Artificial Intelligence

Search trees, BFS and UCS

The notes for the lesson “Search trees, BFS and UCS”: chapters 2.1 to 2.2, with their figures, worked examples, exam traps, quizzes and exam question bank, in one file.

2 chapters | Chapters 2.1 to 2.2

About these notes

These are the notes for the University free n easy video lesson **Search trees, BFS and UCS** on Artificial Intelligence: chapters 2.1 to 2.2, with every figure, worked example, exam trap and quiz, and the long-answer exam questions on these chapters.

2.1 Searching for solutions: search trees and strategies

A search algorithm finds a solution by growing a **search tree**. The root is the initial state, each branch is an action, and each node stands for one path from the root to some state. At every step the algorithm picks one leaf, checks whether it is a goal, and if not, **expands** it by generating its children. The rule that picks the leaf is the **search strategy**, and it is the only thing that changes between the algorithms of this unit. Route planners, puzzle solvers and game programs all run this loop.

You need: problem formulation (initial state, actions, transition model, goal test, path cost) from An earlier lesson, Problem-solving agents and problem formulation, and the state space from An earlier lesson, State-space search, control strategies and production systems.

State space and search tree

The **state space** is a graph in which every state appears once. The search tree grows from it, and one state can appear in many nodes, one per path that reaches it. In Figure 2.1.1 a state space of only four states gives an infinite tree, since S → A → S → A … never ends.

A **node** is a record with five fields: its **state**; its **parent** node (following parents back to the root gives the solution path); the **action** applied to the parent; the **path cost g(n)** from the root; and its **depth**, the number of steps from the root. The **frontier** (also called the open list or fringe)

holds the nodes generated but not yet expanded. The **explored set** (closed list) holds the states already expanded. A **loopy path** visits a state twice, such as S → A → S; a **redundant path** reaches a state by a second, costlier route, such as S → B directly (cost 4) when S → A → B costs 3.

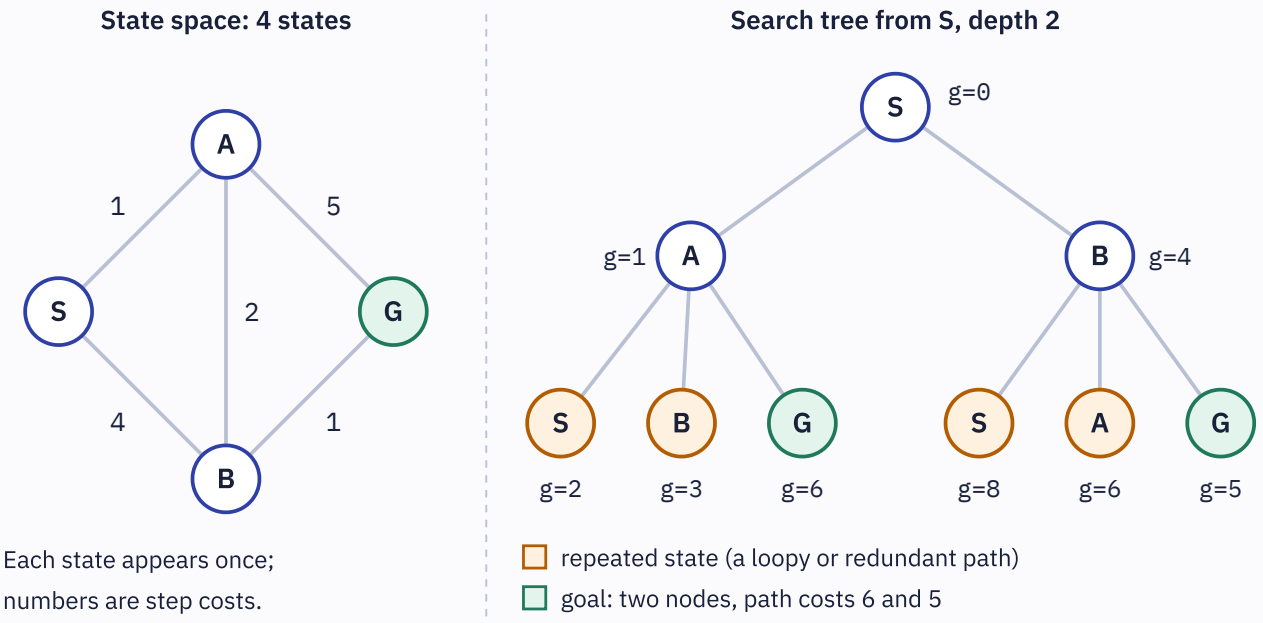


Figure 2.1.1 Left, the state space: four states and five roads with their costs. Right, the first two levels of the search tree from S. The tree already holds S three times and B twice; each copy is a different path with a different cost g. Notice that the goal appears in two nodes, and the cheaper one (g = 5) is not the one on the left.

Tree search and graph search

Every algorithm in Chapters 2.2 to 2.5 is this loop, written here as **graph search**:

1. Put a node for the initial state in the frontier. Make the explored set empty.
2. If the frontier is empty, return failure.
3. Choose a node n and remove it from the frontier. The strategy decides which.
4. If the state of n is a goal, return the path from the root to n by following parent links.
5. Add the state of n to the explored set.
6. Expand n. Add each child to the frontier unless its state is already in the frontier or the explored set. Go to step 2.

Tree search is the same loop without the explored set and the check in step 6. It needs less bookkeeping, but on a state space with cycles it can run forever. Graph search expands each state at most once, so on a finite state space it always stops. Breadth-first search moves the

goal test into step 6 to save one level (Chapter 2.2); uniform-cost search and A* keep it in step 4, and replace a frontier node when they find a cheaper path to its state.

Strategies and how to judge them

The frontier is a queue, and the kind of queue is the strategy. A first-in first-out (FIFO) queue gives breadth-first search; a last-in first-out (LIFO) stack gives depth-first search; a priority queue ordered by g(n) gives uniform-cost search. These are **uninformed** (blind) strategies: they use only the problem definition. **Informed** (heuristic) strategies also use an estimate h(n) of the cost from n to a goal: ordering by h(n) gives greedy best-first search, and by g(n) + h(n) gives A* (Chapter 2.5). This choice of order is called the **control strategy**.

Strategies are compared on four measures: **completeness**, whether it always finds a solution when one exists; **optimality**, whether it finds the cheapest; **time complexity**, the nodes it generates; and **space complexity**, the nodes it stores at once.

Complexity is stated with three numbers: **b**, the branching factor (the largest number of children of any node); **d**, the depth of the shallowest goal; and **m**, the maximum depth of any path, which may be infinite. A full tree to depth d has $1 + b + b^2 + \ldots + b^d = (b^{d+1} - 1)/(b - 1)$ nodes, and the last level dominates, so the count is $O(b^d)$.

Three more ideas that some syllabi list here. **Data-driven** (forward) search starts from the given facts; **goal-driven** (backward) search starts from the goal and works back. Goal-driven suits one stated goal with few ways to reach it; Chapter 2.4 runs both at once. **Generate-and-test** produces a candidate, tests it and repeats; done systematically it is an exhaustive search, and with feedback from the test it becomes hill climbing (Chapter 2.6). When an action can have several outcomes, the solution is a contingency plan, and the tree has **AND nodes**, where every outcome must be handled, besides the usual **OR nodes**, where the agent picks one action.

WORKED EXAMPLE

Question. In Figure 2.1.1 the roads cost S–A 1, S–B 4, A–B 2, A–G 5 and B–G 1. Run graph search from S to goal G with (a) a FIFO frontier and (b) a LIFO frontier, both testing the goal when a child is generated; and (c) a priority queue on g(n), testing the goal when a node is chosen. Generate children in alphabetical order. Give each path and its cost.

STRATEGY	STEP	NODE CHOSEN	WHAT HAPPENS	FRONTIER AFTER
(a) FIFO	1	S	generate A, B	`[A, B]`
	2	A (oldest)	S explored, B already in frontier; generate G: goal	stop
(b) LIFO	1	S	generate A, B	`[A, B]`, `B` on top
	2	B (newest)	S explored, A already in frontier; generate G: goal	stop
(c) priority g	1	S (g 0)	A g = 1, B g = 4	`A 1, B 4`
	2	A (g 1)	B via A: 1 + 2 = 3 < 4, replace; G: 1 + 5 = 6	`B 3, G 6`
	3	B (g 3)	G via B: 3 + 1 = 4 < 6, replace	`G 4`
	4	G (g 4)	goal test passes	stop

Following parent links back from G: (a) **S–A–G**, cost 1 + 5 = **6**; (b) **S–B–G**, cost 4 + 1 = **5**; (c) **S–A–B–G**, cost 1 + 2 + 1 = **4**, the cheapest. Strategy (c) is uniform-cost search. Same graph, same loop, three answers: only the order of expansion changed.

EXAM TRAPS

- A node is not a state. Two nodes can hold the same state with different parents and costs: in strategy (c), state B had a node with g = 4 and another with g = 3.
- Tree search on a graph with cycles can loop forever. If the state space has loops, write graph search with an explored set.
- d is the depth of the shallowest goal and m the depth of the deepest path. Mixing them up is the most common slip in complexity answers.
- Time complexity counts nodes generated, not seconds; space complexity counts nodes stored.
- Favourite question: "Write the general search algorithm and show how the choice of frontier gives breadth-first, depth-first and uniform-cost search."

CHECK YOURSELF

– REMEMBER **Define the frontier and the explored set.** 2 marks

The frontier holds nodes that have been generated but not yet expanded. The explored set holds states that have already been expanded, so graph search does not expand them again.

– REMEMBER **List the four measures used to compare search strategies.** 2 marks

Completeness, optimality, time complexity and space complexity.

– UNDERSTAND **Distinguish a state from a node.** 2 marks

A state is a configuration of the world. A node is a record in the search tree holding a state plus its parent, action, path cost and depth. One state can sit in many nodes, one per path that reaches it.

– UNDERSTAND **Why can tree search fail to stop even when the state space is finite?** 2 marks

Without an explored set it can follow a loopy path such as S → A → S → A forever, so the tree is infinite even though the graph has four states.

– APPLY **How many nodes are in a complete search tree with b = 3 down to depth 4?** 2 marks

1 + 3 + 9 + 27 + 81 = **121**, which matches $(3^5 - 1)/(3 - 1) = 242/2$.

– APPLY **In Figure 2.1.1, what is g for the node reached by S → B → A → G?** 1 mark

4 + 2 + 5 = **11**.

– ANALYZE **A system must decide which of 3 diseases a patient has, using rules over 40 possible symptoms. Choose data-driven or goal-driven search and justify.** 4 marks

Goal-driven. There are only 3 goals, and working back from each asks only for the symptoms that matter to it. Data-driven search would fire every rule matching any of the 40 symptoms and build conclusions nobody asked for.

BEYOND THE SYLLABUS

Not for the exam. The same frontier loop runs inside automated planners that read problems written in PDDL (the Planning Domain Definition Language), inside theorem provers, and in "tree of thoughts" methods that search over the reasoning steps of a large language model. The ACM/IEEE CS2023 curriculum keeps state-space search as core material of its artificial intelligence area. Search next: "AND-OR search tree", "belief-state search", "PDDL planning".

2.2 Breadth-first and uniform-cost search

Breadth-first search (BFS) expands the shallowest node first, so it explores the tree level by level and finds the solution with the fewest steps. **Uniform-cost search** (UCS) expands the cheapest node first, the one with the lowest path cost g(n), so it finds the solution with the lowest total cost even when steps cost different amounts. BFS answers "fewest flight changes from Kochi to Guwahati"; UCS answers "least total travel time", and it is Dijkstra's shortest-path algorithm stopped at the goal.

Breadth-first search

BFS keeps the frontier in a FIFO queue. New nodes join at the back, so every node at depth k is expanded before any node at depth k + 1. Because a goal found at depth k can only be beaten by a shallower one, and all shallower nodes are already done, BFS tests the goal as soon as a child is *generated*.

1. If the start state is a goal, return it. Otherwise put it in a FIFO frontier; make the explored set empty.
2. If the frontier is empty, return failure. Otherwise remove the node n at the front and add its state to the explored set.

3. For each child of n whose state is neither explored nor in the frontier: if it is a goal, return its path; otherwise add it to the back of the frontier. Go to step 2.

Properties. BFS is **complete** when b is finite. It is **optimal** only when every step costs the same, because then fewest steps means cheapest. It generates $b + b^2 + \ldots + b^d$ nodes, so **time is $O(b^d)$**, and since the whole last level waits in the frontier, **space is also $O(b^d)$**. Memory runs out long before patience does. With b = 10, one million nodes generated per second and 1,000 bytes per node:

DEPTH D	NODES ($1 + B + \ldots + B^D$)	TIME	MEMORY
2	111	0.11 ms	111 KB
4	11,111	11 ms	11 MB
6	about 1.1×10^6	1.1 s	1.1 GB
8	about 1.1×10^8	1.9 minutes	111 GB
10	about 1.1×10^{10}	3.1 hours	11 TB
12	about 1.1×10^{12}	13 days	1.1 PB

Uniform-cost search

UCS keeps the frontier in a **priority queue** ordered by g(n) and always removes the node with the smallest g. Two changes to the loop make it correct for unequal costs:

- The goal is tested when a node is *chosen*, not when generated, because a goal reached first by an expensive road may later be reached more cheaply (in the worked example, G appears at g = 11, then at g = 9).
- If a child's state is already in the frontier with a higher g, the cheaper node replaces it. The code below does this by pushing the new copy and skipping the stale one.

Properties. UCS expands nodes in increasing order of g, so the first goal it chooses is the cheapest: it is **optimal**. It is **complete** if every step costs at least some $\varepsilon > 0$; with zero-cost steps it can circle forever. Let C^* be the cost of the optimal solution. UCS may explore every path cheaper than C^*, and such a path can have up to $\lfloor C^*/\varepsilon \rfloor$ steps, so **time and space are $O(b^{1+\lfloor C^*/\varepsilon \rfloor})$**. When all steps cost the same, this becomes $O(b^{d+1})$: UCS then does what BFS does, plus one level, because it tests the goal later.

In Python 3, the two searches differ only in the queue and in where the goal test sits:

```
import heapq
from collections import deque

ROADS = {'S': {'A': 2, 'B': 3, 'F': 2}, 'A': {'S': 2, 'C': 2, 'D': 9},
         'B': {'S': 3, 'D': 4}, 'C': {'A': 2, 'D': 1, 'E': 3},
         'D': {'A': 9, 'B': 4, 'C': 1, 'G': 6}, 'E': {'C': 3, 'G': 2},
         'F': {'S': 2, 'H': 3}, 'G': {'D': 6, 'E': 2}, 'H': {'F': 3}}

def path_to(parent, n):
    p = [n]
    while parent[n]:
        n = parent[n]; p.append(n)
    return p[::-1]

def bfs(start, goal):
    frontier, parent = deque([start]), {start: None}
    while frontier:
        n = frontier.popleft()                    # FIFO: oldest first
        for m in sorted(ROADS[n]):
            if m not in parent:
                parent[m] = n
                if m == goal:                     # test when generated
                    return path_to(parent, m)
                frontier.append(m)

def ucs(start, goal):
    frontier, g, parent, done = [(0, start)], {start: 0}, {start: None}, set()
    while frontier:
        cost, n = heapq.heappop(frontier)         # cheapest g first
        if n in done:
            continue                              # stale, costlier copy
        if n == goal:                             # test when expanded
            return path_to(parent, n), cost
        done.add(n)
        for m, c in ROADS[n].items():
            if cost + c < g.get(m, float('inf')):
                g[m], parent[m] = cost + c, n
                heapq.heappush(frontier, (cost + c, m))

print('BFS:', bfs('S', 'G'))
print('UCS:', ucs('S', 'G'))
```

Output:

```
BFS: ['S', 'A', 'D', 'G']
UCS: (['S', 'A', 'C', 'E', 'G'], 9)
```

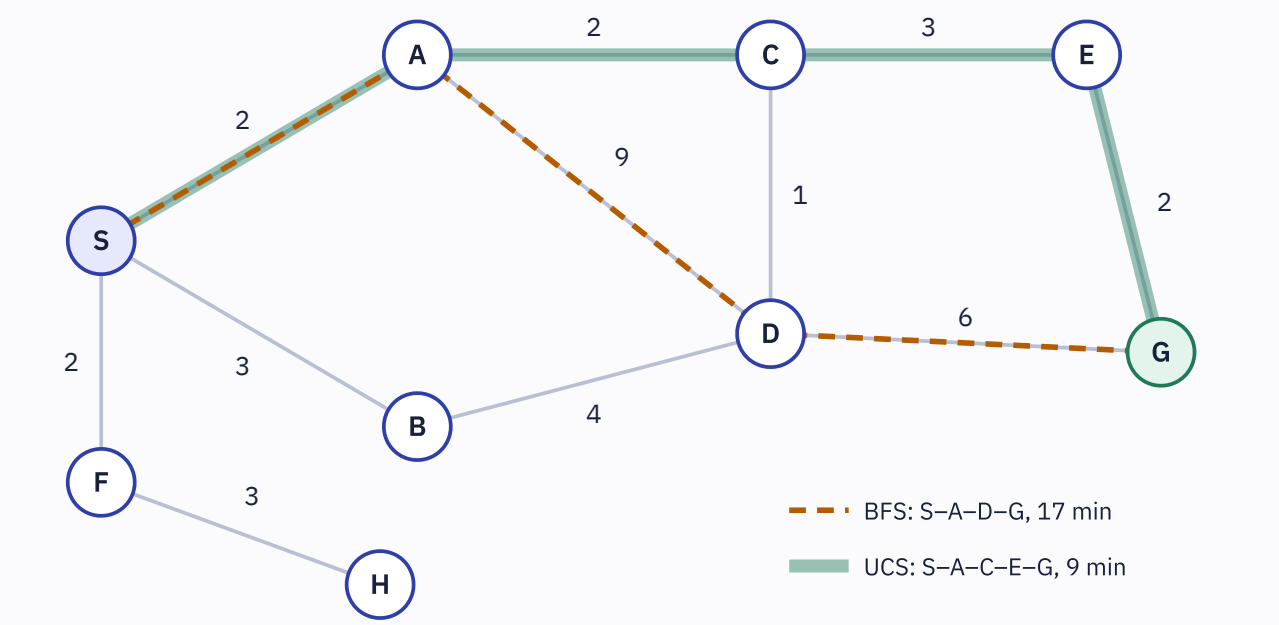


Figure 2.2.1 A delivery rider's map from the shop S to the customer G; numbers are minutes. BFS (orange dashes) returns the route with the fewest roads, 3, which takes 17 minutes. UCS (green) returns a route with 4 roads that takes only 9. Notice the side lane S–F–H: UCS still expands F and H because they are cheap to reach, though they lead nowhere. Chapter 2.5 reuses this map.

WORKED EXAMPLE

Question. For the map in Figure 2.2.1, trace BFS and UCS from S to G with graph search. Break ties alphabetically. Show the frontier after each expansion and give the path, the number of roads and the time.

BFS (FIFO frontier, goal test when generated):

STEP	EXPAND	CHILDREN ADDED	FRONTIER AFTER
1	S	A, B, F	A B F
2	A	C, D	B F C D
3	B	none (D already in frontier)	F C D
4	F	H	C D H
5	C	E (D already in frontier)	D H E
6	D	G is generated: goal	stop

Parents: G ← D ← A ← S. Path **S–A–D–G**, 3 roads, 2 + 9 + 6 = **17 minutes**, after 6 expansions.

UCS (priority queue on g, goal test when chosen):

STEP	EXPAND (G)	UPDATES	FRONTIER AFTER, WITH G
1	S (0)	A 2, B 3, F 2	A2 F2 B3
2	A (2)	C 4, D 2 + 9 = 11	F2 B3 C4 D11
3	F (2)	H 5	B3 C4 H5 D11
4	B (3)	D via B: 3 + 4 = 7 < 11, replace	C4 H5 D7
5	C (4)	D via C: 4 + 1 = 5 < 7, replace; E 7	D5 H5 E7
6	D (5)	G 5 + 6 = 11	H5 E7 G11
7	H (5)	none (dead end)	E7 G11
8	E (7)	G via E: 7 + 2 = 9 < 11, replace	G9
9	G (9)	goal test passes	stop

Parents: G ← E ← C ← A ← S. Path **S–A–C–E–G**, 4 roads, 2 + 2 + 3 + 2 = **9 minutes**, after 9 expansions. Had UCS tested the goal at generation, it would have stopped at step 6 with the 11-minute route.

EXAM TRAPS

- BFS is optimal only when all step costs are equal. On Figure 2.2.1 it returns 17 minutes when 9 is possible.
- UCS tests the goal when a node is chosen, not when generated. Testing early returns a costlier path (11 minutes above).
- BFS time is $O(b^d)$ when the goal is tested at generation; books that test at expansion write $O(b^{d+1})$. Either is accepted if you say which test you use.
- UCS follows cost, not direction: it expands the dead-end lane F–H because it is cheap.
- Favourite question: "Trace BFS and UCS on the given graph, show the frontier at each step, and compare the solutions."

CHECK YOURSELF

– REMEMBER **Which data structure holds the frontier in BFS, and which in UCS?** 1 mark

BFS: a FIFO queue. UCS: a priority queue ordered by path cost g(n).

– REMEMBER **State the time complexity of UCS and define each symbol.** 2 marks

$O(b^{1+\lfloor C^*/\varepsilon \rfloor})$: b is the branching factor, C* the cost of the optimal solution and ε the smallest step cost.

– UNDERSTAND **Why is memory a bigger problem than time for BFS?** 2 marks

BFS keeps the whole deepest level in the frontier, so space grows as fast as time. At d = 10 the table gives 3.1 hours but 11 TB of memory: the time is bearable, the memory is not.

– APPLY **With b = 4 and the shallowest goal at d = 5, how many nodes does BFS generate in the worst case (goal tested at generation)?** 2 marks

4 + 16 + 64 + 256 + 1024 = **1,364** nodes (1,365 with the root).

– APPLY **In Figure 2.2.1 the road C–E is closed for repairs. What route and time does UCS return now?** 3 marks

E can now be reached only from G, so the best route goes through D: S–A–C (4 minutes), C–D (5), D–G (11). UCS returns **S–A–C–D–G, 11 minutes**.

– ANALYZE **One metro rider wants the fewest train changes and another the shortest travel time. Which algorithm suits each, and why would BFS fail the second rider?** 4 marks

Fewest changes: BFS on a graph where each edge is one ride, since all edges count 1. Shortest time: UCS with minutes as step costs. BFS ignores the costs, so it would return the route with fewest edges even if one edge is a 50-minute ride.

BEYOND THE SYLLABUS

Not for the exam. Dijkstra published his shortest-path algorithm in 1959, and it is still the base of road routing. Large map services add preprocessing on top, such as contraction hierarchies, which precompute shortcut edges so that a query across a country explores only a tiny part of the road network. BFS runs in web crawlers and in copying garbage collectors such as Cheney's algorithm. When edge costs are only 0 or 1, a double-ended queue gives "0-1 BFS", which is faster than a priority queue. Search next: "contraction hierarchies", "0-1 BFS", "Dijkstra versus uniform-cost search".

Key terms of this lesson

TERM	MEANING OR FORMULA	CHAPTER
Node	state, parent, action, path cost g(n), depth	2.1
b, d, m	branching factor; depth of the shallowest goal; maximum path depth	2.1
Full tree to depth d	$1 + b + \ldots + b^d = (b^{d+1} - 1)/(b - 1)$	2.1
BFS	FIFO frontier; time and space $O(b^d)$; optimal for equal step costs	2.2
UCS	priority on g(n); $O(b^{1+\lfloor C^*/\varepsilon \rfloor})$; optimal; goal test on expansion	2.2

Exam question bank

Long-answer exam questions on these chapters, each with an outline of the points an examiner looks for. Answer one in full on paper before you read its outline.

Remember and understand

– UNDERSTAND **Explain the general graph-search algorithm. Define node, frontier and explored set, state the four performance measures, and show how the choice of frontier gives BFS, DFS and UCS.** 10 marks

Six-step loop with explored set; node fields; tree search versus graph search (loops); completeness, optimality, time, space in b, d, m; FIFO → BFS, LIFO → DFS, priority on g → UCS; a small example such as the three strategies on Figure 2.1.1 (costs 6, 5, 4). Spans 2.1 to 2.3.

– UNDERSTAND **Explain data-driven and goal-driven search with an example of each, and state when each is the better choice.** 8 marks

Forward from facts versus backward from the goal; goal-driven when the goal is given and few rules lead to it (checking one diagnosis); data-driven when all data is given and the goal is open (configuration, monitoring); bidirectional search as both at once (2.1, 2.4).

Apply

– APPLY **On the delivery map of Figure 2.2.1, trace BFS, DFS, UCS, greedy best-first and A* (heuristic of Figure 2.5.1). Tabulate path, cost and nodes expanded, and comment.** 16 marks

BFS S–A–D–G, 17 min, 6 expansions; DFS S–A–C–D–G, 11 min, 6 visits; UCS S–A–C–E–G, 9 min, 9 expansions; greedy S–B–D–G, 13 min, 4 expansions; A* S–A–C–E–G, 9 min, 7 expansions. Only UCS and A* are optimal; A* saves F and H. Spans 2.2, 2.3 and 2.5.

– APPLY **For b = 5 and the shallowest goal at d = 4, find the nodes generated by BFS, IDS and bidirectional BFS, and the memory each needs.** 8 marks

BFS 5 + 25 + 125 + 625 = **780**, memory up to 625 nodes. IDS 4·5 + 3·25 + 2·125 + 625 = **970**, memory about b·d = 20. Bidirectional 2 × (5 + 25) = **60**, memory about 25 per side. Spans 2.2 to 2.4.

Analyze and evaluate

– ANALYZE **Compare BFS, UCS, DFS, DLS, IDS and bidirectional search in a table. Recommend one for (a) a unit-cost route at depth 8 with b = 10, (b) a puzzle with infinite paths and unknown depth, (c) a road map with unequal costs.** 12 marks

Table of 2.3 plus bidirectional $O(b^{d/2})$. (a) Bidirectional BFS if the goal is explicit, else IDS (BFS would need about 111 GB). (b) IDS: complete, O(bd) memory. (c) UCS, or A* when a heuristic exists. Spans 2.2 to 2.5.

Glossary (terms used in this lesson)

Breadth-first search	Expands the shallowest node first, using a FIFO queue.
Depth-first search	Expands the deepest node first, using a LIFO stack or recursion.
Explored set	The states already expanded (the closed list).
Frontier	Nodes generated but not yet expanded (the open list or fringe).
Generate-and-test	Produce a candidate, test it, repeat until one passes.
Greedy best-first search	Best-first search on f(n) = h(n); fast but not optimal.
Hill climbing	Local search that always moves to the best neighbour and stops when none is better.
Node	A search-tree record: state, parent, action, path cost and depth.
Uniform-cost search	Expands the node with the lowest path cost g(n); optimal for positive step costs.

Sources and further reading

Standard textbooks:

- Stuart Russell and Peter Norvig, *Artificial Intelligence: A Modern Approach*, 4th edition, Pearson (3rd edition 2010): the chapters "Solving Problems by Searching" (uninformed search, A*, heuristics, memory-bounded search) and the following chapter on local search.
- Elaine Rich, Kevin Knight and Shivashankar B. Nair, *Artificial Intelligence*, 3rd edition, McGraw-Hill: the chapters on problem spaces and search and on heuristic search techniques (generate-and-test, hill climbing, best-first search).

- Saroj Kaushik, *Artificial Intelligence*, Cengage Learning: the chapter on state-space search and control strategies, including iterative-deepening A*.
- George F. Luger, *Artificial Intelligence: Structures and Strategies for Complex Problem Solving*, Pearson: state-space search, data-driven and goal-driven search, heuristic search.
- Nils J. Nilsson, *Artificial Intelligence: A New Synthesis*, Morgan Kaufmann: heuristic search and A*.
- David E. Goldberg, *Genetic Algorithms in Search, Optimization, and Machine Learning*, Addison-Wesley; Melanie Mitchell, *An Introduction to Genetic Algorithms*.
- Original papers: P. E. Hart, N. J. Nilsson and B. Raphael, "A formal basis for the heuristic determination of minimum cost paths" (1968); S. Kirkpatrick, C. D. Gelatt and M. P. Vecchi, "Optimization by simulated annealing", *Science* (1983); R. E. Korf, "Depth-first iterative-deepening: an optimal admissible tree search", *Artificial Intelligence* (1985).

Open web resources. The writing environment had no web access, so none of these was opened for this book; the links are from memory, so check them:

- AIMA, the Russell and Norvig book site, and its Python code for every search algorithm: github.com/aimacode/aima-python
- Red Blob Games, "Introduction to the A* algorithm", interactive BFS, Dijkstra and A* on grids: redblobgames.com

All worked-example values, and the 8-queens hill-climbing success rates in Chapter 2.6, were computed with the writer's own Python scripts.

Syllabus: the University free n easy universal syllabus, built from top universities' syllabi.

Built for the University free n easy YouTube channel from the universal syllabus of Artificial Intelligence (Oct 2026). Written with top universities' syllabi and standard textbooks; check a formula against your prescribed book before an exam, and report mistakes on the channel.