached by doing a in s. RESULT(In(Denpasar), Go(Ngerebong)) = In(Ngerebong).
- State space = every state reachable from the initial state. It is set by the initial state, the actions and the transition model. On the road map it is the 7 towns, drawn as a graph with towns as nodes and roads as links.
- Goal test = is this state the goal? In(Padangbai).
- Path cost function = a number for each path, the sum of its step costs. The agent picks a cost that reflects its performance measure: km if it wants the shortest trip, minutes if it wants the fastest.
 - Denpasar, Mengwi, Ubud = 50 + 75 = 125 km.
- Total path cost by hand: list every path to the goal, add the step costs of each, and pick the smallest. Section 3 does this.
- Solution = a path from the initial state to a goal state. Optimal solution = the one with the lowest path cost.
 - Via Ngerebong and Ubud (175 km): a solution, because it reaches Padangbai.
 - Via Mengwi and Ubud (160 km): the optimal solution, because no path is shorter.

3. Road map (km)

- Denpasar-Kuta 10, Denpasar-Ngerebong 40, Denpasar-Mengwi 50.
- Ngerebong-Ubud 100, Ngerebong-Sukawati 150, Mengwi-Ubud 75.
- Ubud-Padangbai 35, Sukawati-Padangbai 40. Kuta is a dead end.
- Paths: via Mengwi, Ubud = 160 (best). Via Ngerebong, Ubud = 175. Via Ngerebong, Sukawati = 230.



SAME MAP AS A SEARCH TREE (start = Denpasar)



Ubud appears twice: two nodes, one state.

4. Abstraction

- Abstraction = remove the details that do not matter for the problem.
 - The road map keeps towns and km and drops weather, radio and scenery.
- Valid = the abstract solution can be carried out in the real world. Any route on the map can be driven.
- Useful = each abstract step is easier than the original problem. "Go(Ubud)" hides every turn of the steering wheel.

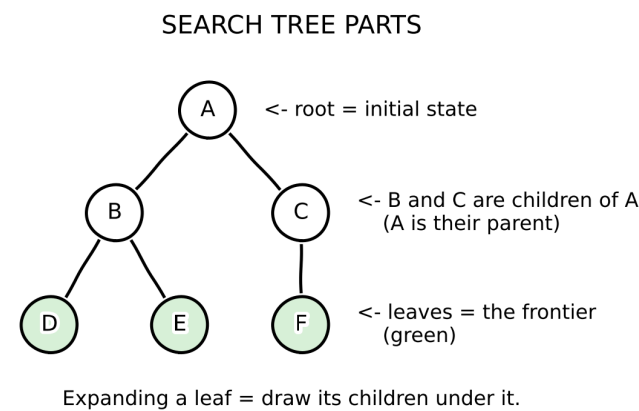
5. Example problems

- Vacuum world: 8 states (2 locations x 4 dirt combinations). Goal: all clean. Cost 1 per action.

- 8-puzzle: 3x3 board, 8 tiles and a blank. Actions: move the blank Up, Down, Left, Right. Cost 1 per move.
- Real world: route finding, travelling salesman (visit each city once, shortest tour), robot navigation.

6. Search tree terms

- The examples use the road tree above.
- Root = the node of the initial state. Denpasar.
- Parent node and child node = a node and the nodes it expands to. Denpasar is the parent node of Kuta, Ngerebong and Mengwi.
- Leaf node = a node with no children yet. Kuta, before anyone expands it.
- Frontier = all possible nodes for expansion, the leaf nodes waiting to be expanded. After expanding Denpasar: Kuta, Ngerebong, Mengwi.
- Expanding = apply ACTION to a node and generate its children. Expanding Mengwi gives Ubud.



7. TREE-SEARCH and GRAPH-SEARCH

- TREE-SEARCH = keep expanding nodes on the Frontier until a solution is found or there are no more states to expand. The slides label it the Search Strategy.
 - Problem: repeated states. Arad -> Sibiu -> Arad -> Sibiu can go on forever, because nothing remembers that Arad was already visited.

```

function TREE-SEARCH(problem) returns a solution, or failure
  initialize the frontier using the initial state of problem
  loop do
    if the frontier is empty then return failure
    choose a leaf node and remove it from the frontier
    if the node contains a goal state
      then return the corresponding solution
    expand the chosen node,
      adding the resulting nodes to the frontier
  
```

- GRAPH-SEARCH = TREE-SEARCH plus an Explored Set of states already expanded. The slides label it the Explored Strategy.

```

function GRAPH-SEARCH(problem) returns a solution, or failure
  initialize the frontier using the initial state of problem
  initialize the explored set to be empty
  loop do
    if the frontier is empty then return failure
    choose a leaf node and remove it from the frontier
    if the node contains a goal state
      then return the corresponding solution
    add the node to the explored set
    expand the chosen node, adding the resulting nodes
      to the frontier only if not in the frontier
      or explored set
  
```

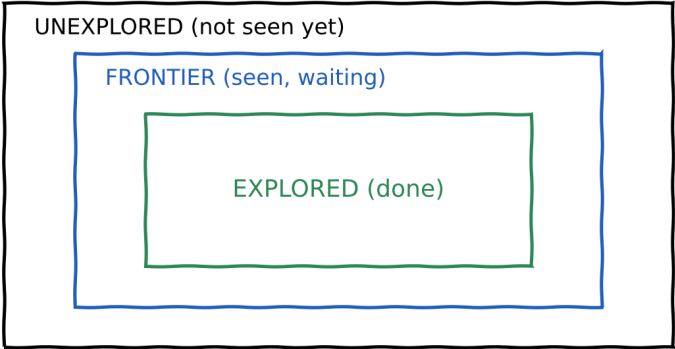
- The same loop in plain steps:

```

Frontier = [start]      Explored Set = { }
loop:
  1. Frontier empty?    -> return FAILURE
  2. PICK a node        -> the rule that differs per method
  3. it is a goal?      -> return the solution
  4. put it in the Explored Set
  5. expand it: add each child to the Frontier
    ONLY IF it is not in the Frontier or Explored Set
  
```

- TREE-SEARCH is the same loop without step 4 and without the check in step 5.
- Implemented data structure: Frontier = queue, with the operations EMPTY?(queue), POP(queue) and INSERT(element, queue). Explored Set = hash table, because checking for a repeated state is fast.
- Step 2 is the only step that changes between BFS, DFS, greedy and A*.

GRAPH-SEARCH: THREE ZONES



Each loop: pick a node from FRONTIER and move it to EXPLORED.
Its new children join FRONTIER.

8. Infrastructure of Search Algorithm: the node

- Each node *n* of the tree has 4 components: *n*.STATE, *n*.PARENT, *n*.ACTION and *n*.PATH-COST, written *g(n)*.
 - Ubud reached via Mengwi: STATE In(Ubud), PARENT the Mengwi node, ACTION Go(Ubud), PATH-COST 125.
- A node is not a state. Ubud reached via Ngerebong (*g* = 140) is a second node holding the same state.

9. Measuring problem-solving performance (4 criteria)

- Completeness = if a solution exists, does it find one? BFS: yes, because it checks every level in turn.
- Optimality = does it find the lowest-cost solution? BFS on the road map: no, because it returns the 175 km path with the fewest roads and misses the 160 km one.
- Time complexity = how many nodes are generated.
- Space complexity = how many nodes are kept in memory.

10. Big O Notation

- Big O = how the work grows with input size *n* in the worst case. Drop constants and smaller terms.
- From best to worst:
 - O*(1), constant operation: read the first item of a list.
 - O*(log *n*): binary search in a sorted list.
 - O*(*n*), linear operation: find the smallest number with one loop.
 - O*(*n* log *n*): merge sort.
 - O*(*n*²), quadratic operation: compare every pair with a nested loop.
 - O*(2^{*n*}): try every subset.
 - O*(*n*!), factorial operation: try every order of *n* cities (travelling salesman).

W2 quick recall

- Problem = initial state + ACTION(s) + RESULT(s, a) + goal test + path cost function.
- GRAPH-SEARCH = Frontier + Explored Set. Judge a method by: complete, optimal, time, space.

W3. Search Methods

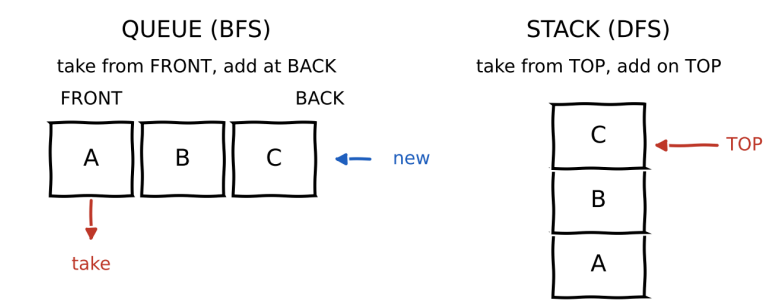
1. One loop, five orders

- Uninformed Search Strategies (blind search) = use only the problem definition. BFS, DFS and uniform-cost are uninformed, because they do not know that Ubud is nearer to Padangbai than Kuta.
- Informed (Heuristic) search strategies = also use problem-specific knowledge beyond the problem definition. Greedy and A* are informed.
- Best-first search = the general informed approach: expand the node with the lowest evaluation function *f(n)*.
- Every method runs the GRAPH-SEARCH loop. Only the order of the Frontier changes.

```
function GRAPH-SEARCH(problem, ORDER) returns a solution, or failure
  frontier <- queue ordered by ORDER, holding the initial node
  explored set <- empty
  loop do
    if EMPTY?(frontier) then return failure
    node <- POP(frontier)
    if node.STATE is a goal then return SOLUTION(node)
    add node.STATE to the explored set
    for each child of node do
      if child.STATE is in neither the frontier nor the explored set
        then INSERT(child, frontier)
      else if child.STATE is in frontier with a higher g
        then replace that frontier node with child
```

Method	Picks from the Frontier	Frontier is a
BFS	the shallowest unexpanded node (oldest)	FIFO queue
DFS	the deepest node (most recently generated)	LIFO queue (stack)
Uniform-cost	smallest <i>g(n)</i>	priority queue

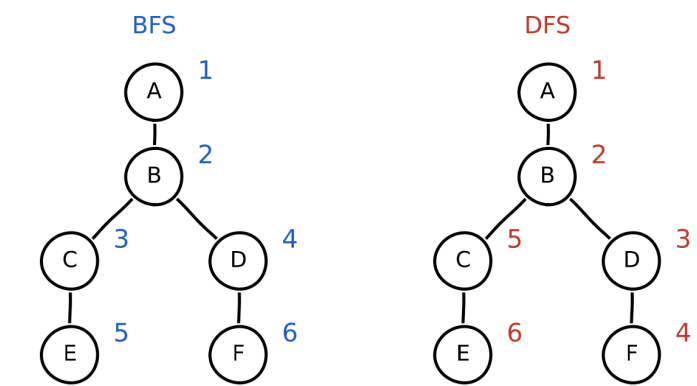
Method	Picks from the Frontier	Frontier is a
Greedy	smallest $h(n)$	priority queue
A*	smallest $g(n) + h(n)$	priority queue



2. BFS and DFS

- Breadth First Search (BFS) = expand the shallowest unexpanded node. It uses a FIFO queue: new nodes go to the back, old nodes are expanded first.
- Depth First Search (DFS) = always expand the deepest node in the current Frontier. It uses a LIFO queue (a stack): the most recently generated node is chosen for expansion.

SAME TREE, TWO VISIT ORDERS (number = order visited)



BFS trace. Frontier written front to back.

Step	Expand	Frontier	Explored Set
1	A	B	A
2	B	C D	A B
3	C	D E	A B C
4	D	E F	A B C D
5	E	F	A B C D E
6	F	empty	A B C D E F

DFS trace. Frontier written bottom to top. The last child pushed is taken first.

Step	Expand	Frontier	Explored Set
1	A	B	A
2	B	C D	A B
3	D	C F	A B D
4	F	C	A B D F
5	C	E	A B D F C
6	E	empty	A B D F C E

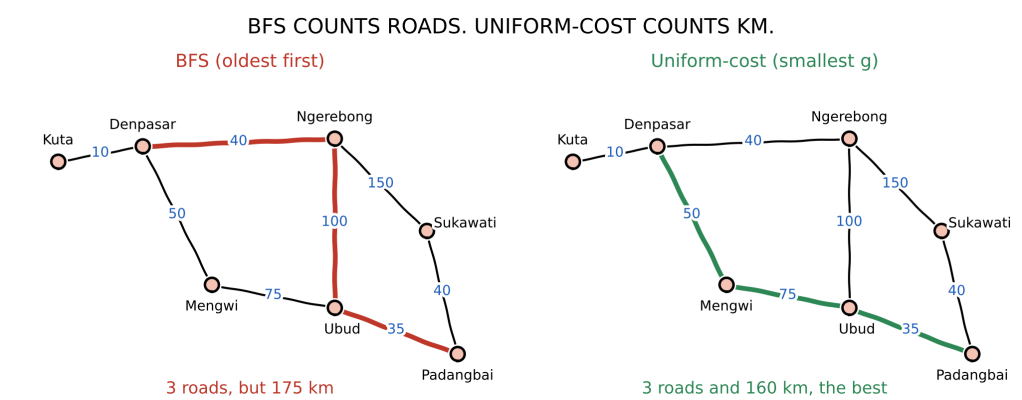
- To go left first, push the children in reverse. The order is then A B C E D F. Follow the order the question states. If the goal is E, as in the slides, BFS stops at step 5 with F still in the Frontier, and DFS finds E at step 6.

Point	BFS	DFS
Expands	shallowest node	deepest node
Advantage	there is always a possible optimal solution (the shallowest)	it can be the fastest algorithm, uses little memory
Disadvantage	takes more time to run	it is possible to get a solution that is not optimal

- Grid practice (DFS): write the direction order first (for example Up, Right, Down, Left) and follow it strictly. Number each cell as you enter it. At a dead end, back up to the last cell with an unvisited neighbour.

3. Uniform-cost search

- Uniform-cost = the same loop, picking the node with the smallest cost so far g(n).
 - BFS on the road map returns Denpasar, Ngerebong, Ubud, Padangbai at 175 km, because it counts roads and not km. Uniform-cost returns the 160 km path.
- BFS is optimal only when every step costs the same. Uniform-cost is optimal for any positive step costs.
- Test for the goal when a node is taken out, not when it is added.

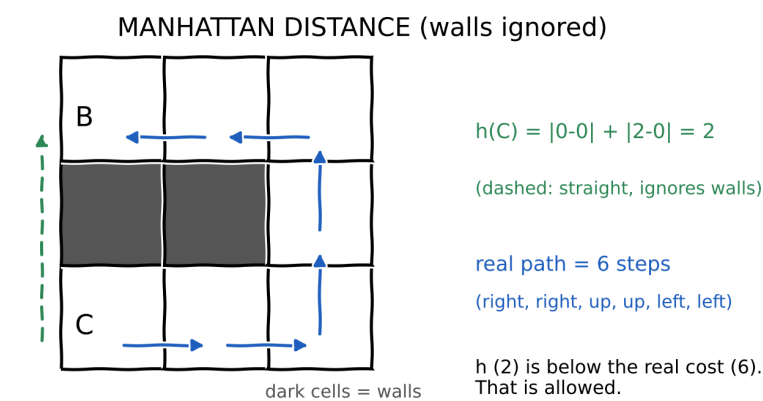


Step	Expand	Frontier, as town (g)
1	Denpasar (0)	Kuta (10), Ngerebong (40), Mengwi (50)
2	Kuta (10)	Ngerebong (40), Mengwi (50)
3	Ngerebong (40)	Mengwi (50), Ubud (140), Sukawati (190)
4	Mengwi (50)	Ubud (125) replaces 140, Sukawati (190)
5	Ubud (125)	Padangbai (160), Sukawati (190)
6	Padangbai (160)	goal, stop

4. Heuristic function

- Heuristic function $h(n)$ = the estimated cost of the cheapest path from n to the goal. $h(goal) = 0$.
 - On a grid: Manhattan distance. Count the cells across plus the cells up and ignore walls. Because walls are ignored, h can be smaller than the real path.

$$h(n) = |x_n - x_{goal}| + |y_n - y_{goal}|$$



- Practice map: the roads from W2, with assumed h values that are each below the real remaining distance.

Town	h	Roads out (km)
Denpasar	120	Kuta 10, Ngerebong 40, Mengwi 50
Kuta	130	none (dead end)
Ngerebong	90	Ubud 100, Sukawati 150
Mengwi	80	Ubud 75
Ubud	30	Padangbai 35
Sukawati	30	Padangbai 40
Padangbai	0	goal

5. Greedy best-first search

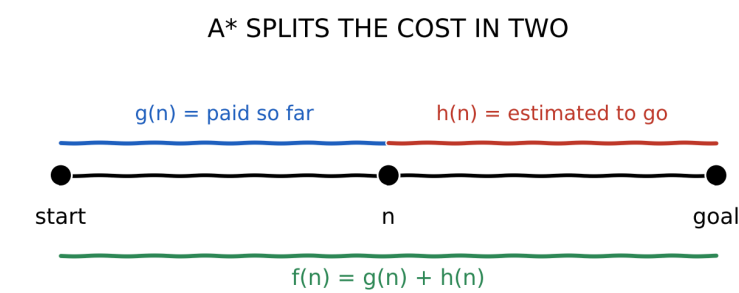
- Greedy best-first search = expand the node that is closest to the goal, because this is likely to lead to a solution quickly. $f(n) = h(n)$.

Step	Expand	Frontier, as town (h)
1	Denpasar	Kuta (130), Ngerebong (90), Mengwi (80)
2	Mengwi (80)	Kuta (130), Ngerebong (90), Ubud (30)
3	Ubud (30)	Kuta (130), Ngerebong (90), Padangbai (0)
4	Padangbai (0)	goal, stop

- Path: Denpasar, Mengwi, Ubud, Padangbai = 160.
- Not optimal in general, because it never counts the cost already paid. It found the best path here only because h happened to point that way.
- Grid practice: each cell shows h. Always step to the neighbour with the smallest number.

6. A* search

- A* Search = the best-known form of best-first search. Expand the node with the smallest $f(n) = g(n) + h(n)$.
 - $g(n)$ = the cost to reach the node from the start. $h(n)$ = estimated cost from n to the goal. $f(n)$ = estimated cost of the cheapest solution through n.



```

Frontier = priority queue, smallest f first
loop:
  1. take the SMALLEST f
  2. goal? stop (test when taken out, not when added)
  3. expand it: child g = parent g + step cost, f = g + h
  4. child already in the Frontier with a bigger g? replace it
  
```

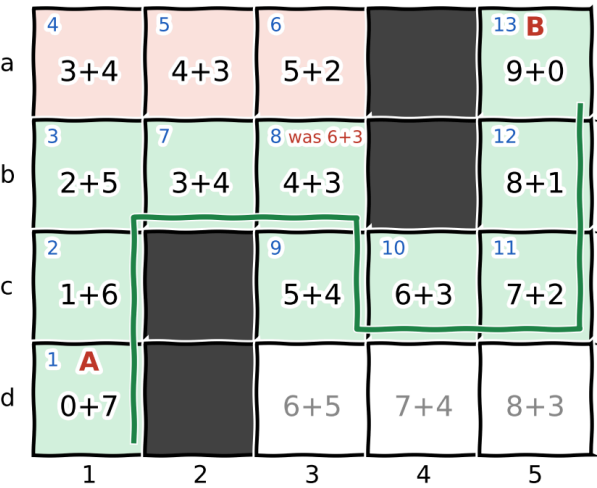
Trace, as town ($g + h = f$). At the tie in step 2, take the town added first.

Step	Expand	Frontier
1	Denpasar (0 + 120 = 120)	Kuta (10 + 130 = 140), Ngerebong (40 + 90 = 130), Mengwi (50 + 80 = 130)
2	Ngerebong (130)	Kuta 140, Mengwi 130, Ubud (140 + 30 = 170), Sukawati (190 + 30 = 220)
3	Mengwi (130)	Kuta 140, Ubud (125 + 30 = 155) replaces 170, Sukawati 220
4	Kuta (140), dead end	Ubud 155, Sukawati 220
5	Ubud (155)	Padangbai (160 + 0 = 160), Sukawati 220
6	Padangbai (160)	goal, stop

- Path: Denpasar, Mengwi, Ubud, Padangbai, cost 160. Sukawati is never expanded.
- Step 3 is the replace rule: Ubud was first reached via Ngerebong ($g = 140$), then cheaper via Mengwi ($g = 125$).
- Grid example, in the style of the slides:

- Start A is cell d1. Goal B is cell a5. Dark cells are walls. Moves are Up, Right, Down, Left, and each costs 1.
- g = steps taken from A. h = Manhattan distance to B, with walls ignored.
- Write $g + h$ in every cell you reach. Always expand the open cell with the smallest total. On a tie, take the smaller h , then the cell found first.

A* ON A GRID: each cell shows $g + h$



blue number = the order A* expands the cells

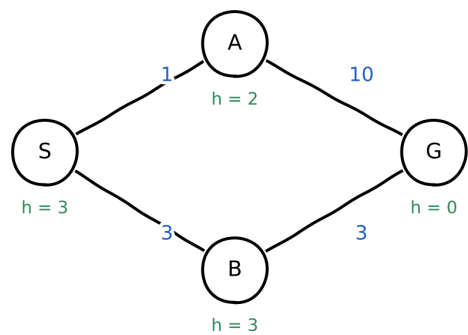
green = final path (9 steps) pink = dead end grey text = seen, never expanded

Step	Expand ($g + h = f$)	New cells written
1	d1 ($0 + 7 = 7$)	c1 ($1 + 6 = 7$)
2	c1 (7)	b1 ($2 + 5 = 7$)
3	b1 (7)	a1 ($3 + 4 = 7$), b2 ($3 + 4 = 7$)
4	a1 (7)	a2 ($4 + 3 = 7$)
5	a2 (7)	a3 ($5 + 2 = 7$)
6	a3 (7), dead end	b3 ($6 + 3 = 9$)
7	b2 (7)	b3 ($4 + 3 = 7$) replaces 9
8	b3 (7)	c3 ($5 + 4 = 9$)
9	c3 (9)	c4 ($6 + 3 = 9$), d3 ($6 + 5 = 11$)
10	c4 (9)	c5 ($7 + 2 = 9$), d4 ($7 + 4 = 11$)
11	c5 (9)	b5 ($8 + 1 = 9$), d5 ($8 + 3 = 11$)
12	b5 (9)	a5 ($9 + 0 = 9$)
13	a5 (9)	goal, stop

- Path: d1, c1, b1, b2, b3, c3, c4, c5, b5, a5. Total cost 9.
- A* tried the top row first, because those cells also had $f = 7$. The wall at a4 made it a dead end.
- f rose from 7 to 9 at c3, because the wall forces two steps away from B. Manhattan distance could not see that wall.
- d3, d4 and d5 were written but never expanded, because their $f = 11$ is above the final 9.
- The slides' A* maze has $g + h$ written in every open cell, including 16 cells A* never reaches. In a real run only the cells you reach get a number. The slide's highlighted cells and its answer (cost 21) are what a real run gives.
- A* is optimal only if h is:
 - Admissible = h should never be overestimated: it is never above the true cost to the goal. $h(\text{Ubud}) = 30$ is admissible, because the real distance is 35. $h(\text{Ubud}) = 50$ would not be, because 50 is more than 35.
 - Consistent = for every node n and successor n' with step cost c , $h(n) \leq h(n') + c$. Mengwi to Ubud: $80 \leq 30 + 75$ holds.
 - Consistent implies admissible.
- When the heuristic is wrong (graph below, S to G): greedy takes S, A, G = 11. A* with an honest h takes S, B, G = 6. A* with $h(B) = 20$ takes S, A, G = 11, because the inflated h hides the best path.

WHY THE COST SO FAR AND AN HONEST h MATTER

blue = road cost, green = h(n)



Greedy (smallest h):
S, A, G = 11. It ignores the 10 on A-G.

A* (g + h):
S, B, G = 6. The best path.

A* with h(B) = 20 (too big):
S, A, G = 11. Wrong, B looked too costly.

7. Scorecard

- b = children per node. d = depth of the shallowest goal. m = deepest level of the tree.

Method	Complete?	Optimal?	Time	Space
BFS	yes, if b is finite	yes, if all step costs are equal	$O(b^d)$	$O(b^d)$
DFS	yes on a finite graph with an Explored Set	no	$O(b^m)$	$O(b \cdot m)$
Uniform-cost	yes, if every step cost is above 0	yes	can exceed b^d	can exceed b^d
Greedy	yes on a finite graph with an Explored Set	no	$O(b^m)$ worst case	$O(b^m)$ worst case
A*	yes	yes, if h is admissible and consistent	exponential worst case	keeps every node
Minimax	yes, if the tree is finite	yes, against a best-playing opponent	$O(b^m)$	$O(b \cdot m)$
Alpha-beta	yes, if the tree is finite	same answer as minimax	$O(b^{(m/2)})$ best order	$O(b \cdot m)$

8. Adversarial search (games)

- Adversarial Search = search in a competitive environment in which the agents' goals are in conflict. The algorithm faces an opponent that tries to achieve the opposite goal.
- Tic-tac-toe is the example. It is:
 - Deterministic, because there are no dice.
 - Fully observable, because both players see the whole board.
 - Two-player and turn-taking, because X and O alternate.
 - Zero-sum, because a win for X (+1) is a loss for O (-1).
- Formulation: players MAX (X, moves first) and MIN (O).
 - S0 = the initial state, how the game is set up at the start. An empty 3x3 board.
 - PLAYER(s) = which player has the move in state s. X or O.
 - ACTION(s) = the set of legal moves in state s. The free squares.
 - RESULT(s, a) = the transition model, the result of a move. The board after move a.
 - TERMINAL-TEST(s) = the terminal test, true when the game is over. Someone has three in a row, or the board is full. Such a state is a terminal state.
 - UTILITY(s) = the utility function, the numeric value of a terminal state. X wins +1, draw 0, O wins -1.

9. Minimax

- Minimax Algorithm = compute the best move from the current state, assuming the opponent also plays its best. MAX is the maximizer, who tries to get the highest score. MIN is the minimizer, who tries to get the lowest.
 - A MAX node takes the biggest value among its children. A MIN node takes the smallest.
 - Values start at the leaves (UTILITY) and flow up to the root.

```
MINIMAX(s):
  if TERMINAL(s): return UTILITY(s)
  if MAX's turn: return the max of MINIMAX(child)
  if MIN's turn: return the min of MINIMAX(child)
```

- Slide version: MAX picks the action a in ACTION(s) with the highest MIN_VALUE(RESULT(s, a)). MIN picks the action with the lowest MAX_VALUE(RESULT(s, a)).

```
function MAX_VALUE(state)
  if TERMINAL(state) return UTILITY(state)
  v = -inf
  for action in ACTION(state)
    v = max(v, MIN_VALUE(RESULT(state, action)))
  return v

function MIN_VALUE(state)
```

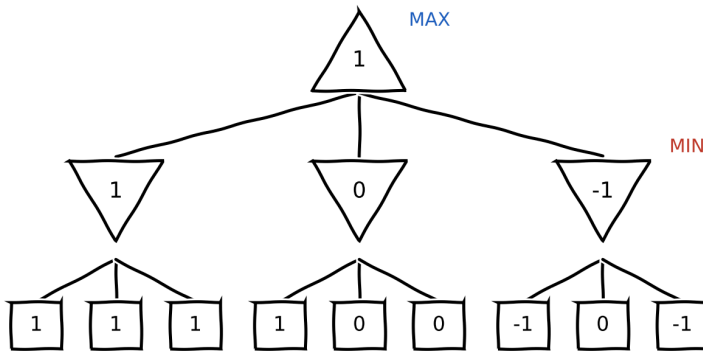
```

if TERMINAL(state) return UTILITY(state)
v = +inf
for action in ACTION(state)
  v = min(v, MAX_VALUE(RESULT(state, action)))
return v

```

- The further from the end of the game, the deeper the tree.
- By hand: draw the tree with the leaf values. Go up one row at a time. The root value is the answer, and MAX plays the branch it came from.
- Draw MAX as an up-triangle and MIN as a down-triangle.

MINIMAX: MAX takes the biggest, MIN takes the smallest



Root = $\max(1, 0, -1) = 1$, so take the left branch.

- Left MIN = $\min(1, 1, 1) = 1$. Middle MIN = $\min(1, 0, 0) = 0$. Right MIN = $\min(-1, 0, -1) = -1$.
- Root MAX = $\max(1, 0, -1) = 1$, so take the left branch.
- Minimax visits every node, so its time grows exponentially with depth.

10. Alpha-beta pruning

- Alpha-Beta Pruning = minimax that prunes away branches which cannot possibly influence the final decision. It returns the same move as minimax.
 - Alpha = the highest value MAX can already guarantee on this path. Starts at $-\infty$.
 - Beta = the lowest value MIN can already guarantee on this path. Starts at $+\infty$.
 - Prune the remaining children when $\alpha \geq \beta$.

```

AB(s, alpha, beta):      start with AB(root, -inf, +inf)
if TERMINAL(s): return UTILITY(s)
MAX node: v = -inf
  for each child:
    v = max(v, AB(child, alpha, beta))
    if v >= beta: return v      <- prune
    alpha = max(alpha, v)
  return v
MIN node: v = +inf
  for each child:
    v = min(v, AB(child, alpha, beta))
    if v <= alpha: return v    <- prune
    beta = min(beta, v)
  return v

```

- Slide version:

```

function ALPHA-BETA-SEARCH(state) returns an action
  v <- MAX-VALUE(state, -inf, +inf)
  return the action in ACTION(state) with value v

```

```

function MAX-VALUE(state, alpha, beta) returns a utility value
  if TERMINAL-TEST(state) then return UTILITY(state)
  v <- -inf
  for each a in ACTION(state) do
    v <- MAX(v, MIN-VALUE(RESULT(state, a), alpha, beta))
    if v >= beta then return v
    alpha <- MAX(alpha, v)
  return v

```

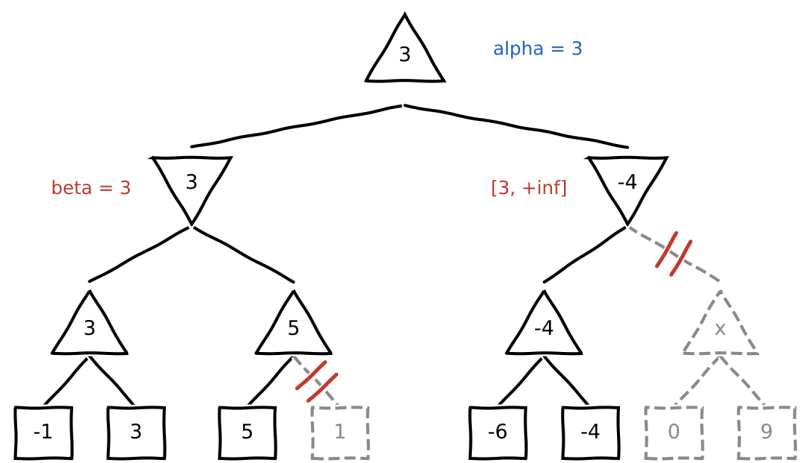
```

function MIN-VALUE(state, alpha, beta) returns a utility value
  if TERMINAL-TEST(state) then return UTILITY(state)
  v <- +inf
  for each a in ACTION(state) do
    v <- MIN(v, MAX-VALUE(RESULT(state, a), alpha, beta))
    if v <= alpha then return v
    beta <- MIN(beta, v)
  return v

```

- By hand: write $[\alpha, \beta]$ beside every node and pass it down. Coming back up, a MAX node raises alpha and a MIN node lowers beta.

ALPHA-BETA: same tree, with [alpha, beta] and cuts



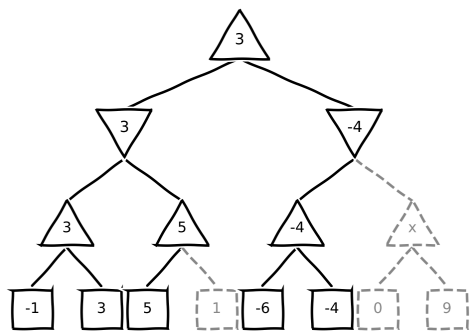
x = never looked at. Red // = cut branch. Root = 3, same as minimax.

Step	Node	[alpha, beta] on arrival	What happens
1	root MAX	$[-\text{inf}, +\text{inf}]$	go down to the left MIN
2	left MIN	$[-\text{inf}, +\text{inf}]$	go down to its first MAX child
3	MAX (-1, 3)	$[-\text{inf}, +\text{inf}]$	reads -1, then 3. Returns 3
4	left MIN		$v = 3$, so $\beta = 3$
5	MAX (5, 1)	$[-\text{inf}, 3]$	reads 5. $5 \geq \beta = 3$, so leaf 1 is pruned. Returns 5
6	left MIN		$v = \min(3, 5) = 3$. Returns 3
7	root MAX		$v = 3$, so $\alpha = 3$
8	right MIN	$[3, +\text{inf}]$	go down to its first MAX child
9	MAX (-6, -4)	$[3, +\text{inf}]$	reads -6, then -4. Returns -4
10	right MIN		$v = -4 \leq \alpha = 3$, so the child (0, 9) is pruned. Returns -4
11	root MAX		$\max(3, -4) = 3$

- Root = 3, the same as minimax. Never looked at: leaf 1 and the whole (0, 9) branch.
- Move order: alpha-beta prunes most when the best move is looked at first. In the slide's order it reads 5 of 8 leaves. In the worst order it reads all 8. In chess, try captures and checks first.

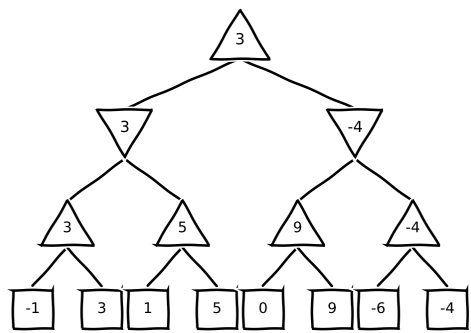
SAME TREE, TWO MOVE ORDERS: pruning depends on the order

Cut-triggering child first



leaves read: 5 of 8. Root = 3.

Cut-triggering child last



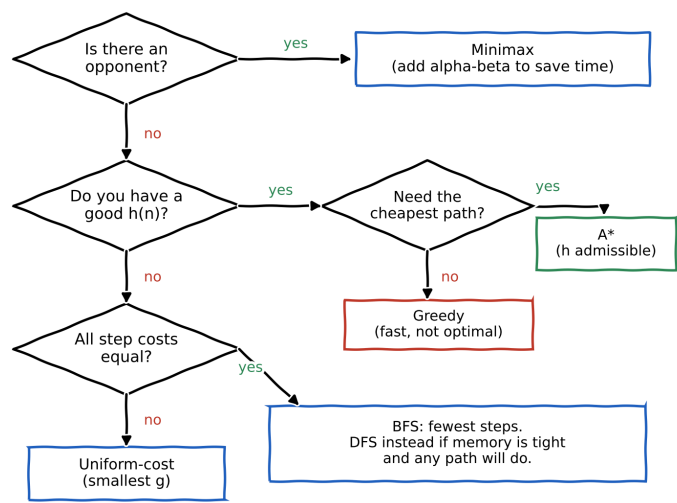
leaves read: 8 of 8. Root = 3.

Same answer both times. Only the amount of work changes. Dashed grey = never looked at.

- State of the art: chess Deep Blue, checkers CHINOOK, Othello LOGISTELLO, bridge Bridge Baron, Go AlphaGo.

11. Which method when

WHICH SEARCH METHOD?



12. Trace pitfalls

- Test for the goal when a node is taken out of the Frontier, not when it is added.
- State your tie-break rule before you start (alphabetical, or the order the question lists).
- GRAPH-SEARCH: never add a state that is already in the Explored Set. If it is in the Frontier with a bigger g, replace it.
- A*: g is the cost along the path from the start, not a straight-line distance.
- Minimax: check whether MAX or MIN moves at the level above the leaves.

13. Medical worked examples

Example A. A clinical pathway as a search problem (A*).

- A patient arrives with chest pain. The hospital wants the fastest route to treatment.
- Initial state: Arrival. Goal test: Treated. Step cost: minutes. Path cost: total minutes.
- ACTION(Arrival) = Go(ECG), Go(Blood test), Go(X-ray). RESULT(Arrival, Go(ECG)) = ECG.
- h = the estimated minutes still needed to reach Treated.

State	h	Steps out (minutes)
Arrival	50	ECG 10, Blood test 25, X-ray 15
ECG	45	Cardiologist 30, Blood test 20
Blood test	30	Cardiologist 10
X-ray	55	Cardiologist 45
Cardiologist	20	Treated 20
Treated	0	goal

- Total path cost of every path:
 - Arrival, Blood test, Cardiologist, Treated = 25 + 10 + 20 = 55 (best).
 - Arrival, ECG, Cardiologist, Treated = 10 + 30 + 20 = 60.
 - Arrival, ECG, Blood test, Cardiologist, Treated = 10 + 20 + 10 + 20 = 60.
 - Arrival, X-ray, Cardiologist, Treated = 15 + 45 + 20 = 80.

A* trace, as state (g + h = f). On a tie, take the state added first.

Step	Expand	Frontier
1	Arrival (0 + 50 = 50)	ECG (10 + 45 = 55), Blood test (25 + 30 = 55), X-ray (15 + 55 = 70)
2	ECG (55)	Blood test 55, X-ray 70, Cardiologist (40 + 20 = 60). Blood test via ECG has g = 30, worse than 25, so keep 25
3	Blood test (55)	X-ray 70, Cardiologist (35 + 20 = 55) replaces 60
4	Cardiologist (55)	X-ray 70, Treated (55 + 0 = 55)
5	Treated (55)	goal, stop

- Answer: Arrival, Blood test, Cardiologist, Treated. Total path cost 55 minutes. X-ray is never expanded.
- h is admissible here, because every h is at most the real remaining time. ECG: h = 45, real = 50.

Example B. Choosing a treatment against the worst case (minimax).

- MAX = the doctor, who picks a treatment. MIN = the disease, which responds in the worst way for the patient.
- UTILITY = recovery score from 0 to 10.

Treatment (MAX's move)	Possible responses (MIN's move)	MIN value
Drug A	7, 5, 6	min = 5
Drug B	9, 2, 8	min = 2
Surgery	6, 6, 4	min = 4

- Root MAX = $\max(5, 2, 4) = 5$, so the doctor picks Drug A.
- Drug B has the best single outcome (9), but minimax rejects it, because its worst case (2) is the lowest.
- To draw it: one MAX up-triangle at the root, three MIN down-triangles below it, and three leaves under each.

W3 quick recall

- Uninformed: BFS (FIFO queue), DFS (LIFO queue), uniform-cost (smallest g).
- Informed: greedy $f = h$. $A^* f = g + h$, optimal if h is admissible and consistent.
- Games: minimax (MAX highest, MIN lowest). Alpha-beta gives the same answer with less work.

Big picture

- W1 defines an agent. W2 turns its task into a search problem. W3 solves it: blind (BFS, DFS, uniform-cost), guided (greedy, A^*), or against an opponent (minimax, alpha-beta).

HOW THE THREE WEEKS FIT

